UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE CIENCIAS EXACTAS
DEPARTAMENTO DE INFORMÁTICA

Una teoría dinámica orientada a objetos como fundamento formal para el proceso de desarrollo de software basado en modelos

Claudia Pons

La Plata, Agosto de 1999

Tesis de doctorado realizada en el Laboratorio de Investigación y Formación en Informática Avanzada (LIFIA),
Universidad Nacional de La Plata
calle 115 esquina 50, 1er Piso
(1900) La Plata
Argentina

Directores

Profesor Gabriel Baum
Dpto. de Informática, Facultad de Ciencias Exactas,
Universidad Nacional de La Plata

Profesor Miguel Felder
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

La Plata, Agosto de 1999
Facultad de Ciencias Exactas, Universidad Nacional de La Plata

A mis padres:

Pablo, María Ofelia, Brígida y Raúl

CONTENIDOS

1. Introducción

1.1 La filosofía del proceso de desarrollo de software

La filosofía del proceso de desarrollo de software, es decir los principios esenciales que guían la construcción de sistemas de software, ha experimentado importantes cambios a través del tiempo. En los comienzos de la computación, el problema de construir programas consistía esencialmente en escribir un conjunto de instrucciones en un lenguaje entendible por una computadora. Cada sistema era un producto intelectual único creado por su propio usuario para realizar una tarea específica dentro de su área de trabajo. Por ejemplo, físicos y matemáticos escribían programas para resolver ecuaciones complejas. El proceso de desarrollo de un programa solamente involucraba al usuario y a la computadora (*figura 1.1*).

A medida que fueron decreciendo los costos y fueron surgiendo lenguajes de alto nivel que facilitaron la comunicación con la máquina, las computadoras se tornaron más populares y la cantidad de usuarios se incrementó rápidamente. En este contexto, la "programación" adquirió jerarquía de profesión: ahora los usuarios tenían la alternativa de contratar a un programador para que escribiera sus programas en vez de escribirlos ellos mismos. Esto introdujo una separación entre el usuario y la computadora (*figura 1.2*). El usuario podía especificar la tarea en un lenguaje menos estricto; luego, el programador interpretaba esta especificación y la traducía en un conjunto preciso de instrucciones expresadas en el lenguaje de la máquina. Por supuesto, existía la posibilidad de que el programador no capturara precisamente las intenciones del usuario y construyera un programa que no satisficiera sus expectativas.

Figura 1.1: desarrollo individual

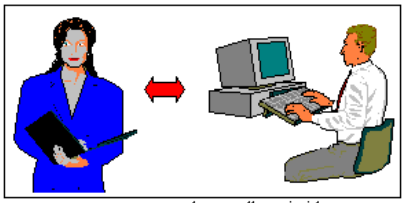


Figura 1.2: desarrollo asistido

Este problema se fue acentuando a medida que el tamaño y la complejidad de los sistemas fueron creciendo. Se descubrió que el problema de construir sistemas de software de gran envergadura no se resolvía simplemente contratando a un grupo numeroso de programadores y reuniendo las líneas de código escritas por cada uno de ellos. El desarrollo de software se había ido transformando en una actividad grupal donde la comunicación e interacción entre cada integrante del grupo era de fundamental importancia (*figura 1.3*).

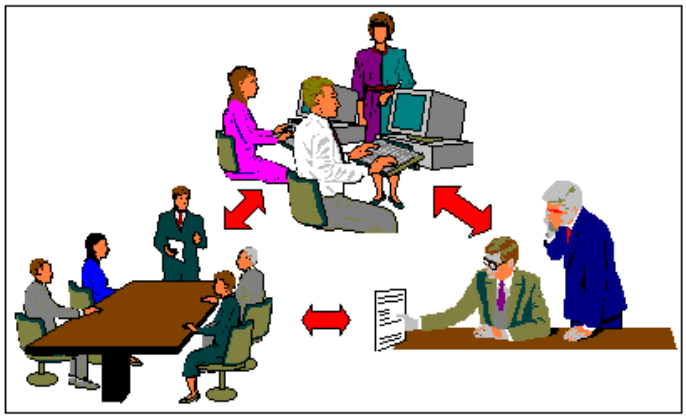


Figura 1.3: desarrollo grupal

La técnica utilizada inicialmente, conocida como “code-and-fix” consistente en escribir el código y luego probarlo una y otra vez para corregir sus errores, fue resultando cada vez menos adecuada, básicamente por dos motivos:

- El tamaño creciente de los sistemas que se desarrollaban hacía difícil tanto localizar y corregir errores como introducir modificaciones en el código para reflejar cambios en la especificación inicial.
- Frecuentemente el producto final no satisfacía los requerimientos del usuario y este problema era descubierto recién en la etapa final del proceso de producción.

Como consecuencia de estos inconvenientes, el proceso de producción de software se volvió no predecible y poco controlable. Los desarrolladores no lograban cumplir ni con los tiempos de entrega ni con los presupuestos establecidos.

Diversas soluciones fueron propuestas y analizadas. Finalmente se llegó a la conclusión de que el problema de construir software debe ser encarado de la misma forma en que los ingenieros construyen otros sistemas complejos, como puentes, edificios, barcos y aviones. La idea básica consistió en observar el sistema de software a construir como un producto complejo y a su proceso de construcción como un trabajo ingenieril. Es decir, un proceso basado en metodologías, técnicas, teorías, herramientas, etc. De esta forma surgió la Ingeniería de Software. Actualmente se conoce a la Ingeniería de Software como una disciplina dentro de las Ciencias de la computación que se ocupa de la construcción de sistemas de software que debido a su gran tamaño y complejidad deben ser construidos por un equipo de ingenieros. Usualmente, estos sistemas de software existen en diferentes versiones y son usados durante varios años. A lo largo de su existencia estos sistemas pueden cambiar, ya sea para corregir errores, introducir mejoras, agregar o eliminar funciones o para adaptase a un nuevo ambiente.

1.2 Las etapas del proceso de desarrollo de software

El proceso de desarrollo de software es el proceso que se sigue para construir un producto de software desde su concepción hasta su retiro. La construcción de un producto de software no sólo involucra escribir código entendible por una máquina, sino que atraviesa por todo un ciclo de vida, comenzando desde la concepción del producto de software, continuando a través del diseño, codificación, mantenimiento y evolución hasta llegar a su retiro.

Los procesos de desarrollo son de fundamental importancia en todas las disciplinas cuyo objetivo sea construir un producto, ya que el proceso permite que la producción sea predecible, confiable y eficiente. Un proceso de desarrollo bien definido (como por ejemplo el proceso de desarrollo de automóviles) permite la automatización de algunas etapas y el uso de componentes standard, lo cual agiliza la producción y reduce costos.

Luego de que la comunidad hubo aceptado que el proceso de desarrollo de software, tal como cualquier otro proceso industrial, está compuesto por varias actividades diferentes, surgió el siguiente interrogante: ¿cómo deben organizarse estas actividades para lograr la calidad deseada en el producto final?

Tratando de dar respuesta a esta pregunta surgieron diversas propuestas. Posiblemente las más aceptadas han sido las siguientes:

Proceso en cascada

El proceso en cascada, ilustrado en la *figura 1.4*, está estructurado como una secuencia de etapas, donde la salida de una etapa constituye la entrada para la etapa siguiente. Cada etapa a su vez está estructurada como un conjunto de actividades que pueden ser ejecutadas por distintas personas concurrentemente. Las etapas mostradas en la figura son las siguientes:

- Estudio de costos y beneficios
- Análisis de requerimientos y especificación
- Diseño
- Codificación y prueba de módulos
- Integración y prueba del sistema
- Instalación
- Mantenimiento

A pesar de que el proceso en cascada no resulta adecuado para el desarrollo de software de nuestros días (por ejemplo, [Parnas and Clements 86] proveen argumentos convincentes de que el proceso en cascada es púramente teórico), su aporte a la ingeniería de software fue muy importante ya que colaboró a que los desarrolladores de software comprendieran dos aspectos importantes, que el proceso de desarrollo de software debe estar sujeto a disciplina, organización y planeamiento, y que la codificación del producto debe posponerse hasta que los objetivos estén bien definidos.

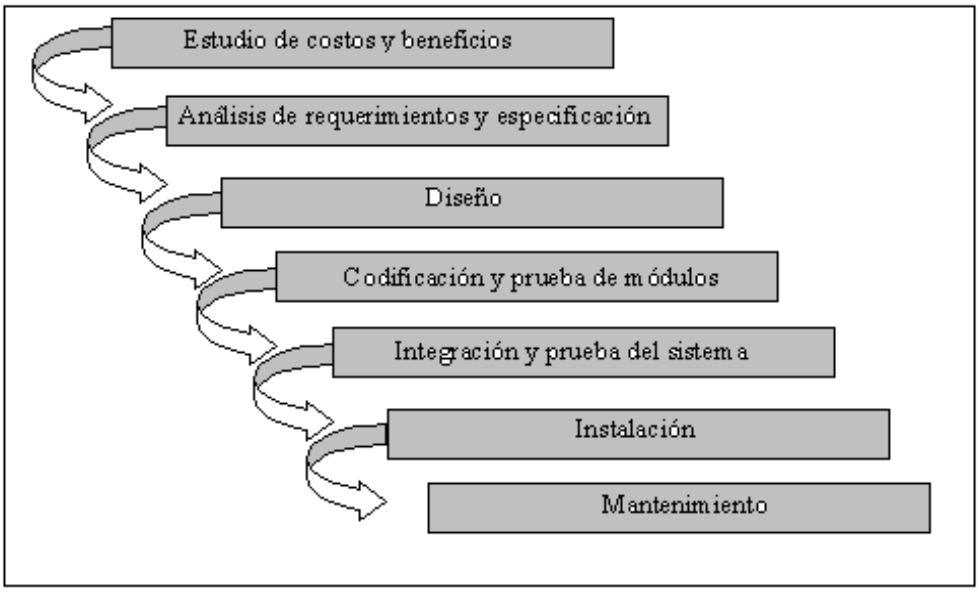


Figura 1.4: proceso de desarrollo en cascada

Proceso incremental por prototipación

Este proceso, ilustrado en la *figura 1.5*, consiste en la construcción de prototipos. Un prototipo es una versión incompleta (o simplificada) del sistema a desarrollar, la cual permite verificar los requerimientos del sistema, medir costos y beneficios, etc. El prototipo puede luego ser descartado permitiendo que el desarrollo del sistema real comience sobre los fundamentos, más sólidos, que se han obtenido. O bien, otra alternativa consiste en transformar progresivamente el prototipo, incorporándole nuevas funciones hasta obtener el sistema final. La principal ventaja de este método es que permite al usuario interactuar con el sistema mucho antes de su finalización, lo cual favorece la detección temprana de errores. Información detallada acerca de esta propuesta puede consultarse en [Gilb 88], [Wong 84] y [Gomaa and Scott 81].

Proceso incremental en espiral

El proceso en espiral fue definido por Boehm [Boehm 88] sobre conceptos de costos de software y análisis de riesgos discutidos en [Boehm and Papaccio 88] y [Boehm 89], respectivamente. El objetivo de este proceso es permitir que las actividades del proceso de desarrollo estén guiadas por los niveles de riesgo de cada proyecto. Riesgos son circunstancias adversas potenciales que pueden perjudicar al proceso de desarrollo y/o a la calidad del producto. Este tipo de ciclo de vida es un proceso dirigido a eliminación de riesgos. Los riesgos técnicos son identificados y priorizados tempranamente en el ciclo de vida y son revisados durante el desarrollo de cada iteración. Los riesgos son vinculados a cada iteración para que al completarla exitosamente se eliminen los riesgos correspondientes. El proceso en espiral es cíclico, involucrando una serie de iteraciones que evolucionan hasta el sistema final, como puede verse en la figura 1.6. Cada iteración se centra en identificar problemas de alto riesgo y darles un trato prioritario, consistiendo de los siguientes procesos: análisis de requerimientos, análisis, diseño, implementación, y prueba. Los desarrolladores no asumen que todos los requerimientos son conocidos al comienzo del ciclo de vida; de hecho el cambio se presupone en todas las fases.

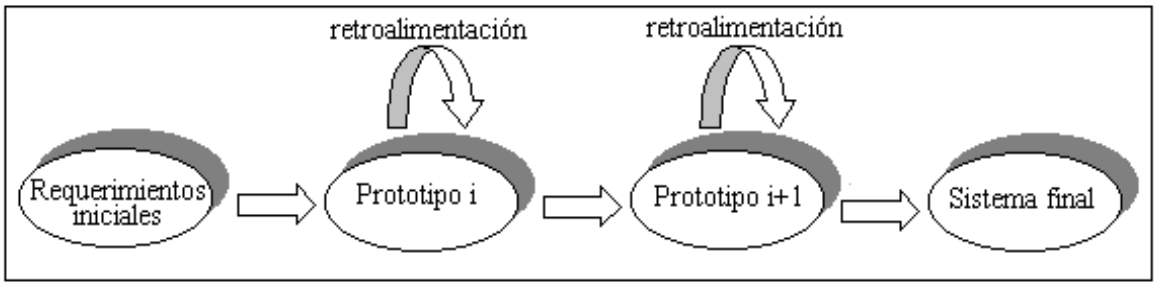


Figura 1.5: proceso de desarrollo incremental por prototipación

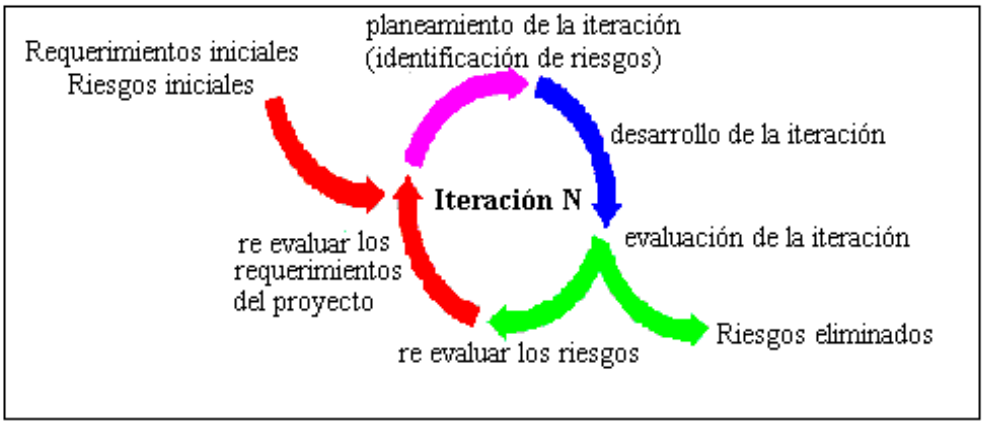


Figura 1.6: proceso incremental en espiral

1.3 Proceso de desarrollo basado en modelos

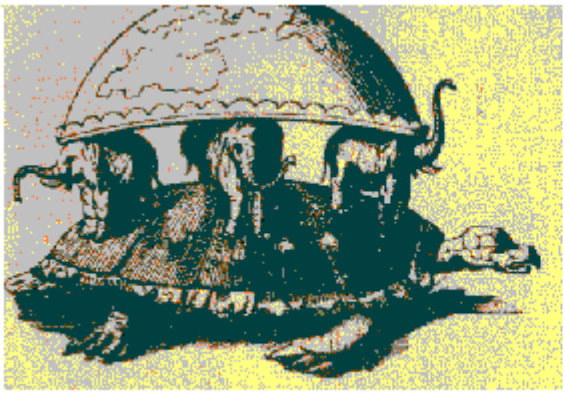
¿Por qué resulta tan difícil construir sistemas de software, aún contando con un proceso de desarrollo bien definido: saber exactamente qué estamos haciendo a medida que avanzamos a través de las etapas del proceso de desarrollo y obtener al final del proceso un sistema que cumpla

con los requerimientos y expectativas del usuario y que sea mantenible, modificable y además que el proyecto pueda concluirse cumpliendo los plazos y recursos asignados?

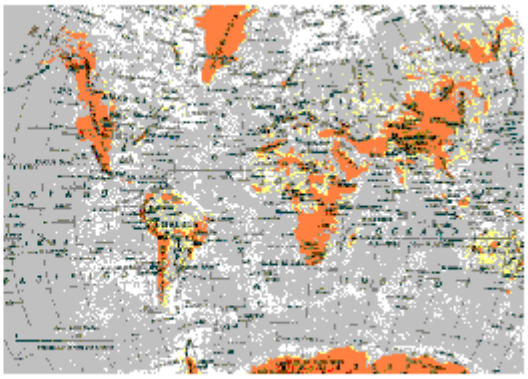
De acuerdo con S. Shlaer y S.Mellor [Shlaer and Mellor] los problemas fundamentales que entorpecen el proceso de desarrollo de software son:

- generalmente los desarrolladores de software no son expertos en el dominio del sistema que deben desarrollar, tales como sistemas financieros, de control de aeropuertos, comunicaciones, etc.
- Aún los expertos en el dominio frecuentemente poseen una idea errónea, ambigua o poco clara de los requerimientos del sistema.
- Las personas involucradas en el proyecto (tanto usuarios como desarrolladores) poseen diferentes esquemas conceptuales del problema y utilizan diferente vocabulario para transmitir sus requerimientos.
- Frecuentemente la especificación del sistema se modifica durante todo su ciclo de vida.

En los finales de los años 70 se observó otro cambio importante en la filosofía del desarrollo de software, tendiente a solucionar los problemas descriptos arriba. Tom DeMarco en su libro *Structured Analysis and System Specification* [DeMarco 79] introdujo el concepto de *ingeniería de software basada en modelos*. DeMarco destacó que la construcción de un sistema de software debe ser precedida por la construcción de un modelo del sistema, tal como se realiza en otros sistemas ingenieriles. De esta forma, el modelo de un sistema provee un medio de comunicación y negociación entre usuarios, analistas y desarrolladores. Actualmente todos los métodos de desarrollo de software han adoptado esta filosofía. Lo que varía de un método a otro es la clase de modelos que deben construirse, la forma de representarlos, manipularlos, etc.



modelo de la tierra según los antiguos hindúes



modelo cartográfico moderno de la tierra

CARACTERÍSTICAS DE LA TIERRA

Área de la superficie terrestre	5,101x10^{8}Km2
Radio dela órbita	1,4953x10^{8}Km2
Radio ecuatorial	6378,38Km
Radio polar	6356,912Km
Achatamiento	1/297
Masa	5,975x10^{24}Kg.
Densidad media	5,517 g/cm3
gravedad media	9,81m/s2
período de traslación	365días y 24hs.
período de rotación	23hs.56'4"

modelo métrico descriptivo de la tierra

modelo medieval de la tierra

Figura 1.7: distintos modelos de la tierra

El punto de partida en el proceso de desarrollo de software es la construcción de un modelo, el cual actúa como una especificación precisa de los requerimientos que el sistema debe satisfacer. Un modelo del sistema consiste en una conceptualización del dominio del problema (ver *figura 1.7*). El modelo se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal.

1.3.1 Utilidad de los modelos

El modelo del sistema se usa básicamente para los siguientes propósitos:

- ***Para definir las necesidades del usuario***. El propósito principal de la especificación de un producto es definir las necesidades de los usuarios del producto.
- ***Como un medio de comunicación y negociación*** entre los usuarios y los desarrolladores y entre los distintos desarrolladores entre sí.
- ***Como un documento de referencia durante la corrección de errores*** en el producto. Luego de introducir modificaciones en el sistema, la especificación es necesaria para chequear que la nueva implementación realmente está corrigiendo los errores contenidos en la versión previa del producto.
- ***Como un documento de referencia durante la evolución*** del producto. En el caso de tener que adaptar el producto debido a cambios en los requerimientos, la especificación original debe ser adaptada para reflejar estos cambios consistentemente.

Se ha observado que la construcción de modelos es una técnica muy efectiva para detectar y resolver discrepancias entre los divergentes puntos de vista de los usuarios acerca de sus requerimientos, brindando así bases firmes para las siguientes etapas del proceso de desarrollo.

1.3.2 Los modelos a través del proceso de desarrollo del software

En Ingeniería de Software, tal como en otras disciplinas, existen distintas metodologías para la construcción de modelos. Los dos principales enfoques son:

- Algorítmico
- Orientado a Objetos

El desarrollo de software tradicional ha tenido un enfoque algorítmico, donde las principales piezas de un sistema son procedimientos y funciones que se ejecutan sobre estructuras de datos estáticas. En cambio el desarrollo de software contemporáneo sigue un enfoque orientado a objetos, donde el elemento central del sistema de software es el Objeto (o clase de objetos). Un objeto es un elemento perteneciente al dominio del problema o al dominio de la solución; una clase es una descripción de un grupo de objetos; cada objeto tiene una identidad (es decir, es distinguible de los otros objetos), tiene un estado (dado por los valores de sus atributos y relaciones) y un comportamiento (dado por la forma en que el objeto reacciona al recibir determinados mensajes). Una descripción detallada del paradigma de objetos puede encontrarse en [Budd 91, Booch 94, Coad and Yourdon 91, Shlaer and Mellor 88].

Durante el proceso de desarrollo de software diferentes modelos del sistema en construcción son creados, tal como es ilustrado en la figura 1.8. Las diferencias entre estos modelos residen en los aspectos del sistema que son contemplados (ningún modelo representa al sistema completo, sino que cada modelo hace énfasis en una parte del sistema) y en el grado de abstracción (en las primeras etapas del ciclo de vida se construyen modelos más abstractos, que luego son sustituidos y/o complementados por modelos más concretos). Por ejemplo los modelos de análisis capturan sólo los requerimientos esenciales del sistema de software, describiendo lo que el sistema hará independientemente de cómo se implemente. Por otro lado, los modelos de diseño y los modelos de implementación describen como el sistema será construido en el contexto de un ambiente de

implementación determinado (plataforma, sistema operativo, bases de datos, lenguajes de programación, interfaces, etc.).

Los distintos modelos están relacionados entre sí en dos direcciones: horizontal y vertical. Cada plano vertical está formado por un grupo de submodelos que conforman una visión completa del sistema a un cierto nivel de abstracción (o en un cierto punto de su ciclo de vida). Las relaciones horizontales representan evolución de un modelo a través del proceso de desarrollo, por ejemplo la relación entre un modelo de análisis y un modelo de diseño.

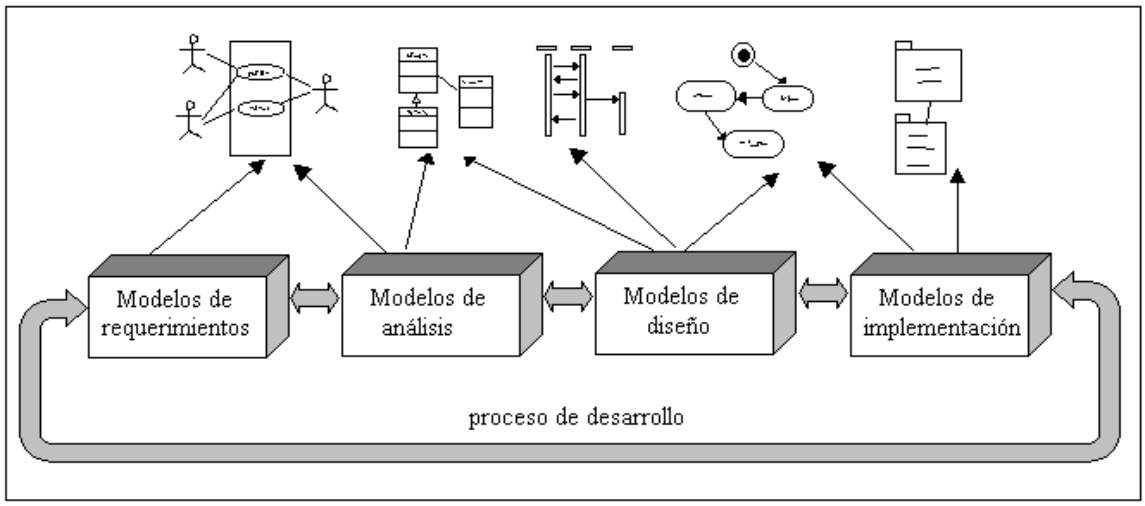


Figura 1.8: los modelos a través del proceso de desarrollo

Los procesos de desarrollo incremental en espiral, como por ejemplo Rational Objectory Process [UML 97 (c)], están estructurados a lo largo de dos dimensiones:

- ***Tiempo*** (división del ciclo de vida en fases e iteraciones).
- ***Componentes*** del proceso (producción de un conjunto de artefactos o modelos específicos para representar un aspecto del sistema).

La dimensión del tiempo involucra las siguientes fases:

- *Introducción*: la especificación de la visión del proyecto.
- *Elaboración*: la planificación de las actividades y recursos requeridos; especificando características y diseñando la arquitectura.
- *Construcción*: la construcción del producto como series de iteraciones incrementales.
- *Transición*: la provisión del producto a la comunidad de usuarios (manufactura, entrega, y entrenamiento).

La dimensión de componentes del proceso incluye las siguientes actividades:

- *Análisis de Requerimientos*: la descripción de lo que el sistema debería hacer.
- *Diseño*: cómo será realizado el sistema en la fase de implementación.
- *Implementación*: la producción del código que resultará en un sistema ejecutable.

- *Prueba*: la verificación del sistema completo.

Cada actividad de la dimensión de componentes del proceso es aplicada en cada fase de la dimensión basada en el tiempo. La figura 1.9 muestra cómo los componentes de proceso son desarrollados a través del tiempo.

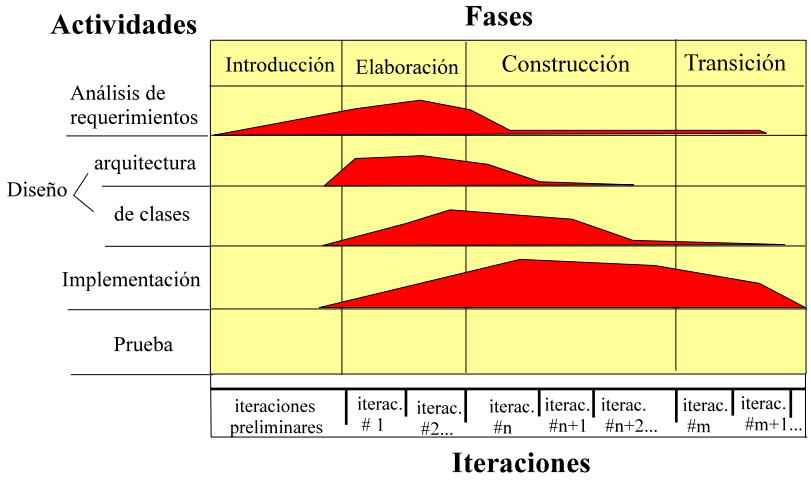


Figura 1.9: las dos dimensiones en el proceso de desarrollo

1.3.3 Cualidades de los modelos

El modelo de un problema es esencial para describir y entender el problema, independientemente de cualquier posible sistema informático que se use para su automatización. El modelo constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema, los analistas y los desarrolladores de software. Por lo tanto es de fundamental importancia que exprese la esencia del problema en forma clara y precisa. Por otra parte, la actividad de construcción del modelo es una parte crítica en el proceso de desarrollo. Los modelos son el resultado de una actividad compleja y creativa y por lo tanto son propensos a contener errores, omisiones e inconsistencias. La verificación del modelo es muy importante, ya que los errores en esta etapa tienen un costoso impacto sobre las siguientes etapas del proceso de desarrollo de software.

En esta sección discutiremos las cualidades relevantes para los modelos de análisis:

- ***El modelo debe ser claro, no ambiguo y entendible***. Para que el modelo resulte útil debe ser accesible (es decir entendible y manejable) para todos sus usuarios y debe poseer una semántica precisa. Como fue destacado por Wittgenstein, “the silent adjustments to understand colloquial language are enormously complicated” [Witt]. La falta de precisión semántica es un problema que no solamente atañe al lenguaje natural, sino que también abarca a los lenguajes gráficos de modelado. Esto se debe principalmente a la existencia de conceptos cuya interpretación semántica difiere para diferentes personas (incluyendo a las personas que crean el modelo y las personas que lo leen) Si el modelo del problema es incompleto, vago o inconsistente, o si se presta a interpretaciones erróneas, el resultado será un sistema de software cuya funcionalidad real difiera considerablemente de la funcionalidad esperada por sus usuarios.
- ***El modelo debe ser consistente,*** es decir que no debe contener información contradictoria. Dado que un sistema es representado a través de diferentes sub-

modelos relacionados horizontal y verticalmente (tal como fue discutido en la sección anterior), debería ser posible especificar precisamente cual es la relación existente entre ellos, de manera que sea posible garantizar la consistencia del modelo compuesto.

- ***El modelo debe ser completo***, es decir que debe documentar todos los requerimientos necesarios. Dado que en general, no es posible lograr un modelo completo desde el inicio del proceso, es importante poder incrementar el modelo. Es decir, comenzar con un documento de especificación incompleto y expandirlo a medida que se obtiene más información acerca del dominio del problema (tal vez aportada por experiencias con prototipos tempranos del sistema).
- ***El modelo debe ser modificable.*** Debido a la naturaleza cambiante de los sistemas actuales, es necesario contar con modelos flexibles, es decir que puedan ser fácilmente adaptados para reflejar las modificaciones en el dominio del problema.
- ***El modelo debe ser reusable***. El modelo de un sistema, además de describir el problema, también debe proveer las bases para el reuso de conceptos y construcciones que se presentan en forma recurrente en una amplia gama de problemas. El reuso permite economizar esfuerzo intelectual, tiempo y dinero.
- ***El modelo debe ser verificable.*** Existen dos aspectos a tener en cuenta, primero el modelo en sí debe ser verificado para asegurar que cumple con las expectativas del usuario. Luego, asumiendo que el modelo es correcto, puede usarse como referencia para verificar la corrección de las implementaciones del sistema. Existen dos formas generales de verificar la funcionalidad de un modelo. Una consiste en observar el comportamiento dinámico del sistema modelado con el objetivo de analizar si corresponde o no con el entendimiento intuitivo que se tiene del sistema a construir. Esta técnica es llamada simulación. Cuando la simulación es automatizable el modelo puede usarse como un prototipo del sistema. Otra forma de verificación consiste en analizar las propiedades que pueden ser deducidas a partir del modelo y confrontarlas contra las propiedades esperadas para el sistema. Por otra parte verificar un modelo no sólo consiste en analizar su funcionalidad, sino que también involucra el análisis de las propiedades de completitud y consistencia.

1.3.4 Modelos formales vs. modelos no-formales

El modelo del sistema se construye utilizando un lenguaje de modelado (que puede variar desde lenguaje natural o diagramas hasta formulas matemáticas). Los modelos informales son expresados utilizando lenguaje natural, figuras, tablas u otras notaciones. Hablamos de modelos formales cuando la notación empleada es un formalismo, es decir posee una sintaxis y semántica (significado) precisamente definidos. Existen estilos de modelado intermedios llamados semi-formales, ya que en la práctica los ingenieros de software frecuentemente usan una notación cuya sintaxis y semántica están sólo parcialmente formalizadas.

El éxito de los lenguajes gráficos de modelado, tales como Object Oriented Analysis (OOA) [Coad and Yourdon 91], Object Oriented system Analysis [Schlaer and Mellor 88], Object Modeling Technique (OMT) [Rumbaugh et al. 91], Booch's design method [Booch 94], and the Unified Method Language (UML) [UML 97(a)(b), UML 98(a)(b)] se basa principalmente en el uso de construcciones gráficas que transmiten un significado intuitivo; por ejemplo un cuadrado representa un objeto, una línea uniendo dos cuadrados representa una relación entre ambos objetos. Estos lenguajes resultan atractivos para los usuarios ya que aparentemente son fáciles de entender y aplicar. Sin embargo, la falta de precisión en la definición de su semántica puede originar problemas, por ejemplo:

- Malas interpretaciones de los modelos: la interpretación que realiza el usuario que lee el modelo puede no coincidir con la interpretación que realizó el creador del modelo.
- Inconsistencia entre los diferentes modelos del sistema: la relación existente entre los diferentes sub-modelos (por ejemplo modelos de la estructura estática, modelos del comportamiento dinámico, etc.) que componen el modelo de un sistema no está precisamente especificada. Por lo tanto no es posible analizar si su integración es consistente o no lo es.
- Discusiones acerca del significado del lenguaje: dado que el significado de algunas construcciones del lenguaje no está precisamente definido, las personas involucradas en el proyecto suelen perder tiempo discutiendo las diferentes posibles interpretaciones que pueden asignarse al lenguaje.

Por otro lado, los lenguajes formales de modelado, tales como Z [Spivey 92], VDM [Jones 90], F-Logic [Kifer and Lausen 90], DS-Logic [Wieringa and Broersen 98] poseen una sintaxis y semántica bien definidas. Sin embargo su uso en la industria es poco frecuente. Esto se debe a la complejidad de sus formalismos matemáticos que son difíciles de entender y comunicar. En la mayoría de los casos los expertos en el dominio del sistema que deciden utilizar una notación formal, focalizan su esfuerzo sobre el manejo del formalismo en lugar de hacerlo sobre el modelo en sí. Esto conduce a la creación de modelos formales que no reflejan adecuadamente al sistema real.

La necesidad de integrar lenguajes gráficos, cercanos a las necesidades del dominio de aplicación con técnicas formales de análisis y verificación puede satisfacerse combinando ambos tipos de lenguaje. La idea básica para obtener una combinación útil consiste en ocultar los formalismos matemáticos detrás de la notación gráfica. De esta manera el usuario solo debe interactuar con el lenguaje gráfico, pero puede contar con la base formal provista por el esquema matemático subyacente. Esta propuesta ofrece claras ventajas sobre el uso de un lenguaje informal así como también sobre el uso de un lenguaje formal, ya que permite que los desarrolladores de software puedan crear modelos formales sin necesidad de poseer un conocimiento profundo acerca del formalismo que los sustenta. Un lenguaje que posea estas características será fácilmente aceptado tanto por parte de los ingenieros de software, como por parte de los usuarios. La figura 1.10 esquematiza los conceptos discutidos en esta sección.

1.4 Acerca de esta tesis

1.4.1 Desarrollo y objetivos

La primera etapa del trabajo consistió en el análisis de las diferentes técnicas de modelado orientado a objetos y su influencia sobre el proceso de desarrollo de software basado en modelos. Este análisis nos condujo a reconocer las claras ventajas que ofrece la integración de técnicas de modelado formales con técnicas no formales aceptadas y usadas por los ingenieros de software típicos.

El siguiente paso consistió en estudiar las propuestas existentes acerca de cómo efectivizar la mencionada integración, con el objetivo de identificar aspectos potencialmente mejorables.

Finalmente y como consecuencia del análisis previo, definimos una nueva propuesta de integración la cual aporta los beneficios esperados para un método de integración standard pero además incorpora ciertas características que no han sido cubiertas satisfactoriamente por las propuestas anteriores, tales como evolución, reusabilidad y métricas de modelos. Además nuestra propuesta se basa en una estructura formal de primer orden que, en contraste con las estructuras de orden superior, facilita los procedimientos para calcular la validez de las fórmulas.

1.4.2 Organización

La parte restante de esta tesis está organizada de la siguiente forma:

En los capítulos 2 y 3 describimos detalladamente un lenguaje gráfico de especificación (UML) y un lenguaje formal (Lógica Dinámica) respectivamente. En el capítulo 4 discutimos las distintas propuestas para lograr la integración de ambas técnicas. En el capítulo 5 presentamos nuestra propuesta: la M&D-theory. El capítulo 6 contiene ejemplos de los principales beneficios standard provistos por la M&D-theory. En los capítulos 7, 8 y 9 demostramos la utilidad de nuestra formalización para expresar evolución de modelos, métricas de calidad y temas de reuso tales como contratos y patrones de diseño. Finalmente el capítulo 10 contiene conclusiones, reflexiones y comentarios acerca del trabajo presentado.

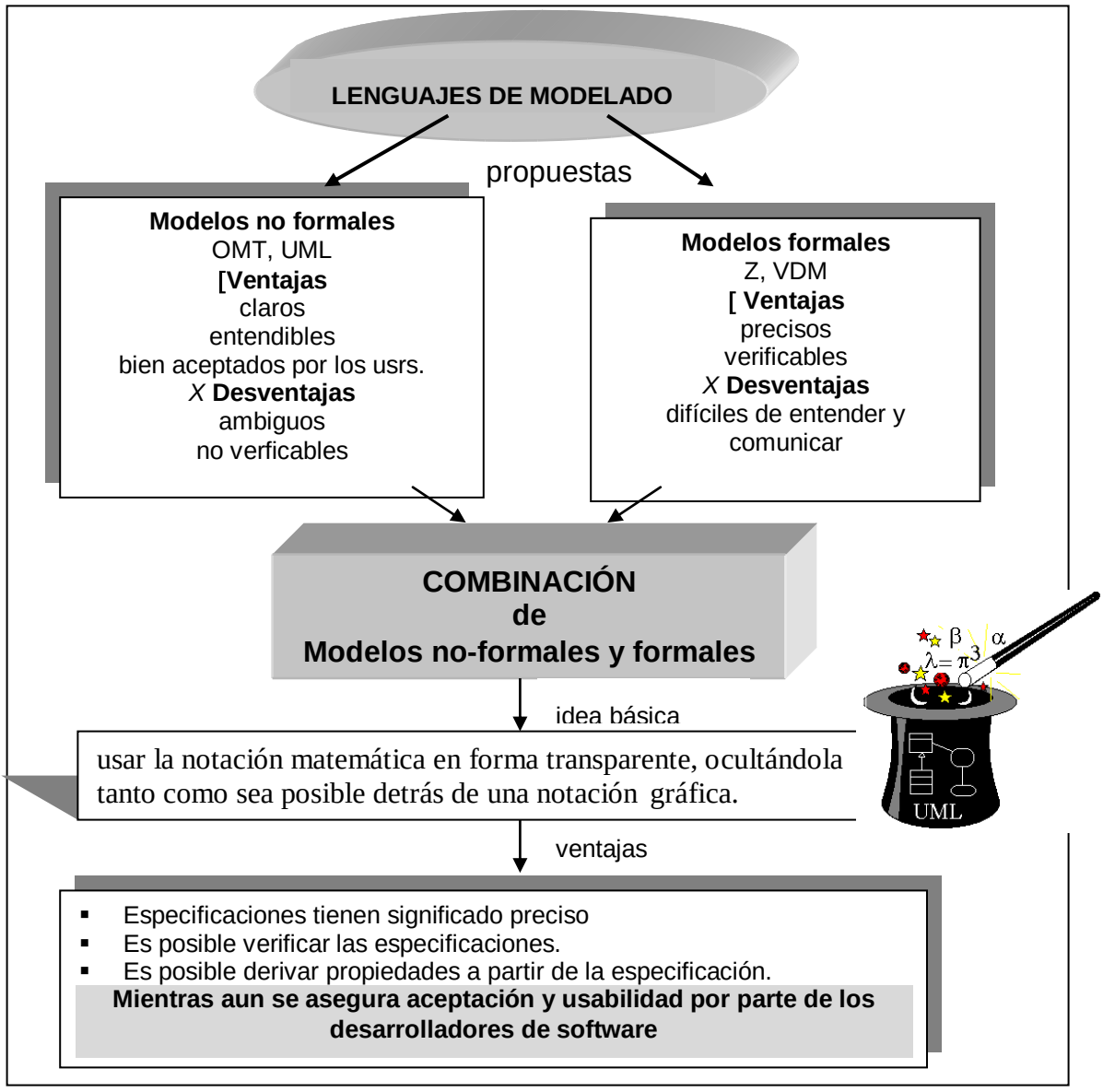


Figura 1.10: combinando lenguajes de modelado

1.5 Publicaciones

Los resultados de esta tesis han sido parcialmente presentados en los siguientes artículos:

- Foundations of Object-oriented modeling notations in a dynamic logic framework, C.Pons, G.Baum, M.Felder, In Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, 1999.

- El proceso de desarrollo de software basado en modelos. C.Pons, proceedings of CACIC'99, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina. Nov 1999.

- Model Evolution and System Evolution, Claudia Pons, Ralf-D Kutsche, proceedings of CACIC'99, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina. Nov 1999.

- Precise semantics of model evolution, C.Pons, R.Giandini. proceedings of IDEAS´99, San Jose, Costa Rica, 24-26 March 1999.

- Integrating object-oriented model with object-oriented meta-model into a single formalism, Claudia Pons, G.Baum and M.Felder, Second ECOOP Workshop on Precise Behavioral Semantics, European Conference on Object-oriented Programming, Brussels, Belgium, Ed: H.Kilov, B.Rumpe, Technische Universitat Munchen, Report TUM-I9813, ECOOP'98 Workshop Readers, Lectures Notes in Computer Science. July 1998.

- Dimensions and Dichotomy in Metamodeling, Robert Geisler, Marcus Klar and Claudia Pons, proc. of Third BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, September 1998, Series in Computing, Springer-Verlag.

- A Formal Approach to Practical Object Oriented Analysis and Design, S.Waldoke, C.Pons, C.Paz and M.Felder, 27avas. Jornadas Argentinas de Informática e Investigación Operativa y ASOO (Simposio en Orientación a Objetos). Buenos Aires, September 1998.

- A dynamic Logic Model for the Formal Foundation of Object-oriented Analysis and Design, C. Pons, G.Baum and M.Felder, XVIII International Conference of the Chilean Computer Science Society (SCCC'98), IEEE Computer Science Press, Chile, November 1998.

- Una herramienta para verificación formal de especificaciones gráficas de objetos, C.Pons, R.Giandini, G.Baum, Actas de II Jornadas de Ingeniería de Software JIS'97, editadas por O.Diaz y P.Lopistéguy, San Sebastián, España, Septiembre 1997.

- A User Friendly Algebraic Proposal for a Formal Verification of Object-Oriented Diagrams, Claudia Pons. Paper presented at the OOPSLA'97 Doctoral Symposium, Atlanta, USA, October 1997.

2. El lenguaje UML

A medida que el valor estratégico del software aumenta en la industria, se torna necesario contar con técnicas para automatizar la producción de software. En respuesta a esta necesidad se buscan técnicas para aumentar la calidad y reducir el costo y el tiempo de implementación. Éstas técnicas incluyen desarrollo basado en componentes, programación visual, patrones de programación y ambientes integrados de desarrollo. También se buscan técnicas para manejar la complejidad de los sistemas a medida que aumentan su alcance y escala. Dado que la complejidad varía según el dominio de aplicación y la fase del desarrollo, se debe hacer énfasis en crear una tecnica que pueda contemplar adecuadamente toda la variedad de complejidad arquitectural a través de todas las fases del desarrollo en los distintos tipos de dominio.

Un aspecto fundamental al modelar sistemas es incluir las mejores prácticas en la industria, pudiendo contemplar diversas vistas basadas en niveles de abstracción, dominios, arquitectura, etapas de desarrollo, técnicas de implementación, etc. También resulta de fundamental importancia contar con un lenguaje de modelización standard que permita expresar la información de manera clara y precisa y que sea conocido y aceptado por la comunidad. Antes de la aparición del Unified Modeling Language (UML), la mayoría de los lenguajes de modelización poseían un conjunto comúnmente aceptado de conceptos pero expresados de formas diferentes. Esta falta de consenso dificultaba el uso de las distintas metodologías orientadas a objetos. Por este motivo nace UML, para unificar las diferentes notaciones. Al converger sobre UML se han unificado muchas de las diferencias, muchas veces sólo aparentes, entre los lenguajes de modelización de métodos previos. También se ha logrado unificar las perspectivas entre muchos diferentes tipos de sistemas, fases de desarrollo y conceptos internos.

UML es un lenguaje gráfico para especificar, construir, visualizar y documentar los componentes de un sistema de software. Está basado en los conceptos de Booch[Booch 94], OMT[Rumbaugh et al. 91] y OOSE[Jacobson]. Conforma un único, común y ampliamente usado lenguaje de modelización para usuarios de estos métodos y otros (figura 2.1). UML está focalizado en estandarizar un lenguaje de modelización, y no en estandarizar los procesos de desarrollo. Aunque puede ser aplicado en el contexto de un proceso en particular, se sabe que diferentes organizaciones y dominios de problema requieren de procesos diferentes. Cada metodología en particular utilizará un conjunto de elementos del lenguaje.

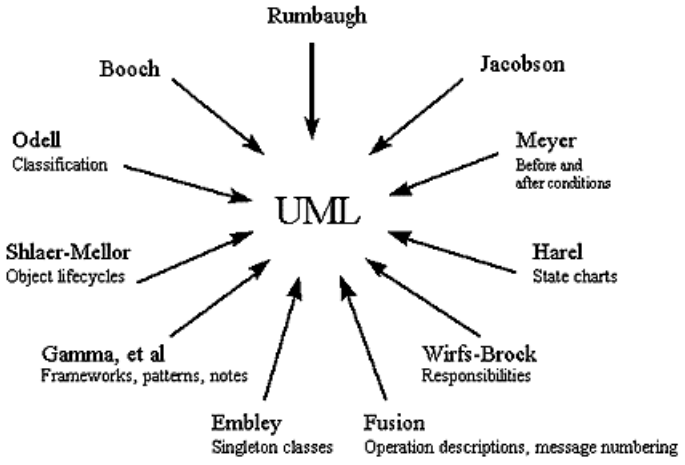


Figura 2.1: Unified Modeling Language

Los objetivos que condujeron a la creación de UML son los siguientes:

- Proveer a los usuarios un lenguaje de modelización expresivo y listo para usarse para que puedan desarrollar e intercambiar modelos. Se busca evitar la personalización de las herramientas para lograr un intercambio en los mismos términos entre los usuarios. Lo único que permanece entre aplicaciones diferentes son los conceptos básicos de modelización; UML estandariza dichos conceptos.
- Proveer mecanismos de extensibilidad y especialización de los conceptos básicos. Mientras que no se quiere que los conceptos básicos sean redefinidos o reimplementados para cada área de aplicación, UML debe ser personalizado para dominios específico. Las adaptaciones de UML serán las mínimas que se requieran para su adaptaciónal dominio. Los usuarios deberían: 1- construir modelos usando conceptos básicos sin usar los mecanismos de extensión para las aplicaciones más normales; 2- agregar nuevos conceptos y notaciones para problemas no cubiertos por los conceptos básicos; y 3- especializar los conceptos, notaciones y restricciones para dominios particulares.
- Mantener independencia de lenguajes de programación y procesos de desarrollo. UML soporta lenguajes de programación y métodos de desarrollo sin excesiva dificultad.
- Proveer una base formal para entender el lenguaje de modelización. UML define formalmente el modelo estático usando un metamodelo expresado en diagramas de clase expresados en el mismo lenguaje UML. Las restricciones se expresan en lenguaje natural complementado con el lenguaje Object Constraint Language [OCL 97].
- Fomentar el crecimiento del mercado de herramientas de orientación a objetos. Para permitir que terceros puedan agregar valor a sus implementaciones, es esencial asegurar la interoperabilidad. Esto requiere que los modelos puedan ser intercambiados entre usuarios y herramientas sin pérdida de información. Esto se logra si el formato y el significado de los conceptos relevantes es compartido por todas las herramientas.
- Proveer de conceptos de desarrollo de más alto nivel como colaboraciones, ambientes de trabajo, patrones y componentes. La definición semántica clara de estos conceptos es esencial para lograr los beneficios de la programación orientada a objetos y para la reusabilidad de los modelos.
- Integrar las mejores prácticas. Una motivación clave detrás del desarrollo de UML ha sido la integración de las mejores prácticas de la industria, logrando tener varias vistas basadas en niveles de abstracción, dominios, arquitectura, etapas de desarrollo del software, técnicas de implementación, etc.

2.1 Historia

UML es un intento de estandarizar las herramientas del análisis y el diseño: modelos semánticos, notación sintáctica, y diagramas. La figura 2.2 muestra la evolución del UML a través del tiempo. El primer borrador público (versión 0.8) fue introducido en Octubre de 1995. El *feedback* del público y el aporte de Ivar Jacobson fueron incluidos en las siguientes dos versiones (0.9 en Julio de 1996, y 0.91 en Octubre de 1996). La versión 1.0 fue presentada al *Object Management Group (OMG) para su estandarización en Enero de 1997.*

Booch, OMT, OOSE y muchos otros métodos poseen procesos bien definidos. UML soporta a la mayoría de ellos. A pesar de existir cierta convergencia en los procesos de desarrollo, aún no hay un consenso para su estandarización. Aunque UML no define un proceso específico, sus desarrolladores han reconocido la importancia de los procesos *Dirigidos por Casos de Uso*, *Centrados en Arquitectura*, *Iterativos* e *Incrementales*. Por este motivo se ha puesto especial cuidado en que UML permita estos procesos, sin exigirlos.

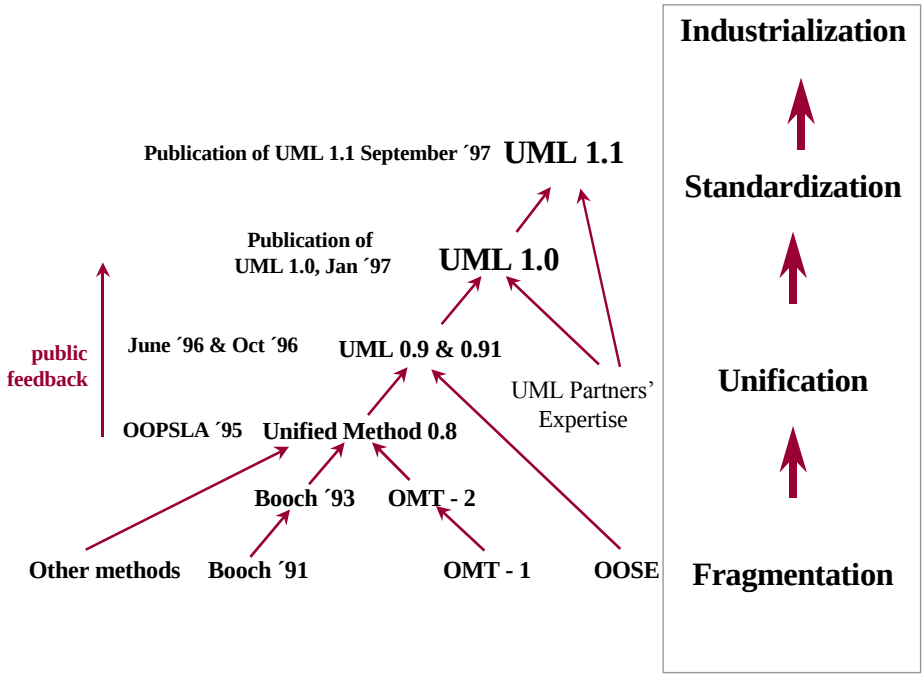


Figura 2.2: evolución del UML

Este esfuerzo de unificación de un lenguaje de modelización está motivado en tres aspectos: Primero, cada uno de estos métodos ya estaba convergiendo uno hacia otro independientemente. Segundo, dar cierta estabilidad al desarrollo orientado a objetos unificando notación y semántica dando un sólido basamento a los proyectos en desarrollo y a los creadores de herramientas. Tercero, esta colaboración podría identificar mejoras a los métodos previos que no habían sido detectadas antes.

Durante esta convergencia, se tuvieron en mente cuatro objetivos:

- Modelar sistemas usando conceptos de orientación a objetos.
- Establecer una relación explícita entre conceptos y artefactos ejecutables.
- Tener en cuenta aspectos de escalabilidad (críticos en sistemas complejos y de misión crítica).
- Crear un lenguaje de modelización manejable tanto para humanos como máquinas.

2.2 Principales Componentes de UML

La elección de qué modelos y diagramas se crean al especificar, tiene un profundo impacto sobre la forma en que un problema es abordado y resuelto. El mecanismo de abstracción es una poderosa herramienta que permite el tratamiento preferencial de los detalles más relevantes ignorando el resto, en particular:

- Cada sistema complejo es más accesible a través de un pequeño conjunto de vistas del modelo casi independientes entre ellas. Ninguna vista es suficiente por sí sola.
- Cada modelo puede ser expresado a distintos niveles de fidelidad con la realidad.
- Los mejores modelos son los conectados a la realidad.

UML permite definir, a través de distintos diagramas, las distintas vistas que componen un modelo. Estos diagramas proveen múltiples perspectivas de un sistema que está siendo analizado o desarrollado. El modelo subyacente integra estas perspectivas permitiendo de esta manera la construcción de un sistema consistente y autocontenido.

Un diagrama es una proyección del modelo; los mismos elementos o conceptos pueden aparecer en más de un diagrama. Cada diagrama puede presentar alguno – pero no necesariamente todos – los detalles de un elemento en particular.

Los principales diagramas incluidos en UML caen dentro de cuatro categorías: de Casos de Uso, de Estructura Estática, de Comportamiento y de Restricciones. A continuación describiremos cada una de estas categorías:

(I) Diagramas de Casos de Uso:

Los diagramas de casos de uso o *use case diagrams* son diseñados a fin de representar la información del análisis de requerimientos. Están conformados por actores y casos de uso. Los actores representan los individuos que interactúan con el sistema, es decir todo lo que necesita intercambiar información con el sistema. En general, los casos de uso representan una secuencia cualquiera de transacciones llevadas a cabo cuando un usuario dialoga con el sistema.

Está compuesto por un grafo de actores, un conjunto de use cases encerrados por un delimitador del sistema, asociaciones de comunicación entre los actores y los use cases, y generalizaciones entre los use cases.

Los use cases se dibujan con elipses, y los actores con un rectángulo de clase o un icono de estereotipo, que por defecto es una figura de un hombrecillo (ver figura 2.3).

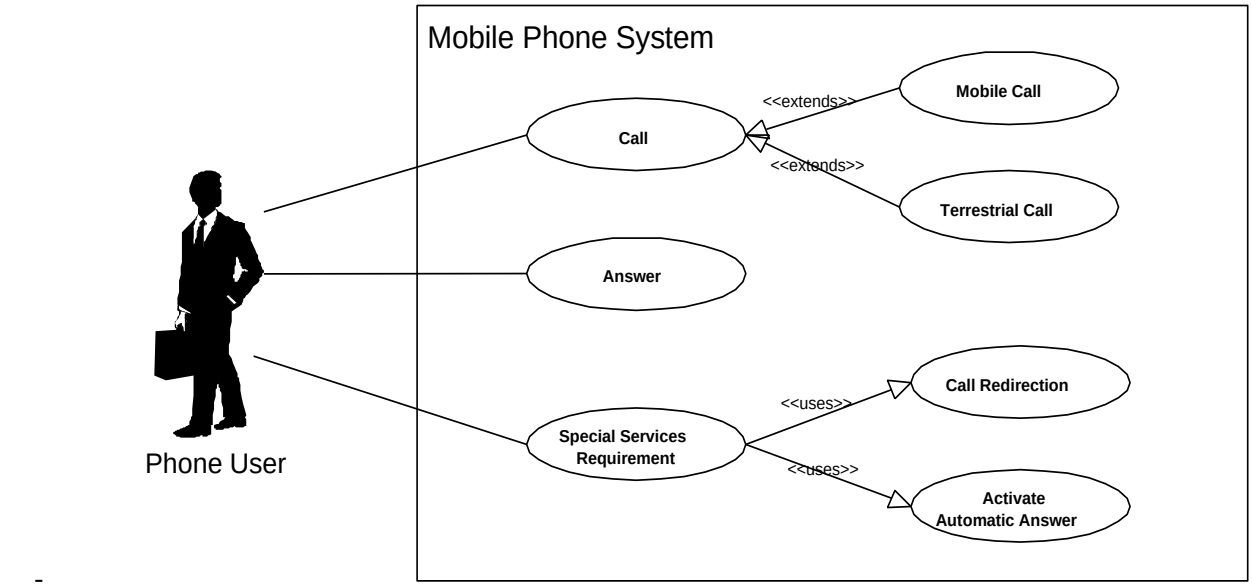


Figura 2.3: Diagrama de Casos de Uso

Hay diversos tipos de relaciones implicados en estos diagramas:

- ***Communicates***: La participación de un actor en un use case se muestra conectándolo al símbolo de use case por un camino sólido.

- ***Extends***: Una relación extends entre use cases es mostrado por una flecha de generalización desde el use case que provee las extensiones hacia el use case base. La flecha se rotula con el estereotipo <<extends>>.
- ***Uses***: Una relación uses entre use cases se muestra por medio de una flecha de generalización desde el use case que usa hacia el que es usado. La flecha se rotula con el estereotipo <<uses>>.

(II) Diagramas de Estructura Estática:

Los diagramas de estructura estática o *Class Diagrams* muestran la estructura estática del modelo, en particular, las cosas que existen (como clases y tipos), su estructura interna y sus relaciones con otras cosas. En estos diagramas (figura 2.4) encontramos, principalmente, clases y sus variaciones (templates o patrones y clases instanciadas) y relaciones entre clases (asociaciones y generalizaciones). Las clases expresan las entidades del sistema con sus atributos y operaciones. Las relaciones de asociación definen una dependencia semántica entre entidades. Las relaciones de generalización muestran la herencia entre entidades; dicha herencia permite modelar el traspaso de estructura y comportamiento de las entidades que actúan en la relación como padre a las entidades que actúan como hijos. Existen otras relaciones, como por ejemplo: dependencias (conexión semántica especial entre entidades), refinamientos (relaciones entre entidades que representan el mismo concepto, pero a distintos niveles de abstracción), etc.

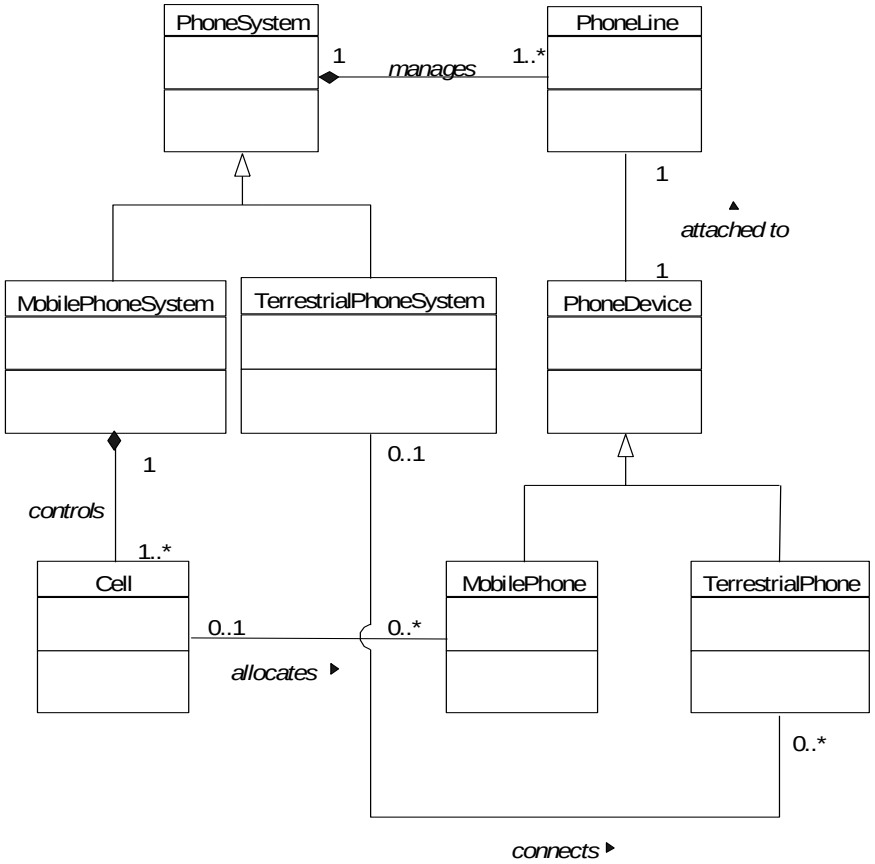


Figura 2.4: Diagrama de estructura estática

Los principales diagramas de este grupo son los siguientes:

- Package (Paquete)
- Class (Clase)
- Relationship (Relación)

No existe ninguna restricción en cuanto a la agrupación de cada uno de estos elementos dentro de un solo diagrama o en varios, depende solamente de la conveniencia al especificar.

Package

Una Package o Paquete permite agrupar elementos de modelado heterogéneos. Un Package define propiedades de alcanzabilidad y visibilidad de los elementos. Se representa mediante un rectángulo con una etiqueta en el vértice superior derecho, el nombre del package puede colocarse dentro de la etiqueta o dentro del cuerpo del package (ver figura 2.5).

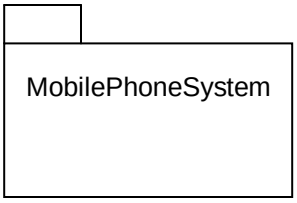


Figura 2.5: Package

Class

Una Clase (o Class) es un descriptor para un conjunto de objetos con similar estructura, comportamiento y relaciones. El nombre de la clase debe ser único dentro del Package al cual pertenece. Una *Clase* está compuesta por métodos y por atributos, se denota como un rectángulo con tres compartimientos: el primero para el nombre y otras propiedades generales; el segundo para la lista de atributos y el último para la lista de métodos u operaciones (ver figura 2.6).

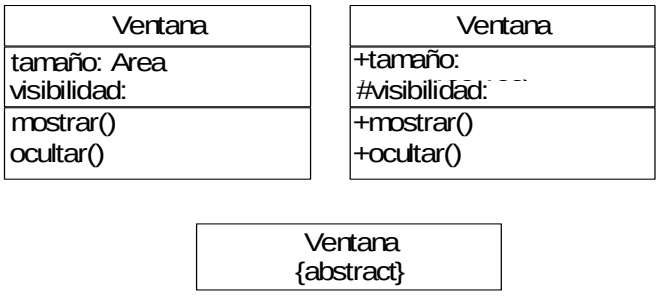


Figura 2.6: Class

Un atributo (o *Attribute*) es una característica estructural de una *Clase*. Cada *Attribute* tiene un tipo asociado, un valor inicial y una visibilidad (*public*, *protected*, *private*, etc.). Por último, se puede notar si está definido a nivel de *Clase* o de *instancia*. Una *operación e*specifica una característica de comportamiento de una *Clase*. Para cada operación se indica si es polimórfica, abstracta y si modifica o no el estado del sistema. Cada *Operation* tiene parámetros, un tipo asociado, valores iniciales y una visibilidad (*public*, *protected*, *private*, etc.). Por último, se puede notar si está definida

a nivel de *Clase* o de *instancia*. La implementación d las operaciones es típicamente especificada explícitamente proveyendo un valor para el cuerpo (típicamente en un lenguaje de programación cuyo alcance está afuera de UML) o implícitamente derivada a partir de los diagramas de comportamiento.

Association

Es una conexión semántica bidireccional entre instancias de tipos (ver figura 2.8). Una *Association* es n-aria, como caso particular puede ser binaria. Esta aridad determina la cantidad de *Association Roles* que la *Association* tiene (por ejemplo, al ser binaria tendrá dos). Es notada como una línea continua que une a las instancias.

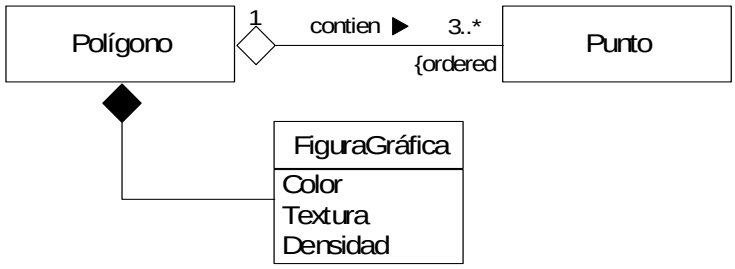


Figura 2.8: Association Role

La responsabilidad de *Association Role* es especificar el papel que un *Type* juega en una *Association*. Cada *Association Role* tiene los siguientes atributos:

- Nombre : Sirve para nombrar el rol en la *Association*.
- Multiplicidad : Indica el número de instancias de un *Type* que participa en la *Association*. Es representada como un conjunto, posiblemente abierto, de enteros positivos.
- Ordenamiento : Los posibles valores para este atributo son *Unordered* (los elementos forman un conjunto) y *Ordered* (los elementos están ordenados en una lista).
- Calificación : Un atributo califica la *Association* con un *Association Role* de multiplicidad mayor a uno cuando éste se usa para designar una instancia específica del tipo (se observa como una modificación de la multiplicidad del rol). Es notado como un rectángulo de menor tamaño al símbolo de clase que incluye el nombre del atributo y en donde está conectada la *Association* (ver figura 2.9).
- Navegabilidad : Indica si la *Association* es navegable hacia el tipo participante; significa que que un tipo dado es directamente alcanzable vía la *Association*. Se nota con una pequeña flecha indicando el sentido.
- Agregación : Especifica la relación *parte/todo*. Existen dos clases. La primera es la *Composite Aggregation* que significa composición o pertenencia de la parte en el todo (la existencia de la parte depende del todo). La segunda es la *Shared Aggregation* que implica una composición compartida (la existencia de la parte no depende del todo). La agregación es notada como un pequeño diamante en el extremo de la línea de la

Association en donde la primera clase de agregación se representa con el diamante "lleno", la segunda con el diamante "vacío".

Cada *Association Role* es notado en el extremo de la *Association* a la que pertenecen.

Association es la única relación que especifica una conexión semántica entre instancias. Todas las otras relaciones (*Dependency*, *Generalization*) especifican conexiones entre *Types*.

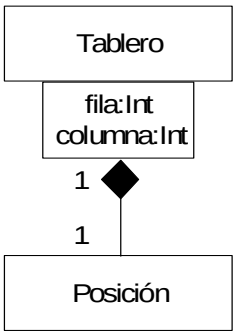


Figura 2.9: Association (qualification)

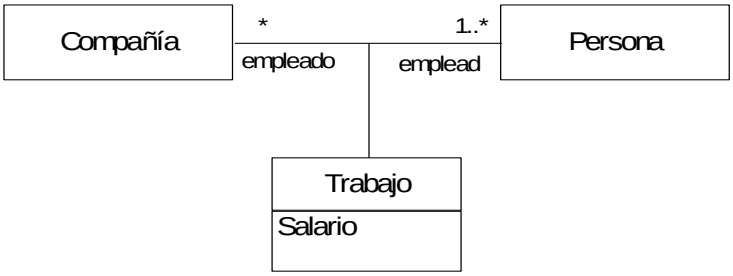


Figura 2.10: Association Class

Generalization

Es la relación taxonómica entre un elemento más general y uno más específico que es totalmente consistente con el primer elemento y que agrega información adicional. Establece una relación de herencia, donde una instancia del subtipo es sustituible por una instancia del supertipo. Esta relación es notada como una línea sólida desde la subclase a la superclase con un triángulo "vacío" en el extremo de la superclase (ver figura 2.10).

Es posible especificar las siguientes restricciones semánticas en esta relación:

- *Overlapping*: Un descendiente puede descender de más de una subclase.
- *Disjoint*: Un descendiente no puede descender de más de una subclase.
- *Complete*: Todas las subclases han sido especificadas, no se esperan más subclases.
- *Incomplete*: Se sabe que solo algunas subclases han sido especificadas y faltan otras.

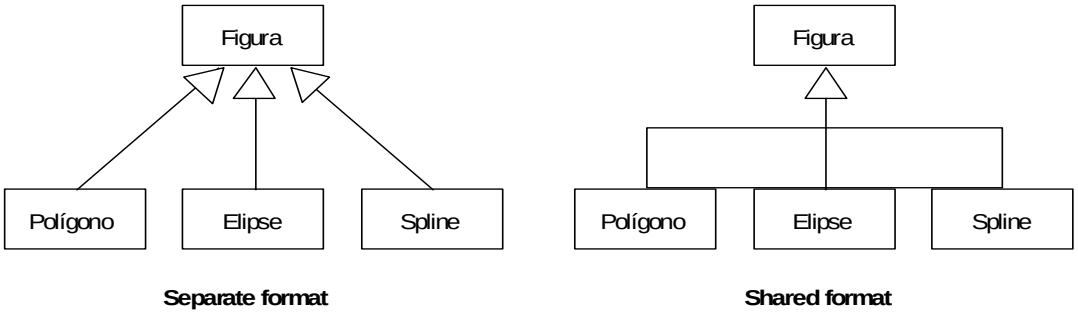


Figura 2.10: Generalization

(III) Diagramas de Comportamiento

Estos diagramas se centran en la dinámica del sistema. Muestran la interacción entre objetos y sus interrelaciones. En general los objetos interactúan mediante el envío y la recepción de mensajes. Básicamente existen dos tipos de diagramas de comportamiento:

- de interacción
- de estados

Diagrama de interacción

Cada diagrama de interacción representa un escenario posible incluido en la funcionalidad total del sistema, es decir muestra un conjunto de interacciones entre objetos en una posible ejecución del sistema. El comportamiento completo del sistema puede deducirse de la composición de todos los posibles escenarios. Existen dos diferentes clases de diagramas de interacción: diagramas de colaboración y diagramas de secuencia.

Un *diagrama de colaboración* (ver figura 2.11) es una construcción que se enfoca en los objetos y mensajes involucrados en cumplir un propósito. Enfatiza las relaciones entre los objetos. Una colaboración involucra dos tipos de construcciones del modelo: una descripción de la estructura estática de los objetos intervinientes, incluyendo sus relaciones relevantes, atributos y operaciones; y una descripción de los mensajes intercambiados entre los objetos para cumplir una tarea.

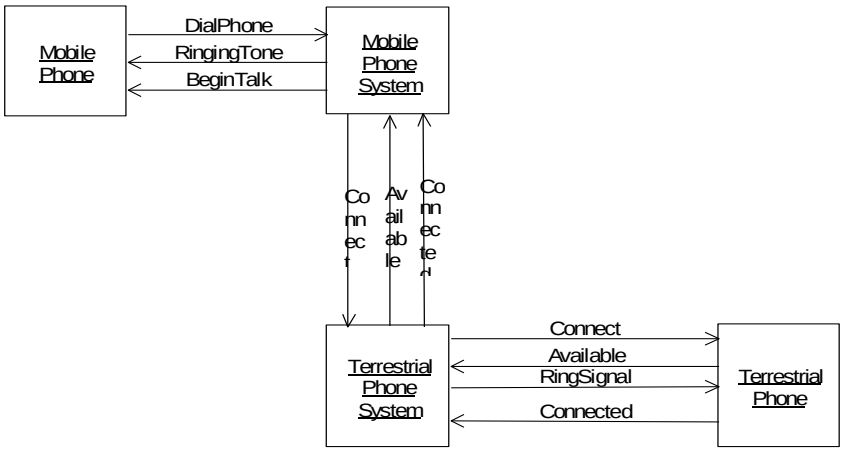


Figura 2.11*:* Diagrama de Colaboración

Un *diagrama de secuencia* (ver figura 2.12) enfatiza la temporalidad de los mensajes mostrando explícitamente su secuencia. Son útiles para especificaciones para tiempo real y para escenarios complejos. Muestran a los objetos participantes en la interacción por medio de su línea de vida (que muestra su intervención en la interacción ordenada en el tiempo) y los mensajes que ellos intercambian en secuencia temporal. Un diagrama de secuencia puede existir en su forma genérica (describe todas las posibles secuencias) y en forma instanciada (describe una secuencia consistente con la forma genérica).

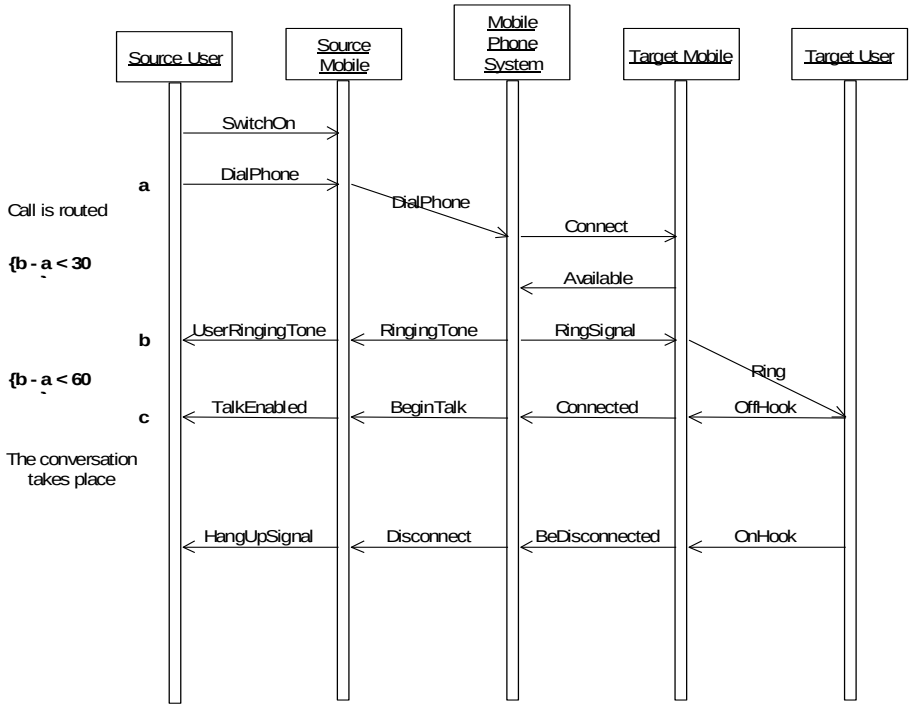


Figura 2.12: Diagrama de Secuencia

Diagrama de Estados

Un diagrama de transición de estados (o *state diagram*) muestra una secuencia de estados por los que un objeto o una interacción pueden pasar durante su existencia en respuesta a estímulos recibidos, y también muestra sus respuestas y acciones (ver figura 2.14). Generalmente, cada clase posee un diagrama de estados para describir el comportamiento dinámico de las instancias de dicha clase.

Un diagrama de estados es un grafo bipartito de estados y transiciones que se vincula generalmente a un tipo, una operación o una clase.

Un estado es una situación particular en la vida del sistema durante la cual se satisface alguna condición, se realiza una acción o se espera un evento. Un estado puede ser compuesto, en este caso puede contener uno o más subestados (posiblemente concurrentes). Una transición a un estado compuesto representa una transición a su pseudo-estado inicial y una transición al pseudo-estado final representa la conclusión de la actividad del estado correspondiente.

Los estados se muestran como un rectángulo con los vértices redondeados (ver figura 2.13). Los pseudo-estados iniciales y finales son representados como un circulo lleno y un círculo lleno con otro círculo envolviéndolo, respectivamente. Pueden contener hasta tres compartimentos: *nombre, variables de estado y actividad interna.*

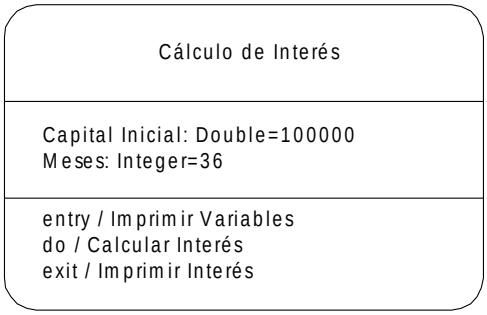


Figura 2.13: Estado con variables de estado y actividad interna

En el compartimento de *actividad interna* se colocan *acciones* de la forma:

event-name / action-expression

Un estado puede ser refinado en subestados concurrentes mediante relaciones *and* o usando relaciones *or* mutuamente excluyentes. En este caso puede crearse un cuarto compartimento donde se alojarán los subestados.

Las transiciones se dibujan mediante flechas comunes entre los estados que están rotuladas de la siguiente forma:

event-signature [guard-condition] / action-expression ^send-clause

donde:

event-signature = event-name(param, ...)

y:

send-clause = dest-object.dest-event (arg, ...)

Por ejemplo:

Mouse-click(x,y) [over-object(x,y)] / obj := pick-object(x,y) ^obj.activate()

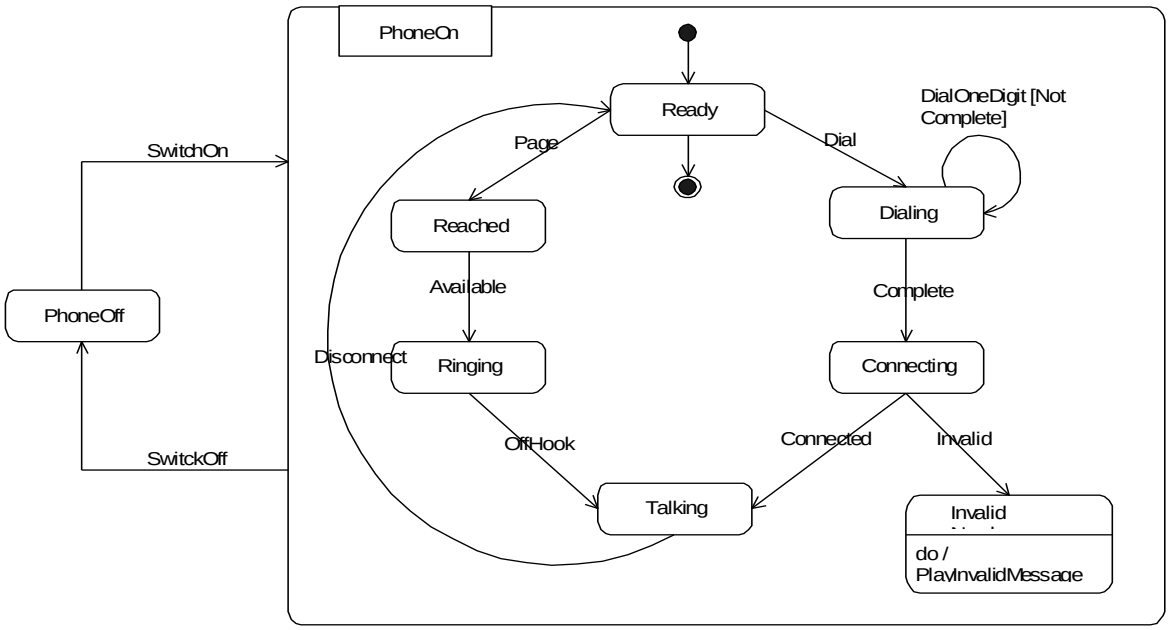


Figura 2.14: Diagrama de Estados

(IV) Constraints

Un constraint es una condición o restricción relacionada a un elemento o a un conjunto de ellos. Un constraint tiene significado semántico ya que representa condiciones que deben ser mantenidas por las instancias del modelo. Se dibuja mediante una nota próxima al elemento condicionado (ver figura 2.15).

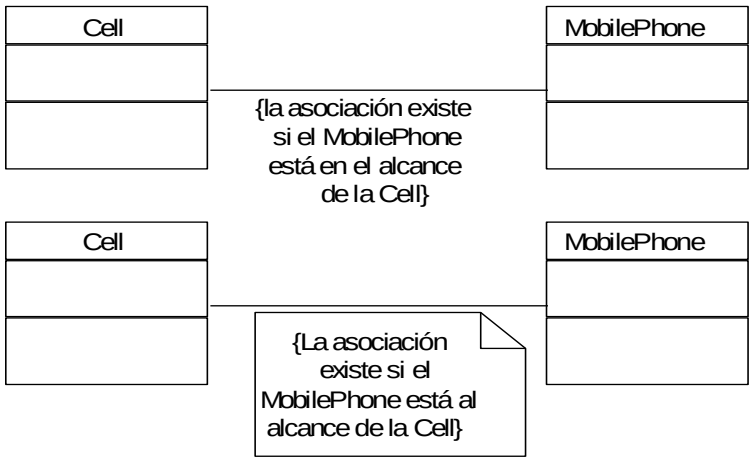


Figura 2.15: formas alternativas de mostrar un constraint

3. Fundamentos de Lógica Dinámica

Una lógica está integrada por tres elementos:

- ***El lenguaje*** es una colección de expresiones bien formadas a las cuales es posible asignarle un significado. Los símbolos del lenguaje juntamente con las reglas formales que permiten distinguir entre expresiones bien formadas y agregaciones arbitrarias de símbolos son llamados la *sintaxis* del lenguaje.
- ***La semántica*** se refiere al significado de las expresiones bien formadas. Usualmente esto se hace explicando las construcciones del lenguaje en términos de conceptos conocidos (y bien entendidos). Este conjunto de conceptos bien entendidos es llamado el dominio semántico. El dominio semántico puede ser un objeto matemático, tal como grupos o grafos o los números naturales o también puede ser un objeto cotidiano, tal como un conjunto de cuentas bancarias, empleados, etc. Las expresiones del lenguaje hablan acerca de propiedades y relaciones entre los objetos del dominio semántico.
- ***El aparato deductivo*** consiste en una colección de reglas que pueden ser aplicadas sobre cierta información inicial para derivar información adicional, en una forma puramente mecánica.

En los siguientes párrafos describimos algunas características importantes de estos tres ingredientes de la lógica.

El lenguaje:

El lenguaje de una lógica típicamente consiste en una parte específica de la aplicación y una parte independiente de la aplicación. La parte específica de la aplicación está integrada por aquellas construcciones que representan objetos de la aplicación; esta parte posiblemente no tenga sentido en otras aplicaciones. Por ejemplo, cuando hablamos de propiedades de los números naturales N={0,1,2,...} el símbolo + denotando adición y el símbolo * denotando multiplicación son muy relevantes, pero estos símbolos no son tan relevantes si estamos discutiendo acerca de las propiedades de los grafos. Por esta razón el lenguaje de la teoría de números y el lenguaje de la teoría de grafos lucen muy diferentes. Estas diferencias son llamadas “diferencias horizontales”. Existen además las llamadas “diferencias verticales”. Aún dentro de un dominio de aplicación especifico pueden existir distintos niveles posibles de expresividad, dependiendo de la complejidad de las propiedades que necesitan ser expresadas y manejadas, por ejemplo las lógicas dinámicas que se limitan a considerar sólo programas while determinísticos son menos expresivas que las que consideran todo tipo de programas regulares.

La semántica:

En las llamadas lógicas bi-valuadas, el significado de cada fórmula del lenguaje se define como su veracidad o falsedad con respecto a objetos matemáticos precisamente definidos. Estos objetos matemáticos son usualmente llamados estructuras o modelos.

Si una sentencia φ es verdadera (true) en una estructura ***U*** entonces decimos que φ se satisface en ***U***, o que ***U*** es un modelo para φ y lo escribimos $\boldsymbol{U} \models \varphi$. Cuando φ se satisface en todas las estructuras posibles decimos que φ es válida y lo escribimos $\models \varphi$.

Si Φ es un conjunto de sentencias, decimos que φ es una consecuencia lógica de Φ, y lo escribimos como $\Phi \models \varphi$, cuando φ se satisface en todo modelo ***U*** de Φ (***U*** es un modelo para Φ cuando ***U*** es un modelo para cada elemento de Φ).

El conjunto de consecuencias lógicas de Φ se denomina la teoría de Φ y se denota como ***Th***Φ.

El aparato deductivo:

Muchos tipos diferentes de sistemas deductivos han sido propuestos, pero sólo nos concentraremos en el estilo de deducción llamado Hilbert-style. Este estilo consiste en un conjunto de axiomas o sentencias del lenguaje que se asumen como verdaderas *a priori* y reglas de inferencia de la forma

$$\frac{\varphi 1\ \varphi 2\ \ldots\ \varphi n}{\psi}$$

A partir de las cuales nuevos teoremas pueden ser derivados. Las sentencias sobre la línea son llamadas *premisas* y la sentencia debajo de la línea es la *conclusión*.

Si Φ es un conjunto de sentencias (o hipótesis), una prueba de φ usando Φ es una secuencia de sentencias $\varphi 1\ \varphi 2\ \ldots\ \varphi m$ tal que $\varphi m = \varphi$ y cada φi es un axioma o es un elemento de Φ o es la conclusión de una regla cuyas premisas aparecen previamente en la secuencia. Cuando existe una prueba de φ usando Φ decimos que φ es una consecuencia deductiva de Φ y lo escribimos como $\Phi \vdash \varphi$. Un caso particular de deducción ocurre cuando el conjunto de hipótesis es vacío; en este caso decimos que φ es un teorema y lo escribimos $\vdash \varphi$.

En relación con el aparato deductivo de una lógica surgen los problemas de corrección y completitud del sistema deductivo, consistencia de las fórmulas, axiomatización y complejidad de las deducciones. A continuación definiremos brevemente cada uno de estos problemas:

Corrección y Completitud: Un sistema deductivo $\vdash$ es correcto (sound) con respecto a una semántica $\models$ si para toda sentencia φ,

$\vdash \varphi$ implica $\models \varphi$, es decir que todo teorema es válido.

Un sistema deductivo $\vdash$ es completo (complete) con respecto a una semántica $\models$ si para toda sentencia φ,

$\models \varphi$ implica $\vdash \varphi$, es decir que toda sentencia válida es un teorema.

Consistencia: Un conjunto de sentencias Φ es inconsistente si $\Phi \vdash$ *false*, donde *false* es el símbolo de falsedad. Un conjunto de sentencias Φ es consistente si no es inconsistente. Una sentencia individual φ es consistente si el conjunto $\{\varphi\}$ es consistente.

Esquemas de axiomas: En muchos casos los axiomas y las reglas de inferencia son escritos como esquemas que permiten infinitas instancias concretas. Por ejemplo, la regla

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi}$$

indica que a partir de la verdad de φ y de ψ puede inferirse la verdad de la sentencia $\varphi \wedge \psi$. Esto representa infinitas reglas una para cada posible elección de φ y de ψ. Por ejemplo, la regla siguiente es una instancia de este esquema:

$$\frac{p \rightarrow q \qquad q \rightarrow p}{p \rightarrow q \wedge q \rightarrow p}$$

***Complejidad*:** Informalmente se dice que un conjunto S es recursivamente enumerable (lo cual se abrevia con r.e.) cuando dado un elemento x cualquiera siempre existe un procedimiento efectivo para determinar si x pertenece a S. Un conjunto S es co-recursivamente enumerable (co-r.e.) cuando dado un elemento x cualquiera siempre existe un procedimiento efectivo para determinar si x no pertenece a S. Un conjunto S es decidible (o recursivo) cuando S es r.e. y co-r.e.

Sistemas Lógicos

Estos tres elementos -lenguaje, semántica y aparato deductivo- pueden ser adaptados para representar aplicaciones particulares y/o para obtener distintos niveles de expresividad, originando diferentes sistemas lógicos. Los sistemas clásicos más conocidos son los siguientes:

- ***Lógica proposicional***, la lógica de las aserciones no-interpretadas y los conectivos proposicionales básicos: and, or, not.
- ***Lógica ecuacional***, la lógica de la igualdad y la sustitución de iguales por iguales.
- ***Lógica de predicados*** de primer orden, la lógica de los cuantificadores.
- ***Lógica multi-sort***, la lógica que permite razonar sobre un dominio integrado por distintos tipos (sorts) de elementos.
- ***Lógica modal***, la lógica de posibilidades y necesidades.

La sintaxis, semántica y aparato deductivo de cada uno de estos sistemas, así como también sus propiedades pueden encontrarse en numerosos libros, tales como [Hamilton 81, Harel et al.99, Enderton 72]. En las siguientes secciones realizaremos una revisión de cada uno de estos sistemas.

3.1 Lógica Proposicional

La Lógica Proposicional (PL) permite realizar deducciones básicas, como por ejemplo:

Si p fuera verdadero entonces q también sería verdadero; pero q no es verdadero; por lo tanto tampoco p lo es. Esta deducción es válida independientemente de la verdad o falsedad de p y q. La Lógica Proposicional formaliza esta clase de razonamientos.

3.1.1 Sintaxis

Los símbolos básicos de PL son los símbolos proposicionales p, q, r, ... representando aserciones atómicas las cuales pueden ser verdaderas o falsas.

Además de estos símbolos el lenguaje tiene operadores proposicionales o conectivos:

$\wedge$	Conjunción “and”
$\vee$	Disyunción “or”
$\neg$	Negación “not”
true	Verdadero
false	Falso
$\rightarrow$	Implicación, “if..then..”
$\leftrightarrow$	Bicondicional “if and only if”

El lenguaje de fórmulas proposicionales $\gamma, \varphi, \kappa,...$ se construye inductivamente a partir de los símbolos básicos de acuerdo a las siguientes reglas:

- todas las proposiciones atómicas p, q, r,..y ***true*** y ***false*** son proposiciones;
- si γ y φ son proposiciones entonces $\gamma\wedge\varphi$, $\gamma\vee\varphi$, $\neg\gamma$, $\gamma\rightarrow\varphi$, $\gamma\leftrightarrow\varphi$, también los son.

3.1.2 Semántica

La verdad o falsedad de una proposición depende del la verdad o falsedad de las proposiciones atómicas que aparecen en ella. Existen dos posibles valores de verdad que denotamos con **0** (falso) y **1** (verdadero). Una *asignación de valores de verdad* es una función que asocia cada proposición atómica con un elemento del conjunto {**0**,**1**}:

$$u: \{p,q,r,...\} \rightarrow \{\mathbf{0},\mathbf{1}\}$$

Toda asignación se extiende inductivamente sobre todo el conjunto de proposiciones de la siguiente forma:

$u(\gamma \wedge \varphi)$ $=_{def}$ **1**, si $u(\gamma)=$**1** y $u(\varphi)=$**1 ; 0**, en los demás casos.

$u(\gamma \vee \varphi)$ $=_{def}$ **1**, si $u(\gamma)=$**1** ó $u(\varphi)=$**1 ; 0**, en los demás casos.

$u(\neg\gamma)$ $=_{def}$ **1**, si $u(\gamma)=$**0 ; 0**, en los demás casos.

$u($***true***$)$ $=_{def}$ **1**

$u($***false***$)$ $=_{def}$ **0**

$u(\gamma \rightarrow \varphi)$ $=_{def}$ **1**, si $u(\gamma)=$**0** ó $u(\varphi)=$**1 ; 0**, en los demás casos.

$u(\gamma \leftrightarrow \varphi)$ $=_{def}$ **1**, si $u(\gamma)=u(\varphi)$ **; 0**, en los demás casos.

Decimos que la asignación de valores de verdad u satisface ϕ, o que ϕ es u-válida, o u es un modelo de ϕ, cuando $u(\phi)=1$ y lo notamos con $u \models \phi$. Si Φ es un conjunto finito o infinito de proposiciones, decimos que u satisface Φ y lo notamos con $u \models \Phi$ si u satisface a cada una de las proposiciones en Φ. Una proposición (o conjunto de proposiciones) es satisfactible cuando existe una asignación de valores de verdad que la satisface (es decir, tiene un modelo). Si una proposición ϕ es verdadera en todos los modelos de Φ, entonces escribimos $\Phi \models \phi$ y decimos que ϕ es una consecuencia lógica de Φ. Si $\varnothing \models \varphi$ escribimos simplemente $\models \varphi$ y decimos que φ es válida o que es una tautología.

3.1.3 Sistema deductivo para PL

Existen distintos sistemas deductivos correctos y completos para PL, en esta sección mostraremos un sistema deductivo muy simple extraído de [Hamilton 81] que solamente considera fórmulas construidas usando los conectivos $\neg$ y $\rightarrow$. Esto no implica pérdida de generalidad ya que este conjunto de conectivos es *adecuado*, es decir toda proposición es lógicamente equivalente a una proposición construida usando solamente los conectivos de dicho conjunto.

Axiomas

(L1) $\gamma \rightarrow (\varphi \rightarrow \gamma)$

(L2) $(\gamma \rightarrow (\varphi \rightarrow \kappa)) \rightarrow ((\gamma \rightarrow \varphi) \rightarrow (\gamma \rightarrow \kappa))$

(L3) $(\neg\gamma \rightarrow \neg\varphi) \rightarrow (\varphi \rightarrow \gamma)$

Regla de inferencia

(MP) $\dfrac{\varphi \quad \varphi \rightarrow \gamma}{\gamma}$

3.1.4 Decidibilidad y Complejidad

Dada una fórmula proposicional ϕ el proceso para determinar su validez es decidible; la implementación ingenua de este proceso toma tiempo exponencial en el tamaño de ϕ ya que existen 2^n posibles asignaciones de valores de verdad, siendo n la cantidad de proposiciones atómicas diferentes que aparecen en ϕ.

3.2 Lógica de Predicados de primer orden

La lógica de predicados de primer orden es la lógica de las cuantificaciones sobre los elementos de una estructura.

3.2.1 Sintaxis

La signatura Σ del lenguaje consiste en un conjunto F de símbolos de función y un conjunto P de símbolos de predicado (o relación); cada símbolo tiene asociada una aridad que indica la cantidad de argumentos. Símbolos de aridad 0,1,2,3,..,n son llamados nularios (o constantes), unarios, binarios, ternarios y n-arios respectivamente.

El alfabeto del lenguaje está formado por:

- Los símbolos de función y los símbolos de predicado en Σ;
- Un conjunto numerable X de variables individuales x, y, z,...;
- Los conectivos proposicionales;
- Los cuantificadores $\forall$ y $\exists$;
- Paréntesis

Los términos s,t,... del lenguaje se construyen inductivamente de la siguiente forma:

- cada variable es un término
- si $t_1,...,t_n$ son términos y f es un símbolo de función n-ario entonces $f(t_1,...,t_n)$ es un término.

El conjunto de todos los términos sobre Σ y X se denota $T_\Sigma(X)$. Los términos que no tienen variables se denominan atómicos o *ground*. El conjunto de términos atómicos sobre Σ se denota T_Σ.

Las fórmulas $\phi, \gamma, \varphi,..$ se definen inductivamente de la siguiente forma:

- si t1,...,tn son términos y p es un símbolo de predicado n-ario entonces $p(t_1,...,t_n)$ es una fórmula (en este caso atómica).
- si γ y φ son fórmulas entonces $\gamma\wedge\varphi$, $\gamma\vee\varphi$, $\neg\gamma$, $\gamma\rightarrow\varphi$, $\gamma\leftrightarrow\varphi$, también los son.
- si φ es una fórmula y x es una variable entonces $\forall x\varphi$ y $\exists x\varphi$ son fórmulas.

3.2.2 Semántica

Σ-álgebras

Los términos y fórmulas definidos sobre un alfabeto (o signatura) $\Sigma=(F,P)$ toman significado al ser interpretados sobre una estructura algebraica llamada Σ-álgebra

$U=(A,m_U)$

donde A es un conjunto no-vacío llamado el carrier o dominio del álgebra y m_U es la función de interpretación semántica que asigna una función $f^U:A^n \rightarrow A$ para cada símbolo de función n-ario de F y asigna una relación n-aria $p^U \subseteq A^n$ para cada símbolo de predicado p en P. Notemos que las constantes son interpretadas como elementos de A, es decir $c^U \in A$.

Valuaciones

Una valuación es un homomorfismo definido sobre el conjunto de términos

$u:T_\Sigma(X) \rightarrow U$

Una valuación está unívocamente determinada por los valores de las variables en X, es decir, toda función $\underline{u}$: $X \rightarrow U$ se extiende unívocamente a un homomorfismo $u:T_\Sigma(X) \rightarrow U$ por inducción de la siguiente forma:

- si $x \in X$ entonces $u(x)=\underline{u}(x)$
- si $f(t_1,..., t_n) \in T_\Sigma(X)$ entonces $u(f(t_1,..., t_n))= f^U(u(t_1),..., u(t_n))$

Sustituciones

Dada una valuación u, se define que u[x/a] es una nueva valuación que se obtiene a partir de u cambiando el valor de x por a y dejando intactos los valores de las restantes variables, es decir:

$u[x/a](y) = u(y), y \neq x$

$u[x/a](x) = a$

Satisfacción de fórmulas

La relación de satisfacción $\models$ se define inductivamente de la siguiente forma:

$U,u \models p(t_1,...,t_n)$	sii	$p^U(u(t_1),..., u(t_n))$
$U,u \models \gamma \wedge \varphi$	sii	$U,u \models \gamma$ y $U,u \models \varphi$
$U,u \models \gamma \vee \varphi$	sii	$U,u \models \gamma$ ó $U,u \models \varphi$
$U,u \models \neg\varphi$	sii	$U,u \not\models \varphi$
$U,u \models \forall x\varphi$	sii	para todo $a \in A$ se cumple que $U,u[x/a] \models \varphi$
$U,u \models \exists x\varphi$	sii	existe algún $a \in A$ tal que $U,u[x/a] \models \varphi$

Si $U,u \models \varphi$ decimos que φ es verdadero en U bajo la valuación u, o que el par (U,u) es un modelo de φ ó que U,u satisface φ. Si Φ es un conjunto de fórmulas entonces denotamos $U,u \models \Phi$ si $U,u \models \varphi$ para todo φ de Φ y decimos que U,u satisface Φ. Si φ es verdadera en todos los modelos de Φ , entonces escribimos $\Phi \models \varphi$ y decimos que φ es una consecuencia lógica de Φ. Si $\varnothing \models \varphi$ escribimos simplemente $\models \varphi$ y decimos que φ es válida.

3.2.3 Sistema deductivo

En esta sección mostraremos un sistema deductivo estilo Hilbert para la Lógica de Primer Orden extraído de [Hamilton 81].

Axiomas

(K1) $\gamma\rightarrow(\varphi\rightarrow\gamma)$

(K2) $(\gamma\rightarrow(\varphi\rightarrow\kappa))\rightarrow((\gamma\rightarrow\varphi)\rightarrow(\gamma\rightarrow\kappa))$

(K3) $(\neg\gamma\rightarrow\neg\varphi)\rightarrow(\varphi\rightarrow\gamma)$

(K4) $(\forall x_i)\varphi\rightarrow\varphi$ si x_i no ocurre libre en φ.

(K5) $(\forall x_i)\varphi(x_i)\rightarrow\varphi(t)$, si $\varphi(x_i)$ es una fórmula donde x_i ocurre libre y t es un término libre para x_i en $\varphi(x_i)$.

(K6) $(\forall x_i)(\varphi\rightarrow\gamma)\rightarrow(\varphi\rightarrow(\forall x_i)\gamma)$ si φ no tiene ocurrencias libres de x_i.

Regla de inferencia

(MP) $\dfrac{\varphi \quad \varphi\rightarrow\gamma}{\gamma}$

(GEN) $\dfrac{\varphi}{(\forall x_i)\varphi}$

La regla (MP) es la misma regla *modus ponens* de la Lógica Proposicional. La regla (GEN) se conoce como la *regla de generalización*.

3.2.4 Decidibilidad y complejidad

Dada una fórmula φ de primer orden, el problema de determinar su validez (es decir $\models\varphi$) es r.e. pero no-decidible.

3.3 Lógica Ecuacional de primer orden

La Lógica ecuacional es una formalización del razonamiento ecuacional. Las propiedades de la igualdad son capturadas en los axiomas para las relaciones de equivalencia: reflexividad, simetría y transitividad y la propiedad de sustitución de iguales por iguales. El lenguaje de la lógica ecuacional es un sub-lenguaje de la lógica de primer orden en el cual existe un símbolo de predicado especial de aridad 2 llamado el *símbolo de igualdad*. Este símbolo se representa mediante = y generalmente se utiliza la notación infija, es decir $t_1=t_2$ en lugar de $=(t_1,t_2)$. La interpretación del = es la siguiente:

$U,u \models (t_1=t_2)$ sii $u(t_1)=u(t_n)$

3.4 Lógica Order-sorted

Las lógicas con sorts (o tipos) permiten razonar sobre un dominio integrado por distintas clases (o tipos) de elementos. En particular la Lógica order-sorted es una lógica que permite representar un ordenamiento parcial entre los sorts. Esto resulta útil por ejemplo para realizar la siguiente deducción: se sabe que la propiedad p vale para todos los elementos de sort s, también se sabe que el sort s´ es menor que el sort s, es decir $s' \leq s$. A partir de esta información puede deducirse que la propiedad p también vale para todos los elementos de sort s´.

3.4.1 Sintaxis

La signatura de una lógica order-sorted [Goguen and Meseguer 92] es una terna

$$\Sigma=((S,\leq),F,P)$$

formada por los siguientes elementos:

- Un conjunto parcialmente ordenado $(S, \leq)$ de nombres de sorts.
- Un conjunto F de declaraciones de funciones de la forma, $f:s_1,...,s_n \rightarrow s$ para $s_i \in S$.
- Un conjunto P de declaraciones de predicados de la forma $p:s_1,...,s_n$ para $s_i \in S$.

El conjunto de variables X es una familia finita de conjuntos $X=\{X_s \mid s \in S\}$, donde X_s el conjunto de variables de sort s.

El conjunto de todos los términos de sort s sobre Σ y X se denota $T_{\Sigma,s}(X)$ y se define inductivamente de la siguiente forma:

- $X_S \subseteq T_{\Sigma,S}(X)$
- si $(f:s_1 x...x\ s_n \rightarrow s) \in F$ y $t_i \in T_{\Sigma,Si'}(X)$ y $si' \leq si$ para i=1..n, entonces $f(t_1,..., t_n) \in T_{\Sigma,S}(X)$.

Las fórmulas $\phi, \gamma, \varphi,..$ sobre Σ y X se definen inductivamente de la siguiente forma:

- si p es un símbolo de predicado, $p:s_1,... ,s_n \in P$ y $t_i \in T_{\Sigma,Si'}(X)$ y $si' \leq si$ para i=1..n, entonces $p(t_1,...,t_n)$ es una fórmula (en este caso atómica).
- si γ y φ son fórmulas entonces $\gamma \wedge \varphi$, $\gamma \vee \varphi$, $\neg\gamma$, $\gamma \rightarrow \varphi$, $\gamma \leftrightarrow \varphi$, también los son.
- si φ es una fórmula y x es una variable de sort s entonces $(\forall x:s)\varphi$ y $(\exists x:s)\varphi$ son fórmulas.

3.4.2 Semántica

Los términos y fórmulas definidos sobre un alfabeto (o signatura) $\Sigma=((S\leq), F, P)$ toman significado al ser interpretados sobre una estructura algebraica llamada Σ-álgebra

$$U=(A,m_U)$$

donde A es un conjunto no-vacío llamado el universo del álgebra y m_U es la función de interpretación semántica que asigna una función f^A: $A_{s1},..,A_{sn} \rightarrow A_s$ para cada símbolo de función $f \in F_{s1,..,sn,s}$ y asigna una relación n-aria p^A: $A_{s1},..,A_{sn}$ para cada símbolo de predicado $p \in P_{s1,..,sn}$.

La única diferencia con la semántica de la Lógica de primer orden clásica reside en que las lógicas order-sorted se interpretan sobre álgebras heterogéneas o multi-sort cuyo universo está integrado por más de un carrier o dominio. Es decir, el universo de una Σ-álgebra es una familia finita de conjuntos $A=(A_s)_{s \in S}$ donde cada A_s es un conjunto llamado el carrier (o dominio) del sort s. Además existe una relación de inclusión entre los carriers respetando el ordenamiento entre sus sorts:

para todo sort s_i en S, si $s_i \leq S_j$ entonces $A_{si} \subseteq A_{sj}$

3.5 Lógica Modal

Lógica modal [Chellas 80, Harel et al.99, Popkorn 94] es la lógica que permite expresar los conceptos de posibilidad y necesidad. Existen numerosos sistemas de lógica modal dependiendo del dominio de aplicación.

Dada una lógica (*L*,*M*, |=) donde *L* es el conjunto definido inductivamente de fórmulas (es decir el lenguaje de la lógica), *M* es el dominio de las interpretaciones y |=es la relación de satisfacción de fórmulas (es decir que |= es un subconjunto de *M*x*L*). Por ejemplo *L* puede ser el conjunto de fórmulas proposicionales y *M* el conjunto de valuaciones de símbolos proposicionales primitivos.

Una lógica modal se construye de la siguiente forma:

El lenguaje modal:

El lenguaje modal *L´* se obtiene agregando las siguientes cláusulas a la definición inductiva de *L* :

- Si ϕ es una fórmula entonces $\Diamond\phi$ también es una fórmula.
- Si ϕ es una fórmula entonces $\Box\phi$ también es una fórmula.

La fórmula $\Diamond\phi$ se lee como “es posible que ϕ” o “existe un mundo en el cual se satisface ϕ”. La fórmula $\Box\phi$ se lee como “es necesario que ϕ” o “en todos los mundos se satisface ϕ”.

La semántica:

Las estructuras sobre las cuales las fórmulas modales son interpretadas se denominan modelos de Kripke (o Kripke-frames) en honor al inventor de la semántica formal de las lógicas modales. Un Kripke-frame es un par *k*= (U,R), donde

- U es un conjunto no vacío incluido en *M*.
- R es una relación binaria sobre U, es decir $R \subseteq UxU$.

El conjunto U se denomina el universo de k y sus elementos son llamados estados o mundos posibles. Intuitivamente R especifica que mundos son accesibles o posibles a partir de un mundo dado. Es decir , $(u,v) \in R$ indica que v es un mundo posible desde el punto de vista de u. Para mayor claridad algunas veces escribimos uRv en lugar de $(u,v) \in R$.

Para cada Kripke-frame k la relación de satisfacción de fórmulas |= es extendida a U x *L´* agregando dos cláusulas a su definición inductiva:

- $u \models \Diamond\phi$ sii existe un v tal que uRv y $v \models \phi$
- $u \models \Box\phi$ sii para todo v tal que uRv, $v \models \phi$

Usaremos la notación $k,u \models \phi$ si $u \models \phi$ en el Kripke-frame k. Cuando $k,u \models \phi$ para todo estado u de k, escribiremos $k \models \phi$. Y cuando $k \models \phi$ para todo frame k, escribiremos $\models \phi$.

3.5.1 Lógica Multimodal

Sea A={a,b,c,...} un conjunto de nombres. El conjunto A se denomina conjunto de modalidades y dado (*L*,*M*, |=)es posible agregar las cláusulas:

- Si ϕ es una fórmula y $a \in A$ entonces $\langle a \rangle \phi$ también es una fórmula.
- Si ϕ es una fórmula y $a \in A$ entonces $[a]\phi$ también es una fórmula.

Sea L´ el nuevo conjunto de fórmulas obtenido luego de agregar estas cláusulas. Un Kripke-frame es ahora una estructura $k=(U,R_A)$ donde R_A es una función asociando una relación binaria $R_a \subseteq UxU$ con cada modalidad $a \in A$.

3.5.2 Lógica modal y programas

La lógica de primer orden clásica es estática en el sentido que los valores de verdad de sus sentencias son inmutables. Las sentencias son interpretadas sobre una estructura individual (equivalente a un único estado o mundo posible). En lógica modal una interpretación consiste en una colección U de mundos posibles o estados. Y dado que una sentencia puede tener asignados distintos valores de verdad en cada estado, las lógicas modales son aptas para razonar en situaciones dinámicas, es decir situaciones en las cuales los valores de verdad de las sentencias no son fijos, sino que pueden variar.

Una interpretación de las lógicas modales que ha tenido gran aplicación es la Lógica Temporal. En esta lógica un estado t es accesible desde otro estado s si t ocurre en el futuro de s. La relación de accesibilidad generalmente se toma como un orden lineal de U (linear-time temporal logic) o como un árbol (branching-time temporal logic).

Esta idea también resulta adecuada para razonar sobre la ejecución de programas. Podemos tomar a U como el universo de todos los posibles estados en la ejecución de un programa. Con cada programa p es posible asociar una relación binaria de accesibilidad sobre U tal que (s,t) está en la relación si y sólo si t es un posible estado final del programa p con estado inicial s. Es decir, si y sólo si existe una computación de p comenzando en s y terminando en t.

Estas ideas pueden representarse utilizando lógica multimodal, considerando a los programas como modalidades. De esta forma, los programas se convierten en una parte del lenguaje de la lógica. La expresión $\langle p \rangle \phi$ dice que es posible ejecutar p y llegar a un estado que satisface ϕ, la expresión $[p]\phi$ dice que siempre que p termine estará en estado que satisface ϕ. El sistema resultante se denomina **Lógica Dinámica**. La lógica dinámica no se limita a enriquecer la lógica clásica con modalidades fijas para cada programa; ya que esto no iría mas allá de la lógica multimodal, sino que usa varios cálculos de programas, los cuales en conjunción con las reglas de la lógica clásica proveen un poderoso sistema para analizar la interacción entre programas y fórmulas. El cálculo de programas permite la construcción de programas complejos a partir de programas atómicos (tal como la instrucción de asignación). Existen también reglas para analizar el comportamiento de un programa en términos del comportamiento de sus sub-programas. La lógica dinámica es una poderosa herramienta para razonar sobre programas y entender su poder y complejidad.

3.6 Lógica Dinámica Proposicional

Con respecto a la Lógica Dinámica (DL), la Lógica Dinámica Proposicional (PDL) juega el mismo rol que la Lógica Proposicional juega dentro de la Lógica de Predicados clásica. PDL [Pratt 76, Goldblatt 92, Harel et al.99] permite expresar propiedades de las interacciones entre programas y proposiciones que son independientes del dominio de computación. Dado que PDL es un subsistema de DL todas las propiedades válidas en PDL también son válidas en DL.

Dado que en PDL no existe la noción de dominio de computación, tampoco puede existir la noción de asignación a una variable. En lugar de esto, los programas primitivos se interpretan como relaciones binarias sobre un conjunto abstracto de estados *K*. Este nivel de abstracción puede parecer demasiado general para expresar alguna propiedad de interés. Sin embargo numerosas relaciones entre programas y proposiciones pueden ser expresadas naturalmente utilizando este nivel de abstracción. Por ejemplo, consideremos la siguiente fórmula PDL:

$$[\alpha]\ (\phi \wedge \tau)\ \leftrightarrow [\alpha]\phi \wedge [\alpha]\tau$$

La parte izquierda indica que la fórmula $(\phi \wedge \tau)$ debe satisfacerse luego de la ejecución del programa α y la parte derecha indica que la fórmula ϕ debe satisfacerse luego de la ejecución del programa α, así como también la fórmula τ. La fórmula completa establece la equivalencia de ambos sentencias y su validez es universal independientemente del dominio de computación sobre el cual se aplique así como también de la naturaleza de α, ϕ ó τ.

La siguiente fórmula proporciona otro ejemplo:

$$[\alpha]\ \phi\ \leftrightarrow [\beta]\phi$$

Si esta fórmula es verdadera bajo todas las interpretaciones entonces α y β son dos programas equivalentes, en el sentido que se comportan idénticamente con respecto a cualquier propiedad expresable en PDL o cualquier otro sistema formal que contenga a PDL como subsistema. Por ejemplo los siguientes programas son equivalentes en este sentido:

if σ **then** α **else** β

if $\neg\sigma$ **then** β **else** α

3.6.1 Sintaxis

Sintácticamente PDL es una combinación de tres ingredientes: Lógica Proposicional, Lógica Modal y el álgebra de eventos regulares. Existen distintas versiones de PDL, dependiendo de la elección de operadores permitidos. En esta sección presentaremos la versión básica llamada PDL regular.

El lenguaje de PDL regular contiene expresiones de dos tipos distintos: proposiciones o fórmulas $\psi,\varphi,...$ y programas $\alpha, \beta,...$ Los programas atómicos son denotados a,b,c... y el conjunto de todos los programas atómicos es denotado Π_0. Las proposiciones atómicas son denotadas p,q,r,...y el conjunto de todas las proposiciones atómicas es denotado Φ_0. El conjunto de todos los programas es denotado Π y el conjunto de todas las proposiciones es denotado Φ. Los programas y las

fórmulas se construyen inductivamente a partir de programas y fórmulas atómicos usando los siguientes operadores:

Operadores proposicionales:

$\vee$	disyunción
$\neg$	negación

Operadores de programas:

;	composición
$\cup$	choice
*	iteración

Operadores combinados:

$\langle\rangle$	posibilidad
?	test

El conjunto Π de todos los programas y el conjunto Φ de todas las fórmulas se definen por inducción mutua como los menores conjuntos tales que:

- $\Phi_0 \subseteq \Phi$
- $\Pi_0 \subseteq \Pi$
- Si $\varphi, \psi \in \Phi$ entonces $\varphi \vee \psi \in \Phi$ y también $\neg\varphi \in \Phi$
- Si $\alpha, \beta \in \Pi$ entonces $\alpha;\beta \in \Pi$ y $\alpha \cup \beta \in \Pi$ y $\alpha* \in \Pi$
- Si $\alpha \in \Pi$ y $\varphi \in \Phi$ entonces $\langle\alpha\rangle \varphi \in \Phi$
- Si $\varphi \in \Phi$ entonces $\varphi? \in \Pi$

Los programas compuestos y las proposiciones tienen el siguiente significado intuitivo:

$\langle\alpha\rangle \varphi$	"es posible ejecutar α y terminar en un estado donde se satisface φ "
$\alpha;\beta$	"ejecute α y luego ejecute β"
$\alpha \cup \beta$	"elija no-determinísticamente entre α y β y luego ejecute el elegido"
$\alpha*$	"ejecute α un número finito (elegido no-determinísticamente) de veces"
$\varphi?$	"testee φ, prosiga si se satisface y falle si no se satisface"

Las fórmulas escritas utilizando sólo los operadores primitivos pueden resultar difíciles de leer. Esto se debe a que los operadores primitivos han sido elegidos teniendo en cuenta su simplicidad matemática y no su legibilidad. Con el objetivo de facilitar el manejo de las fórmulas se han definido otros operadores no primitivos, los cuales se definen a partir de los primitivos. Estos operadores son:

$\varphi \rightarrow \psi$	=	$\neg\varphi \vee \psi$
$\varphi \wedge \psi$	=	$\neg(\varphi \rightarrow \neg\psi)$
$\varphi \leftrightarrow \psi$	=	$(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
false	=	$\varphi \wedge \neg\varphi$
true	=	$\neg$***false***
skip	=	***true***?
fail	=	***false***?
$[\alpha]\varphi$	=	$\neg\langle\alpha\rangle\neg\varphi$
if φ **then** α **else** β	=	$\varphi?;\alpha \cup \neg\varphi?;\beta$
while φ **do** α	=	$(\varphi?;\alpha)^* ; \neg\varphi?$
$\{\varphi\}\ \alpha\ \{\psi\}$	=	$\varphi \rightarrow [\alpha]\psi$

Los programas **skip** y **fail** son el programa que "hace nada" y el programa que "falla" respectivamente. El operador de necesidad [ ] es el dual modal de $\langle\ \rangle$. La proposición $[\alpha]\varphi$ tiene el siguiente significado intuitivo: "siempre que α termina debe encontrarse en un estado en el cual se satisfaga φ". Pero la fórmula $[\alpha]\varphi$ no implica que α termina, contrariamente a lo que ocurre con el operador de posibilidad $\langle\ \rangle$. Notemos que la fórmula $\langle\alpha\rangle$***true*** indica la existencia de una computación de α que termina, mientras que la fórmula $[\alpha]$***false*** indica que ninguna computación de α termina.

El operador ternario **if-then-else** y el operador binario **while-do** son las construcciones convencionales que se encuentran en los lenguajes de programación. La construcción $\{\varphi\}\ \alpha\ \{\psi\}$ se denomina aserción de corrección parcial.

3.6.2 Semántica

La semántica de PDL deriva de la semántica de la Lógica Modal. Las estructuras sobre las cuales los programas y proposiciones de PDL son interpretados se denominan Kripke-frames. Un Kripke-frame para PDL es un par $k = (U, m_k)$, donde

- U es un conjunto no vacío de estados.
- m_k es la función de valuación (meaning function).

La función de valuación asigna un subconjunto de U a cada proposición atómica y una relación binaria sobre U a cada programa atómico. Es decir:

$m_k(p) \subseteq U$, $p \in \Phi_0$

$m_k(a) \subseteq U \times U$, $a \in \Pi_0$

La definición de la función se extiende por inducción para dar significado a todos los elementos de los conjuntos Φ y Π, tal que:

$m_k(\varphi) \subseteq U \quad , \varphi \in \Phi$

$m_k(\alpha) \subseteq UxU \quad , \alpha \in \Pi$

Intuitivamente podemos pensar que el conjunto $m_k(\varphi)$ es el conjunto de estados en los cuales la proposición φ es verdadera y podemos pensar en la relación binaria $m_k(\alpha)$ como el conjunto de pares entrada⁄salida del programa α.

El significado de las proposiciones y programas compuestos se define de la siguiente forma:

- $m_k(\varphi \vee \psi) = m_k(\varphi) \cup m_k(\psi)$
- $m_k(\neg\varphi) = U - m_k(\varphi)$
- $m_k(\langle\alpha\rangle\ \varphi) = dom(m_k(\alpha) \circ \{(u,u) \mid u \in m_k(\varphi)\})$
- $m_k(\alpha;\beta\ ) = m_k(\alpha) \circ m_k(\beta)$
- $m_k(\alpha \cup \beta) = m_k(\alpha) \cup m_k(\beta)$
- $m_k(\alpha*) = m_k(\alpha)*$
- $m_k(\varphi?) = \{(u,u) \mid u \in m_k(\varphi)\}$

El operador $\circ$ es la composición relacional. Los operadores $\cup$ y - son los operadores usuales de unión y resta de conjuntos. La función dom retorna el dominio de una relación. El programa α* es interpretado como la clausura transitiva reflexiva de $m_k(\alpha)$.

La relación k,u |= φ significa "φ es true en el estado (o mundo) u del frame k" o también "u satisface φ en el frame k" y está definida de la siguiente forma:

k,u |= p	sii	$u \in m_k(p)$
k,u |= $\varphi \vee y$	sii	k,u |= φ ó k,u |=y
k,u |= $\neg\varphi$	sii	k,u $\not\models \varphi$
k,u |= $\langle\alpha\rangle\ \varphi$	sii	$\exists v\ (u,v) \in m_k(\alpha)$ y k,v |=φ

El significado de los programas compuestos puede también definirse de la siguiente forma:

$(u,v) \in m_k(\alpha;\beta)$ sii $\exists w\ (u,w) \in m_k(\alpha)$ y $(w,v) \in m_k(\beta)$

$(u,v)\in m_k(\alpha\cup\beta)$	sii	$(u,v)\in m_k(\alpha)$ ó $(u,v)\in m_k(\beta)$
$(u,v)\in m_k(\alpha*)$	sii	$\exists n\geq 0\ \exists u_0,...,u_n,\ u=u_0,\ v=u_n\ ,\ (u_i\ ,\ u_{i+1})\ )\in m_k(\alpha),\ 0\leq i\leq n-1$
$(u,v)\in m_k(\varphi?)$	sii	$u=v$ y $k,u \models \varphi$

Una fórmula φ es verdadera (o válida) en un frame k cuando es verdadera en todos sus mundos posibles:

k $\models\varphi$ sii para todo mundo u en *k*, k,u $\models\varphi$

Una fórmula es verdadera en una clase *C* de frames cuando es verdadera en todos los miembros de la clase:

C $\models\varphi$ sii para todo miembro *M* de *C*, *M* $\models\varphi$

Una fórmula es una tautología sii es verdadera en la clase de todos los Kripke-frames:

$\models\varphi$ sii para toda clase *C*, *C* $\models \varphi$.

Una fórmula φ es falsa en un frame k cuando φ es falsa en algún mundo del frame. Una fórmula φ es falsa en un mundo w de un frame k cuando no se cumple *k,w* $\models\varphi$.

Una fórmula φ es satisfactible cuando existe algún estado w y algún frame *k*, tal que *k,w* $\models\varphi$.

3.6.3 Un sistema deductivo para PDL

Los siguientes axiomas y reglas de inferencia constituyen un sistema deductivo Hilbert-style para PDL. Es posible demostrar que este sistema es correcto y completo.

Axiomas para PDL

(i)	axiomas de la Lógica Proposicional		
(ii)	$\langle\alpha\rangle\phi \wedge [\alpha]\tau$	$\rightarrow$	$\langle\alpha\rangle(\phi\wedge\tau)$
(iii)	$\langle\alpha\rangle(\phi\vee\tau)$	$\leftrightarrow$	$\langle\alpha\rangle\phi \vee \langle\alpha\rangle\tau$
(iv)	$\langle\alpha\cup\beta\rangle\phi$	$\leftrightarrow$	$\langle\alpha\rangle\phi \vee \langle\beta\rangle\phi$
(v)	$\langle\alpha;\beta\rangle\phi$	$\leftrightarrow$	$\langle\alpha\rangle\langle\beta\rangle\phi$
(vi)	$\langle\varphi?\rangle\phi$	$\leftrightarrow$	$\varphi\wedge\phi$
(vii)	$(\varphi\vee \langle\alpha\rangle\langle\alpha*\rangle\varphi)$	$\rightarrow$	$\langle\alpha*\rangle\varphi$
(viii)	$\langle\alpha*\rangle\varphi$	$\rightarrow$	$(\varphi\vee\langle\alpha*\rangle(\neg\varphi\wedge\langle\alpha\rangle\varphi))$

Reglas de Inferencia

(MP) $\dfrac{\varphi \quad \varphi\rightarrow\psi}{\psi}$

(GEN) $\dfrac{\varphi}{[\alpha]\varphi}$

Los axiomas (ii) y (iii) y las dos reglas de inferencia no son particulares de PDL sino que provienen de la Lógica Modal. Las reglas (MP) y (GEN) se denominan *modus ponens* y *generalización modal* respectivamente.

El axioma (viii) se denomina *axioma de inducción* de PDL y es mejor conocido por su forma dual:

$$(\varphi \wedge [\alpha*](\varphi \rightarrow [\alpha]\varphi)) \quad \rightarrow \quad [\alpha*]\varphi$$

y su significado es similar al de la regla de inducción de la aritmética de Peano. Este axioma establece que si φ es verdadero inicialmente y si la verdad de φ es preservada por el programa α entonces φ será verdadero luego de cualquier número de iteraciones de α.

3.6.4 Decidibilidad en PDL

PDL posee la propiedad del modelo finito. Esta propiedad dice que si una fórmula φ es satisfactible, entonces φ es satisfactible en un estado de un modelo con a lo sumo $2^{|\varphi|}$ estados, donde $|\varphi|$ es la cantidad de símbolos de φ. Esta propiedad proporciona un procedimiento de decisión para PDL: para determinar si vale $\models \varphi$, deben inspeccionarse todos los modelos con a lo sumo $2^{|\varphi|}$ estados. La técnica que se utiliza se denomina *filtración*, informalmente consiste en considerar interpretaciones solamente de las fórmulas primitivas y de los programas primitivos que aparecen en φ, por lo tanto se considera sólo un número finito de símbolos y por lo tanto un número finito de posibles modelos.

La versión ingenua de este algoritmo -que construye todos los Kripke-frames de a lo sumo $2^{|\varphi|}$ estados y chequea si φ se satisface en cada uno de sus estados- toma tiempo exponencial no determinístico, realiza $2^{2|\varphi|}$ pasos en el peor de los casos. Sin embargo se han desarrollado otros algoritmos más eficientes, por ejemplo ver [Harel et al. 99, capítulo 9: Complexity of PDL].

3.7 Lógica Dinámica de Primer Orden

La principal diferencia entre Lógica Dinámica de primer orden (first-order DL) [Goldblatt 92, Harel et al.99] y la versión proposicional PDL discutida anteriormente reside en la presencia de una estructura de primer orden *U* llamada el *dominio de computación* sobre el cual pueden realizarse cuantificaciones de primer orden. Los estados ya no son puntos abstractos sino que ahora son valuaciones de un conjunto de variables en el dominio *U*. Los programas atómicos de DL ya no son relaciones binarias abstractas sino que representan modificaciones sobre los valores de las variables; el ejemplo más básico es el programa primitivo x:=t donde x es una variable y t es un término; este programa asigna el valor de t a la variable x.

3.7.1 Sintaxis

El lenguaje de la Lógica Dinámica de primer orden se construye sobre el lenguaje de la Lógica de Primer Orden clásica. Siempre existe en forma subyacente un vocabulario de primer orden Σ que involucra símbolos de funciones y símbolos de predicados. Sobre este vocabulario se define el conjunto de los programas y el conjunto de las fórmulas dinámicas.

Sea Σ=(...,f,g,...,p,r,...) un vocabulario de primer orden finito, donde f y g denotan símbolos de función y p y r denotan símbolos de predicado. Cada símbolo en Σ tiene asociada una aridad(cantidad de argumentos). Asumimos que Σ siempre contiene el símbolo de igualdad = cuya aridad es dos. Usaremos un conjunto numerable de variables individuales V={x_0,x_1,...}.

Fórmulas atómicas y Programas atómicos

En general, las fórmulas atómicas de DL son las fórmulas atómicas del vocabulario de primer orden Σ. Es decir, fórmulas de la forma:

$p(t_1,...,t_n)$

donde p es un símbolo de predicado n-ario de Σ y $t_1,...,t_n$ son términos de Σ.

Tal como en PDL, los programas se definen inductivamente a partir de programas atómicos usando los constructores de programas. El significado de un programa compuesto se define inductivamente en términos del significado de las partes que lo integran. Diferentes clases de programas se obtienen variando la elección de los programas atómicos y de los constructores de programas permitidos. En la versión básica de DL los programas atómicos son las asignaciones simples de la forma x:=t, donde x es una variable y t es un término de Σ. Esto representa la instrucción de asignación presente en los lenguajes de programación convencionales.

Tests

Como en PDL, DL contiene un operador de test ?, el cual convierte una fórmula en un programa. En la mayoría de las versiones de DL sólo fórmulas sin cuantificadores son permitidas en los tests. Estas versiones se denominan *basic test*. Alternativamente cualquier fórmula de primer orden puede ser admitida como test. Y finalmente podría no colocarse ninguna restricción a la forma de los test, permitiendo que cualquier fórmula actúe como test, inclusive fórmulas conteniendo programas (y posiblemente otros tests). Estas versiones de DL se denominan *rich test*. De aquí en adelante sólo trabajaremos con las versiones *basic test* de DL.

Programas Regulares

Para un conjunto dado de programas atómicos y tests el conjunto de programas regulares se define como en PDL:

(i) los programas atómicos son programas

(ii) los tests son programas

(iii) si α y β son programas entonces α ; β es un programa

(iv) si α y β son programas entonces $\alpha \cup \beta$ es un programa

(v) si α es un programa entonces α^* es un programa

Programas While

Gran parte de la literatura sobre DL trata con la clase de los programas **while** determinísticos. Los programas while determinísticos son una subclase de los programas regulares en los cuales los operadores $\cup$,? y * están obligados a aparecer sólo en las siguientes formas:

skip	=	***true***?
fail	=	***false***?
if φ **then** α **else** β	=	$\varphi?;\alpha \cup \neg\varphi?;\beta$
while φ **do** α	=	$(\varphi?;\alpha)^* ; \neg\varphi?$

La clase de los programas **while** determinísticos es importante dado que captura las construcciones de programación básicas, presentes en la mayoría de los lenguajes de programación imperativos convencionales. Sobre la estructura standard de los números naturales $\mathbb{N}$ los programas while determinísticos son suficientemente poderosos como para definir todas las funciones parciales recursivas y por lo tanto sobre $\mathbb{N}$ son tan expresivos como los programas regulares. Un resultado similar se obtiene sobre numerosos modelos similares a N, sin embargo se sabe que los programas **while** tienen menor poder expresivo que los programas regulares cuando se consideran otras estructuras distintas de $\mathbb{N}$.

Programas y Fórmulas

El lenguaje de DL regular contiene expresiones de dos tipos distintos: proposiciones o fórmulas $\psi,\varphi,...$ y programas $\alpha, \beta,...$ Los programas atómicos son denotados a,b,c... y el conjunto de todos los programas atómicos es denotado Π_0. Las fórmulas atómicas son denotadas p,q,r,...y el conjunto de todas las fórmulas atómicas es denotado Φ_0

Consideremos el conjunto Π de todos los programas **while** determinísticos construídos a partir de los programas atómicos y usando como tests a las fórmulas de Φ sin cuantificadores (es decir *basic tests*). El conjunto Φ de todas las fórmulas se define por inducción mutua como el menor conjunto tal que:

- $\Phi_0 \subseteq \Phi$
- Si $\varphi, \psi \in \Phi$ entonces $\varphi \vee \psi \in \Phi$ y también $\neg\varphi \in \Phi$
- Si $\alpha \in \Pi$ y $\varphi \in \Phi$ entonces $\langle\alpha\rangle \varphi \in \Phi$
- Si $\varphi \in \Phi$ y $x \in V$ entonces $\forall x\, \varphi \in \Phi$

El cuantificador existencial se define en términos del cuantificador universal. $\exists x\, \varphi$ es una abreviatura de $\neg\forall x \neg\varphi$. Similarmente el constructor modal [ ] y los demás conectivos lógicos $\wedge$, $\rightarrow$, etc. se definen como en PDL.

3.7.2 Semántica

En esta sección describiremos el significado de las construcciones sintácticas presentadas en las secciones previas. Programas y fórmulas se interpretan en términos de una estructura de primer orden U. Para la descripción de la semántica de los programas adoptamos un punto de vista operacional: los programas modifican los valores de las variables mediante secuencias de asignaciones simples de la forma x:=t u otras asignaciones más complejas y el flujo de control es determinado por los valores de verdad de los tests que se realizan durante la computación.

Estados como valuaciones

En cada instante durante la computación, toda la información relevante queda determinada por los valores de las variables del programa. Por lo tanto es razonable identificar estados con valuaciones de las variables V sobre los dominios de la estructura U. La definición formal asociará el par (u,v) de valuaciones con el programa α si es posible comenzar en una valuación u, ejecutar el programa α y terminar en una valuación v. En este caso (u,v) se denomina un par entrada/salida para α y se

denota con $(u,v) \in m_U(\alpha)$. Esta interpretación de los programas da origen a Kripke-frames como fue descripto para PDL.

Sea

$$U = (A , m_U)$$

una estructura de primer orden para el vocabulario Σ. U se denomina el dominio de computación. Aquí A es un conjunto llamado el carrier de U y m_U es la función de interpretación, tal que $m_U(f)$ es una función n-aria $m_U(f):A^n \rightarrow A$ interpretando el símbolo de función n-ario f de Σ y $m_U(p)$ es una relación n-aria $m_U(p) \subseteq A^n$ interpretando el símbolo de predicado n-ario p de Σ. También suele usarse la notación f^U y p^U para referirse a las interpretaciones de los símbolos de función y predicado.

La estructura U determina un Kripke-frame que llamaremos U de la siguiente forma:

$$U = (S^U, m_U)$$

donde el conjunto de estados del Kripke-frame, que denotamos como S^U, es el conjunto de todas las posibles valuaciones para las variables de V (es decir, todas las posibles funciones u que asignan valores a las variables, tal que $u(x) \in A$, para cada $x \in V$).

Programas y fórmulas

Con cada programa α se asocia una relación binaria llamada la relación entrada/salida de α :

$$m_U(\alpha) \subseteq S^U \times S^U$$

y con cada fórmula φ se asocia un conjunto, que intuitivamente representa los estados donde la fórmula es verdadera:

$$m_U(\varphi) \subseteq S^U$$

Estos conjuntos $m_U(\alpha)$ y $m_U(\varphi)$ se definen por inducción mutua sobre la estructura de α y φ, de la misma forma que para PDL. La base de la inducción es la semántica de la instrucción de asignación básica: $m_U(x:=t) = \{ (u,u[x/u(t)]) \mid u \in S^U \}$

El significado de los programas regulares compuestos se define de la siguiente forma:

- $m_U(\alpha;\beta) = m_U(\alpha) \circ m_U(\beta)$
- $m_U(\alpha \cup \beta) = m_U(\alpha) \cup m_U(\beta)$
- $m_U(\alpha^*) = m_U(\alpha)^*$
- $m_U(\varphi?) = \{(u,u) \mid u \in m_U(\varphi)\}$

Ahora describimos el significado de las fórmulas compuestas de DL. La semántica de las fórmulas atómicas de primer orden es el usual:

- $m_U(p(t_1,..,t_n)) =$ $\{u \mid u \in S^U$ y $p^U(u(t_1),..., u(t_n))\}$
- $m_U(\varphi \vee \psi) =$ $m_U(\varphi) \cup m_U(\psi)$
- $m_U(\neg\varphi) =$ $U - m_U(\varphi)$
- $m_U(\langle\alpha\rangle \varphi) =$ $m_U(\alpha) \circ m_U(\varphi)$
- $m_U(\forall x \varphi) =$ $\{u \mid \forall a \in A\ u[x/a] \in m_U(\varphi)\}$

Es importante destacar que para programas determinísticos α, $m_U(\alpha)$ es una función parcial de estados en estados. Es decir que para cada estado u existe a lo sumo un estado v tal que $(u,v) \in m_U(\alpha)$. La parcialidad de la función se debe a la posibilidad de que el programa no termine. Por ejemplo, m_U(**while *true* do skip**) es la relación vacía.

Satisfacción y validez de fórmulas

Los conceptos de satisfacción y validez se definen como los correspondientes conceptos en PDL y en Lógica de Primer Orden.

Sea $U = (S^U, m_U)$ un Kripke-frame y sea u un estado en S^U. Para una fórmula φ escribimos

$$U,u \models \varphi \text{ si } u \in m_U(\varphi)$$

y esto significa "φ es true en el estado (o mundo) u del frame U" o también "u satisface φ en el frame U" .

3.7.3 Niveles de Razonamiento en DL

La Lógica Dinámica de primer orden posibilita dos niveles distintos de razonamiento: interpretado y no-interpretado.

Como ya hemos mencionado un estado puede verse como una función que asigna valores -pertenecientes a un dominio dado- a las variables y el efecto de los programas puede interpretarse como modificaciones sobre los valores de esas variables. En el ***nivel de razonamiento no-interpretado*** no se asumen propiedades particulares de las estructuras, sino que se razona sobre propiedades que son verdaderas independientemente del dominio de valores que se considere. Por ejemplo la fórmula

$$p(f(x),g(y,f(x))) \rightarrow \langle z:=f(x)\rangle p(z,g(y,z))$$

es verdadera sobre cualquier dominio, sin importar cuales sean las interpretaciones concretas de los símbolos p, f y g.

Este nivel de razonamiento es apropiado para comparar propiedades de los lenguajes de programación ya que la comparación no depende de propiedades particulares del dominio de discurso, sino que dependen del propio poder expresivo del lenguaje. Por ejemplo, si abandonamos el nivel no-interpretado y asumimos el dominio de los números naturales $\mathbb{N}$ con cero, adición y multiplicación, todos los lenguajes de programación razonables son equivalentes en su poder computacional (computan las funciones recursivas parciales). Sin embargo, se sabe que en el nivel no-interpretado, la recursión es una construcción computacional más poderosa que la iteración.

Otro hecho importante es que en el nivel no-interpretado, DL posee mayor poder expresivo que la Lógica de Primer Orden.

Por otro parte, en el ***nivel de razonamiento interpretado***, consideraremos las estructuras aritméticas. Estas estructuras son las más cercanas al proceso de razonamiento real acerca de los programas concretos. Sintácticamente los programas y las fórmulas son los mismos en ambos niveles, no-interpretado e interpretado, lo que cambia es que en el nivel interpretado se asume una clase fija de estructuras sobre las cuales programas y fórmulas son interpretados. La estructura específica $\mathbb{N}$ puede ser generalizada lo cual conduce a la clase de las *estructuras aritméticas*. Una estructura U es aritmética si contiene una copia definible en primer orden de $\mathbb{N}$ y posee funciones definibles en primer orden para codificar secuencias finitas de elementos de U en elementos simples y también posee las funciones correspondientes para decodificar. Las estructuras aritméticas son importantes debido a que al interpretar la Lógica Dinámica sobre ellas no se pierde generalidad, ya que:

(i) La mayoría de las estructuras que surgen naturalmente en ciencias de la computación (por ejemplo estructuras discretas con tipos de datos definidos recursivamente) son aritméticas.

(ii) Toda estructura puede ser extendida a una estructura aritmética mediante el agregado de las funciones de codificación y decodificación.

En este contexto es posible estudiar programas cuyo comportamiento depende (en algunos casos fuertemente) de las propiedades de la estructura particular que se considera. En general, la tarea de verificar la corrección de un programa real se reduce a algún tipo de razonamiento sobre la estructura subyacente. Por ejemplo, la siguiente formula DL es válida en la estructura de los números naturales:

$$(x=x' \wedge y=y' \wedge xy \geq 1) \rightarrow \langle \alpha \rangle x=gcd(x',y')$$

Esta fórmula establece la corrección y terminación de un programa α sobre $\mathbb{N}$ computando el máximo común divisor, donde α podría ser el siguiente programa:

$$(x \neq y?;((x>y?,x:=x-y) \cup (x<y?;y:=y-x)))^*x=y?$$

Con respecto al poder expresivo de la lógica, considerando sólo las estructuras aritméticas DL posee el mismo poder expresivo que la Lógica de Primer Orden.

3.7.4 Complejidad de DL

En esta sección discutiremos brevemente cual es la complejidad de la Lógica Dinámica de primer orden, es decir cual es la dificultad para establecer la verdad o falsedad de una fórmula en DL. En el nivel no-interpretado nos enfrentamos con el problema de validar una fórmula sobre todos las posibles estructuras, mientras que en el nivel de razonamiento interpretado solamente debemos considerar las estructuras aritméticas. Esta característica determina diferencias en la complejidad de las pruebas en los diferentes niveles.

Ya que todas las versiones de DL contienen a la Lógica de Primer Orden, las pruebas en DL no pueden ser más fáciles que en Lógica de Primer Orden. Esto nos da un límite inferior para la complejidad de DL. Es decir que lo mejor que puede esperarse es que las pruebas sean recursivamente enumerables. Y desde ya se sabe que son no-decidibles.

En esta sección enunciaremos los principales resultados acerca de la complejidad del problema de validación en DL. Las demostraciones de estos teoremas pueden encontrarse en [Harel et al.99], [Harel et al.77], [Harel and Kozen 84] y [Pratt 76].

El nivel no-interpretado:

Teorema 1: el problema de validar una fórmula en DL es Π^1_1-hard, aún para fórmulas de la forma

$$\exists x[\alpha]\varphi$$

donde α es un programa regular y φ es una fórmula de primer orden. Π^1_1-hard denota la clase de los conjuntos co-recursivamente enumerables en la jerarquía analítica (no aritmética, de segundo orden).

Este resultado parece muy desalentador, sin embargo se ha demostrado que existen casos especiales donde el problema de validación es menos complejo, tal como se expresa en los siguientes teoremas:

Teorema 2: el problema de validación para el sub-lenguaje de DL que consiste en fórmulas de la forma $\langle\alpha\rangle\varphi$, donde φ es una fórmula de primer orden y α es un programa regular es recursivamente enumerable.

Teorema 3: el problema de validación para el sub-lenguaje de DL que consiste en fórmulas de la forma $[\alpha]\varphi$, donde φ es una fórmula de primer orden y α es un programa regular, es co-r.e. en B para algún B que es co-r.e.

Corolario: el resultado de este teorema también vale para fórmulas de corrección parcial de la forma $\phi \rightarrow [\alpha]\varphi$, donde ϕ es también de primer orden.

Notemos que el problema de validación de fórmulas en el nivel no-interpretado no es recursivamente enumerable, aún para un sub-lenguaje simple. Por lo tanto no es posible dar una axiomatización completa para realizar tales validaciones.

El nivel interpretado:

Teorema 4: el problema de validación de fórmulas DL en el nivel interpretado de las estructuras aritméticas tiene complejidad recursiva y no pertenece a la jerarquía aritmética (esto se debe a que la teoría de primer orden de $\mathbb{N}$ no es aritmética).

3.7.5 Axiomatización de DL

En esta sección comentaremos el problema de axiomatizar DL.

Dado que el problema de validación en DL en el nivel no-interpretado es Π^1_1–completo no resulta posible obtener una axiomatización en el sentido standard, sin embargo en [Harel et al.99] se provee una axiomatización no-finita mediante un conjunto infinito de axiomas y una regla de inferencia con un número infinito de premisas. Por otro lado, cuando nos restringimos a determinados sub-lenguajes de DL la situación mejora y es posible dar una axiomatización finita completa, por ejemplo fórmulas de la forma $\langle\alpha\rangle\varphi$ las cuales, en el caso de programas determinísticos, expresan corrección total (ver teorema 2). En [Harel et al.99] se presenta una axiomatización finita correcta y completa para este sub-lenguaje.

Por otro lado, probar propiedades de los programas reales frecuentemente involucra razonamiento en el nivel interpretado, es decir probar validez de una fórmula sobre una estructura particular U. Una prueba típica podría usar inducción sobre la longitud de la computación para establecer un invariante para corrección parcial o mostrar un valor decreciente sobre algún conjunto bien fundado para probar terminación. En cada caso el problema se reduce a verificar ciertos hechos dependientes del dominio usualmente llamados *condiciones de verificación*. Matemáticamente hablando, esta clase de actividad es realmente una transformación efectiva de aserciones acerca de programas en aserciones acerca de la estructura subyacente.

Esta transformación puede ser guiada por inducción directa sobre la estructura del programa usando un sistema de axiomas que es completo con respecto a cualquier estructura matemática U dada. La idea esencial consiste en usar la siguiente propiedad: “para toda fórmula DL existe una fórmula de primer orden equivalente en U”, ya que en el nivel interpretado DL y la Lógica de primer orden estática tienen el mismo poder expresivo. En [Harel et al.99] se construye un sistema de axiomas que toma todas las fórmulas de primer orden U-válidas como axiomas adicionales, logrando de esta manera que las pruebas sean finitas y efectivas.

Consideremos el siguiente sistema deductivo S_3:

Axiomas

(i) todas las instancias de fórmulas válidas de PDL

(ii) $\langle x:=t\rangle\varphi \leftrightarrow \varphi[x/t]$, para φ de primer orden

Reglas de Inferencia

(MP) $$\frac{\varphi \quad \varphi\rightarrow\psi}{\psi}$$

(GEN) $$\frac{\varphi}{[\alpha]\varphi}$$

Notemos que S_3 es el sistema de axiomas para PDL presentado anteriormente con el agregado del axioma de asignación. Dada una fórmula DL φ y una estructura U denotamos con $U|\text{-}_{S3}\ \varphi$ cuando φ puede probarse en el sistema obtenido a partir de S_3 agregando el siguiente conjunto de axiomas:

(iii) todas las sentencias de primer orden válidas en U.

Teorema 5: para toda estructura expresiva y para toda fórmula ϕ de DL de la forma $\varphi \rightarrow [\alpha]\psi$, donde φ y ψ son fórmulas de primer orden y si α contiene tests son de primer orden, vale que el sistema S_3 es correcto y completo es decir:

$U|= \phi$ sii $U|\text{-}_{S3}\ \phi$

Una estructura es expresiva para un lenguaje de programación K si para todo α de K y para toda fórmula de primer orden φ existe una fórmula de primer orden ψ tal que:

$U|= \psi \leftrightarrow [\alpha]\ \varphi$

Ejemplos de estructuras expresivas para los lenguajes de programación convencionales son las estructuras finitas y las estructuras aritméticas.

3.8 Lógica Temporal

Lógica Temporal (TL) es una aplicación alternativa de la Lógica Modal para especificar y verificar programas. Fue propuesta inicialmente por Pnuelli [Pnuelli 81] y desde entonces ha sido estudiada y desarrollada por numerosos autores, por ejemplo la Lógica Temporal de Acciones TLA [Lamport 94]. Una visión general sobre este tema puede encontrarse en [Emerson 90]. TL difiere de DL básicamente en que es una lógica endógena, es decir que los programas no se expresan explícitamente en el lenguaje de la lógica.

Consideremos la siguiente descripción de un programa concurrente: existen n procesos diferentes actuando en paralelo, usando un ambiente de memoria compartida, tal que cada proceso puede alterar los valores de las variables que están siendo usadas por los demás procesos. Los procesos pueden verse como flowcharts disjuntos con nodos rotulados. Un nodo típico del proceso i-ésimo es denotado m^i. Cada proceso tiene un nodo de entrada m^i_0. Si las variables del programa son v1,..,vk entonces un estado puede ser definido como un vector

$$s= (m^1,.....,m^n, a_1,.....,a_k)$$

especificando un rótulo para cada proceso (denotando el punto en el cual se encuentra el proceso) y un valor a_i para cada variable v_i.

Cada estado se obtiene a partir de un estado anterior permitiendo que exactamente un proceso ejecute una transición definida en su flowchart. Es decir, a partir de un estado inicial

$$s_0 = (m^1_0,.....,m^n_0, a_1,.....,a_k)$$

muchas secuencias de ejecución diferentes s_0, s_1,.... pueden ser generadas, dependiendo de cual proceso sea elegido para actuar en cada paso.

Existen principalmente dos versiones: *linear time* TL y *branching time* TL. En la primera versión, un modelo es una secuencia lineal de estados de computación, representando la ejecución de un programa determinístico o bien una posible ejecución de un programa no-determinístico o de un programa concurrente. En la segunda versión un modelo es un árbol de estados, representando el espacio de todas las posibles secuencias de computación de un programa concurrente o no-determinístico.

3.8.1 Linear time TL

Las construcciones modales usadas en *linear time* TL incluyen las siguientes:

$\Box\varphi$ "φ vale en todos los estados futuros"

$\Diamond\varphi$ "φ vale en algún estado futuro"

$O\varphi$ "φ vale en el estado siguiente"

Un modelo para una fórmula en *linear time* TL es un par $U = (A , m_U)$ donde A es una secuencia de estados y m_U es una función de interpretación, de la forma usual.

La relación $U,j \models \varphi$ significa "φ es verdadera en el estado j-ésimo del modelo U" y para los conectivos temporales está definida por:

$U,j \models \Box\varphi$ sii para todo $k \geq j$, $U,k \models \varphi$

$U,j \models O\varphi$ sii $U,j+1 \models \varphi$

El conectivo $\Diamond$ se define en términos de $\Box$, como $\Diamond = \neg\Box\neg$.

Propiedades "Safety"

Las propiedades "Safety" (o invariantes) son propiedades que valen en todos los estados de la computación de un programa. Pueden expresarse como $\Box\varphi$. Veremos algunos ejemplos de tales propiedades, donde el predicado at_i será usado con la siguiente semántica:

$s \models at_i(m)$ si y sólo si $m^i = m$, o sea que el proceso i-ésimo se encuentra en el nodo m.

- ***Ejecución limpia*** (o sin errores), por ejemplo una pila nunca se desborda o las instrucciones de división nunca dividen por cero.
- ***Corrección parcial***. Si el programa termina entonces determinadas propiedades son verdaderas.
- ***Exclusión mutua***. Dos procesos nunca acceden a sus secciones críticas simultáneamente. La exclusión mutua asegura que dos procesos nunca accederán simultáneamente a los nodos m y m'. Esta propiedad puede expresarse mediante la fórmula $\Box\neg(at_i(m) \wedge at_j(m'))$
- ***Ausencia de deadlock***: deadlock ocurre cuando se llega a un estado en el cual ningún proceso puede actuar. El requerimiento de que no ocurra un deadlock puede ser expresado por la fórmula $\Box(at_1(m^1) \wedge ... \wedge at_n(m^n) \rightarrow E_1 \vee ... \vee E_n)$ donde E_i es la condición de exit del nodo m^i, esta condición es la disyunción de las proposiciones ligadas a las aristas que salen del

nodo (una proposición ligada a una arista establece los requisitos que el proceso debe cumplir para avanzar a través de ella).

Propiedades "liveness"

Las propiedades "liveness" establecen que algún hecho ocurrirá eventualmente y se expresan como $\Diamond\phi$. Los siguientes son ejemplos de esta clase de propiedades:

- ***Corrección total***. Un programa eventualmente termina y produce una salida que es correcta.
- ***Fairness o no-starvation***. Si un proceso está esperando por un recurso, eventualmente le será provisto. Por ejemplo, un sistema operativo puede recibir pedidos provenientes de varios agentes. Al recibir un pedido (p_i) el sistema enviará una señal (q_i) anunciando al agente que su pedido fue atendido. La fórmula $\Box(p_i \rightarrow \Diamond q_i)$ expresa que todo pedido será eventualmente atendido.
- ***Liveness de las variables***. Si a una variable x se le asigna un valor mediante una instrucción de asignación x:=e, entonces la variable será usada en algún estado futuro.

3.8.2 Branching Time TL

La teoría discutida hasta aquí se refiere a propiedades lógicas de una sola secuencia $s_0,s_1,...$ generada por procesos actuando en paralelo. Sin embargo cada estado tiene varios posibles estados sucesores y por lo tanto habrá varias secuencias diferentes a partir de un estado inicial determinado s_0. Cada secuencia es una rama en el árbol que representa todas las posibles ejecuciones del programa. Existen interesantes conectivos modales que pueden ser usados para razonar formalmente sobre el comportamiento de estos programas:

$[\forall F]\ \Phi$	'Φ es inevitable' (en toda rama futura hay un estado en el cual Φ es true)
$[\exists F]\Phi$	'Φ es posible' (en alguna rama futura hay un estado donde Φ es true)
$[\forall G]\Phi$	'Φ es true en todos los posibles estados futuros'
$[\exists G]\Phi$	'en alguna rama futura, Φ es true en todos los estados'.
$[\forall X]\Phi$	'Φ es true en todos los estados (inmediatamente) sucesores'
$[\exists X]\ \Phi$	'Φ es true en alguno de los estados (inmediatamente) sucesores'

Un sistema lógico con estas características, conocido como Computational Tree Logic (CTL) fue presentado por [Clarke and Emerson 81] y también fue considerado por [Ben-Ari et al.83, Manna and Pnuelli 92]. [Emerson and Halpern 85] establecieron decidibilidad y completitud para CTL, usando un método de eliminación de estados.

3.9 Lógica Dinámica Order-sorted de primer orden con igualdad

La Lógica Dinámica Order-sorted de primer orden con igualdad (que llamaremos order-sorted DL) reúne ingredientes provenientes de la Lógica order-sorted, la Lógica Ecuacional y la Lógica Dinámica. En esta sección comentaremos una versión de esta lógica que hemos obtenido adaptando levemente la definición dada por Spruit y Wieringa [Spruit et al. 93, Wieringa and Broersen 98]. Esta versión de Lógica Dinámica será la que utilizaremos en el desarrollo de esta tesis.

3.9.1 Sintaxis

El lenguaje de la order-sorted DL se construye sobre el lenguaje de la Lógica order-sorted de primer orden clásica. Siempre existe en forma subyacente un vocabulario de primer orden Σ que involucra símbolos de funciones y símbolos de predicados. Sobre este vocabulario se define el conjunto de los programas y el conjunto de las fórmulas dinámicas.

Sea Σ=((S, $\leq$), F, P, A) un vocabulario de primer orden finito, donde S es un conjunto parcialmente ordenado de nombres de sorts, F es un conjunto de símbolos de función, P es un conjunto de símbolos de predicado (incluyendo el símbolo de igualdad) y A es un conjunto de símbolos de acción (o programas atómicos). La signatura Σ tiene las siguientes propiedades:

- $S=S_{data} \cup S_{act}$, y dos elementos de distintos conjuntos no están relacionados por el orden parcial, es decir que para todo sort $s \in S_{data}$ y $a \in S_{act}$ no ocurre $a \leq s$ ni $s \leq a$. Además cada uno de estos conjuntos tiene un límite superior, es decir existe un sort act, tal que $a \leq act$ para todo $a \in S_{act}$ y existe un sort data, tal que $d \leq data$ para todo $d \in S_{data}$.
- Las declaraciones de funciones son de la forma, $f:s_1,...,s_n \rightarrow s_{n+1}$ con $s_i \in S_{data}$ para i=1..n+1. Además, F está particionado en dos conjuntos disjuntos de símbolos updatables (modificables) y nonupdatables (no-modificables). $F = F_U \cup F_N$,
- Las declaraciones de predicados son de la forma $p:s_1,...,s_n$ con $s_i \in S_{data}$ para i=1..n. Además P está particionado en dos conjuntos disjuntos de símbolos updatables (modificables) y nonupdatables (no-modificables). $P = P_U \cup P_N$
- A es el conjunto de los símbols de acción. La signatura de estos símbolos está restringida al siguiente formato: $a:s_1,... ,s_n \rightarrow s$ con $s_i \in S_{data}$ para i=1..n y $s \in S_{act}$.

Dado un conjunto de variables individuales X, el conjunto de términos de sort s sobre Σ y X, el cual denotamos mediante $T_{\Sigma,s}(X)$, se obtiene de la manera usual.

Fórmulas atómicas y Programas atómicos

Las fórmulas atómicas de order-sorted DL son las fórmulas atómicas del vocabulario de primer orden Σ. Es decir, si p es un símbolo de predicado, $p:s_1,... ,s_n \in P$ y t1,...,tn son términos, $t_i \in T_{\Sigma,Si'}(X)$ y $si' \leq si$ para i=1..n, entonces $p(t_1,...,t_n)$ es una fórmula atómica.

Tal como en DL, los programas se definen inductivamente a partir de programas atómicos usando los constructores de programas. El significado de un programa compuesto se define inductivamente en términos del significado de las partes que lo integran. En la versión básica de DL los programas atómicos son las asignaciones simples de la forma x:=t, donde x es una variable y t

es un término de Σ. En esta versión de lógica dinámica existe un conjunto mucho más rico de programas atómicos definidos de la siguiente forma:

si a es un símbolo de acción, $a:s_1,\ldots,s_n \rightarrow s \in A$ y t1,...,tn son términos, $t_i \in T_{\Sigma,Si'}(X)$ y $si' \leq si$ para i=1..n, entonces $a(t_1,\ldots,t_n)$ es un programa (o acción) atómico.

Programas y Fórmulas

El lenguaje de order-sorted DL regular contiene expresiones de dos tipos distintos: proposiciones o fórmulas $\psi,\varphi,\ldots$ y programas $\alpha, \beta,\ldots$ Los programas atómicos son denotados a,b,c... y el conjunto de todos los programas atómicos es denotado Π_0. Las fórmulas atómicas son denotadas p,q,r,...y el conjunto de todas las fórmulas atómicas es denotado Φ_0

El conjunto Π de todos los programas y el conjunto Φ de todas las fórmulas se definen por inducción mutua, tal como en DL, como los menores conjuntos tales que:

- $\Phi_0 \subseteq \Phi$
- $\Pi_0 \subseteq \Pi$
- Si $\varphi, \psi \in \Phi$ entonces $\varphi \vee \psi \in \Phi$ y también $\neg\varphi \in \Phi$
- Si $\alpha, \beta \in \Pi$ y si $\varphi \in \Phi$ y φ está libre de cuantificadores
  entonces
  skip $\in \Pi$
  fail $\in \Pi$
  if φ **then** α **else** $\beta \in \Pi$
  while φ **do** $\alpha \in \Pi$
- Si $\varphi \in \Phi$ y $x \in Vs$ entonces $(\forall x:s)\varphi \in \Phi$
- Si $\alpha \in \Pi$ y $\varphi \in \Phi$ entonces $\langle\alpha\rangle \varphi \in \Phi$
- Si $\varphi \in \Phi$ entonces $\Box\varphi \in \Phi$

El cuantificador existencial se define en términos del cuantificador universal. $\exists x\, \varphi$ es una abreviatura de $\neg\forall x \neg\varphi$. Similarmente el constructor modal [ ] y los demás conectivos lógicos $\wedge$, $\rightarrow$, etc. se definen como en DL.

El conectivo $\Box$ corresponde a la lógica temporal clásica, donde $\Box\varphi$ significa que la fórmula φ es siempre verdadera en el futuro. Además puede definirse el conectivo $\Diamond$ de la siguiente forma $\Diamond\varphi = \neg\Box\neg\varphi$, así la fórmula $\Diamond\varphi$ significa que existe un momento en el futuro en el cual φ es verdadera.

3.9.2 Semántica

La semántica de esta lógica es similar a la semántica standard para lógicas dinámicas de primer orden. Es decir que las fórmulas se interpretan sobre Kripke-frames. La diferencia con la semántica standard reside en la existencia de *términos updatable* y *non-updatable* los cuales son interpretados de diferente forma. En las lógicas dinámicas clásicas los términos reciben una interpretación fija, la cual no varía de un estado a otro; en cambio en esta lógica la interpretación de los *términos updatable* varía entre estados (mientras que la interpretación de los *símbolos non-*

updatable permanece fija). Es decir que los *términos updatables* se comportan en forma similar a las *variables*, cuyos valores sufren modificaciones a lo largo de la ejecución del programa.

Formalmente, sea Σ=((S, $\leq$), F, P, A) una signatura dinámica order-sorted de primer orden y sea Σ_N=((S, $\leq$), F_N, P_N) la parte non-updatable de Σ. Sea U=(A,m_U) una $\mathbf{\Sigma_N}$-álgebra, que provee el dominio de valores y la interpretación de los términos estáticos. Las fórmulas del lenguaje se interpretan sobre Kripke-frames de la forma:

$U=(S^U,w_o,m_U)$

donde:

- S^U es el conjunto de estados. Cada estado $w \in S^U$, es una función de interpretación de términos sobre el álgebra U, de la siguiente forma:

  si $f \in F_N$ entonces $w(f)=f^U$ (es decir, la interpretación fija dada por el álgebra U).

  si $f \in F_U$ y $f:s_1,..,s_n \rightarrow s$ entonces $w(f): U_{s1},..,U_{sn} \rightarrow U_s$.

  si $p \in P_N$ entonces $w(p)=p^U$ (es decir, la interpretación fija dada por el álgebra U).

  si $p \in P_U$ y $p:s_1,..,s_n$ entonces $w(p): U_{s1},..,U_{sn}$

  si x es una variable de sort s, entonces $w(x) \in U_s$.
- $w_0 \in S^U$ es el estado inicial.
- m_U asocia cada programa α una relación binaria llamada la relación entrada/salida de α:

  $m_U(\alpha) \subseteq S^U \times S^U$

El conjunto $m_U(\alpha)$se definen por inducción mutua sobre la estructura de α de la misma forma que para PDL. La base de la inducción es la semántica de los programas atómicos: $m_U(a(t_1,..,t_n))$:

- $m_U(\alpha;\beta)$ = $m_U(\alpha) \circ m_U(\beta)$
- $m_U(\alpha \cup \beta)$ = $m_U(\alpha) \cup m_U(\beta)$
- $m_U(\alpha^*)$ = $m_U(\alpha)^*$
- $m_U(\varphi?)$ = $\{(u,u) \mid u \in m_U(\varphi)\}$

Tal como en DL, la relación U,u |= φ significa "φ es true en el estado (o mundo) u del frame U" o también "u satisface φ en el frame U" y está definida de la siguiente forma:

$U,u \models p(t_1,..,t_n)$	sii	$u(p)(u(t_1),..., u(t_n))$
$U,u \models \varphi \vee y$	sii	$U,u \models \varphi$ ó $U,u \models y$
$U,u \models \neg\varphi$	sii	$U,u \not\models \varphi$
$U,u \models (\forall x\ \varphi)$	sii	para todo a de A se cumple que $U,u[x/a] \models \varphi$
$U,u \models \langle\alpha\rangle\ \varphi$	sii	existe un v tal que $(u,v) \in m_U(\alpha)$ y $U,v \models \varphi$

Un modelo para una fórmula ϕ es una estructura $U=(S^U,w_o,m_U)$ tal que $U,w_0 \models \phi$.

Semántica del mínimo cambio (minimal change semantics)

El modelo minimal de una fórmula se define como el modelo en el cual las transiciones causan el menor cambio de estado posible necesario para satisfacer la fórmula.

Supongamos que dos acciones α_1 y α_2 están definidas por los siguientes fórmulas $[\alpha_1]\phi$ and $[\alpha_2]\varphi$ y en el estado corriente del sistema, sea w, tenemos que $w|=\neg\phi\wedge\neg\varphi$. Entonces la interpretación de mínimo cambio de la acción α_1 lleva a un estado donde se satisface $\phi\wedge\neg\varphi$ y la interpretación de mínimo cambio de α_2 lleva a un estado donde se satisface $\varphi\wedge\neg\phi$. Notemos que en la interpretación libre de una acción es "la acción cumple sus post-condiciones y cualquier otra cosa puede cambiar"; mientras que la interpretación del modelo minimal es "la acción cumple sus post-condiciones y ninguna otra cosa puede cambiar".

Semántica de máxima alcanzabilidad (maximal reachability semantics)

El ordenamiento de alcanzabilidad se define de la siguiente manera:

Sean M=(U, W,w,R) y M'=(U, W',w',R') dos Kripke-frames para una signatura dinámica de primer orden order-sorted Σ=((S, $\leq$), F, P, A). La relación $\leq$**r** se define de la siguiente forma:

$M \leq_{\mathbf{r}} M'$ sii $W \subseteq W'$ and $R_\alpha \subseteq R'_\alpha$ para todo término de acción $\alpha \in U_{Act}$.

Un modelo que es maximal bajo el orden de alcanzabilidad definido arriba, contiene el máximo no-determinismo permitido por las fórmulas que actúan como guardas (una guarda es una fórmula de la forma $<\alpha>$***true*** $\rightarrow\phi$). Notemos que cada fórmula guarda especifica una pre-condición necesaria para la ejecución del programa α.

En un modelo maximal bajo el orden de alcanzabilidad la conjunción de todas las pre-condiciones necesarias es una condición suficiente para la ejecución de α.

Semántica del paso máximo (maximal step semantics)

Un paso es un conjunto finito de acciones que se ejecutan simultáneamente. Usaremos la notación $w \rightarrow_s w'$ para indicar que el conjunto de acciones s define una transición del estado w al estado w´. Notemos que si $\{\alpha 1,\alpha 2\}$ ocurre en una transición, las post-condiciones de $\alpha 1$ son consistentes con las post-condiciones de $\alpha 2$ (en este caso decimos que $\alpha 1$y $\alpha 2$ son no-conflictivos). Esto permite definir la siguiente regla de inclusión de pasos:

$s' \subseteq s$ entonces $[s']\phi \rightarrow [s]\phi$

En un estado dado, un paso es maximal cuando contiene un conjunto maximal de acciones mutuamente no-conflictivas habilitadas en ese estado.

Semántica intuitiva (intended semantics)

En [Wieringa and Broersen 98] se define una semántica intuitiva para fórmulas dinámicas. Esta semántica refleja apropiadamente el comportamiento intuitivo de los sistemas orientados a objetos. Entre todos los posibles modelos de una fórmula esta semántica selecciona los modelos que cumplen las siguientes propiedades:

- *Mínimo cambio*
- *Máxima alcanzabilidad*
- *Paso máximo*

Dada una fórmula ϕ de una lógica dinámica order-sorted de primer orden; el modelo *min-max* para ϕ se define como el elemento $\leq_{\mathbf{r}}$ maximal del conjunto de modelos minimales de ϕ. Minimalización toma los modelos en los cuales las transiciones causan el menor cambio de estado posible necesario para satisfacer la fórmula. Entre estos modelos maximalización $\leq_{\mathbf{r}}$ toma el modelo que es maximal bajo el orden de alcanzabilidad. En [Wieringa and Broersen 98] se demuestra que el elemento máximal existe y es único.

Luego, el modelo intuitivo para ϕ se obtiene a partir del modelo min-max eliminando, para cada estado w en el modelo min-max, todas las transiciones $w \rightarrow_s w'$ para las cuales existe una transición $w \rightarrow_{s'} w''$ con $s \subseteq s'$. Por lo tanto en el modelo intuitivo, sólo los pasos máximos son tomados. Nótese que el modelo intuitivo puede ser no-determinístico.

4. Combinando técnicas de modelado

4.1 Introducción

En el capítulo anterior hemos destacado la necesidad de integrar lenguajes de modelado gráficos, cercanos a las necesidades del dominio de aplicación con lenguajes de modelado formales provistos de herramientas de análisis y verificación. Con el objetivo de clarificar el significado y los alcances del problema consistente en la integración de ambos tipos de lenguaje, dedicaremos esta sección a discutir y comparar las diferentes soluciones que han sido propuestas al mencionado problema.

Hemos identificado básicamente cuatro propuestas diferentes para realizar la integración:

- **Suplemento**. La propuesta de suplemento (ver figura 4.1) consiste en enriquecer un modelo informal con conceptos formales. Buenos ejemplos de esta propuesta son Syntropy [Cook and Daniels 94] y los trabajos de Lano [Goldsack and Kent 96] y Weber [Weber 96] los cuales proponen utilizar la notación formal Z[Spivey 92] para enriquecer notaciones semi-formales.
- **Extensión**. La propuesta de extensión (ver figura 4.2) consiste en extender una notación formal existente, con conceptos más cercanos al dominio de la aplicación, tales como los conceptos adoptados por el paradigma de orientación a objetos. De esta forma la notación formal se vuelve más fácil de entender y manejar por parte de los desarrolladores de software. Los ejemplos más relevantes de esta propuesta son las extensiones del lenguaje Z, como por ejemplo Z++[Lano 91] y Object-Z[Duke et al. 91]. También corresponden a este grupo los lenguajes TROLL [Jungclaus et al. 96] OOZE [Alencar and Goguen 91] y MAUDE [Meseguer and Winkler 91] inspirados sobre lenguajes de especificaciones algebraicas.
- **Interfaz**. Dado un método de modelado formal, esta propuesta consiste en desarrollar una notación gráfica alternativa para facilitar la creación y visualización de los modelos (ver figura 4.3). Algunos ejemplos de esta propuesta son las interfaces gráficas provistas por los lenguajes formales OASIS [Pastor and Ramos 95] y LTL [Reggio and Larosa 97].
- **Semántica**. Esta propuesta (ver figura 4.4) consiste en definir formalmente la semántica de un lenguaje de modelado conocido y aceptado por la comunidad. Sus principales componentes son reglas para asociar estructuras sintácticas del lenguaje de modelado con elementos en un dominio semántico formalmente definido.

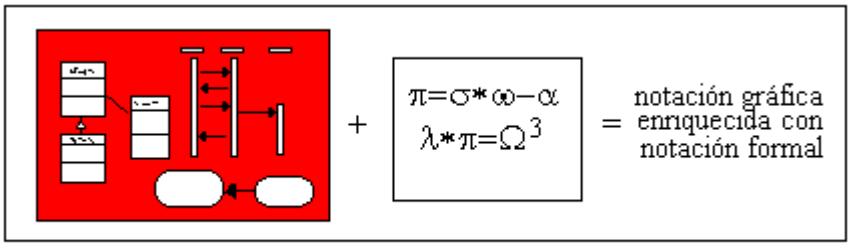


Figura 4.1: integración por suplemento

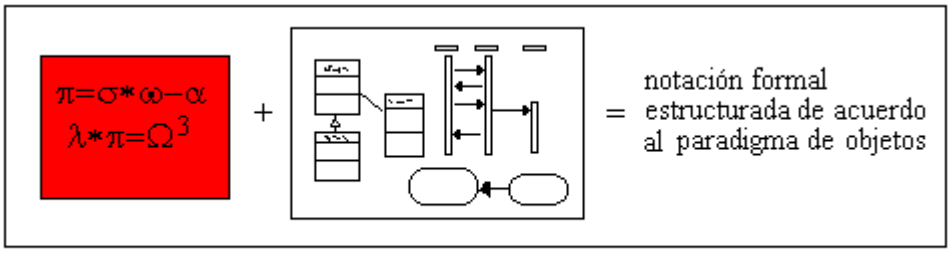


Figura 4.2: integración por extensión

De acuerdo con nuestra opinión, "semántica" constituye la propuesta más adecuada, dado que permite que especificaciones expresadas en una notación conocida y aceptada por los desarrolladores de software adquiera significado preciso a través de su "traducción" en un dominio formal. Nuestra opinión se basa en que tanto las propuestas de integración por suplemento, como las propuestas de integración por extensión requieren que los desarrolladores conozcan la notación formal, dado que ésta constituye una parte visible de la especificación. Por otra parte, la desventaja de las propuestas de integración por interfaz es que el usuario se ve forzado a adoptar un nuevo lenguaje gráfico, el cual generalmente posee notorias influencias provenientes del formalismo, que lo tornan poco intuitivo.

La principal ventaja de la propuesta semántica sobre las demás propuestas reside en que el

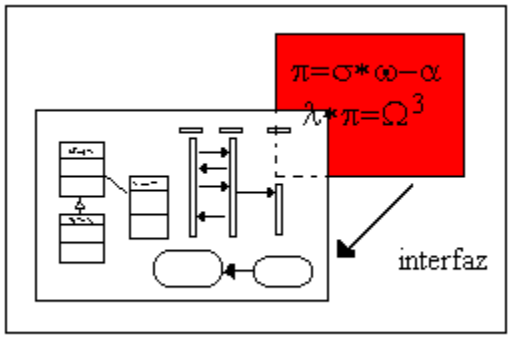


Figura 4.3: integración por interfaz

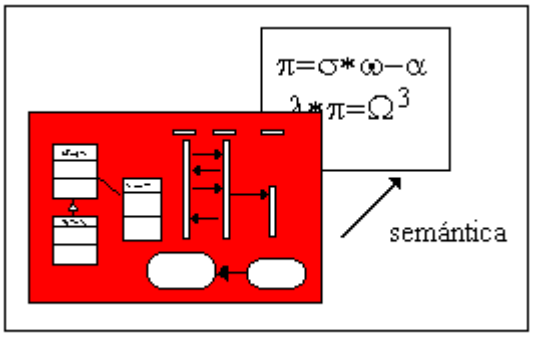


Figura 4.4: integración semántica

lenguaje gráfico se convierte en un lenguaje formal y por lo tanto las especificaciones escritas utilizando el lenguaje gráfico pueden ser formalmente analizadas para detectar contradicciones y ambigüedades tempranamente en el proceso de desarrollo del software. Una de las claves para el éxito de esta propuesta reside en ocultar la notación matemática tanto como sea posible tras la notación gráfica. Por ejemplo, debería ser posible utilizar la semántica formal para desarrollar herramientas CASE. Sólo los desarrolladores del lenguaje deberían usar el formalismo para construir las herramientas CASE y justificar su corrección, mientras que los desarrolladores de software de aplicación podrían manejar los modelos gráficos sin necesidad de conocer el formalismo matemático subyacente.

A partir de la estandarización del lenguaje gráfico de modelado Unified Modeling Language (UML) [UML 97 (a)(b), UML 98 (a) (b)] han surgido activas discusiones acerca de la precisión semántica de sus construcciones. Mientras que el OMG fue responsable por la estandarización de UML como notación, la semántica de UML aún es un tema de investigación. Existe un número importante de trabajos teóricos (ver por ejemplo [UML-conference 98]) que tratan diferentes partes de UML definiendo formalmente su semántica. Sin embargo todavía resta un largo camino por recorrer. En particular, es difícil comparar los resultados de los respectivos artículos y es aun más difícil

combinar dichos resultados con el objetivo de obtener una semántica standard para UML. Esta dificultad surge especialmente porque los diferentes trabajos utilizan diferentes métodos (o lenguajes) formales, o cubren un subconjunto de la notación o asumen una subclase particular de sistemas a ser especificados. Sin embargo, surge una clara clasificación de las propuestas en dos grupos: formalizaciones basadas en el modelo y formalizaciones basadas en el metamodelo, tal como explicaremos en las siguientes secciones de este capítulo.

4.2 Una arquitectura de dos niveles: modelo vs. metamodelo

Los lenguajes de modelado visuales son lenguajes gráficos que se usan para la especificación, visualización, documentación de productos de software como paso previo a su construcción. Generalmente el marco conceptual de las notaciones de modelado se basa sobre una arquitectura formada por distintos niveles (ver figura 4.5 [Odell95]). La descripción de esta arquitectura puede encontrarse por ejemplo en [UML 97(b)].

Existen dos niveles principales:

1. **El nivel del metamodelo**. El metamodelo es un modelo para la información que puede ser expresada durante la construcción del modelo de un sistema. Básicamente, define los elementos de modelado tales como *Class_diagrams, State_machines, Sequence_diagrams* en el metamodelo de UML. Además define en que forma estos elementos se relacionan para conformar el modelo de un sistema. El metamodelo es una descripción del lenguaje de modelado en sí. Su semántica es el conjunto de todos los modelos bien formados. El metamodelo es independiente de cualquier modelo en particular, él describe los elementos del lenguaje y las restricciones que deben cumplir todos los modelos, por ejemplo: “dentro de un Classifier los nombres de los atributos no se repiten”.
2. **El nivel del modelo**. Por otro lado, el modelo es una instancia del metamodelo. El modelo (en realidad un modelo no es una unidad sino que es un conjunto de diversos modelos relacionados, tal como fue explicado en la sección 1.3) describe los objetos inherentes al dominio de la aplicación que está siendo modelada, por ejemplo: *Employer, Employee*, *BankAccount, Client,* etc. e incluye restricciones que deben ser satisfechas por los objetos del sistema, por ejemplo “no se permiten extracciones cuando el saldo de una cuenta es menor que cero”.

Existen además otros dos niveles relacionados con los anteriores:

1. ***El nivel de los datos.*** En el nivel de los datos, las entidades son instancias de las clases del modelo, por ejemplo los objetos *Robert* y *Chris* son instancias de la Clase Employee.
2. **El nivel del meta-metamodelo.** Es necesario contar con un nivel superior describiendo el lenguaje utilizado para expresar el metamodelo. Este nivel superior es llamado meta-metamodelo. Por ejemplo en esta tesis utilizaremos Dynamic Logic como meta-lenguaje.

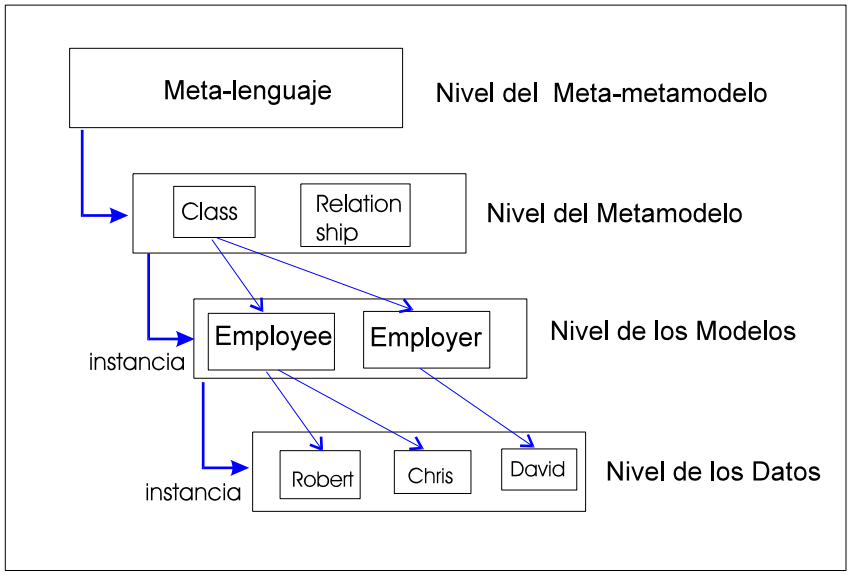


Figura 4.5 : arquitectura de cuatro niveles

4.3 Semántica de Lenguajes

En el área de semántica de lenguajes es reconocida la existencia de distintas dimensiones que conforman la descripción de un lenguaje: por un lado se encuentra la sintaxis y la semántica del lenguaje y por otro lado se tiene la dicotomía entre conceptos estáticos y conceptos dinámicos. A continuación describiremos cada una de estas dimensiones.

Sintaxis y Semántica:

En notación textual convencional, la sintaxis de un lenguaje es descripta por el conjunto de caracteres usados (alfabeto) y las posibles secuencias de esos símbolos (palabras, frases). Llamamos lenguaje al conjunto de todas las secuencias correctas. Cuando la notación involucra diagramas, su sintaxis es más compleja dado que ya no se limita a una secuencia lineal de caracteres, sino que comprende elementos gráficos de dos (o aún tres) dimensiones.

Por otra parte, la semántica de un lenguaje nos habla acerca del significado de cada construcción del lenguaje. Usualmente esto se hace explicando las construcciones del nuevo lenguaje en términos de conceptos ya conocidos (y bien entendidos). Este conjunto de conceptos bien entendidos es llamado el dominio semántico. Por ejemplo, dado que UML se utiliza para especificar sistemas orientados a objetos, es esperable que un dominio semántico para UML contenga todos los conceptos de la orientación a objetos tales como objetos, mensajes, generalización y especialización, composición, etc.

Observemos cómo estos conceptos se corresponden con los distintos niveles de la arquitectura de las notaciones de modelado (ver figura 4.5): la sintaxis de UML es descripta en el nivel del metamodelo. El nivel del modelo contiene modelos particulares, los cuales son construcciones correctas del lenguaje. Finalmente en el nivel inferior (el nivel de los datos) encontramos entidades semánticas tales como objetos. De acuerdo con esta identificación de conceptos de aquí en adelante llamaremos *modelo* a las construcciones (sintácticas) del lenguaje UML y llamaremos *sistema modelado* a la interpretación semántica de un modelo.

Conceptos estáticos y conceptos dinámicos:

En la descripción de un lenguaje existen además otras dos dimensiones ortogonales a sintaxis y semántica: aspectos estáticos y aspectos dinámicos.

La diferenciación entre semántica estática y semántica dinámica es bien conocida y aceptada. Mientras que la semántica estática caracteriza propiedades estáticas (invariantes en el tiempo) de las interpretaciones del lenguje, la semántica dinámica describe la evolución o comportamiento de dichas interpretaciones.

Sin embargo, hablando de sintaxis, generalmente sólo se definen reglas de buena formación para las construcciones del lenguaje, pero no se trata el tema de su evolución o dinamismo. Es decir, la sintaxis es analizada sólo desde el punto de vista estático. La dimensión faltante, es decir sintaxis-dinámica, cobra importancia cuando se intenta dar semántica a un lenguaje de modelado en un contexto donde los modelos (las construcciones del lenguaje) pueden evolucionar (sufrir modificaciones) durante su ciclo de vida. Es importante aclarar que no estamos hablando de cambios en la sintaxis del lenguaje (lo cual representaría evolución en el nivel del metamodelo), sino de cambios en los modelos construidos utilizando el lenguaje(es decir evolución en el nivel del modelo).

La siguiente tabla esquematiza la relación entre ambas dimensiones, donde spec es un modelo (o sea un elemento sintáctico) y sys es el sistema modelado por spec (o sea un elemento semántico).

	Modelo (dominio sintáctico)	Sistema modelado (dominio semántico)
Aspectos estáticos	Reglas de buena formación de los modelos.	Reglas de buena formación de los objetos en el sistema
Aspectos dinámicos	Evolución de los modelos.	Evolución de los objetos del sistema

En la siguiente tabla mostramos ejemplos de los conceptos generales expresados arriba:

	Modelo (spec)	Sistema modelado (sys)
Aspectos estáticos	spec no debe contener dos atributos con el mismo nombre dentro de la misma clase.	Los valores de los atributos de los objetos de sys deben corresponderse con las declaraciones en las respectivas clases.
Aspectos dinámicos	Refinamiento del modelo spec por el agregado de una nueva clase.	Los objetos de sys reaccionan al recibir un mensaje.

En relación con UML, es importante destacar que la dimensión estática del sistema modelado constituye la semántica de los diagramas de estructura, tales como diagrama de clases y diagrama de relaciones. Mientras que la dimensión dinámica del sistema modelado puede (y debe) utilizarse para definir la semántica de los diagramas de comportamiento, tales como máquinas de estado y diagramas de interacción.

4.4 Clasificación de las propuestas para dar semántica a los lenguajes de modelado

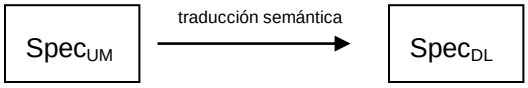
Como hemos mencionado anteriormente, los trabajos de formalización realizados sobre notaciones de modelado (y en particular sobre UML) pueden clasificarse en dos grupos: basados en el modelo o basados en el metamodelo. El 'foco' de la formalización constituye el elemento discriminatorio entre ambos grupos. Las formalizaciones del primer grupo centran su atención sobre el nivel del modelo, mientras que las formalizaciones del segundo grupo lo hacen sobre el nivel del metamodelo.

En los métodos pertenecientes al grupo basado en el modelo (ver por ejemplo [Moreira and Clark 94, France et al. 97(a), Goldsack and Kent 96, Waldoke et al. 98, Wieringa and Broersen 98, Lano and Biccaregui 98]) las especificaciones formales son generadas a partir de modelos orientados a objetos no-formales. Las especificaciones formales así obtenidas describen a los objetos del dominio de la aplicación (por ejemplo, cuentas y clientes en una aplicación bancaria). Es decir, la formalización se centra en el sistema particular que ha sido descripto utilizando la técnica de modelado.

En los métodos pertenecientes al grupo basado en el metamodelo (ver por ejemplo [France et al. 97 (b), Breu et al. 97, UML 97(b), Evans et al. 98]), el objetivo es dar una descripción precisa de los elementos que componen la técnica de modelado y proveer reglas para analizar sus propiedades. Las especificaciones formales describen a los elementos de modelado, tal como clases, asociaciones, generalizaciones, maquinas de estado, etc. Es decir, la formalización se centra en la técnica de modelado en sí misma.

En esta sección ilustraremos las principales diferencias entre ambas propuestas mediante un ejemplo. Usaremos UML [UML 97 (a)(b)] como notación gráfica y Dynamic Logic [Wieringa and Broersen 98] como lenguaje formal. La figura 4.6 muestra la especificación gráfica de un sistema bancario, la cual llamaremos $Spec_{UML}$. El ejemplo sólo contiene las principales estructuras, otras son omitidas con el fin de ganar simplicidad y claridad.

En las siguientes secciones mostraremos la especificación formal correspondiente a $Spec_{UML}$, la cual llamaremos $Spec_{DL}$, es decir:



El contenido y la forma de $Spec_{DL}$ será diferente dependiendo de la propuesta aplicada.

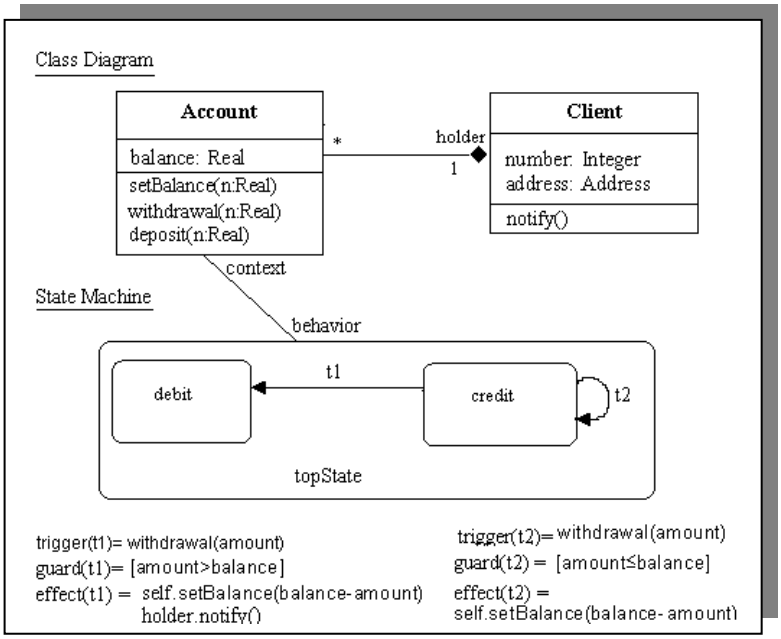


Figura 4.6: modelo UML del sistema bancario

4.4.1 Propuestas basadas en el modelo

De acuerdo con las propuestas basadas en el modelo, la formalización $Spec_{DL}$ describe a los objetos del dominio de la aplicación, es decir Account, Client, etc. Este tipo de formalización permite la especificación de reglas de buena formación de los objetos y su comportamiento, por ejemplo: las cuentas están en estado 'credit' cuando su saldo es mayor que cero, luego de un depósito el saldo de la cuenta se incrementa, etc. A continuación mostramos una parte de $Spec_{DL}$.

Specification of Account

Sorts	BankAccount
functions	balance: BankAccount → Real
	debit, credit: BankAccount →Bool
	holder: BankAccount →Client
Actions	deposit: BankAccount x Real →Act
	withdrawal: BankAccount x Real →Act
Axioms	∀c:BankAccount ∀n:Real
	credit(c) ↔ 0≤balance
	[deposit(c,n)] balance(c)= balance(c)+n
	[withdrawal(c,n)] balance(c)= balance(c) –n
	(balance<n)→[withdrawal(c,n)]enabled(notify(holder(c)))

End specification of Account

Specification of Client

Sorts	Client
functions	number: Client → Integer
Actions	notify: Client → Act
Axioms	∀c:Client
	.....

End specification of Client

Las falencias de esta propuesta son las siguientes: $Spec_{DL}$ no incluye reglas de consistencia entre diferentes elementos de modelado, por ejemplo no es posible expresar propiedades de las clases (ej. los nombres de los atributos no se repiten en una clase) o relaciones entre la descripción estructural de una clase (dada por el diagrama de clases) y la descripción de su comportamiento (dada por la máquina de estados). Por otra parte, modificar el modelo implicaría modificar la teoría lógica obtenida, es decir que resulta imposible representar evolución de modelos como parte de la teorìa.

Entre las propuestas basadas en el modelo hemos seleccionado los trabajos de Kevin Lano y Jean Biccaregui [Lano and Biccaregui 98] y el método de Roel Wieringa [Wieringa 98], los cuales consisten en obtener modelos formales a partir de modelos expresados en la notación gráfica UML. En ambas propuestas los modelos formales obtenidos describen a los objetos de la aplicación:

- Lano y Biccaregui proponen una semántica axiomática para la notación UML, usando teorías estructuradas en lógica temporal. Transformaciones sobre modelos UML, tales como agregar una clase o relación son representadas en el formalismo mediante extensiones de teorías. El tratamiento que esta propuesta da a cada una de las cuatro dimensiones discutidas antes es esquematizado en la siguiente tabla:

	Modelo	Sistema modelado
Aspectos estáticos	no considerado	axiomas en lógica de primer orden
Aspectos dinámicos	transformaciones de teorías	acciones y axiomas en lógica temporal

- Roel Wieringa define una semántica formal para UML en términos de sistemas de transición rotulados. Su propuesta se enmarca dentro de una metodología para el desarrollo de software. El tratamiento que esta propuesta da a cada una de las cuatro dimensiones es esquematizado en la siguiente tabla:

	Modelo	Sistema modelado
Aspectos estáticos	no considerado	axiomas en lógica de primer orden
Aspectos dinámicos	no considerado	acciones y axiomas en lógica dinámica

4.4.2 Propuestas basadas en el metamodelo

De acuerdo con las propuestas basadas en el metamodelo, la formalización $Spec_{DL}$ describe a los elementos de modelado en sí, tal como, clases (class), atributos de las clases (attribute), operaciones (operation), relaciones (por ejemplo: association y generalization), etc. Este tipo de formalización permite la especificación de reglas de buena formación del lenguaje, por ejemplo en la especificación de GeneralizableElement se incluyen las siguientes reglas:

[1] La raíz de una jerarquía de generalizaciones no puede tener ninguna generalización.
[2] Los elementos generalizables que son hojas no pueden tener subtipos.

Y la especificación de Classifier muestra los siguientes axiomas:

[1] Dentro de un Classifier los nombres de los atributos no se repiten.
[2] Dentro de un Classifier dos operaciones no pueden tener la misma signatura.
[3] Herencia circular no está permitida.
[4] Herencia múltiple no debe provocar conflictos de nombres.

Specification of Generalization
Sorts Generalization
Taxonomy functions Generalization ≤ModelElement
discriminator: Generalization →Name
supertype: Generalization →GeneralizableElement.
subtype : Generalization →GeneralizableElement.
Axioms ∀g: Generalization
[1] ¬isRoot(subtype(g))
[2] ¬isLeaf(supertype(g))
End specification of Generalization

Specification of Classifier
Sorts Classifier
Taxonomy functions Classifier≤GeneralizableElement
attributes: Classifier→Seq of Attribute
operations: Classifier →Seq of Operation
Axioms ∀c,c1,c2,c3:Classifier ∀a1,a2:Attribute
[1] (a1∈ attributes(c) ∧a2∈attributes(c) ∧name(a1) = name(a2)) → a1 =a2
[2] ∀f,g∈operations(c) (hasSameSignature(f,g) →f = g)
[3] IsA(c1,c2)∧IsA(c2 ,c1) → c2 = c1
[4] (IsA(c1,c2) ∧IsA(c1,c3)∧c3≠c2) →
∀f:Feature (f∈allFeatures(c2)∧f∈allFeatures(c3)
→ ∃c:Classifier(IsA(c2,c) ∧ IsA(c3,c) ∧f∈features(c)))
End specification of Classiffier

La descripción de los objetos del dominio es introducida por medio de axiomas de instanciación como se muestra en el siguiente ejemplo:

Instantiation axioms ϕ_{INST_BANK}

$\exists c_1, c_2$:Class
$(c_1 \in elements(m) \wedge name(c_1)=Account \wedge c_2 \in elements(m) \wedge name(c_2)=Client$
$\wedge \exists a$:Attribute
$(a \in attributes(c1) \wedge name(a)=balance\ type(a)=Integer)$
$\wedge \exists p1,p2$:Operation
$(p1 \in operations(c1) \wedge name(p1)=deposit \wedge size(parameters(p1))=1 \wedge type(first(parameters(p1))=Real \wedge$
$p2 \in operations(c1) \wedge name(p2)=withdrawal \wedge size(parameters(p2))=1 \wedge type(first(parameters(p2))=Real)\ )$
$\wedge \exists h$:StateMachine
$(h \in elements(m) \wedge context(h)= c_1 \wedge states(h)=\{s_1,s_2\} \wedge name(s_1)=debit \wedge name(s_2)=credit \wedge transitions(h)=\{t1,t2\} \wedge trigger(t1)= p_2 \wedge source(t1)=s_2 \wedge target(t1)=s_1 \wedge guard(t1)=(....)\ effect(t1)= (\ ...) \wedge$
$trigger(t2)= p_2 \wedge source(t2)=s_2 \wedge target(t2)=s_2 \wedge .....)))$

La falencia de esta propuesta reside en que la teoría lógica describe metaentidades lo cual dificulta la representación de las entidades correspondiente al dominio de las aplicaciones (esta información particular está presente en los axiomas de instanciación).

Entre las propuestas basadas en el metamodelo hemos seleccionado los trabajos de los creadores de UML [UML 97(a)(b), UML 98 (a) (b)] y la propuesta del "PUML group" [Evans et al. 98] integrado por Andy Evans, Robert France, Kevin Lano, Bernhard Rumpe y otros. El objetivo de estos trabajos consiste en proveer una definición precisa de la notación UML. En ambas propuestas los modelos formales se centran en la notación de modelado en sí y en las reglas para analizar sus propiedades:

- El lenguaje UML ha sido descripto básicamente en dos documentos: UML11-notation [UML 97(a)] y UML11-semantics [UML 97 (b)]. Estos documentos dan una noción clara de la sintaxis de UML, pero no logran definir precisamente la semántica del lenguaje. El documento llamado "semantics" en realidad sólo define reglas de buena formación de los modelos (es decir restricciones sobre la sintaxis) y explica la semántica de las construcciones UML de una forma poco precisa y en muchos casos contradictoria, usando lenguaje natural. El tratamiento que esta propuesta da a cada una de las cuatro dimensiones discutidas antes es esquematizado en la siguiente tabla:

	Modelo	Sistema modelado
Aspectos estáticos	Reglas OCL de buena formación de metaentidades	parcialmente considerado mediante reglas OCL de buena formación de entidades
Aspectos dinámicos	no considerado	parcialmente considerado mediante lenguaje natural

- El “PUML group” ha construido un modelo semántico preciso del lenguaje UML. El modelo incluye la descripción de la sintaxis abstracta de UML usando el lenguaje formal Z y la definición formal (también usando Z) de una función que interpreta las construcciones sintácticas en un dominio semántico formalmente definido (el cual está descripto en [Rumpe 96]). El tratamiento que esta propuesta da a cada una de las cuatro dimensiones se muestra en la siguiente tabla:

	Modelo	Sistema modelado
Aspectos estáticos	reglas de buena formación del dominio sintáctico expresadas en Z	definición de la estructura del dominio semántico expresado en Z.
Aspectos dinámicos	no considerado	función semántica del dominio sintáctico en el dominio semántico

4.5 Conclusiones

En este capítulo hemos descripto las diferentes propuestas para la integración de lenguajes de modelado gráficos (en particular el lenguaje standard UML) con lenguajes de modelado formales. Hemos destacado que la técnica consistente en definir formalmente la semántica de un lenguaje de modelado conocido y aceptado por la comunidad constituye la propuesta más adecuada, dado que permite que el lenguaje gráfico se convierta en un lenguaje formal y por lo tanto las especificaciones escritas utilizando el lenguaje gráfico pueden ser formalmente analizadas para detectar contradicciones y ambigüedades tempranamente en el proceso de desarrollo del software.

Entre los numerosos trabajos teóricos que tratan diferentes partes de UML definiendo formalmente su semántica hemos seleccionado los más representativos y los hemos descripto y clasificado en dos grupos: formalizaciones basadas en el modelo y formalizaciones basadas en el metamodelo. Las principales ventajas y desventajas de cada propuesta son esquematizadas en la tabla de la figura 4.7. En esta tabla hemos incluido las ventajas y desventajas inherentes al enfoque modelo vs.metamodelo. Sin embargo y tal como puede deducirse de la observación de los cuatro ejemplos presentados en las secciones anteriores, dentro de cada grupo existen propuestas que han superado parcialmente las desventajas de su enfoque; mientras que por otro lado existen propuestas que no han explotado todo su potencial.

Propuestas	Ventajas	Desventajas
basadas en el modelo	▪ Es apropiado para la especificación de información inherente al dominio de la aplicación. ▪ Permite detectar inconsistencias y errores en las especificaciones escritas en el lenguaje de modelado.	▪ No permite expresar reglas de consistencia entre diferentes elementos de modelado, por ejemplo, propiedades acerca de la estructura del sistema, tales como relaciones entre clases. ▪ No permite representar evolución del modelo en un formalismo de primer orden.

Propuestas	Ventajas	Desventajas
basadas en el metamodelo	▪ Permite especificar reglas de consistencia entre diferentes elementos de modelado (por ejemplo entre clases y maquinas de estado). ▪ Permite detectar inconsistencia y errores en la definición del lenguaje de modelado en sí. ▪ Provee un marco formal claro y simple para expresar evolución y refinamiento de modelos.	▪ Es difícil representar y analizar la información correspondiente al dominio de las aplicaciones descriptas con el lenguaje.

Figura 4.7: ventajas y desventajas de cada grupo

En los capítulos siguientes presentaremos una teoría dinámica de primer orden, que integra ambos enfoques (es decir modelo y metamodelo). Esta integración permite representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden.

5. Nuestra propuesta: la M&D-theory

5.1 Introducción

En este capítulo presentaremos una nueva propuesta para definir formalmente la semántica de UML. La idea básica de esta nueva formalización consiste en utilizar un dominio semántico que integra los dos niveles inferiores de la arquitectura de las notaciones de modelado (es decir el nivel del modelo y el nivel de los datos), permitiendo de esta manera representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden, que llamamos M&D-theory.

Dicotomía de entidades

Las entidades descriptas por la M&D-theory se clasifican en dos conjuntos disjuntos:

- Entidades de modelado
- Entidades modeladas

Esta dicotomía puede observarse en la figura 5.1. Las entidades de modelado se corresponden con construcciones (sintácticas) correctas del lenguaje UML, tales como clases (Class) y a máquinas de estados (StateMachine). En contraste, las entidades modeladas, tales como objetos (Object) y conexiones (Link) representan los datos del sistema modelado.

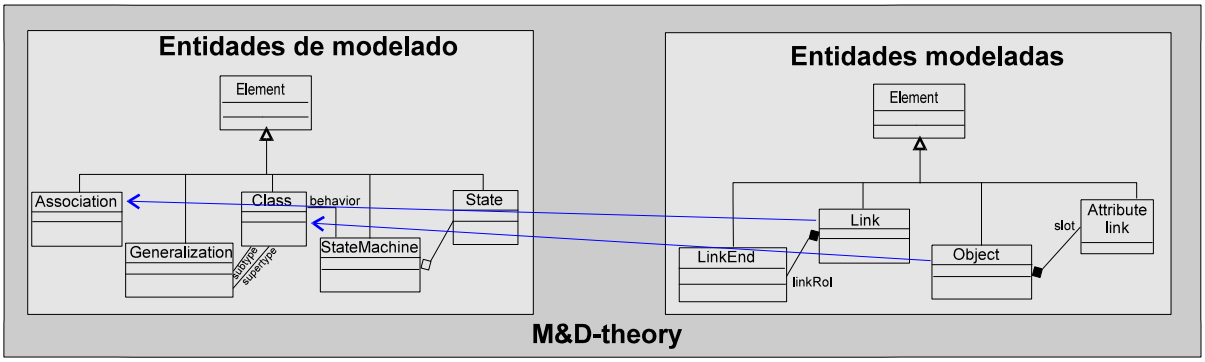


Figura 5.1: dicotomía de entidades

Estas entidades se relacionan de distintas formas:

- *Algunas entidades de modelado se relacionan con otras entidades de modelado*. Por ejemplo las clases (Class) están relacionadas con las máquinas de estado (StateMachine) a través de la relación rotulada con el nombre 'behavior'. Esto indica que las máquinas de estado se utilizan para definir el comportamiento de las instancias de cada clase. Otro ejemplo puede observarse en la relación entre máquina de estados y estados (State), la cual indica que una máquina de estados está compuesta por un conjunto de estados.

- *Algunas entidades modeladas se relacionan con otras entidades modeladas*. Por ejemplo, la relación rotulada con el nombre 'slot' entre Object y AttributeLink, indica que un objeto se relaciona con los valores de sus atributos.

- *Algunas entidades de modelado se relacionan con entidades modeladas*. Existe una relación muy especial entre algunas entidades modeladas y su correspondiente entidad de modelado. Esta relación representa 'instanciación', como por ejemplo los objetos (Object) son instancias de una Clase (Class), mientras que las conexiones (Link) son instancias de una asociación (Association).

Diferentes relaciones de instanciación

La M&D-theory provee dos clases diferentes de relación de instanciación:

- **Instanciación intra-nivel**: es la relación entre una entidad modelada y la entidad principal que la modela. En la figura 5.2 las líneas azules representan algunas instanciaciones intra-nivel. Por ejemplo, Object es instancia de Class, Link es instancia de Association.

- **Instanciación inter-nivel:** es la relación de instanciación provista por el metalenguaje (dynamic logic en este caso). En la figura 5.2 las líneas rojas punteadas representan algunas instanciaciones inter-nivel. Por ejemplo, BankAccount es instancia de Class, holder es instancia de Association, C1 es instancia de Object, S2 es instancia de Object.

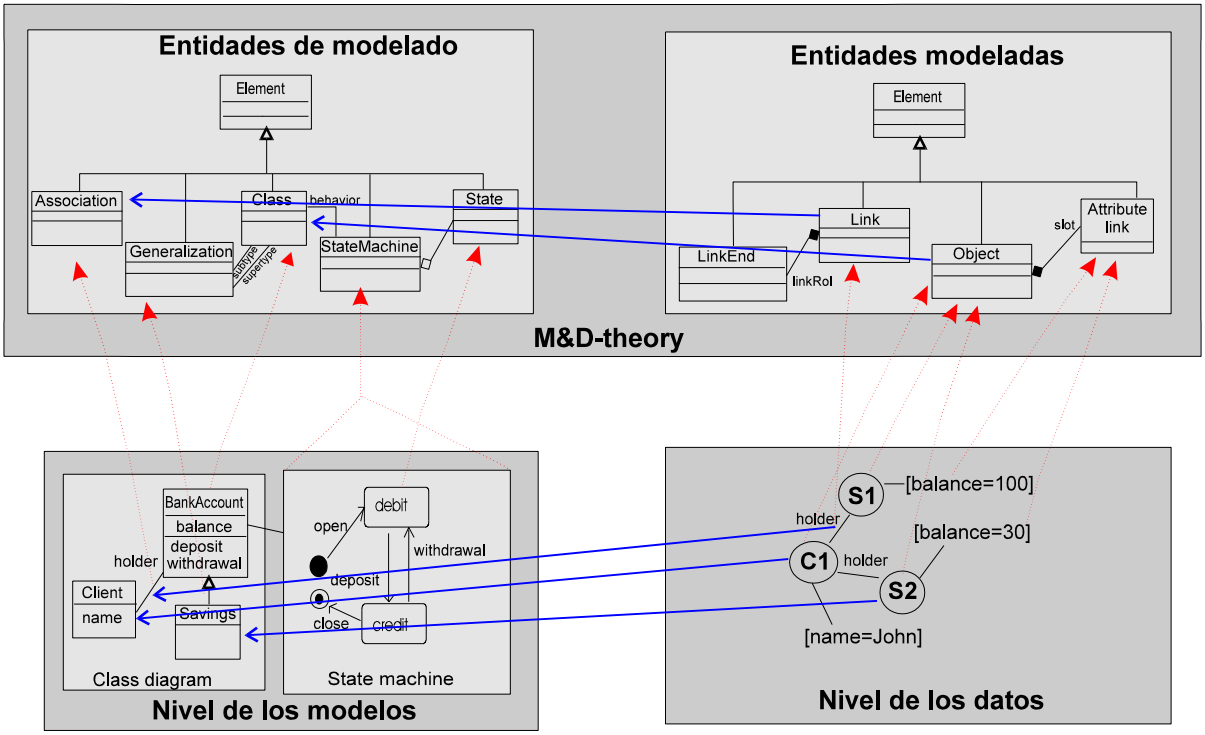
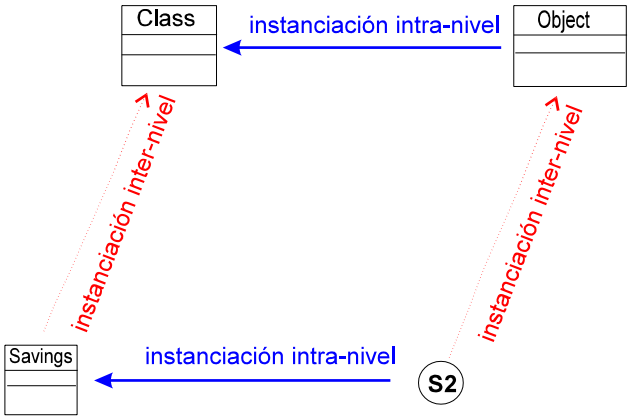


Figura 5.2: distintas relaciones de instanciación

Es interesante destacar que las relaciones de instanciación intra-nivel se preservan a través de la relación de instanciación inter-nivel. Por ejemplo S2 es una instancia (intra-nivel) de Savings, por lo tanto deben existir M1 y M2 tales que S2 es una instancia (inter-nivel) de M2 y Savings es una instancia (inter-nivel) de M1 mientras que M2 es una instancia (intra-nivel) de M1. En el ejemplo de

la figura 5.2 M1 es Class y M2 es Object. Esta propiedad de preservación de la relación de instanciación puede observarse en el siguiente diagrama y su demostración formal se encuentra en [Geisler et al. 98].



Ventajas de la integración

La integración de entidades de modelado y entidades modeladas dentro de la misma teoría permite representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden. la siguiente tabla esquematiza el tratamiento que nuestra propuesta da a cada una de las cuatro dimensiones discutidas en el capítulo anterior:

	Modelo	Sistema modelado
Aspectos estáticos	Axiomas de primer orden sobre entidades de modelado	Axiomas de primer orden sobre entidades modeladas
Aspectos dinámicos	Acciones y axiomas modales sobre entidades de modelado.	Acciones y axiomas modales sobre entidades modeladas

Contar con una estructura formal de primer orden, en contraste con una estructura de orden superior, facilita los procedimientos para calcular la validez de las fórmulas. A pesar de que la lógica de primer orden es no-decidible (y por lo tanto también lo es la lógica dinámica de primer orden), los sistemas de computación satisfacen ciertas propiedades (por ejemplo, se interpretan sobre estructuras aritméticas, el estado de un programa en un momento dado queda determinado por un conjunto finito de valores) las cuales permiten calcular la validez de las fórmulas dinámicas en forma efectiva.

Organización del capítulo

Este capítulo está organizado de la siguiente forma: en la sección 5.2 describimos la formalización (Σ_{UML}, ϕ_{UML}) de las entidades de modelado. En la sección 5.3 presentamos la formalización (Σ_{SYS}, ϕ_{SYS}) de las entidades modeladas. La sección 5.4 contiene consideraciones acerca de la integración de ambos niveles. Luego, en la sección 5.5 describimos la función de interpretación que asocia las construcciones de UML con los elementos en el dominio semántico. Finalmente, en la sección 5.6 presentamos las principales conclusiones.

5.2 Nivel de los Modelos

5.2.1 Introducción

ELEMENTOS

En el lenguaje UML, los diagramas de clases (Class diagrams) modelan los aspectos estructurales del sistema. Estos diagramas incluyen gráficos para representar clases (Class) y relaciones entre ellas, tales como generalizaciones (Generalization), agregaciones (Aggregation) y asociaciones (Association).

Por otro lado, la parte dinámica del sistema es modelada mediante diagramas de colaboración (Collaboration diagrams), los cuales describen el comportamiento de un grupo de instancias en términos de envíos de mensajes y mediante máquinas de estados (States Machines) las cuales muestran dinamismo interno de los objetos en términos de transiciones entre estados.

Las relaciones existentes entre los diferentes diagramas deben ser definidas formalmente con el objetivo de asegurar la consistencia del modelo. Además es necesario especificar las formas en que estos diagramas pueden evolucionar, mostrando el impacto que una modificación realizada sobre un diagrama produce sobre los restantes diagramas que constituyen el modelo del sistema.

EVOLUCIÓN

La especificación de un sistema puede evolucionar por diversas razones, a lo largo de su ciclo de vida. Una de las formas más comunes de evolución es la que involucra cambios estructurales, tales como agregar nuevas clases de objetos, agregar o eliminar atributos de clases existentes, modificar la jerarquía de herencia, etc. En el otro extremo encontramos formas de evolución que no solo modifican la estructura del sistema, sino que modifican el comportamiento especificado para los objetos. Los cambios de comportamiento se reflejan en las modificaciones realizadas sobre diagramas de comportamiento, tales como máquinas de estado.

En la teoría formal, las distintas formas de evolución en el nivel de los modelos serán denominadas ModelEvolutions.

ESTRUCTURA DE LA TEORÍA

La M&D-theory está organizada en paquetes, siguiendo intencionalmente la estructura del metamodelo de UML. Esta división en paquetes (Packages) permite dominar la complejidad de la teoría. La mayor parte de la información contenida en cada paquete puede ser comprendida y analizada independientemente de los otros paquetes. Sin embargo es necesario prestar atención a las relaciones existentes entre los diferentes paquetes. La figura 5.3 muestra los paquetes de mas alto nivel en los que la teoría se encuentra dividida: Foundation, BehavioralElements y ModelManagement.

Este capítulo contiene la descripción de cada uno de los paquetes. La descripción está integrada por las siguientes secciones:

- ***Sintax*is *Abstracta.*** La sintaxis abstracta es representada mediante diagramas UML mostrando las metaclases contenidas en el paquete juntamente con sus relaciones. Estos diagramas muestran además algunas reglas de buena formación, tales como requerimientos de multiplicidad.

- ***Especificación en Lógica Dinámica.*** En esta sección incluimos la especificación formal de las metaclases usando un lenguaje formal basado en *Dynamic Logic* (DL). Esta especificación consta de una signatura $\sum_{UML}$**= ((S_{UML},≤), F_{UML}, P_{UML}, A_{UML})** y una fórmula ϕ_{UML} sobre Σ_{UML}. Los elementos del álgebra inicial denotada por la especificación son elementos de modelado, tales como clases, asociaciones y máquinas de estados. La relación de transición entre posibles mundos representa modificaciones sobre la especificación del sistema, por ejemplo el agregado de una nueva clase, la modificación de una clase existente, etc. La fórmula ϕ_{UML} es la conjunción de dos conjuntos disjuntos de fórmulas, ϕ_S y ϕ_D de fórmulas estáticas y fórmulas dinámicas respectivamente. El primer conjunto consiste en fórmulas no modales que deben satisfacerse en todos los estados posibles del sistema (son invariantes o propiedades estáticas o reglas de buena formación de los modelos). Estas reglas se usan para realizar análisis de la estructura del sistema y reportar posibles errores en su diseño. Mientras que el segundo conjunto consiste en fórmulas modales que definen la semántica de las acciones, es decir de la evolución de los modelos.
- ***Descripción Informal.*** Finalmente, se incluye una breve descripción informal, utilizando lenguaje natural. Para cada metaclase enumeramos sus atributos y asociaciones conjuntamente con una breve explicación de su significado.

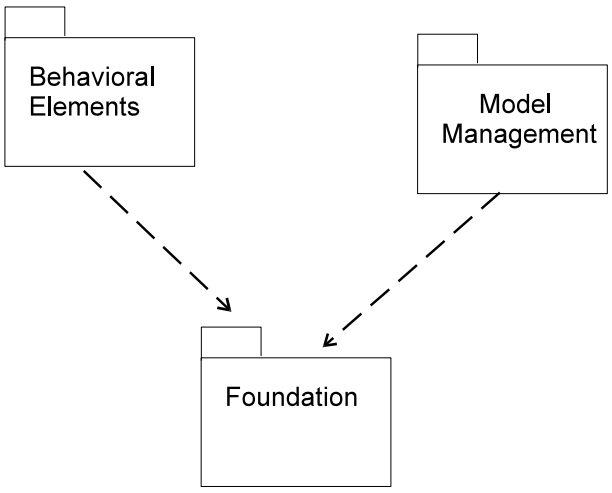


Figura 5.3: Paquetes de más alto nivel

5.2.2 Paquete FOUNDATION

El paquete Foundation define la infraestructura de UML. Se descompone en dos subpaquetes: Core y UML_Data Types (ver figura 5.4). El paquete UML_Data Types define los tipos de datos básicos de la teoría, tales como Booleans, Integer, etc. El paquete Core especifica los principales elementos para modelar la estructura de un sistema, tales como clases y asociaciones.

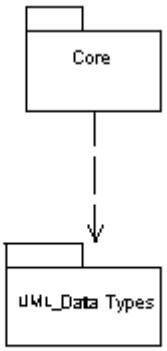


Figura 5.4: **Foundation Packages**

5.2.2.1 Paquete FOUNDATION: UML_DATA TYPES

El paquete UML_Data Types especifica los diferentes tipos de datos básicos utilizados por UML. Las siguientes secciones describen al paquete UML_DataTypes.

Sintaxis Abstracta

La figura 5.5 muestra la descripción gráfica de la sintaxis abstracta de los elementos que componen el paquete Data Types.

Es importante destacar la diferencia entre estos tipos de datos básicos y los tipos de datos utilizados por los desarrolladores de sistemas. Estos tipos en el paquete UML_DataType son necesarios para definir el lenguaje en si, mientras que los desarrolladores (utilizando el lenguaje) pueden crear otros tipos de datos, instanciando la metaclase Data Type.

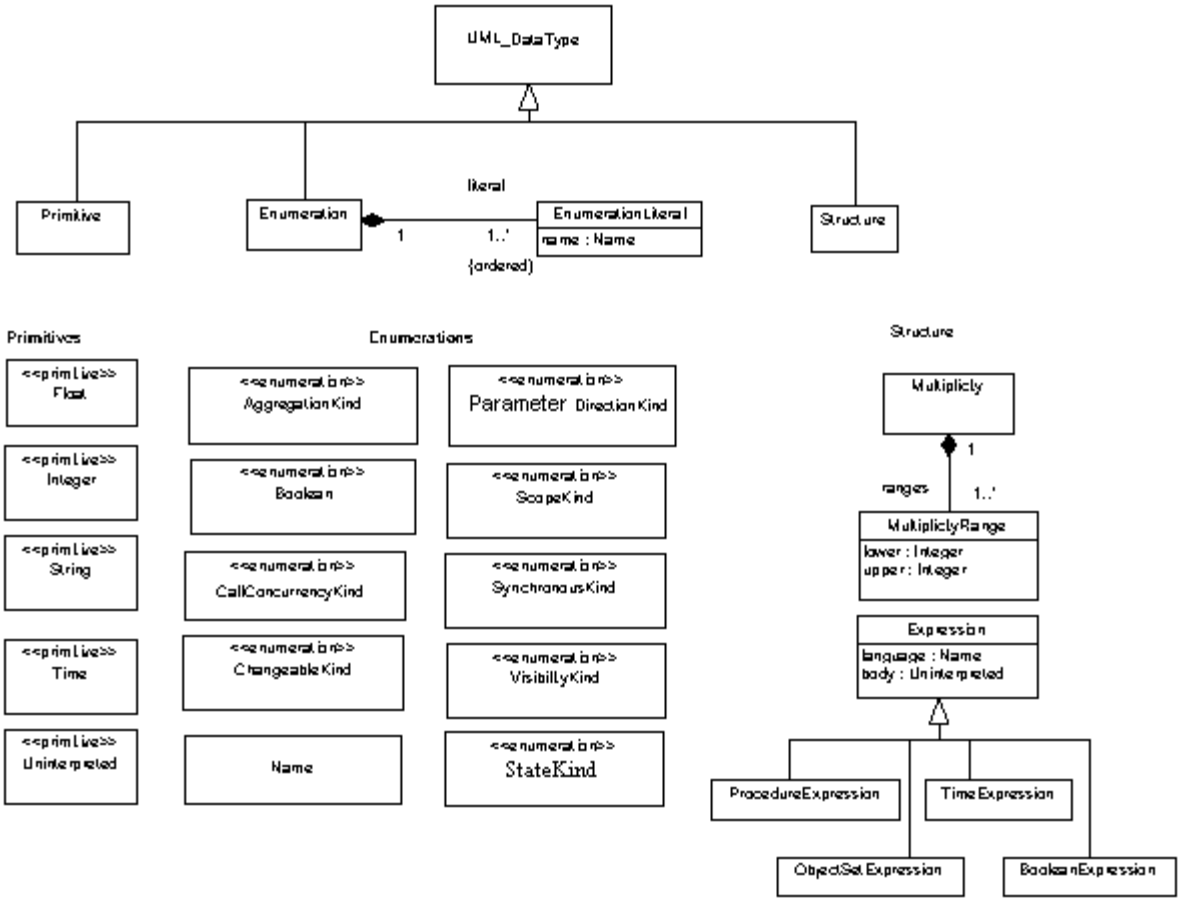


Figura 5.5: **UML-Data Types**

Especificación en Lógica Dinámica

Specification of UML_DataType	
Sorts	UML_DataType, Primitive, Enumeration, Structure
Taxonomy	Primitive≤UML_DataType Enumeration≤UML_DataType Structure≤UML_DataType
End specification of UML_DataType	

Specification of Primitive UML_DataTypes:

Specification of Boolean	
Sorts	Boolean
Taxonomy	Boolean ≤Primitive
NonUpdatable functions	
	true: → Boolean false: → Boolean ∧: Boolean x Boolean →Boolean ¬: Boolean →Boolean
Axioms	∀b,c: Boolean
Static axioms	
	¬true=false b∧true=b ¬¬b=b b∧c=c∧b
End specification of Boolean	

Specification of Enumeration UML_DataTypes:

Specification of AggregationKind	
Sorts	AggregationKind
Taxonomy	AggregationKind ≤Enumeration
NonUpdatable functions	
	none: → AggregationKind shared: → AggregationKind composite: → AggregationKind
End specification of AggregationKind	

.

Specification of CallConcurrencyKind	
Sorts	CallConcurrencyKind
Taxonomy	CallConcurrencyKind ≤Enumeration
NonUpdatable functions	
	sequential: → CallConcurrencyKind concurrent: → CallConcurrencyKind
End specification of CallConcurrencyKind	

Specification of ChangeableKind	
Sorts	ChangeableKind
Taxonomy	ChangeableKind ≤Enumeration
NonUpdatable functions	
	none: → ChangeableKind frozen: → ChangeableKind addOnly: → ChangeableKind
End specification of ChangeableKind	

Specification of ParameterDirectionKind	
Sorts	ParameterDirectionKind
Taxonomy	ParameterDirectionKind ≤Enumeration
NonUpdatable functions	
	in: → ParameterDirectionKind out: → ParameterDirectionKind inout: → ParameterDirectionKind returns: → ParameterDirectionKind
End specification of ParameterDirectionKind	

Specification of ScopeKind	
Sorts	ScopeKind
Taxonomy	ScopeKind ≤Enumeration
NonUpdatable functions	
	classifier: → ScopeKind instance: → ScopeKind
End specification of ScopeKind	

Specification of StateKind	
Sorts	StateKind
Taxonomy	StateKind ≤Enumeration
NonUpdatable functions	
	initial: → StateKind final: → StateKind normal: → StateKind
End specification of ScopeKind	

Specification of SynchronousKind	
Sorts	SynchronousKind
Taxonomy	SynchronousKind ≤Enumeration
NonUpdatable functions	
	synchronous: → SynchronousKind asynchronous: → SynchronousKind
End specification of SynchronousKind	

Specification of VisibilityKind	
Sorts	VisibilityKind

Taxonomy	VisibilityKind ≤ Enumeration	
NonUpdatable functions		
	public: → VisibilityKind protected: → VisibilityKind private: → VisibilityKind	
End specification of VisibilityKind		

Specification of Structure UML_DataTypes:

Specification of Multiplicity		
Sorts	Multiplicity	
Taxonomy	Multiplicity ≤ Structure	
NonUpdatable functions		
	ranges: Multiplicity → Set of MultiplicityRange	
End specification of Multiplicity		

Specification of MultiplicityRange		
Sorts	MultiplicityRange	
Taxonomy	MultiplicityRange ≤ Structure	
NonUpdatable functions		
	lower: MultiplicityRange → Integer upper: MultiplicityRange → Integer	
Axioms	∀m: MultiplicityRange	
	lower(m) ≤ upper(m)	
End specification of MultiplicityRange		

Descripción Informal

La especificación del paquete UML_ DATA TYPES contiene los siguientes elementos. Cada tipo diferente de elemento es representado mediante un sort en la teoría order-sorted.

UML_Data Type

Esta parte del metamodelo especifica los tipos de datos que se necesitan para definir UML, tales como tipos de datos primitivos (por ejemplo Integer y String), tipos de datos enumerativos (por ejemplo AggregationKind, VisibilityKind) y tipos de datos estructurados (por ejemplo Multiplicity).

AggregationKind

El tipo AggregationKind define una enumeración cuyos valores son ***none***, ***shared***, y ***composite***. Sus valores denotan el tipo de agregación representado por una Asociación.

Boolean

Booelean es el tipo de dato primitivo cuyos valores son ***false*** y ***true***. Soporta las operaciones booleanas clásicas $\wedge$ y $\neg$.

BooleanExpression

El sort BooleanExpression define una expresión que al ser evaluada retorna una instancia de Boolean. El predicado *consistent* definido sobre un conjunto de expresiones booleanas es verdadero cuando las expresiones son consistentes, es decir existe alguna valuación que satisface a cada elemento en el conjunto.

CallConcurrencyKind

El sort CallConcurrencyKind define una enumeración cuyos valores son:

sequential indicando que las invocaciones a la operación deben ser secuenciales.

concurrent indicando que múltiples invocaciones de la operación pueden ocurrir simultáneamente.

ChangeableKind

El sort ChangeableKind define una enumeración cuyos valores son ***none***, ***frozen***, y ***addOnly***. Sus valores denotan la forma en que un atributo o asociación pueden ser modificados.

Enumeration

El sort Enumeration define un tipo de dato cuyo rango es una lista de valores llamados literales o EnumerationLiterals.

EnumerationLiteral

EnumerationLiteral define un átomo que puede ser comparado por igualdad.

Expression

El sort Expression define una expresión que puede evaluarse en un contexto.

La función referencedElements(e) retorna el conjunto de elementos(por ejemplo atributos, asociaciones) que están involucrados en la expresión.

El predicado syntactic-consistent definido sobre una clase y una expresión es verdadero cuando los elementos involucrados en la expresión se corresponden con los atributos definidos en la clase (o expresiones de camino válidas).

Integer

El sort Integer representa al tipo de dato primitivo cuyos valores están en el rango (…-1, 0, 1, 2…).

Multiplicity

El sort Multiplicity define un conjunto no vacío de rangos de multiplicidad.

MultiplicityRange

El sort MultiplicityRange define un rango de enteros. El límite superior del rango no puede ser menor que el límite inferior.

Name

El sort Name define un átomo que es usado para nombrar Elementos. Cada nombre tiene una representación como String.

ObjectSetExpression

El sort ObjectSetExpression define una expresión que al ser evaluada retorna un objeto.

ParameterDirectionKind

El sort ParameterDirectionKind define una enumeración cuyos valores son ***in, inout, out***, y ***return***. Sus valores indican si un parámetro es usado para la entrada y/o salida de datos o para retornar un valor.

Primitive

Representa a los tipos de datos primitivos, tales como Integer o Boolean.

ProcedureExpression

El sort ProcedureExpression define una expresión que al ser evaluada retorna un Procedimiento.

ScopeKind

El sort ScopeKind define una enumeración cuyos valores son ***instance*** y ***classifier***. Sus valores indican si un feature está definido para las instancias de una clase o para la clase en sí.

StateKind

El sort StateKind define una enumeración cuyos valores son ***initial***, ***final***, y ***normal***. Sus valores indican el tipo de un estado en una StateMachine.

String

Un String es una secuencia de caracteres.

Structure

A Structure defines una clase especial de tipo de dato que tiene una cantidad fija de partes.

SynchronousKind

El sort SynchronousKind define una enumeración cuyos valores son ***synchronous*** and ***asynchronous***.

Time

El sort Time define un valor que representa un momento en el tiempo.

TimeExpression

El sort TimeExpression define una expresión que al ser evaluada retorna una instancia del sort Time.

Uninterpreted

Este sort se utiliza para indicar elementos que no tienen interpretación dentro de UML.

VisibilityKind

El sort VisibilityKind define una enumeración cuyos valores son ***public***, ***protected***, y ***private***. Sus valores indican como es visto un elemento desde afuera de su contexto.

5.2.2.2 Paquete FOUNDATION: CORE

El paquete Core define las estructuras básicas necesarias para la construcción de modelos de objetos. Las metaclases concretas contenidas en este paquete, tales como Class, Attribute, Operation y Association son instanciables y reflejan elementos de modelado usados por los desarrolladores de modelos. Por otra parte, en este paquete existen otras metaclases que no son concretas (son abstractas). Las clases abstractas como por ejemplo, ModelElement, GeneralizableElement y Classifier, no son instanciables y se utilizan para organizar el metamodelo, compartir conceptos y estructuras.

El Core es extendido en otros paquetes mediante el agregado de nuevas clases las cuales son relacionadas con las clases ya existentes mediante los mecanismos de generalización y asociación.

Las siguientes secciones describen al paquete Core.

Sintaxis Abstracta

Las figuras 5.6 y 5.7 muestran la sintaxis abstracta, expresada en notación gráfica, de los elementos que componen el paquete Core. La figura 5.6 muestra los principales elementos que se usan para modelar la estructura de un sistema, mientras que la figura 5.7 muestra los elementos que permiten representar relaciones.

Figura 5.6: Structural Backbone

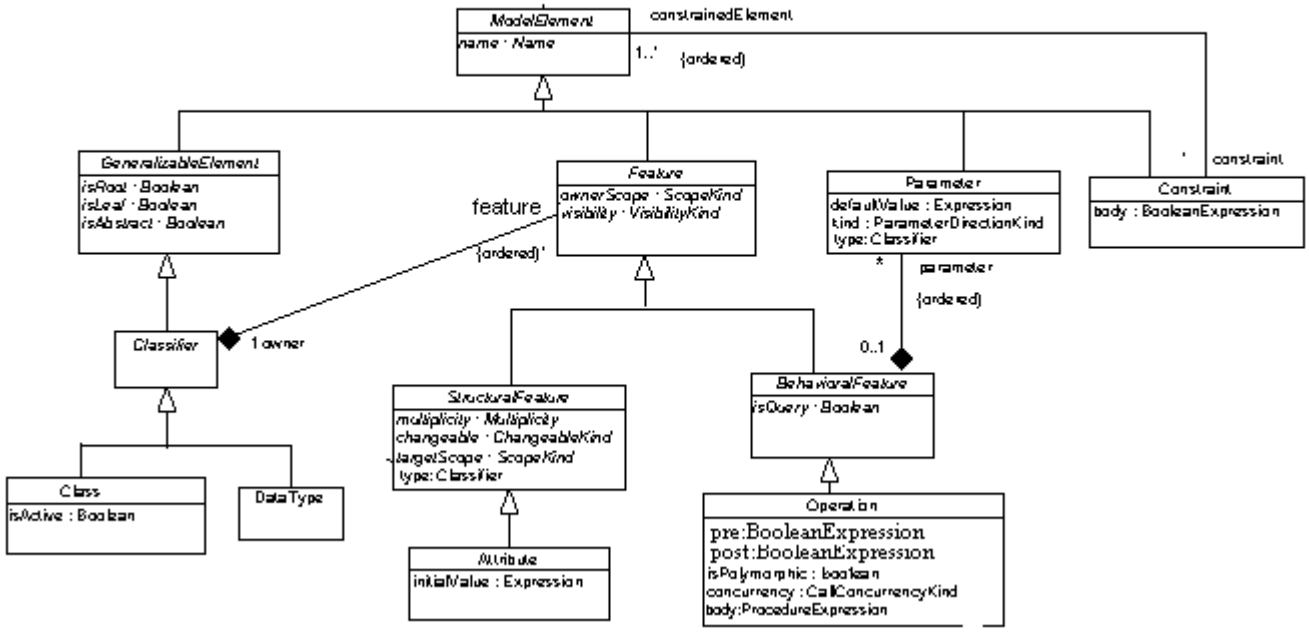
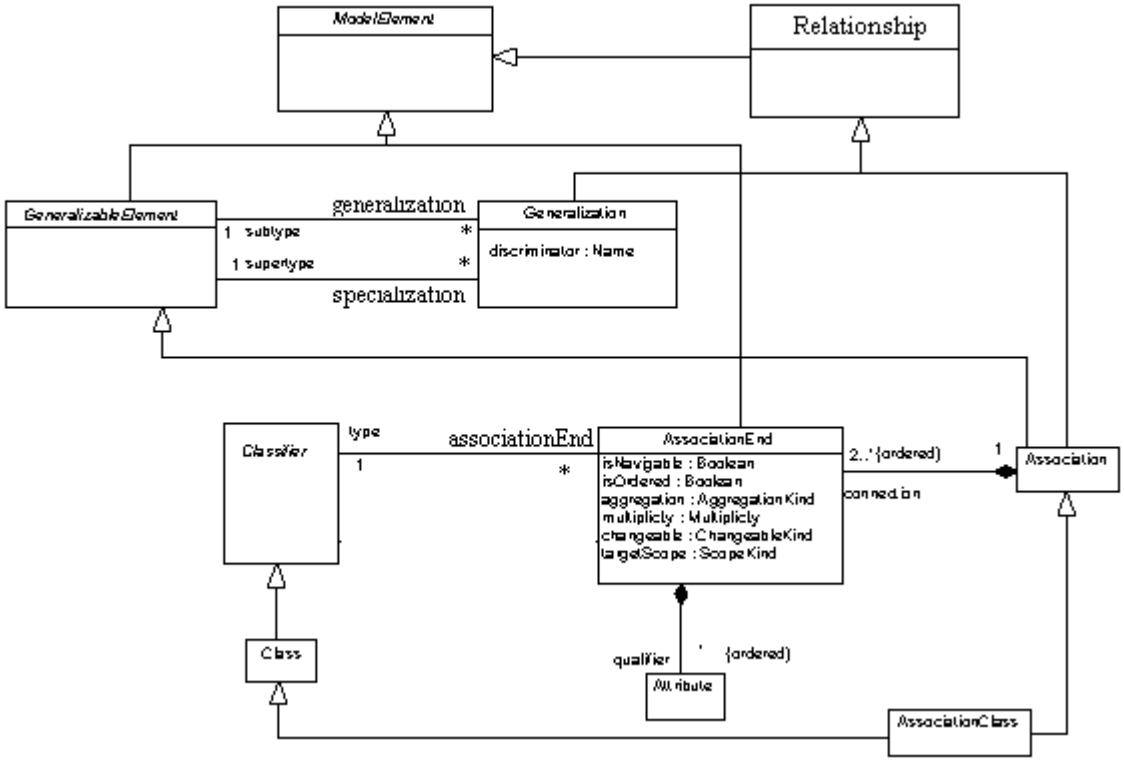


Figura 5.7 : Relationships



Especificación en Lógica Dinámica

Specification of Association	
Sorts	Association
Taxonomy	Association ≤GeneralizableElement Association ≤ Relationship
Updatable functions	
	connections: Association → Seq of AssociationEnd *Additional functions* allConnections:Association → Seq of AssociationEnd
Updatable predicates	
Actions	
	addConnection: Association x AssociationEnd →ModelEvolution deleteConnection: Association x AssociationEnd →ModelEvolution
Axioms	∀a: Association ∀e,e1,e2:AssociationEnd
Static axioms	
	axioms for additional functions: [1] allConnections returns the set of all AssociationEnds of the Association itself and all its inherited AssociationEnds. allConnections(a) = connections(a) ∪ (∪s∈supertypes(a) allConnections(s)) [2]connectedElements (a) returns a sequence containing all the classifiers connected by the association.

	connectedElements(a) = map type connections(a) [3] allConnectedElements (a) returns a sequence containing all the classifiers connected by the association itself and by its supertypes. allConnectedElements(a) = map type allConnections(a) *well-formedness axioms:* [1] The AssociationEnds must have a unique name within the Association. $\forall$e1,e2$\in$allConnections(a) name(e1)=name(e2) $\rightarrow$ e1=e2 [2] At most one AssociationEnd may be an aggregation or composition. $\forall$e1,e2$\in$allConnections(a) aggregation(e1)<>#none$\wedge$aggregation(e2)<>#none$\rightarrow$ e1=e2 [3] If an Association has 3 or more AssociationEnds then no AssociationEnd may be an aggregation or composition. size(allConnections(a))>2$\rightarrow$ $\forall$e$\in$allConnections(a)aggregation(e)=#none [4] An Association must have 2 or more AssociationEnds size(allConnections(a))>1 [5] life dependency Exists(a) $\leftrightarrow$($\forall$e$\in$connections(a)Exists(e)) [6] symmetry e$\in$connections(a)) $\leftrightarrow$ association(e)=a
Dynamic axioms	
	$\langle$addConnection(a,e)$\rangle$*true* $\rightarrow$e$\notin$allConnections(a) [addConnection(a,e)] Exists(e) $\wedge$ e=last(connections(a)) $\wedge$association(e)=a $\wedge$ e$\in$associationEnds(type(e)) $\langle$deleteConnection(a,e)$\rangle$*true*$\rightarrow$e$\in$connections(a) [deleteConnection(a,e)] $\neg$Exists(e)$\wedge$e$\notin$connections(a))$\wedge$e$\notin$associationEnds(type(e))
End specification of Association	

Specification of AssociationClass		
Sorts	AssociationClass	
Taxonomy	AssociationClass $\leq$ Association AssociationClass $\leq$Class	
Updatable functions		
Updatable predicates		
Actions		
Axioms	$\forall$a: AssociationClass	
Static axioms		
	[1] The names of the AssociationEnds and the StructuralFeatures do not overlap. $\forall$e1,e2$\in$allConnections(a)$\cup$allAttributes(a) name(e1)=name(e2) $\rightarrow$ e1=e2 [2] An AssociationClass cannot be defined between itself and something else. $\neg$a$\in$allConnectedElements(a)	
Dynamic axioms		
End specification of AssociationClass		

Specification of AssociationEnd	
Sorts	AssociationEnd
Taxonomy	AssociationEnd ≤ ModelElement
Updatable functions	
	aggregation: AssociationEnd → AggregationKind changeable: AssociationEnd → ChangeableKind multiplicity: AssociationEnd → Multiplicity targetScope: AssociationEnd → ScopeKind qualifier: AssociationEnd →Seq of Attribute type: AssociationEnd → Class association: AssociationEnd → Association
Updatable predicates	
	isOrdered: AssociationEnd isNavigable: AssociationEnd
Actions	
	setAggregation: AssociationEnd x AggregationKind → ModelEvolution setChangeable: AssociationEnd x ChangeableKind → ModelEvolution setMultiplicity: AssociationEnd x Multiplicity → ModelEvolution setTargetScope: AssociationEnd x ScopeKind → ModelEvolution setQualifier: AssociationEnd xSeq of Attribute → ModelEvolution setOrdered: AssociationEnd → ModelEvolution setNavigable: AssociationEnd → ModelEvolution
Axioms	∀e: AssociationEnd ∀c:Classifier
Static axioms	
Dynamic axioms	
	[setAggregation(a,g)] aggregation(a)=g [setChangeable(a,h)] changeable(a)=h [setMultiplicity(a,m)] multiplicity(a)=m [setTargetScope(a,s)] targetScope(a)=s [setQualifier(a,q)]qualifier(a)=q [setOrdered (a)]isOrdered(a) [setNavigable(a)]isNavigable(a)
End specification of AssociationEnd	

Specification of Attribute	
Sorts	Attribute
Taxonomy	Attribute ≤ StructuralFeature
Updatable functions	
	initialValue: Attribute→ Expression
Updatable predicates	
Actions	
	setInitialValue: Attribute x Expression → ModelEvolution
Axioms	∀a: Attribute ∀e:Expression
Static axioms	
Dynamic axioms	

	[setInitialValue(a,e)] initialValue(a)=e
End specification of Attribute	

Specification of BehavioralFeature		
Sorts	BehavioralFeature	
Taxonomy	BehavioralFeature ≤ Feature	
Updatable functions		
	parameters: BehavioralFeature →Seq of Parameter	
Updatable predicates		
	isQuery: BehavioralFeature *additional predicates* hasSameSignature: BehavioralFeature x BehavioralFeature	
Actions		
	addParameter: BehavioralFeature x Parameter → ModelEvolution deleteParameter: BehavioralFeature x Parameter → ModelEvolution	
Axioms	∀b: BehavioralFeature ∀ p,p1, p2:Parameter	
Static axioms		
	axioms for additional functions and predicates [1] The additional predicate hasSameSignature checks if two behavioralFeatures have the same signature. hasSameSignature(b,b') ↔ (name(b)=name(b') ∧ areEquivalent(parameters(b), parameters(b'))) areEquivalent(∅,∅)=true areEquivalent(p-ps,∅)=false areEquivalent(∅,p-ps)=false areEquivalent(p1-ps,p2-ps´)=equivalent(p1,p2)∧areEquivalent(ps,ps´) *well-formedness axioms* [1] All Parameters should have a unique name. ∀p1,p2∈parameters(b) name(p1)=name(p2) → p1= p2 [2] The type of the Parameters should be included in the Package of the Classifier. ∀p∈parameters(b) type(p)∈allContents(package(owner(b))) [7] life dependency Exists(b) ↔(∀p∈parameters(b)Exists(p))	
Dynamic axioms		
	⟨addParameter(b,p)⟩*true* → p∉parameters(b) [addParameter (b,p)] Exists(p) ∧p∈ parameters(b) ⟨deleteParameter(b,p)⟩*true*→p∈parameters(b) [deleteParameter(b,p)] ¬Exists(p) ∧ p∉parameters(b)	
End specification of BehavioralFeature		

Specification of Class		
Sorts	Class	
Taxonomy	Class ≤Classifier	
Updatable functions		
	behavior: Class →StateMachine	
Updatable predicates		
	isActive: Class	
Actions		
	activate: Class→ ModelEvolution	

	desactivate: Class→ ModelEvolution
Axioms	∀c:Class
Static axioms	
	[1] If a Class is concrete, all the Operations in the full descriptor of the Class should have a realizing Method. ¬isAbstract(c) →∀p∈allOperations(c) body(p)≠nullElement [2] symmetry context(behavior(c))=c
Dynamic axioms	
	[activate(c)] isActive(c) [desactivate(c)] ¬isActive(c)
End specification of Class	

Specification of Classifier	
Sorts	Classifier
Taxonomy	Classifier≤GeneralizableElement
Updatable functions	
	features: Classifier→Seq of Feature associationEnds: Classifier→Set of AssociationEnd *Additional functions* associations: Classifier→Set of Association oppositeAssociationEnds: Classifier→Set of AssociationEnd allFeatures : Classifier→Set of Feature allOperations : Classifier→Set of Operation allAttributes : Classifier→Set of Attribute allAssociationEnds: Classifier→Set of AssociationEnd allAssociations: Classifier→Set of Association allOppositeAssociationEnds: Classifier→Set of AssociationEnd
Updatable predicates	
	additional predicates *DirectPartOf*: Classifier x Classifier *PartOf*: Classifier x Classifier
Actions	
	addFeature: Classifier x Feature → ModelEvolution deleteFeature: Classifier x Feature → ModelEvolution
Axioms	∀c:Classifier ∀f,f1,f2:Feature ∀e:AssociationEnd
Static axioms	
	axioms for additional functions [1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features. allFeatures(c)= features(c) ∪ ($\cup_{ci \in supertypes(c)}$ allFeatures(ci)) [2] The operation allOperations results in a Set containing all Operations of the Classifier itself and all its inherited Operations. ∀p:Operation o∈allOperations(c) ↔ o∈allFeatures(c) [3] The operation allAttributes results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes. ∀t:Attribute t∈allAttributes(c) ↔ t∈allFeatures(c) [4] The operation allAssociationEnds results in a Set containing all AssociationEnds of the Classifier itself and all its inherited AssociationEnds. allAssociationEnds(c)= associationEnds(c)∪($\cup_{ci \in supertypes(c)}$ allAssociationEnds(ci)) [5] allAssociations returns the associations in which the classifier participates.

	allAssociations(c)= map association allAssociationEnds(c) [6] oppositeAssociationEnds returns the opposite AssociationEnds of the Classifier itself. oppositeAssociationEnds(c) = $\{e \in (\cup_{a \in associations(c)}$ allConnections(a)) / type(e)$\neq$c$\}$ [7] allOppositeAssociationEnds returns the opposite AssociationEnds of the Classifier itself and all its inherited opposite AssociationEnds. allOppositeAssociationEnds(c)=oppositeAssociationEnds(c) $\cup$ $(\cup_{ci \in supertypes(c)}$ allOppositeAssociationEnds(ci)) *axioms for additional predicates* [1] the *DirectPartOf* predicate: The set of Association with associationsEnds that are aggregation or composition defines a relation which we will call *DirectPartOft*. *DirectPartOf*(c1,c2) means that instances of c1 are part of instances of c2. It is true when there exists an Association (aggregation or composition) connecting c1 with c2 . *DirectPartOf*(c1,c2) $\leftrightarrow$ $\exists$e:AssociationEnd(e$\in$allOppositeAssociationEnds(c1)$\wedge$aggregation(e)$\neq$#none$\wedge$ type(e)=c2) [2] the *PartOf* predicate: *PartOf* is the transitive closure of *DirectPartOf*. *PartOf=DirectPartOf** *Well-formedness axioms* [1]No Attributes may have the same name within a Classifier $\forall$f,g$\in$attributes(c) (name(f) = name(g) $\rightarrow$ f=g) [2] No Operations may have the same signature in a Classifier. $\forall$f,g$\in$operations(c) (hasSameSignature(f,g) $\rightarrow$f = g) [3]No opposite AssociationEnds may have the same rol-name within a Classifier $\forall$f,g$\in$oppositeAssociationEnds(c) (name(f) = name(g) $\rightarrow$ f=g) [4] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd. $\forall$f$\in$oppositeAssociationEnds(c) $\forall$g$\in$allAttributes(c) name(f) $\neq$ name(g) [5] life dependency Exists(c) $\leftrightarrow$ $\forall$f$\in$features(c)Exists(f) [6] symmetry f$\in$features(c) $\leftrightarrow$owner(f)=c e$\in$associationEnds(c) $\leftrightarrow$type(e)=c
Dynamic axioms	
	$\langle$addFeature(c,f)$\rangle$*true* $\rightarrow$f$\notin$allFeatures(c) [addFeature(c,f)] Exists(f) $\wedge$ f$\in$features(c) $\wedge$ owner(f)=c $\langle$deleteFeature(c,f)$\rangle$*true*$\rightarrow$ f$\in$features(c) [deleteFeature(c,f)] $\neg$Exists(f)$\wedge$f$\notin$features(c)
End specification of Classiffier	

Specification of Constraint		
Sorts	Constraint	
Taxonomy	Constraint $\leq$ModelElement	
Updatable functions		
	body: Constraint$\rightarrow$BooleanExpression constrainedElements: Constraint$\rightarrow$Seq of ModelElement *additional functions* eval: Constraint$\rightarrow$Boolean	

	referencedElements: Constraint →Set of ModelElements
Updatable predicates	
	consistent: Set of Constraint syntactic-compatible: Class x Constraint
Actions	
	setBody: Constraint x BooleanExpression → ModelEvolution
Axioms	∀c,c1,c2: Constraint ∀b:BooleanExpression ∀S:Set of Constraint
Static axioms	
	axioms for additional functions [1]referencedElements(c)=referencedElements(body(c)) [2] eval(c) =eval(body(c)) [3] consistent(S) ↔consistent(⦃body(s) | s∈S ⦄) [4] ∀l:Classifier syntactic-compatible(l,c) ↔syntactic-compatible(l,body(c)) *well-formedness axioms* [1] A Constraint cannot be applied to itself. ¬ c∈constrainedElements(c)
Dynamic axioms	
	[setBody(c,b)] body(c)=b
End specification of Constraint	

Specification of DataType	
Sorts	DataType
Taxonomy	DataType ≤Classifier
Axioms	∀d:DataType
Static axioms	
	[1] A DataType can only contain Operations. size(allAttributes(d))=0 [2]All Operations must be queries. ∀p∈allOperations(d) isQuery(p)
Dynamic axioms	
End specification of DataType	

Specification of Element and NullElement	
Sorts	Element, NullElement
Taxonomy	
NonUpdatable functions	
	nullElement: → NullElement
End specification of Element and NullElement	

Specification of Feature	
Sorts	Feature
Taxonomy	Feature ≤ModelElement
Updatable functions	
	owner: Feature→Classifier ownerScope: Feature→ScopeKind visibility: Feature→VisibilityKind
Updatable predicates	

Actions	
	setOwnerScope: Feature x ScopeKind→ ModelEvolution setVisibility: Feature x VisibilityKind → ModelEvolution
Axioms	∀f: Feature ∀c:Classifier ∀s:ScopeKind ∀v:VisibilityKind
Static axioms	
Dynamic axioms	
	[setOwnerScope(f,s)] ownerScope(f)=s [setVisibility(f,v)] visibility(f)=v
End specification of Feature	

Specification of GeneralizableElement	
Sorts	GeneralizableElement
Taxonomy	GeneralizableElement ≤ModelElement
Updatable functions	
	generalizations: GeneralizableElement →Set of Generalization specializations: GeneralizableElement →Set of Generalization *additional functions* supertypes: GeneralizableElement →Set of GeneralizableElement subtypes: GeneralizableElement →Set of GeneralizableElement allSupertypes : GeneralizableElement → Set of GeneralizableElement allConstraints: GeneralizableElement →Set of Constraint
Updatable predicates	
	isAbstract: GeneralizableElement isLeaf: GeneralizableElement isRoot: GeneralizableElement *IsA* : GeneralizableElement x GeneralizableElement
Actions	
Axioms	∀ c_1,c_2 ,c: GeneralizableElement
Static axioms	
	axioms for additional functions [1] supertypes(c) returns the set of direct supertypes of c. supertypes(c) = map supertype generalizations(c) [2]subtypes(c) returns the set of direct subtypes of c. subtypes(c) = map subtype specializations(c) [3] allSupertypes(c) returns the set of (direct and indirect) supertypes of c (i.e the transitive closure of supertypes) allSupertypes(c)=supertypes(c) ∪ ($\cup_{ci \in supertypes(c)}$ allSupertypes(ci)) [4]allConstraints returns the constraints of the model element itself and all its inherited constraints allConstraints(c)=constraints(c) ∪ ($\cup_{ci \in supertypes(c)}$ allConstraints(ci)) [5] Definition of IsA predicate IsA(c,c1) ↔ c=c1 ∨ c1∈allSupertypes(c) *well-formedness axioms* [1] Circular inheritance is not allowed. $IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$ [2] multiple inheritance does not lead to name conflict (IsA(c1,c2) ∧IsA(c1,c3)∧c3≠c2) →

	∀f:Feature (f∈allFeatures(c2)∧f∈allFeatures(c3) → ∃c:Classifier(IsA(c2,c) ∧ IsA(c3,c) ∧f∈features(c))) [3] Cyclic situations: if aggregation is recursive in a superclass, it must have base case. This means that some subclass of c1 simpler than c2 must exist (*PartOf*(c_1,c_2) → ¬*IsA*(c_1,c_2)) ∧ ((*PartOf*(c_1,c_2) ∧*IsA*(c_2,c_1)) → ∃c_3:Classifier(*IsA*(c_3,c_1) ∧ c_3≠c_2 ∧ ¬*PartOf*(c_2,c_3))) [4] Constraint consistency *IsA*(c_1,c_2) → consistent(constraints(c_1) ∪constraints(c_2)) [5] Behavioral consistency *IsA*(c_1,c_2) → refinement(behavior(c_1) , behavior(c_2))
Dynamic axioms	
End specification of GeneralizableElement	

Specification of Generalization	
Sorts	Generalization
Taxonomy	Generalization ≤Relationship
Updatable functions	
	discriminator: Generalization →Name supertype: Generalization →GeneralizableElement subtype : Generalization →GeneralizableElement
Updatable predicates	
Actions	
Axioms	∀g: Generalization
Static axioms	
	axioms for additional functions [1] connectedElements(g) returns a sequence containing the two GeneralizableElements connected by the generalization. connectedElements(g)={supertype(g), subtype(g)} [2] allConnectedElements (a) returns the same result as function connectedElements because Generalizations are not GeneralizableElements (they do not participate in inheritance hierarchies). allConnectedElements(g)= connectedElements(g) *well-formedness axioms* [1] A root cannot have any Generalizations. ¬isRoot(subtype(g)) [2] No GeneralizableElement which is a leaf can have a subtype ¬isLeaf(supertype(g)) [3] A GeneralizableElement may only be a subclass of GeneralizableElement of the same kind. SameKind(subtype(g) , supertype(g))
Dynamic axioms	
End specification of Generalization	

Specification of ModelElement	
Sorts	ModelElement
Taxonomy	ModelElement≤Element
Updatable functions	

	name: ModelElement→Name constraints: ModelElement→Set of Constraint package: ModelElement→ (Package+NullElement) *Additional functions* packages : ModelElement →Set of Package
Updatable predicates	
	sameKind: ModelElement x ModelElement
Actions	
	setName: ModelElement x Name →ModelEvolution addConstraint: ModelElement x Constraint → ModelEvolution deleteConstraint: ModelElement x Constraint → ModelEvolution
Axioms	∀m: ModelElement ∀c:Constraint ∀p:Package
Static axioms	
	axioms for additional functions [1] The operation packages results in all the Packages to which a ModelElement belongs. packages = {package(m)} ∪ allSurroundingPackages(package(m)) *well-formedness axioms* [1] Constraint consistency consistent(allConstraints(m)) [2] ∀c∈allConstraints(m) syntactic-compatible(m,c)
Dynamic axioms	
	[setName(m,n)] name(m)=n ⟨addConstraint(m,c)⟩*true* → consistent({c} ∪ allConstraints(m)) [addConstraint(m,c)] Exists(c) ∧ c∈constraints(m) ∧ m∈constrainedElements(c) ⟨deleteConstraint(m,c)⟩*true* →c∈constraints(m) [deleteConstraint(m,c)] ¬Exists(c) ∧ c∉constraints(m)
End specification of ModelElement	

Specification of Operation	
Sorts	Operation
Taxonomy	Operation≤BehavioralFeature
Updatable functions	
	concurrency: Operation → CallConcurrencyKind precondition: Operation → Constraint postcondition: Operation → Constraint body: Operation → ProcedureExpression
Updatable predicates	
	isPolymorphic: Operation
Actions	
	setConcurrency: Operation x CallConcurrencyKind → ModelEvolution setPrecondition: Operation x BooleanExpression → ModelEvolution setPostcondition: Operation x BooleanExpression → ModelEvolution setImplementation: Operation x ProcedureExpression → ModelEvolution
Axioms	∀o: Operation ∀c:CallConcurrencyKind ∀b: BooleanExpression ∀p:ProcedureExpression
Static axioms	
	[1] Consistency between preconditions and constraints. consistent({precondition(o)}∪allConstraints(owner(o)))

	[2] Consistency between postconditions and constraints. consistent({postcondition(o)} ∪ allConstraints(owner(o)))
Dynamic axioms	
	[setConcurrency(o,c)] concurrency(o)=c [setPrecondition (o,b)] precondition (o)=b [setPostcondition (o,b)] postcondition (o)=b [setEmplementation(o,p)] body(o)=p
End specification of Operation	

Specification of Parameter	
Sorts	Parameter
Taxonomy	Parameter ≤ModelElement
Updatable functions	
	defaultValue:Parameter → Expression kind: Parameter → ParameterDirectionKind type: Parameter → Classifier
Updatable predicates	
	equivalent: Parameter x Parameter
Actions	
Axioms	∀p: Parameter
Static axioms	
	equivalent(p1,p2) ↔ defaultValue(p1)=defaultValue(p2) ∧ kind(p1)=kind(p2) ∧ type(p1)=type(p2)
Dynamic axioms	
End specification of Parameter	

Specification of Relationship	
Sorts	Relationship
Taxonomy	Relationship ≤ModelElement
Updatable functions	
	Additional functions connectedElements: Relationship → Seq of ModelElement allConnectedElements: Relationship → Seq of ModelElement
Updatable predicates	
End specification of Relationship	

Specification of StructuralFeature	
Sorts	StructuralFeature
Taxonomy	StructuralFeature ≤ Feature
Updatable functions	
	changeable: StructuralFeature →ChangeableKind multiplicity: StructuralFeature →Multiplicity targetScope: StructuralFeature →ScopeKind type: StructuralFeature → Classifier
Updatable predicates	

Actions		
	setChangeable: StructuralFeature x ChangeableKind → ModelEvolution setMultiplicity: StructuralFeature x Multiplicity → ModelEvolution setTargetScope: StructuralFeature x ScopeKind → ModelEvolution	
Axioms	∀e: StructuralFeature ∀c: ChangeableKind ∀m: Multiplicity ∀s: ScopeKind	
Static axioms		
Dynamic axioms		
	[setChangeable(e,c)] changeable(e)=c [setMultiplicity(e,m)]multiplicity(e)=m [setTargetScope(e,s)]targetScope(e)=s	
End specification of StructuralFeature		

Descripción informal

El paquete CORE incluye la especificación de los siguientes elementos de modelado, donde cada elemento diferente es representado mediante un sort de la teoría order-sorted.

Association

Una Association (o asociación) define una relación semántica entre Classifiers (generalmente clases). Una Association tiene al menos dos AssociationEnds. Cada AssociationEnds está conectado con un Classifier y representa la conexión entre la Association y el Classifier. Cada AssociationEnd define un conjunto de propiedades que caracterizan a la relación.

Associations

connections es el conjunto de AssociationEnds de la Association.

Acciones

addConnection: agrega un nuevo AssociationEnd a la Association. La acción addConnection(a,e) representa la introducción de una nueva conexión en el modelo. Esta acción afecta a tres elementos del modelos: la Association en sí, el AssociationEnd que es agregado y el Classifier que está siendo conectado, el cual está definido mediante type(e).

deleteConnection: elimina un AssociationEnd de una Association.

AssociationClass

Un AssociationClass es una Association que al mismo tiempo es una Clase. No sólo conecta a un conjunto de Classifiers sino que además define un conjunto de propiedades (o features) propios de la relación. AssociationClass es subclase de Association y de Class. Consecuentemente, una AssociationClass tiene al mismo tiempo AssociationEnds y Features.

AssociationEnd

Un AssociationEnd es un conector que conecta una asociación con un elemento asociado. Los AssociationEnds de una Association están ordenados y son parte de ella.

Un AssociationEnd tiene un nombre y define un conjunto de propiedades de la conexión, tal como multiplicidad.

Attributes

aggregation especifica la clase de conexión. Sus posibles valores son:

none el End no es una agregación.

shared El End es una agregación compartida; por lo tanto el otro extremo de la asociación es una parte de este extremo y consecuentemente el valor de su

atributo aggregation debe ser **none**. Esta parte puede estar contenida en otras agregaciones (es decir que puede estar compartida).
composite El End es una composición; por lo tanto el otro extremo de la asociación es una parte de este extremo y consecuentemente el valor de su atributo aggregation debe ser **none**. Esta parte no puede estar contenida en otras agregaciones (es decir que es no-compartida).

changeable Indica la forma en que la conexión puede ser modificada. Las posibilidades son:
none No se define ninguna restricción sobre la forma de modificación.
frozen luego de establecida la conexión no es posible agregar nuevos links.
addOnly Es posible agregar nuevos links, pero no se permite eliminar links existentes.

isOrdered Cuando su valor es true indica que existe un orden entre los objetos asociados. En otro caso los objetos forman un conjunto sin orden.

isNavigable Si su valor es true indica que es posible navegar (es decir acceder) desde la asociación hacia ese End.

multiplicity Indica la cantidad de instancias que pueden estar asociadas mediante esta conexión.

name Es un nombre que permite hacer referencia a los objetos conectados, desde el Classifier opuesto (o source). Se lo denomina rol-name y representa un pseudo-atributo del Classifier opuesto, es decir que puede usarse en la misma forma que un atributo. Por lo tanto este nombre no puede ser igual al de algún otro atributo en dicho Classifier.

targetScope Sus posibles valores son **instance** y **classifier** indicando si la conexión corresponde a instancias del Classifier o al Classifier en sí mismo.

Asociaciones

qualifier Una lista (posiblemente vacía) de atributos calificados)

type Indica el Classifier que se conecta mediante este End.

association Representa la Association a la cual pertenece el AssociationEnd.

Acciones

setAggregation	modifica el valor del atributo aggregation
setChangeable	modifica el valor del atributo changeable
setMultiplicity	modifica el valor del atributo multiplicity
setTargetScope	modifica el valor del atributo targetScope
setQualifier	modifica el valor del atributo qualifier
setOrdered	especifica que la asociación es ordenada.
setNavigable	especifica que la asociación es navegable.

Notar que el tipo (type) y la asociación (association) de un AssociationEnd son inmutables.

Attribute

Un atributo (o Attribute) representa a un casillero dentro de un classifier. Tiene un nombre y describe el rango de valores posibles que las instancias del Classifier pueden almacenar en dicho casillero. Attribute es una subclase de StructuralFeature.

Attributes

initialValue Es una Expression especificando el valor que tendrá el atributo en las instancias recién creadas.

Acciones

setInitialValue Asigna un valor inicial al atributo con una expresión dada.

BehavioralFeature

Un BehavioralFeature se refiere a una propiedad dinámica de los elementos, tal como una operación de un Classifier. BehavioralFeature es una metaclase abstracta.

Attributes

isQuery Si su valor es true indica que luego de la ejecución del Behavioralfeature el estado del sistema no habrá cambiado, mientras que si su valor es false indica que pueden ocurrir efectos laterales.

Asociaciones

parameters Es una lista ordenada de parámetros (o Parameters) para la operación. Para invocar a una operación, el llamador debe proveer una lista de valores compatibles con los tipos de los parámetros.

Acciones

addParameter Agrega un nuevo parámetro al Behavioral Feature.
deleteParameter Elimina un parámetro del Behavioral Feature.

Class

Una clase (o Class) es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones y relaciones. Una Clase describe entonces la colección de Features, incluyendo Operaciones (Operations), atributos (Attributes), que son comunes a todo el conjunto de objetos. Una clase puede ser abstracta indicando que no pueden crearse instancias directas de esta clase.

Attributes

isActive Si su valor es true indica que los objetos de esta clase son activos, es decir tienen su propia línea de vida y se comportan concurrentemente con otros objetos activos.

Acciones

activate La clase se vuelve activa.
desactivate La clase se desactiva.

Classifier

Un Classifier declara una colección de Features, tales como Attributes y Operations. Tiene un nombre, el cual es único. Classifier es una metaclase abstracta. Sus subclases son por ejemplo DataType y Class.

Asociaciones

features La lista de Features que posee el Classifier.
associationEnds Es la inversa de type. Es la lista de AssociationEnds en los que participa el Classifier.

Acciones

addFeature Agrega un nuevo Feature al Classifier
deleteFeature Elimina un Feature del Classifier
Notar que no se especifican acciones para agregar o eliminar AssociationEnds. Estas acciones están especificadas en la clase Association.

Constraint

Un Constraint es una restricción o condición semántica que el sistema debe satisfacer. Es una expresión booleana.

Attributes

body Es la expresión booleana que define el Constraint.

Asociaciones

constrainedElement La lista de elementos afectados por el Constraint.

Acciones

setBody Asigna una nueva expresión booleana al body del Constraint.

DataType

Un tipo de dato (o DataTtype) es un tipo cuyos valores, a diferencia de los objetos, no tienen identidad, es decir son valores puros. Las operaciones de un DataType son funciones puras, es decir que retornan valores pero no modifican el estado del sistema.

Feature

Un Feature declara una característica estructural o de comportamiento de una instancia de un Classifier o de un Classifier en sí. Feature es una metaclase abstracta.

Attributes

ownerScope Sus posibles valores son: ***instance*** (indica que el Feature corresponde a las instancias del Classifier) y ***classifier*** (indica que el Feature corresponde al Classifier en sí).

visibility Especifica la visibilidad del Feature. Las posibilidades son:
public Indica que puede ser usado desde afuera del Classifier.
protected Indica que puede ser usado por el Classifier y sus descendientes.
private Indica que sólo el Classifier puede usar el Feature.

Asociaciones

owner El Classifier que contiene al Feature.

Acciones

setOwnerScope Asigna un nuevo ScopeKind como OwnerScope del Feature.

setVisibility Asigna una nueva VisibilityKind como Visibility del Feature.

Notar que un Feature pertenece al mismo Classifier durante toda su vida (es decir, owner es inmutable). Sin embargo un Classifier puede modificar sus Features (por ejemplo, mediante la acción addFeature o delFeature).

GeneralizableElement

El sort GeneralizableElement representa elementos que son generalizables, es decir que pueden ser definidos en forma jerárquica mediante el mecanismo de herencia. En una jerarquía de elementos generalizables cada elemento hereda todos las propiedades definidas en sus ancestros. GeneralizableElement es una metaclase abstracta.

Attributes

isAbstract True indica que el GeneralizableElement es una declaración incompleta (abstracta), false indica que es completa (concreta). Un GeneralizableElement abstracto no es instanciable.

isLeaf True indica que no es posible definir descendientes de este elemento, false indica que el elemento puede tener descendientes.

isRoot True indica que el elemento no puede tener ancestros; false indica que sí puede tenerlos.

Asociaciones

generalizations Conjunto de Generalization cuyo supertipo es el ancestro inmediato del GeneralizableElement.

specializations Conjunto de Generalization cuyo subtipo es un descendiente inmediato del GeneralizableElement.

Generalization

Una Generalization es una relación taxonómica entre un elemento más general y un elemento más específico. El elemento más específico es consistente con el más general (es decir que posee todas las propiedades del más general) y puede contener información adicional.

Attributes

discriminator Los subtipos de un GeneralizableElement están divididos en una o más particiones. El discriminador es un nombre que identifica a la partición. Cada

grupo de links que comparten un discriminador representa una dimensión ortogonal de especialización del supertipo.

Asociaciones

supertype — Designa al GeneralizableElement que es una versión generalizada del GeneralizableElement.

subtype — Designa al GeneralizableElement que es una versión especializada del GeneralizableElement..

ModelElement

Un ModelElement es una entidad del Modelo. ModelElement es una metaclase abstracta que sirve como base para todas las metaclases de UM; todas las demás metaclases de UML son subclases directas o indirectas de ModelElement.

Attributes

name — Un nombre que identifica al ModelElement el cual debe ser único dentro de un contexto.

Asociaciones

constraint — El conjunto de Constraints que afectan al elemento.

package — El paquete en el cual el ModelElement está definido. Su valor es el elemento nulo cuando el ModelElement es el paquete principal (es decir, el modelo en sí).

Acciones

setName — Asigna un nuevo nombre al ModelElement.

addConstraint — Agrega un Constraint al conjunto constraints.

deleteConstraint — Elinina un Constraint del conjunto constraints.

Notar que no se especifican acciones para modificar el paquete en el cual el ModelElement está definido. Estas acciones están especificadas en la clase Package.

Operation

Una operación (Operation) es un servicio que puede solicitarse a un objeto. Una operación tiene un nombre y una signatura, la cual describe los parámetros de la operación. El sort Operation es subsort de BehavioralFeature.

Attributes

concurrency — Sus posibles valores son *sequential* (secuencial) y *concurrent* (concurrente).

isPolymorphic — Si es true indica que puede ser redefinida en las subclases; si es false entonces es hereda sin cambios por todas las subclases.

precondition — Describe las precondiciones para ejecutar la Operation, es una expresión booleana que debe ser verdadera antes de la ejecución de la operación.

postcondition — Describe las postcondiciones para ejecutar la Operation, es una expresión booleana que debe ser verdadera al finalizar la ejecución de la operación.

.

body — Es una expresión que representa la implementación de la operación.

Acciones

setConcurrency — Asigna un CallConcurrencyKind a la operación.

setPrecondition — Asigna una Expression a la precondición de la operación.

setPostcondition — Asigna una Expression a la postcondición de la operación.

setImplementation — Asigna una ProcedureExpression a la implementación de la operación.

Parameter

Un parámetro (Parameter) es la declaración de un argumento a ser pasado y/o retornado por una operación. Un parámetro tiene un nombre, un tipo y dirección de comunicación.

Attributes

defaultValue	Es una expresión cuyo valor será usado cuando no se provea un argumento para el parámetro.
kind	Especifica la dirección del parámetro. Las posibilidades son: in, out, inout y return.

Asociaciones

type	Designa al Classifier del cual el valor del parámetro debe ser instancia.

Relationship

Una relación (Relationship) representa una conexión entre ModelElements. Relationship es una metaclase abstracta. Sus subclases concretas son Generalization y Association.

StructuralFeature

StructuralFeature declara un aspecto estructural de una instancia de un Classifier, tal como un Attribute.Define ciertas propiedades del aspecto, tales como su multiplicidad.

Attributes

changeable	Indica si el valor del feature puede ser modificado luego de su creación. Las posibilidades son: **none** No se definen restricciones sobre la modificación. **frozen** no se permiten modificaciones. **addOnly** (tiene sentido sólo cuando la multiplicidad es mayor que uno) indica que elementos adicionales pueden ser agregados al conjunto, pero no eliminados.
multiplicity	La cantidad posible de valores que puede almacenar el atributo para cada instancia.

Asociaciones

type	Indica de que tipo deben ser los valores del atributo.

Acciones

setChangeable	Asigna un valor ChangeableKind al StructuralFeature.
setMultiplicity	Asigna un valor de Multiplicity al StructuralFeature.
setTargetScope	Asigna un valor de ScopeKind al StructuralFeature.

5.2.3 Paquete BEHAVIORAL ELEMENTS

El paquete Behavioral Elements define los elementos que permiten modelar el comportamiento (behavior) de los objetos del sistema. Este paquete está integrado por tres subpaquetes: Collaborations, Use Cases, y State Machines. El subpaquete Collaboration describe los diagramas de colaboración y secuencias de envíos de mensajes; El subpaquete Use Cases describe los diagramas de actores y casos de uso; Finalmente, el subpaquete State Machine describe las máquinas de estados finitos.

De estos tres subpaquetes sólo nos ocuparemos de State Machines, los otros dos paquetes no serán considerados dentro de este trabajo. Su inclusión en la M&D-theory constituye parte de nuestros planes de trabajo futuro. Sin embargo, es importante destacar que su no-inclusión no le resta poder expresivo al lenguaje de modelado UML, en el sentido de que toda la información representada mediante estos diagramas puede también ser expresada mediante otros diagramas presentes en la teoría.

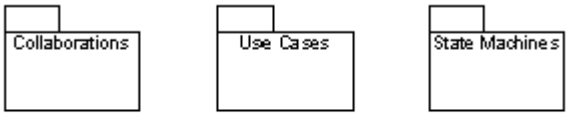


Figura 5.8: Behavioral Elements Packages

5.2.3.1 Paquete BEHAVIORAL ELEMENTS: STATE MACHINES

El paquete State Machine especifica un conjunto de conceptos que pueden ser usados para modelar comportamiento de los objetos del sistema mediante máquinas de estados finitos.

Una máquina de estados permite especificar el comportamiento completo de su contexto (el contexto es la clase a la cual la máquina está especificando). Cuando un objeto (instancia de la clase) recibe un mensaje, la máquina de estados ligada a la clase determina cual es el efecto asociado al mensaje recibido. En la máquina de estados esta información es representada de la siguiente forma: cada transición es disparada por un evento (trigger) el cual generalmente representa la recepción de un determinado mensaje. Además cada transición define una secuencia de acciones (effect) a desplegar como respuesta al evento.

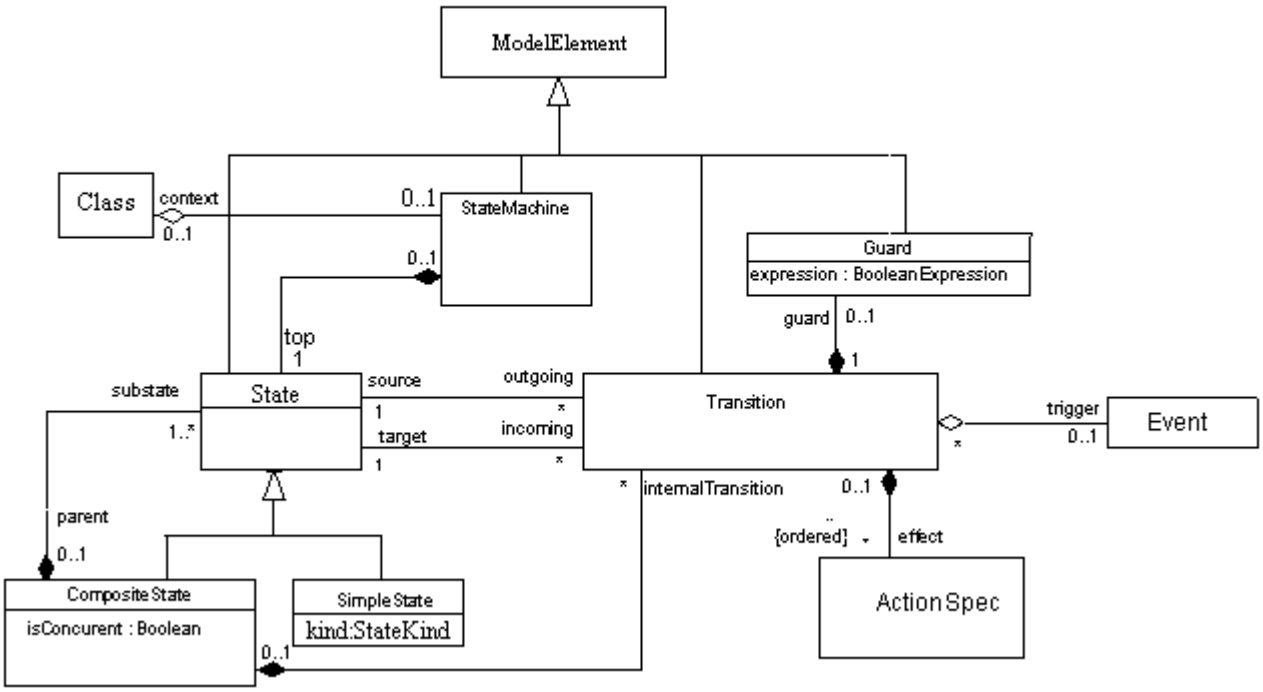
Por otra parte, la máquina de estados permite la especificación del protocolo de una clase de objetos, mostrando el orden en el cual las operaciones pueden ser invocadas sobre un objeto. Dado que en cada estado sólo un conjunto de eventos puede ser aceptado, los cuales están representados por las transiciones con origen en el estado dado (outgoing transitions).

Las siguientes secciones describen al paquete State Machines.

Sintaxis Abstracta

En la figura 5.9 se muestra, usando notación gráfica, la sintaxis abstracta de los elementos del paquete State Machines.

Figura 5.9: State Machine



Especificación en Lógica Dinámica

Specification of ActionSpec	
Sorts	ActionSpec, CreateActionSpec, LocalInvocationSpec, CallActionSpec, DestroyActionSpec
Taxonomy	ActionSpec ≤ModelElement CreateActionSpec≤ActionSpec CallActionSpec≤ActionSpec LocalInvocationSpec≤ActionSpec DestroyActionSpec≤ActionSpec
NonUpdatable functions	
	Creator functions create: Classifier →CreateActionSpec destroy: ObjectExpression → DestroyActionSpec call: Operation x Seq of ObjectExpression →CallActionSpec update:ObjectExpressionx AttributeName x Expression → LocalInvocationSpec *Observer functions* actualArguments: ActionSpec →Seq of Expression *additional functions*

	referencedElements: ActionSpec →Set of ModelElement
Axioms	∀s: Seq of ObjectExpression ∀c:Classifier ∀exp,exp1:ObjectExpression ∀op:Operation ∀a:AttributeName
	actualArguments(create(c))=<c> actualArguments(destroy(exp))=<exp> actualArguments(call(op,s))=s actualArguments(update(exp,a,exp1))= <exp,exp1> referencedElements(create(c))=c referencedElements(destroy(exp))=referencedElements(exp) referencedElements(call(op,s))= {op} ∪ ($\cup_{expi \in s}$referencedElements(expi)) referencedElements(update(exp,a,exp1)=referencedElements(exp)∪{a} ∪referencedElements(exp1)
End specification of ActionSpec	

Specification of CompositeState	
Sorts	CompositeState
Taxonomy	CompositeState ≤State
Updatable functions	
	substates: CompositeState → Set of State internalTransitions: CompositeState →Set of Transition *additional functions* allSubstates: CompositeState → Set of State allInternalTransitions: CompositeState →Set of Transition
Updatable predicates	
	isConcurrent: CompositeState internalTransitions: State→Set of Transition *additional predicates* isTop: CompositeState
Actions	
	addSubState: CompositeState x State →ModelEvolution deleteSubState: CompositeState x State → ModelEvolution addInternalTransition: CompositeState x Transition → ModelEvolution deleteInternalTransition: CompositeState x Transition → ModelEvolution
Axioms	∀c: CompositeState ∀s:State ∀t:Transition
Static axioms	
	Axioms for additional functions [1] allSubstates(c) returns all the nested substates contained in c. allSubstates(c)= substates(c) ∪ ($\cup_{s:CompositeState \in substates(c)}$ allSubStates(s)) [2] allInternalTransitions(c) returns all the nested transitions contained in c. allInternalTransitions(c)= internalTransitions(c) ∪ ($\cup_{s:CompositeState \in substates(c)}$ allInternalTransitions(s)) *Axioms for additional predicates* [1] isTop(c) ↔parent(c)=nullElement *Well-formedness axioms* [1] A composite state has one initial substate ∃s1:SimpleState (s1∈subStates(c) ∧ kind(s1)= #initial) ∧ ∀s1,s2:SimpleState ((s1∈subStates(c) ∧ kind(s1)= #initial ∧ s2∈subStates(c) ∧ kind(s2)= #initial) → s1=s2)

	[2] There have to be at least two composite substates in a concurrent composite state isConcurrent(c) → ∃s1,s2:CompositeState (s1≠s2 ∧ s1∈subStates(c)∧ s2∈subStates(c)) [3] life dependency Exists(c) ↔ (∀s∈substates(c)Exists(s) ∧ ∀t∈internalTransitions(c)Exists(t))
Dynamic axioms	
	⟨addSubState (c,s)⟩*true* →s∉substates(c) [addSubState (c,s)] Exists(s) ∧s∈substates(c) ∧ parent(s)=c ∧ outgoings(s)=∅ ∧incomings(s)=∅ ⟨deleteSubState (c,s)⟩*true* →s∈substates(c) ∧ outgoings(s)=∅ ∧incomings(s)=∅ [deleteSubState (c,s)] ¬Exists(s) ∧ s∉substates (c) ⟨addInternalTransition(s,t)⟩*true* → t∉internalTransitions(s) ∧ source(t)∈substates(c) ∧ target(t)∈substates(c) [addInternalTransition(s,t)] Exists(t) ∧ t∈internalTransitions(s) ∧ t∈incomings(target(t)) ∧t∈outgoings(source(t)) ⟨deleteInternalTransition(s,t)⟩*true* → t∈ internalTransitions(s) [deleteInternalTransition(s,t)] ¬Exists(t) ∧t∉internalTransitions(s) ∧ t∉incomings(target(t)) ∧t∉outgoings(source(t))
End specification of CompositeState	

Specification of Event	
Sorts	Event, TimeEvent, ChangeEvent,CreationEvent, CallEvent, DestructionEvent
Taxonomy	Event ≤ModelElement CreationEvent≤Event CallEvent≤Event DestructionEvent≤Event TimeEvent≤Event ChangeEvent≤Event
NonUpdatable functions	
	creator functions create: →CreationEvent destroy: →DestructionEvent call: Operation →CallEvent timeOut: → TimeEvent changed: →ChangeEvent *additonal functions* referencedElements: Event →Set of ModelElement
Axioms	
	referencedElements(create)=∅ ∧ referencedElements(destroy)=∅ referencedElements(changed)=∅ ∧ referencedElements(timeOut)=∅ referencedElements(call(op)) = {op}
End specification of Event	

Specification of Guard	
Sorts	Guard
Taxonomy	Guard ≤ModelElement
Updatable functions	

	expression: Guard → BooleanExpression *additional functions* referencedElements: Guard →Set of ModelElement
Updatable predicates	
	guard-compatible: Class x Guard
Actions	
	setExpression: Guard x BooleanExpression →ModelEvolution
Axioms	∀g: Guard ∀e:BooleanExpression
Static axioms	
	axioms for additional functions and predicates referencedElements(g)=referencedElements(expression(g)) guard-compatible(c,g)↔ syntactic-compatible(c,expression(g))
Dynamic axioms	
	[setExpression (g,e)] expression(g)=e
End specification of Guard	

Specification of SimpleState	
Sorts	SimpleState
Taxonomy	SimpleState ≤State
Updatable functions	
	kind:StateKind
Updatable predicates	
Actions	
Axioms	∀s:SimpleState
Static axioms	
Dynamic axioms	
End specification of SimpleState	

Specification of State	
Sorts	State
Taxonomy	State≤ModelElement
Updatable functions	
	parent: : State→CompositeState outgoings: State→Set of Transition incomings: State→Set of Transition
Updatable predicates	
	isRegion: State
Actions	
Axioms	∀s: State
Static axioms	
	axioms for additional predicates [1] isRegion(s) ↔parent(s)≠ nulObject
Dynamic axioms	

End specification of State	

Specification of StateMachine	
Sorts	StateMachine
Taxonomy	StateMachine≤ModelElement
Updatable functions	
	context: StateMachine→ Class top: StateMachine→ CompositeState *additional functions* allStates: StateMachine→ Set of State allTransitions: StateMachine→ Set of Transition specifiedOperations: StateMachine→ Set of Operation referencedElements: StateMachine→ Set of ModelElement
Updatable predicates	
	refinement: StateMachine x StateMachine syntactic-compatibility: Class x StateMachine
Actions	
	setBehavior: Class x StateMachine → ModelEvolution cancellBehavior: Class → ModelEvolution
Axioms	∀h:StateMachine, ∀c:Classifier
Static axioms	
	axioms for additional functions [1] allStates returns all the nested states of the state machine. allStates(h)={top(h)} ∪ allSubStates(top(h)) [2] allTransitions returns all the transitions contained into the top state and its nested substates. allTransitions(h)= allInternalTransitions(top(h)) [3] the predicate syntactic-compatible is true if only features of the class are used within the state machine. syntactic-compatible(c,h) ↔∀t∈allTransitions(h) syntactic-compatible(c,t) [4]∀op:Operation op∈specifiedOperations(h) ↔∃t∈allTransitions(h) trigger(t)=call(op) [5] referencedElements(h)= $\cup_{t\in allTransitions(h)}$ referencedElements(t) [6] the predicate refinement(h1,h2) is true if StateMachine h1 is a refinement of StateMachine h2. *Well-formedness axioms* [1] A top state cannot have parent parent(top(h)) = nullElement [2] The top state cannot be the source or target of a transition isEmpty(outgoings(top(h))) ∧ isEmpty(incomings(top(h))) [3] The top state must have (one or more) final state. ∃s:SimpleState (s∈subStates(top(h)) ∧ kind(s)= #final) [4] source and target states must belong to the state machine ∀t:Transition (t∈allTransitions(h) → source(t)∈allStates(h) ∧ target(t)∈allStates(h)) [5] symmetry. ∀t:Transition∀s:State (t∈outgoings(s)↔ source(t)=s ∧ t∈incomings(s)↔ target(t)=s) [6] compatibility between views: only features of its context class can be used within a state machine. syntactic-compatible(context(h),h) [7] life dependency Exists(h) ↔ Exists(top(h))

Dynamic axioms		
	[setBehavior(c,h)](Exists(h)∧ package(h)=package(c) ∧ h∈ownedElements(package(c)) ∧ context(h)=c ∧ behavior(c)=h) h=behavior(c) → [cancellBehavior(c)] (¬Exists(h) ∧ h∈ownedElements(package(c)) ∧ behavior(c)=nullElement)	
End specification of StateMachine		

Specification of Transition		
Sorts	Transition	
Taxonomy	Transition ≤ModelElement	
Updatable functions		
	trigger: Transition → Event + NullElement guard: Transition → Guard effect: Transition → Seq of ActionSpec source: Transition → State target: Transition → State *additional functions* referencedElements: Transition → Set of ModelElements	
Updatable predicates		
	syntactic-compatible: Classifier x Transition	
Actions		
	setTrigger: Transition x Event → ModelEvolution setGuard: Transition x Guard→ ModelEvolution setEffect: Transition x Seq of ActionSpec → ModelEvolution	
Axioms	∀t:Transition ∀e:Event ∀g:Guard ∀a:ActionSpec	
Static axioms		
	Axioms for additional functions [1]referencedElements(t)=referencedElements (trigger(t))∪referencedElements(guard(t)) ∪ ($\cup_{a\in effect(t)}$ referencedElements(a)) [2] ∀c:Classifier syntactic-compatible(c,t) ↔ (trigger-compatible(c,trigger(t)) ∧ guard-compatible(c,guard(t)) ∧ ∀a∈effect(t) effect-compatible(c,a)) [3.1] an event is trigger-compatible with a class if it is an event that can be received for the instances of that class, (i.e creation or destruction or reception of a message) ∀c:Class ∀e:Event (trigger-compatible(c,e) ↔ (e=create ∨ e=destroy ∨ e= timeOut ∨ ∃op∈allOperations(c)e=call(op)) [3.2] an actionSpec is effect-compatible with a class if it represents an action that can be triggered by the instances of that class, (either creation, destruction or message sendings to other objects in the scope of the sender, or localInvocations to self) *Well-formedness axioms* [1] An initial transition at the topmost level may have a trigger with a create event. Apart from this case, an initial transition never has a trigger. (kind(source(t))=#initial ∧ isTop(parent(source(t)))) → trigger(t)=create (kind(source(t))=#initial ∧ ¬isTop(parent(source(t)))) → trigger(t)=nullElement [2] source and target of the transitions have the same parent. parent(source(t)) =parent(target(t))	
Dynamic axioms		
	[setTrigger(t,e)] trigger(t)=e [setGuard(t,g)]guard(t)=g	

	[setEffect(t,s)]effect(t)=s	
End specification of Transition		

Descripción informal

El paquete STATE MACHINES incluye la especificación de los siguientes elementos de modelado. Cada elemento diferente es representado mediante un sort de la teoría order-sorted.

ActionSpec

Una ActionSpec es la especificación estática de la ejecución de una acción. Estas especificaciones se encuentran dentro de las cláusulas de efecto de las transiciones de la máquina de estados. Una ActionSpec tiene una lista de expresiones que representan los argumentos actuales de la acción. Durante la ejecución del sistema, los argumentos se evalúan retornando instancias y estas especificaciones se instancian con acciones ejecutadas sobre los datos (DataEvolution actions).
Existen tres clases de ActionSpecs:

- CreateActionSpec (especifica la creación de un nuevo objeto de alguna clase)
- CallActionSpec (especifica la invocación de una operación o envío de mensaje)
- LocalInvocationSpec (especifica la invocación de una operación local sobre el objeto)
- DestroyActionSpec (especifica la destrucción de un objeto)

CompositeState

Un estado compuesto (CompositeState) es un estado que contiene sub-estados.

Asociaciones

substates Subconjunto de estados que están contenidos en el estado compuesto.
internalTransitions Conjunto de transiciones que ocurren completamente dentro del estado. Si uno de los triggers es invocado, entonces la transición se dispara sin cambiar de estado.

Atributos

isConcurrent Si es true, entonces el estado compuesto se divide directamente en dos o más componentes ortogonales (usualmente se asocia con ejecución concurrente), si es false entonces exactamente uno de los subestados puede estar activo en un momento dado (es decir que la ejecución es secuencial).
isTop Es un valor booleano que indica si el estado es el más externo (el top) en la máquina de estados.

Acciones

addSubstate Agrega un nuevo subestado al estado compuesto.
deleteSubState Elimina un subestado del estado compuesto.
addInternalTransition Agrega una transición entre dos subestados internos.
deleteInternalTransition Elimina una transición entre dos subestados internos.

Event

Un evento es la especificaicón de un acontecimiento significativo que ocurrer en el sistema y que produce un cambio de estado.

Consideramos cinco clases especiales de eventos:

- CreationEvent (representa la creación de un nuevo objeto)
- CallEvent (representa la atención de un mensaje ya recibido),
- DestructionEvent (representa la destrucción de un objeto existente)
- TimeEvent (representa un evento temporal tal como un deadline)
- ChangeEvent (evento vinculado con la relación de dependencias)

Guard

Una guarda es una expresión booleana que puede asociarse a una transición para especificar las oncidicones bajo las cuales la transición estará abilidtada o desabilitada. La guarda se evalúa cuando se produce un evento que dispara la transición. La transición se disparará sólo si la guarda es verdadera en ese momento.

Atributos

expression Es una expresión booleana que especifica la condición de la guarda.

Acciones

setExpression Asigna una expresión booleana al cuerpo de la guarda.

SimpleState

Un estado simple (o SimpleState) es un estado que no tiene sub-estados.

Atributos

kind Determina el tipo del estado. Sus posibles valores son: #initial, #final, #simple.

State

Un estado es una situación durante la vida de un objeto, el cual satisface determinadas condiciones y espera la ocurrencia de ciertos eventos. Un estado puede ser origen (source) y/o destino (target) de transiciones.

Atributos

isRegion Es un valor booleano que indica si el estado es un sub-estado de un estado concurrente.

Asociaciones

parent Especifica el estado compuesto que contiene al estado.
outgoing Especifica las transiciones que salen del estado.
incoming Especifica las transiciones que entran en el estado.

StateMachine

Una StateMachine está integrada por estados (States) y transiciones (Transitions). Generalmente una StateMachine se utiliza para especificar el comportamiento de las instancias de un Classifier (el cual se denomina context). La StateMachine es dueña en forma directa únicamente del estado compuesto más externo (el top state), los demás estados se encuentran anidados en los respectivos sub-estados. El comportamiento de las instancias se especifica como las posibles secuencias de cambios de estados que pueden producirse durante su vida. Las transiciones se disparan como consecuencia de la ocurrencia de eventos.

Asociaciones

context El context es el elemento cuyo comportamiento es especificado por la StateMachine. UML considera que dicho elemento puede ser un Classifier o un BehavioralFeature, pero en este trabajo nos limitamos a representar sólo Clases. Además UML permite asociar más de una StateMachine a cada elemento, mientras que nosotros permitimos a lo sumo una.

top El top designa el estado compuesto más externo de la StateMachine. Los demás estados están contenidos por los estados padres. Su multiplicidad es 1.

Acciones

setBehavior Agrega una nueva máquina de estados en el modelo y la liga con una clase. La máquina define el comportamiento de las instancias de dicha clase.
cancellBehavior Elimina del modelo a la máquina de estados ligada a la clase.

Transition

Una Transition es una relación entre un estado fuente (source) y un estado destino(target).

Asociaciones

trigger Especifica el evento que activa la transición.
guard Es un predicado que debe evaluar a true cuando la transición se activa.
effect Secuencia de acciones a ser ejecutadas cuando la transición se activa.
source Estado donde se inicia la transición.
target Estado al cual conduce la transición.

Acciones

setTrigger Asigna un evento como trigger de la transición.
setGuard Asigna una nueva guarda a la transición.
setEffect Asigna una secuencia de especificaciones de acciones como efecto para la transición.

5.2.4 Paquete MODEL MANAGEMENT

Los modelos actúan como una especificación precisa de los requerimientos que el sistema debe satisfacer. Un modelo del sistema consiste en una conceptualización del dominio del problema y se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal.

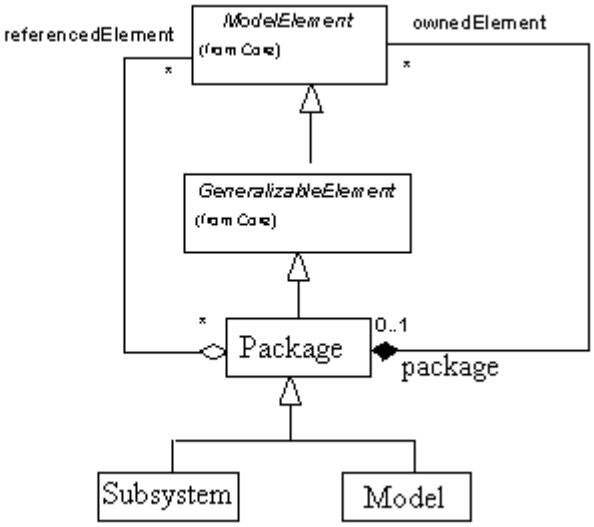
Durante el proceso de desarrollo de software diferentes modelos del sistema son creados. La diferencia entre estos modelos reside en los aspectos del sistema que son contemplados y en su grado de abstracción. Tal como hemos comentado anteriormente, estos modelos están relacionados entre sí de distintas formas.

El paquete Model Management contiene la descripción de elementos de modelado que permiten reunir diferentes modelos y expresar sus relaciones.

Sintaxis Abstracta

La figura 5.10 muestra, usando notación gráfica, la sintaxis abstracta de los elementos del paquete Model Management.

Figura 5.10: Model Management



Especificación en Lógica Dinámica

Specification of Package	
Sorts	Package
Taxonomy	Package ≤GeneralizableElement
Updatable functions	
	referencedElements: Package → Set of ModelElement ownedElements: Package → Set of ModelElement *additional functions*

	contents : Package →Set of ModelElement allContents : Package →Set of ModelElement allSurroundingPackages : Package →Set of Package
Updatable predicates	
Actions	
	addReferencedElement: Package x ModelElement →ModelEvolution addSubpackage: Package x Package → ModelEvolution addClassifier: Package x Classifier → ModelEvolution addRelationship: Package x Relationship → ModelEvolution deleteReferencedElement: Package x ModelElement → ModelEvolution deleteSubpackage: Package x Package → ModelEvolution deleteClassifier: Package x Classifier → ModelEvolution deleteRelationship: Package x Relationship → ModelEvolution
Axioms	∀p: Package
Static axioms	
	axioms for additional functions [1] The operation contents results in a set containing all ModelElements owned or referenced by the Package. contents(p) = ownedElements(p) ∪ referencedElements(p) [2] The operation allContents results in a Set containing all ModelElements contained by the Package itself and all its inherited elements. allContents(p) = contents(p) ∪ (∪s∈supertypes(c) allContents(s)) [3] The operation allSurroundingPackages results in a Set containing all surrounding Packages. allSurroundingPackages(p) = {package(p)} ∪ allSurroundingPackages(package(p)) *well-formedness axioms* [1] in a Package the Classifier names and the Package names are unique $\forall c_1,c_2$: Classifier (($c_1 \in$contents(p)$\wedge c_2 \in$contents(p) $\wedge$ name(c_1) = name(c_2)) $\rightarrow c_1 = c_2$) $\forall p_1,p_2$: Package (($p_1 \in$contents(p) $\wedge$ $p_2 \in$contents(p) $\wedge$name(p_1) = name(p_2)) $\rightarrow p_1 = p_2$) [2] in a Package the association names are unique $\forall a_1,a_2$: Association (($a_1 \in$contents(p) $\wedge$ $a_2 \in$contents(p)$\wedge$name(a_1)=name(a_2)) $\rightarrow a_1 = a_2$) [3] The supertype must be included in the Package of the GeneralizableElement. ∀g:Generalization supertype(g) ∈ allContents(package(subtype(g))) [4] The connected type should be included in the current Package ∀f:StructuralFeature type(f)∈allContents(package(owner((f))) [5] The connected Classifiers of the AssociationEnds must be included in the Package of the Association. ∀a:Association (a∈ownedElements(p) →∀r∈allConections(a) type(r)∈allContents(p)) [6] The context Classifiers of the StateMachine must be included in the Package of the StateMachine. (also the behavior StateMachine of the Classifiers must be included in the Package of the Classifier). ∀s:StateMachine context(s)∈allContents(package(s)) ∀c:Classifier behavior(c)∈allContents(package(c)) [7] life dependency Exists(p) ↔(∀e∈ownedElements(p)Exists(e))
Dynamic axioms	
	[addReferencedElement(p,e)] e∈referencedElements(p) [deleteReferencedElement(p,e)] e∉referencedElements(p)

	[addSubpackage(p,e)] Exists(e) $\wedge e \in$ ownedElement (p) $\wedge$ package(e)=p [addClassifier(p,e)] Exists(e) $\wedge e \in$ ownedElement (p) $\wedge$ package(e)=p [addRelationship (p,e)] Exists(e) $\wedge e \in$ ownedElement (p) $\wedge$ package(e)=p [deleteSubpackage (p,e)] $\neg$Exists(e) $\wedge e \notin$ ownedElement (p) [deleteClassifier (p,e)] $\neg$Exists(e) $\wedge e \notin$ ownedElement (p) [deleteRelationship(p,e)] $\neg$Exists(e) $\wedge e \notin$ ownedElement (p)
End specification of Package	

Specification of Model	
Sorts	Model
Taxonomy	Model $\leq$Package
End specification of Model	

Specification of Subsystem	
Sorts	Subsystem
Taxonomy	Subsystem $\leq$Package
End specification of Subsystem	

Descripción informal

Un paquete (Package) reúne a un grupo de elementos de modelado. El atributo referencedElements en un Package M se refiere a elementos que han sido definidos en otro Package, pero que pueden ser accedidos desde M. En cambio el atributo ownedElements se refiere a elementos que han sido definidos dentro de M.

Dentro de un Package los elementos de modelado están relacionados, por ejemplo a través de la relación de generalización, la cual ha sido formalizada mediante el predicado IsA(), o de la relación de agregación, la cual ha sido formalizada mediante el predicado partOf(), u otro tipo de relaciones.

Además dentro de un Package los elementos deben cumplir ciertas normas de convivencia, por ejemplo sus nombres no deben repetirse, la máquina de estados definiendo el comportamiento de una clase debe estar incluida en el mismo paquete que dicha clase, etc.

En la jerarquía de sorts, los sorts correspondientes a modelos (Models) y subsistemas (Subsystems) están definidos como subsorts de Package.

5.3 Nivel de los Datos

ELEMENTOS

Los elementos en el nivel de los datos son básicamente instancias (valores y objetos) y mensajes. En este nivel el sistema puede verse como un conjunto de objetos colaborando concurrentemente. Los objetos están relacionados (por medio de relaciones de asociación) y se comunican a través de mensajes que son almacenados en espacios semi-públicos llamados mailboxes. Cada objeto posee un mailbox donde los demás objetos pueden dejarle mensajes. Existen requerimientos de privacidad para asegurar que los únicos mensajes que un objeto puede leer son aquellos contenidos en su propio mailbox.

EVOLUCIÓN

En este nivel el sistema puede evolucionar por la ejecución de tres clases diferentes de acciones:

- **modification**: en general callActions que representan la recepción de un mensaje, lo cual causa que una operación sea invocada en el objeto receptor. La ejecución de una operación puede causar modificaciones en el estado interno del receptor (mediante invocaciones locales o LocalInvocations), así como también el envío de mensajes a otros objetos.
- **creation**: estas acciones provocan la creación de una nueva instancia de una clase.
- **cancellation**: estas acciones provocan que instancia deje de existir en el sistema.

En la teoría formal las distintas acciones de evolución en el nivel de los datos son representadas mediante el sort DataEvolution, el cual tiene tres sub-sorts: Creation, Cancellation y Modification.

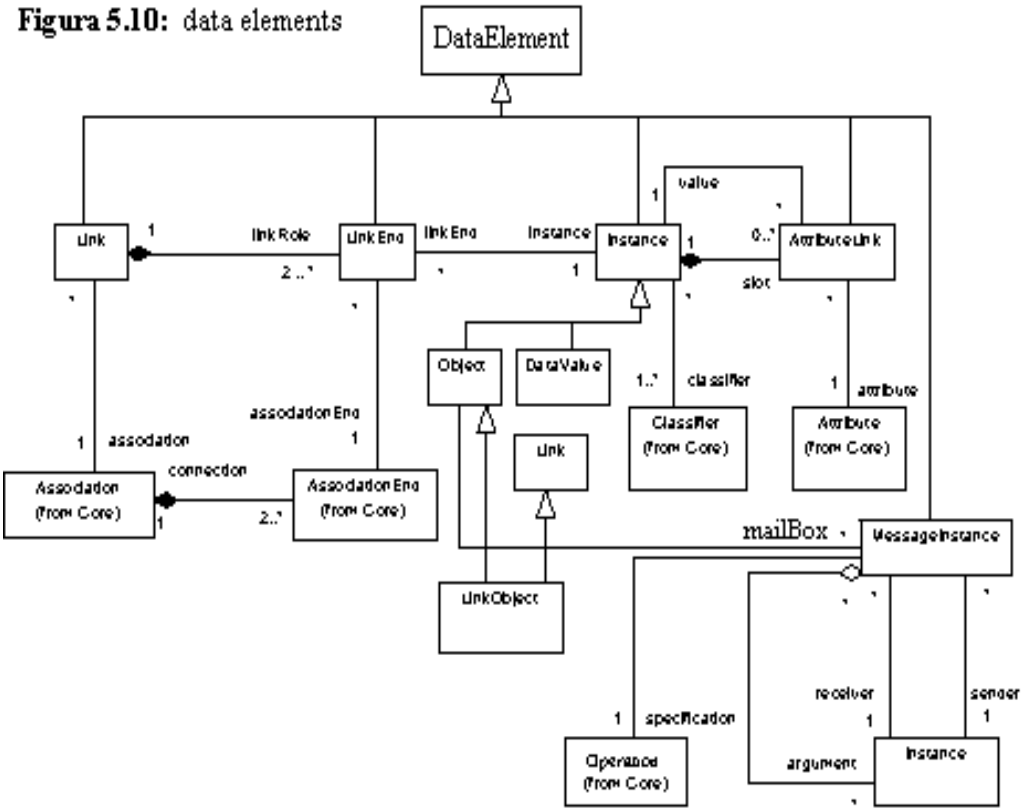
SINTAXIS ABSTRACTA

La figura 5.10 define, usando notación gráfica, la sintaxis abstracta de los elementos en el nivel de los datos, tales como instancias (Instances), conexiones (Links) y mensajes (Messages).

ESPECIFICACIÓN EN LÓGICA DINÁMICA

En esta sección incluimos la especificación formal de los elementos en el nivel de los datos. Esta especificación consta de una signatura Σ_{SYS}**= (** (**S**$_{SYS,\leq}$)**,** $\mathbf{F}_{SYS}$**,** $\mathbf{P}_{SYS}$, $\mathbf{A}_{SYS}$**)** y una fórmula γ_{SYS} sobre Σ_{SYS}. Los elementos del álgebra inicial denotada por la especificación son los datos del sistema y sus relaciones, tales como objetos, conexiones entre objetos, mensajes, etc. La relación de transición entre posibles mundos representa evolución de los datos, por ejemplo cambios en los valores de los atributos de los objetos. La fórmula ϕ_{SYS} es la conjunción de dos conjuntos disjuntos de fórmulas, γ_S y γ_D de fórmulas estáticas y fórmulas dinámicas respectivamente. El primer conjunto consiste en fórmulas no modales que deben satisfacerse en todos los estados posibles del sistema (son invariantes o propiedades estáticas o reglas de buena formación de los objetos). Mientras que el segundo conjunto consiste en fórmulas modales que definen la semántica de las acciones, es decir de la evolución de los datos.

Figura 5.10: data elements



Specification of AttributeLink	
Sorts	AttributeLink
Taxonomy	AttributeLink ≤ DataElement
Updatable functions	
	value: AttributeLink →Instance attribute: AttributeLink→ Attribute
Updatable predicates	
Actions	
Axioms	∀a:AttributeLink
Static axioms	
	[1] the type of the value of the Attribute must match the type of the Attribute. IsA(classifier(value(a)) , type(attribute(a)))
Dynamic axioms	
End specification of AttributeLink	

Specification of DataElement	
Sorts	DataElement
Taxonomy	DataElement ≤Element
Nonupdatable functions	

Updatable predicates	
Actions	
Axioms	
Static axioms	
Dynamic axioms	
End specification of DataElement	

Specification of DataValue	
Sorts	DataValue, Primitive, Number, Boolean, String
Taxonomy	DataValue≤Instance, Primitive≤DataValue, Integer≤ Primitive, Boolean≤ Primitive, String≤ Primitive,
Nonupdatable functions	
	...specification of the built-in primitive data types...
Nonupdatable predicates	
Axioms	
Static axioms	
	[1] ...axioms for primitive data types... [2] The Classifier of a DataValue must be a DataType ∀d:DataValue ∃t:DataType classifier(d)=t
Dynamic axioms	
End specification of DataValue	

Specification of Instance	
Sorts	Instance
Taxonomy	Instance≤DataElement
Updatable functions	
	slots: Instance → Set of AttributeLink linkEnds: Instance → Set of LinkEnd classifier: Instance → Classifier *additional functions:* value:Instance x Name → Set of Instance allLinks: Instance →Set of Link allOppositeLinkEnds: Instance →Set of LinkEnd parts: Instance → Set of Instance allParts: Instance → Set of Instance
Updatable predicates	
Actions	
Axioms	∀i:Instance
Static axioms	

	axioms for additional functions [1] value returns the value of an attribute or an association. Notice that since attribute names and opposite role names do not overlap, one of the two sets is always empty. value(i,n)= { value(l) | l∈slots (i) ∧ name(attribute(l))=n } ∪ { instance(l) | l∈allOppositelinkEnds(i) ∧ name(associationEnd(l))=n } [2] allLinks returns a set containing all links in which the instance participates. allLinks (i)= map link linkEnds(i) [3] allOppositelinkEnds returns a set containing all opposite LinkEnds of the instance. allOppositelinkEnds(i) = { e∈(∪l∈links(i) linkRoles(l)) | instance(e)≠i} [4] parts returns a set containing the parts of a composite instance. parts(i)={p:Instance | ∃k:Link ∃l1∈linkRoles(k)∃l2∈linkRoles(k) (instance(l1)=i ∧ instance(l2)=p ∧ aggregation(associationEnd(l1))=#composite)} [5] allParts returns all nested parts of a composite instance. allParts(i)= parts(i) ∪ (∪p∈parts(i) allParts(p)) *well-formedness axioms:* [1] the AttributeLinks matches the declarations in the Classifier. ∀l:AttributeLink(l∈slots(i) ↔attribute(l)∈allAttributes(classifier(i))) [2] the links matches the declarations in the Classifier. ∀l:Link(l∈allLinks(i) →association(l)∈allAssociations(classifier(i))) [3] An Instance may not belong by composition to more than one composite Instance. ∃e1,e2∈oppositeLinkEnds(i) ((aggregation(associationEnd(e1))=#composite ∧ aggregation(associationEnd(e2))=#composite) → e1=e2) [4] Satisfaction of Constraints. Constraints always evaluate true. ∀c∈allConstraints(classifier(i)) (eval(c)[self:=i] = true) [5] symmetry l∈linkEnds(i)↔ instance(l)=i
End specification of Instance	

Specification of Link		
Sorts	Link	
Taxonomy	Link≤DataElement	
Updatable functions		
	association: Link → Association linkRoles: Link → Seq of LinkEnd *additional functions* connectedElements: Link → Seq of Instance	
Updatable predicates		
Actions		
Axioms	∀l:Link	
Static axioms		
	axioms for additional functions [1]connectedElements(l) returns a sequence containing all the instances connected by the link. connectedElements(l)= map instance linkRoles(l) *well-formedness axioms* [1] The set of LinkEnds must match the set of AssociationEnds of the Association.	

	∀e:AssociationEnd(e∈map associationEnds linkRoles(l)↔e∈allConnections(association(l))) [2] There are not two Links of the same Association which connects the same set of Instances in the same way. ∀l1,l2:Link((association(l1)=association(l2)∧connectedElements(l1)=connectedElements(l2)) →l1=l2)
Dynamic axioms	
End specification of Link	

Specification of LinkEnd	
Sorts	Object
Taxonomy	LinkEnd ≤ DataElement
Nonupdatabel functions	
Updatable functions	
	instance: LinkEnd→Instance associationEnd: LinkEnd→AssociationEnd
Updatable predicates	
Actions	
Axioms	∀l:LinkEnd
Static axioms	
	[1] The type of the Instance must match the type of the AssociationEnd type(associationEnd(l))=classifier(instance(l))
Dynamic axioms	
End specification of LinkEnd	

Specification of LinkObject	
Sorts	LinkObject
Taxonomy	LinkObject ≤ Link , LinkObject ≤ Object
End specification of LinkObject	

Specification of Message	
Sorts	Message
Taxonomy	Message ≤ DataElement
Nonupdatable functions	
	<>: Operation x Instance x Instance x Seq of Instance →Message specification: Message → Operation. sender: Message → Instance receiver : Message → Instance arguments: Message → Seq of Instance
Updatable functions	
NonUpdatable predicates	
	IsLocal: Message
Actions	
Axioms	∀m:Message
Static axioms	

	[1] specification <op,s,r,p> = op [2] sender<op,s,r,p> = s [3] receiver <op,s,r,p> = r [4] arguments<op,s,r,p> = p *well-formedness axioms* [1] the receiver understands the message. specification(m)∈operations(classifier(receiver(m))) [2] the types and order of actual arguments for a message must match the parameters of the Operation . match(arguments(m),parameters(specification(m))) where, match(<>,<>)=true match(a-args , p-pars) = classifier(a)=type(p) ∧ match(args,pars)
Dynamic axioms	
End specification of Message	

Specification of Object	
Sorts	Object
Taxonomy	Object ≤ Instance
Nonupdatabel functions	
Updatable functions	
	mailBox: Object → Seq of Message currentStates:Object→Set of Name
Updatable predicates	
Actions	
	newObject: Class x Object →Creation update: Object x Name x Instance → Modification -.-: Object, Message → Modification destroy: Object→Cancellation
Axioms	∀o:Object,
Static axioms	
	well-formedness axioms [1] ∀m∈mailBox(o) receiver(m)=o [2] ∀m∈mailBox(o) specificaton(m)∈operations(classifier(o))
Dynamic axioms	
	[3] local invocations modifying the value of an attribute or link of object o. [update(o,a,v)] value(o,a)=v [4] call actions: reception of a message m ⟨o.m⟩*true* →m=first(mailBox(o)) Enabled(o.m) → m∈mailBox(o) [5] call actions: reception of a message m ⟨o.m⟩*true* → ∃t:Transition t∈firingTransitions(behavior(classifier(o)),m) [6] effect of call actions. [o.m](currentStates(o)=currentStates(o)-{source(t) | t∈firing}∪{target(t) | t∈firing} ∧∀ t∈firing ∀a∈effect(t) sent(a) ∧ mailBox(o) = mailBox(o) – firt(mailBox(o))) where, firing=firingTransitions(behavior(classifier(o)),m)

	[7] fairness conditions ∀m:Message ∀o:Object (m∈mailBox(o) → ◊⟨o.m⟩*true*) [8] destruction of objects [destroy(o)] ¬Exists(o) [9] Behavioral correctness: the behavior of operations (defined by the state machine) satisfies the pre and post conditions of the corresponding specifications. ∀<op,s,r,p>:Message (classifier(r)=owner(op) → (eval(precondition(op)[self/r, parameters/p])=true →[r.<op,s,r,p>]eval(postcondition(op)[self/r, parameters/p])=true))
End specification of Object	

Descripción informal

Los siguientes sorts están contenidos en la especificación formal del nivel de los datos:

AttributeLink

Un AttributeLink almacena el valor de un atributo de una instancia

Asociaciones

value El valor del AttributeLink.
attribute El atributo representado por el AttributeLink.

DataValue

Los Valores (DataValue) son instancias sin identidad y cuyo estado interno nunca cambia (todas las operaciones aplicables sobre valores son funciones (queries), es decir no producen modificaciones. Además un valor tampoco puede cambiar su clase.

Instance

Una instancia que se origina a partir de un Classifier, el cual define su estructura y comportamiento. Todas las instancias de un mismo Classifier están estructuradas de la misma forma, aunque cada una de ellas posee su propio conjunto de slots. Cada slot almacena un valor para un atributo definido en el classifier de la instancia. Cada instancia está conectada a un conjunto de conexiones (links). El conjunto de slots mas el conjunto de conexiones determinan el estado de la instancia.

Instance es una metaclase abstracta.

Asociaciones

slots	El conjunto de AttributeLinks que almacenan los valores de los atributos de la instancia.
linkEnds	El conjunto de LinkEnds de los Links conectados a la instancia.
classifier	El Classifier que declara la estructura de la instancia.

Link

Un link es una conexión entre instancias. Cada link es una instancia de una asociación (Association) del nivel del modelo, es decir que un link conecta instancias de (subclases de) las clases asociadas. Cada link posee un conjunto de linkEnds, cada uno de ellos está ligado con exactamente una instancia. Una instancia se puede comunicar con las instancias ligadas a sus linkEnds opuestos (siempre que este sea navegable).

Asociaciones

association	La Association que declara al Link.
linkRole	La secuencia de LinkEnds que constituyen el Link.

LinkEnd

Un link end es el extremo de un link, conectado directamente a una instancia. Se corresponde con el concepto de AssociationEnd en el nivel del modelo.

Asociaciones

instance	La instancia conectada al LinkEnd.
AssociationEnd	El AssociationEnd que declara al LinkEnd.

LinkObject

Un link object es una clase especial de link que al mismo tiempo es un objeto. Surgen cuando la conexión en sí misma es un objeto, es decir posee atributos y es posible aplicarle operaciones.

LinkObject es subclase de ambos Object y Link.

MessageInstance

Un mensaje representa una comunicación entre dos instancias. Es una terna compuesta por:

- El nombre del mensaje (que se corresponde con el nombre de la operación invocada por el mensaje)
- La identidad del destinatario del mensaje
- Los argumentos actuales para los parámetros de la operación invocada.

La recepción de un mensaje produce la invocación de una operación sobre el receptor del mensaje. Esto puede causar transiciones de estado y efectos sobre otros objetos, tal como es especificado mediante la máquina de estado de la clase del receptor.

Asociaciones

specification	La operación representada por el mensaje.
sender	La instancia que envía el mensaje.
receiver	La instancia que recibe el mensaje.
arguments	La secuencia de instancias ligadas a los argumentos del mensaje.

Object

Object es una subclase de Instance. Los valores de los slots de un objeto pueden cambiar a lo largo de su vida. La clase de un objeto puede también cambiar. Esto significa que las propiedades definidas por la nueva clase son dinámicamente agregadas al objeto, mientras que las propiedades definidas en la antigua clase son dinámicamente eliminadas del objeto.

Durante su vida un objeto está definido por:

- Su identidad (identity);
- Su clase (Class);
- Su estado interno, es decir los valores de sus atributos y conexiones;
- Su mailbox privado.

Asociaciones

mailBox	Su mailbox privado conteniendo los mensajes que el objeto ha recibido y que aún no ha procesado.

DataEvolution

En este nivel el sistema puede evolucionar por la ejecución de tres clases diferentes de acciones: creación de objetos (Creation), destrucción de objetos (Cancellation) y modificación de objetos mediante el envío de mensajes (CallAction) o invocaciones locales (LocalInvocation). Estos sorts se definen como subsorts del sort DataEvolution (ver capítulo 7).

El símbolo **.** (dot) denota CallActions. La fórmula [(obj_term**.**message_term)] Pred_term significa que inmediatamente después de que el objeto denotado por obj_term recibe y procesa el mensaje denotado por message_term, el predicado Pred evalúa a verdadero.

La recepción de un mensaje implica que el objeto está preparado para recibir el mensaje y actuar en consecuencia. Este tipo de fórmulas especifica cual es el comportamiento esperado del objeto.

En UML el comportamiento de los objetos se especifica mediante máquinas de estado (**StateMachine**). De acuerdo con la semántica de las máquinas de estado, para procesar un mensaje el receptor debe estar en un estado apropiado y la guarda asociada con el mensaje (donde los parámetros fueron reemplazados por los argumentos actuales) debe satisfacerse. La ejecución de un mensaje m puede habilitar la ejecución de otras acciones en el estado siguiente.

Las siguientes fórmulas, cuantificadas con ($\forall$o:Object), especifican la semántica de la recepción y procesamiento de mensajes en el sistema:

[1]Requerimientos de Privacidad:

Sólo los mensajes destinados al dueño de un mailbox pueden ingresar a dicho mailbox.

$\forall m \in mailBox(o)$ receiver(m)=o

[2] No Dangling behavior:

Los objetos no aceptan mensajes que no estén definidos en su protocolo (es decir en la interfaz de su classifier).

$\forall m \in mailBox(o)$ specificaton(m)$\in$operations(classifier(o))

[3] Invocaciones locales.

Existe una clase especial de mensajes llamado invocaciones locales (LocalInvocations). Estos mensajes son enviados por un objeto a sí mismo con el objetivo de producir modificaciones en su estado interno. Este tipo de invocaciones tiene lugar sin la mediación de una máquina de estado. En nuestra teoría, localInvocations son representadas mediante acciones *update* definidas mediante la siguiente fórmula:

[*update*(o,a,v)] value(o,a)=v, expresando la modificación del valor de un atributo del objeto o.

[4] Acciones habilitadas:

Los únicos mensajes que se reciben y procesan son los contenidos en algún mailbox. El orden de procesamiento es fifo (primero en llegar es el primero en ser atendido):

Enabled(o.m) $\rightarrow$ m$\in$mailBox(o)

$\langle$o.m$\rangle$*true* $\rightarrow$m=first(mailBox(o))

[5] Ejecución de CallActions

En UML las acciones CallActions se especifican mediante transiciones en una máquina de estados. La función firingTransitions se aplica sobre una máquina de estados y una acción que actúa como trigger, retornando el conjunto de transiciones habilitadas por dicho trigger. Esta función está definida formalmente en el capítulo 6.

$\langle$o.m$\rangle$true $\rightarrow$ $\exists$t:Transition t$\in$firingTransitions(behavior(classifier(o)),m)

[6] Efectos:

Cuando una transición se dispara, la secuencia de acciones denotada por effect(t) debe ser ejecutada (sólo estamos considerando asyncronous actions). Además, luego del disparo el objeto receptor debe cambiar de estados (deja los source states y pasa a los target states, considerando estados concurrentes). Finalmente, el mensaje que disparó la transición es borrado del mailbox, pues ya ha sido procesado:

[o.m](currentStates(o)=currentStates(o)-{source(t) | t$\in$firing}$\cup${target(t) | t$\in$firing}

$\wedge \forall$ t$\in$firing $\forall$a$\in$effect(t) sent(a)
$\wedge$ mailBox(o) = mailBox(o) – first(mailBox(o)))

donde firing es el conjunto de transiciones abilitadas corrientemente y está definido de la siguiente forma:
firing=firingTransitions(behavior(classifier(o)),m)

y donde sent indica la ejecución del efecto de la transición. Cada ActionSpec contenida en la secuencia effect(t) puede especificar: la creación de un objeto, o la destrucción de un objeto, el envío de un mensaje o la invocación de una operación local.
El predicado sent() se define de la siguiente forma para cada tipo de acción:

sent(create(c))=Enabled(newObject(c))
sent(destroy(exp))=Enabled(destroy(eval(exp)))
sent(update(exp1,a,exp2))=Enabled(update(eval(exp1),a,eval(exp2)))
sent(call(op,args))=<op,eval(self),eval(head(args)),eval(tail(args))>$\in$mailBox(eval(head(args)))

Por ejemplo, ejecutar un evento de invocación involucra evaluar las expresiones de objetos para obtener al receptor del mensaje y a sus parámetros reales. Con estos objetos se crea un nuevo mensaje que es colocado en el mailBox del receptor.

[7] Condiciones fairness:
Todo mensaje que fue enviado será eventualmente recibido y procesado.
$\forall$m:Message $\forall$o:Object (m$\in$mailBox(o) $\rightarrow \Diamond\langle$o.m$\rangle$*true)*

[8]Destrucción de objetos.
Como consecuencia de la acción de destrucción el objeto deja de existir.
[destroy(o)] $\neg$Exists(o)

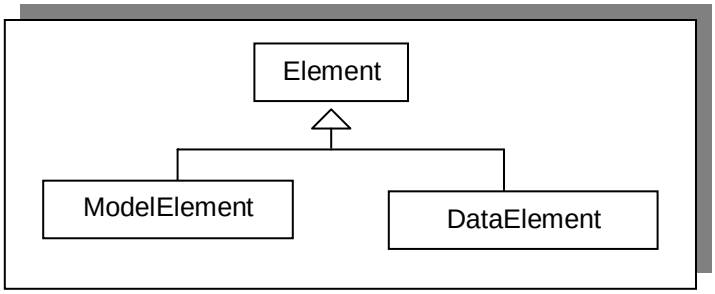
5.4 Integración de ambos niveles: la M&D-theory

La M&D-theory (*Models&Data theory)* es una teoría dinámica de primer orden, formada por una signatura (la cual define el lenguaje de la teoría) y un conjunto de axiomas:

$$\text{M\&D-theory}=(\Sigma_{M\&D}, \phi_{M\&D})$$

La signatura de la teoría, $\Sigma_{M\&D}$=((**S**, ≤), **F, P, A**), es una signatura en lógica dinámica con las siguientes características especiales:

- La signatura$\Sigma_{M\&D}$ incluye a la signatura Σ_{UML}. Esto significa, por ejemplo que existe un conjunto distinguido de sorts $\mathbf{S}_{UML}\subseteq\mathbf{S}$ y un conjunto distinguido de funciones $\mathbf{F}_{UML}\subseteq\mathbf{F}$. Estos símbolos representan a los elementos de modelado, tales como Class y StateMachine. Usualmente son llamados metaclases o metasorts.
- La signatura $\Sigma_{M\&D}$ incluye a la signatura Σ_{SYS}. Esto significa, por ejemplo que existe un conjunto distinguido de sorts $\mathbf{S}_{SYS}\subseteq\mathbf{S}$. Estos símbolos representan a los objetos del sistema y sus relaciones, tales como Object y Message.
- Existe un sort universal llamado Element. Es decir, Element$\in\mathbf{S}\wedge(\forall s\in\mathbf{S})s\leq$Element.
- Los conjuntos de sorts $\mathbf{S}_{UML}$ y $\mathbf{S}_{SYS}$ son disjuntos y sus elementos no están relacionados por ≤**.** Es decir que $(\forall u\in\mathbf{S}_{UML})(\forall s\in\mathbf{S}_{SYS})\neg(u\leq s\vee s\leq u)$. Además cada uno de estos dos conjuntos tiene un sort distinguido (DataElement y ModelElement respectivamente) encabezando la jerarquía, es decir, DataElement$\in\mathbf{S}_{SYS}\wedge(\forall s\in\mathbf{S}_{SYS})s\leq$DataElement y por otro lado, ModelElement$\in\mathbf{S}_{UML}\wedge(\forall s\in\mathbf{S}_{UML})s\leq$ModelElement. La siguiente figura ilustra esta jerarquía de sorts:



- Hay un símbolo de predicado Exists:Element→Bool. La extensión de un sort (el conjunto de todas los posibles instancias del sort) es siempre el mismo conjunto en todos los mundos posibles. Sin embargo, sólo algunas de estas instancias existen realmente (han sido creadas y aún no han sido destruidas). El predicado Exists tiene una interpretación diferente en cada mundo posible definiendo el conjunto de instancias existentes en los mundos correspondientes. En el mundo inicial sólo existen elementos de modelado, pero no existen instancias, es decir que se cumple el siguiente axioma: (∀i:Instance) ¬Exists(i).
- Hay un símbolo de predicado *Enabled*:Action→Bool. Este predicado define el conjunto de Acciones (Actions) cuya ejecución está permitida en cada mundo, es decir el conjunto de acciones habilitadas.
- Existen dos símbolos de acción que permite efectivizar la relación de instanciación inter-nivel (la cual fue descripta al inicio de este capítulo). La signatura de estos símbolos es: newObject:ClassifierxObject→Creation y newLink:AssociationxLink→Creation . El término newObject(c,o) denota la creación de una nueva instancia (referenciada por o) de la clase

denotada por el término c. El término newLink(a,k) denota la creación de una nueva conección (referenciada por k) correspondiente a la asociación denotada por el término a. Las funciones polimórficas: instances:Classifier $\rightarrow$Set of Instance y instances:Association $\rightarrow$Set of Link, representan los conjuntos de instancias (o links) creados a partir de un Classifier (o Association). Están definidos de la siguiente forma:

$\forall$c:Classifier $\forall$i:Instance (i$\in$instances(c) $\leftrightarrow$ classifier(i)=c)

Por otra parte los axiomas de la teoría, están integrados por distintos grupos de axiomas. Es decir, $\phi_{M\&D}$ es la conjunción de tres fórmulas: $\phi_{M\&D} = \phi_{UML} \wedge \gamma_{SYS} \wedge \phi_{JOINT}$. Primeramente, ϕ_{UML} es la fórmula sobre Σ_{UML} que define las características de los elementos de modelado (esta fórmula se obtiene mediante la conjunción de todos los axiomas del nivel de los modelos). Luego, γ_{SYS} es la fórmula construida sobre Σ_{sys} que describe las características de los elementos modelados (esta fórmula se obtiene mediante la conjunción de todos los axiomas del nivel de los datos). Finalmente, ϕ_{JOINT} es una fórmula construida utilizando el lenguaje integrado *M&D* y por lo tanto puede expresar tanto propiedades de los modelos, como propiedades de los datos, como propiedades relacionando ambos niveles. La fórmula ϕ_{JOINT} se obtiene uniendo dos grupos de axiomas:

- $\phi_{GENERAL}$ (describe características generales a todos los sistemas. Un ejemplos de este tipo de fórmulas es el axioma [Ax-new] descripto arriba)
- $\phi_{SPECIFIC}$ (describe características específicas de cada sistema). Esta fórmula es la conjunción de:
  - axiomas de instanciación ϕ_{INST} .
  - axiomas de completación ϕ_{COMP}.

La figura 5.11 ilustra la estructura de la M&D_theory.

5.5 La Interpretación semántica de UML

Las principales componentes de la interpretación semántica de UML son reglas para asociar estructuras sintácticas del lenguaje de modelado con elementos en un dominio semántico formalmente definido. En las siguientes secciones describiremos el dominio semántico y las correspondientes reglas de interpretación para las construcciones de UML.

5.5.1 El dominio semántico

El dominio semántico donde las construcciones de UML serán interpretadas está formado por sistemas de transición de la forma $U=(S^U, w_o, m_U)$. Un sistema de transición es un conjunto de mundos posibles con una relación de transición entre ellos, tal como fue descripto en el capítulo 3. Formalmente, sea $\Sigma=(S, \leq), F, P, A)$ una signatura dinámica order-sorted de primer orden y sea $\Sigma_N=(S, \leq), F_N, P_N)$ la parte non-updatable de Σ. Sea $U=(A, m_U)$ una $\mathbf{\Sigma_N}$-álgebra, que provee el dominio de valores y la interpretación de los términos estáticos. Las fórmulas del lenguaje se interpretan sobre Kripke-frames de la forma:

$$U=(S^U, w_o, m_U)$$

donde:

- S^U es el conjunto de estados. Cada estado $w \in S^U$, es una función de interpretación de términos sobre el álgebra U, de la siguiente forma:

  si $f \in F_N$ entonces $w(f)=f^U$ (es decir, la interpretación fija dada por el álgebra U).

si $f \in F_U$ y $f:s_1,..,s_n \rightarrow s$ entonces $w(f): U_{s1},..,U_{sn} \rightarrow U_s$.

si $p \in P_N$ entonces $w(p)=p^U$ (es decir, la interpretación fija dada por el álgebra U).

si $p \in P_U$ y $p:s_1,..,s_n$ entonces $w(p): U_{s1},..,U_{sn}$

si x es una variable de sort s, entonces $w(x) \in U_s$.

- $w_0 \in S^U$ es el estado inicial.
- m_U asocia cada acción α una relación binaria llamada la relación entrada/salida de α:

  $m_U(\alpha) \subseteq S^U \times S^U$

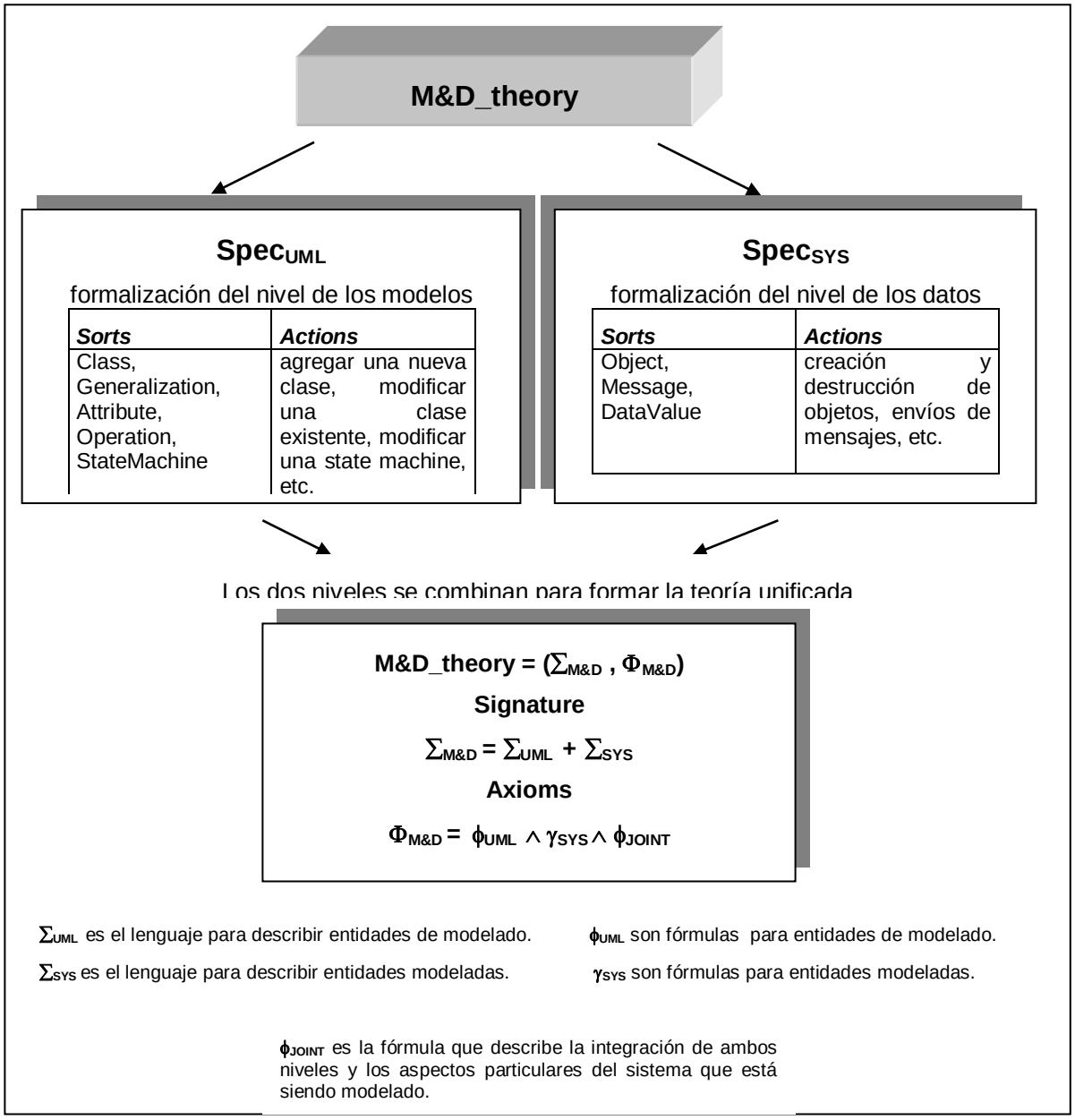


Figura 5.11: M&D-theory

Notemos que el dominio de las interpretaciones es un álgebra heterogénea (es decir una Σ_N-álgebra) cuyos elementos incluyen tanto datos (por ejemplo objetos) como meta-datos (por ejemplo clases). En la figura 5.12 mostramos una estructura de mundos posibles, donde se observa la dicotomía datos (sobre fondo gris) vs. meta-datos (sobre fondo blanco).

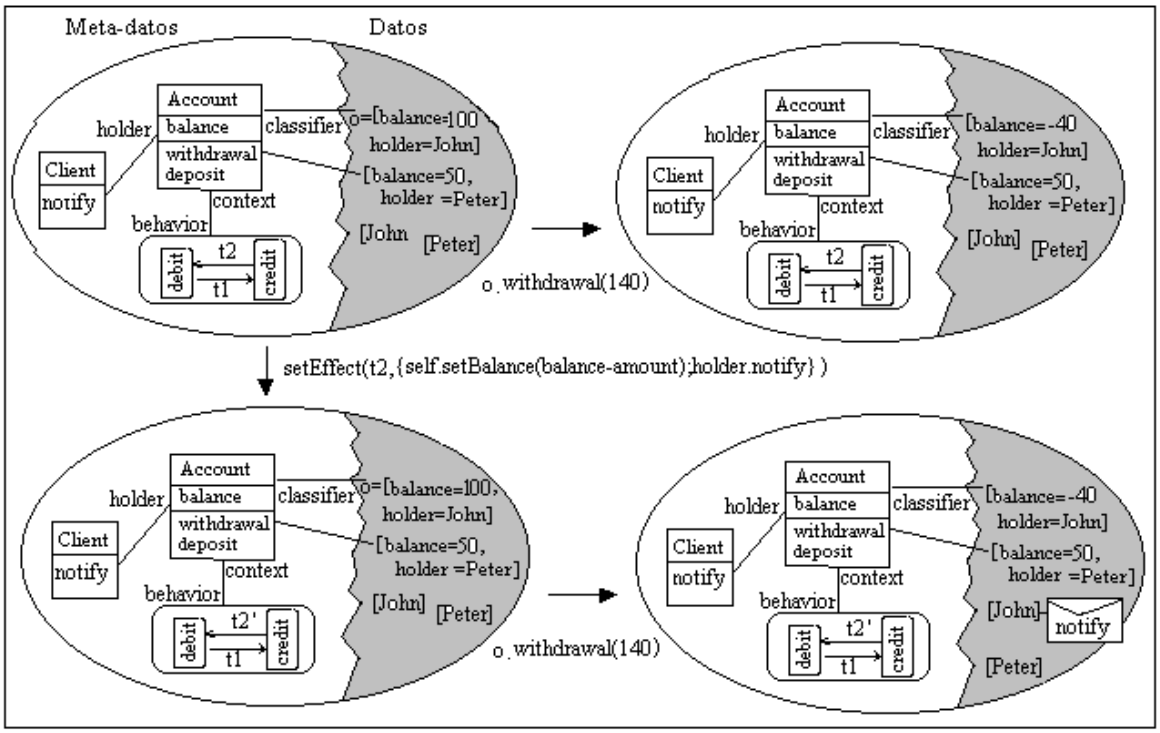


Figura 5.12 : evolución en ambas direcciones

El conjunto de relaciones de transición entre mundos está particionado en dos conjuntos disjuntos:

- Un conjunto de transiciones que representan modificaciones sobre la especificación del sistema (evolución de los meta-datos).
- Un conjunto de transiciones que representan modificaciones sobre los datos del sistema (evolución de los datos).

La figura 5.12 muestra un ejemplo de evolución en ambas direcciones. Es importante notar que como consecuencia de la evolución de la especificación (es decir la modificación de la transición t2 agregándole un nuevo efecto: enviar el mensaje notify al holder) el comportamietno del objeto ***o*** ha sufrido una modificación co-lateral.

5.5.2 Función de interpretación semántica

En las secciones anteriores hemos definido el dominio semántico donde interpretaremos al lenguaje de modelado UML. El paso siguiente consiste en establecer las relaciones entre los conceptos sintácticos de UML y los conceptos del dominio semántico, a través de la definición de la función de interpretación semántica que llamaremos **Sem**.

Sem: ConstruccionesUML $\rightarrow$ DominioSemántico

Es importante destacar que la interpretación semántica de UML que proporcionaremos no es directa, sino que se obtiene en dos pasos:

1- interpretación (o traducción) del lenguaje UML en una teoría M&D.

2- interpretación semántica de la teoría M&D.

Es decir,

ConstruccionesUML $\rightarrow^{translation}$ **M&D-theory** $\rightarrow^{semantics}$ **Dominio-semántico**

Por lo tanto la función de interpretación semántica **Sem** es la composición de ambas funciones, **Sem**=***semantics*** o ***translation***

La función ***semantics*** que asocia una teoría en lógica dinámica con su semántica ha sido explicada en el capítulo 3. Recordemos que la semántica para una especificación dinámica spec es el conjunto de todos los modelos (sistemas de transición entre mundos posibles) que son isomorfos al modelo min-max (el modelo min-max es el elemento $\leq_{\mathbf{r}}$ maximal del conjunto de modelos minimales de spec). Es decir, ***semantics***(spec) = {M | M≡min-max(spec) }

Para obtener la interpretación semántica de UML sólo nos resta definir la función ***translation***. La función ***translation*** asocia cada modelo UML bien formado con su correspondiente M&D-theory. Esta función está determinada por medio de un conjunto de reglas que permiten crear una M&D-theory a partir de los distintos submodelos relacionados que componen un modelo UML. Las reglas trabajan sobre una teoría básica (Llamamos básica a una M&D-theory cuyos únicos axiomas son ϕ_{UML} , ϕ_{SYS} y $\phi_{GENERAL}$, es decir que no contiene axiomas específicos $\phi_{SPECIFIC}$). Las reglas van enriqueciendo progresivamente esta teoría, agregando dos clases de axiomas específicos:

- de instanciación ϕ_{INST} (describen cuales elementos de modelado son usados en este modelo)
- y de completación ϕ_{COMP} (describen características especiales del sistema modelado)

Es importante destacar que los axiomas de completación no corresponden a la interpretación semántica del modelo UML, sino que permiten enriquecer la teoría obtenida mediante el agregado de nueva información que no estuviera presente en el modelo UML original.

Ilustraremos la función ***translation*** mediante un ejemplo. La figura 5.13 muestra el modelo UML de un sistema bancario mientras que la figura 5.14 contiene la teoría asociada al mismo.

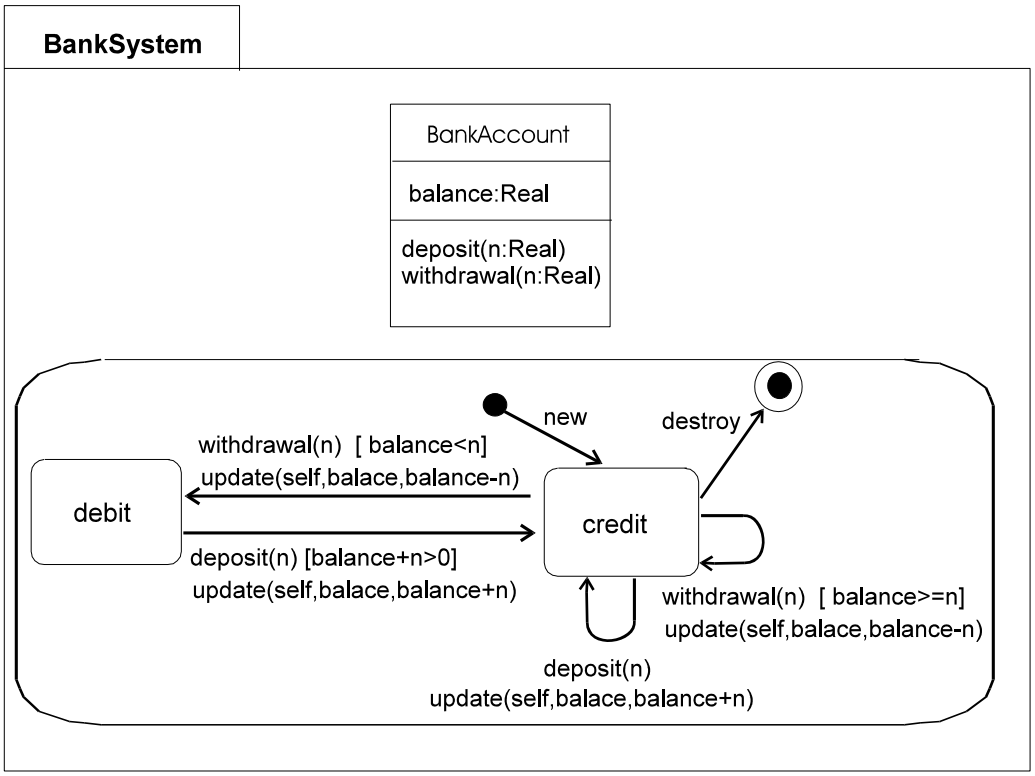


Figure 5.13: modelo UML de un sistema bancario

BANK-theory = $(\Sigma_{BANK}, \phi_{BANK})$

Donde

$\Sigma_{BANK} = \Sigma_{M\&D}$

$\phi_{BANK} = \phi_{UML} \wedge \gamma_{SYS} \wedge \phi_{JOINT}$

$\phi_{JOINT\_} = \phi_{GENERAL} \wedge \phi_{SPECIFIC}$

$\phi_{SPECIFIC} = \phi_{INST-BANK} \wedge \phi_{COMP-BANK}$

Axiomas de instanciación $\phi_{INST-BANK}$ =

$\exists$m:Model $\exists c_1$:Class $\exists$h:StateMachine

(

Exists(m)

$\wedge c_1 \in$ ownedElements(m) $\wedge$ name(c_1)=BankAccount

$\wedge \exists$a:Attribute (a$\in$features(c1) $\wedge$ name(a)=balance $\wedge$type(a)=Real

$\wedge$ownerScope(a)=#instance $\wedge$ visibility(a)=#private)

$\wedge \exists$p1,p2:Operation $\exists$a1,a2:Parameters

(

p1$\in$features(c1)$\wedge$name(p1)=deposit$\wedge$ownerScope(p1)=#instance$\wedge$visibility(p1)=#public

$\wedge$parameters(p1)=<a1> $\wedge$ type(a1)=Real $\wedge$ kind(a1)=#in

$\wedge$p2$\in$features(c1)$\wedge$name(p2)=withdraw$\wedge$ownerScope(p2)=#instance$\wedge$visibility(p2)=#public

$\wedge$parameters(p2)=<a2> $\wedge$ type(a2)=Real $\wedge$ kind(a2)=#in

)

$\wedge$ h$\in$ownedElements(m) $\wedge$ context(h)= $c_1 \wedge$

$\exists$s:CompositeState $\exists s_1,s_2,s_3,s_4$:SimpleState $\exists$ t1,t2,t3,t4,t5,t6:Transition

(

top(h)=s $\wedge$ substates(s)={s_1,s_2,s_3,s_4}

$\wedge$ kind(s_1)=initial$\wedge$ kind(s_2)=normal $\wedge$kind(s_3)=normal $\wedge$kind(s_4)=final

$\wedge$ name(s_3)=debit $\wedge$ name(s_2)=credit

$\wedge$ internalTransitions(s)={t1,t2,t3,t4,t5,t6}

$\wedge$ trigger(t1)=#create $\wedge$ source(t1)=$s_1 \wedge$ target(t1)=$s_2 \wedge$ guard(t1)=*true* effect(t1)=<>

$\wedge$trigger(t2)=call(p_2) $\wedge$ source(t2)=$s_2 \wedge$ target(t2)=$s_3 \wedge$ guard(t2)=balance<n

$\wedge$effect(t2)=update(self,balance,balance-n)

$\wedge$trigger(t3)=call(p_1) $\wedge$ source(t3)=$s_3 \wedge$ target(t3)=$s_2 \wedge$ guard(t3)=balance+n$\geq$0

$\wedge$effect(t3)=update(self,balance,balance+n)

$\wedge$trigger(t4)= call(p_1) $\wedge$ source(t4)=$s_2 \wedge$ target(t4)=$s_2 \wedge$ guard(t4)=*true*

$\wedge$effect(t4)=update(self,balance,balance+n)

$\wedge$trigger(t5)= call(p_2) $\wedge$ source(t5)=$s_2 \wedge$ target(t5)=$s_2 \wedge$ guard(t5)=balance$\geq$n

$\wedge$effect(t5)=update(self,balance,balance-n)

$\wedge$trigger(t6)=#destroy $\wedge$ source(t2)=$s_2 \wedge$ target(t2)=$s_4 \wedge$ guard(t2)=*true* effect(t2)=<>

)

)

Axiomas de competación $\phi_{COMP-BANK}$ = true (no se agregan axiomas de completación

Figure 5.14: teoría asociada al modelo UML del sistema bancario

5.6 Conclusiones

Ventajas de la integración: Hemos presentaremos una nueva propuesta para definir formalmente la semántica de UML. La idea básica de esta nueva formalización consiste en utilizar un dominio semántico que integra a las entidades de modelado y a las entidades modeladas, permitiendo de esta manera representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden, que llamamos M&D-theory.

La siguiente tabla esquematiza el tratamiento que nuestra propuesta da a cada una de las cuatro dimensiones discutidas en el capítulo 4:

	Modelo	Sistema modelado
Aspectos estáticos	Axiomas de primer orden sobre entidades de modelado	Axiomas de primer orden sobre entidades modeladas
Aspectos dinámicos	Acciones y axiomas modales sobre entidades de modelado.	Acciones y axiomas modales sobre entidades modeladas

En la siguiente tabla pueden observarse ejemplos de algunos aspectos estáticos y dinámicos, formalizados en la M&D-theory:

	Modelo (spec)	Sistema modelado (sys)
Aspectos estáticos	spec no debe contener dos atributos con el mismo nombre dentro de la misma clase: $\forall$ c:Classifier $\forall$a1,a2:Attribute (a1$\in$ attributes(c) $\wedge$ a2$\in$attributes(c) $\wedge$name(a1) = name(a2)) $\rightarrow$ a1 =a2	Los valores de los atributos de los objetos de sys deben corresponderse con las declaraciones en las respectivas clases: $\forall$a:AttributeLink $\forall$i:Instance a$\in$slots(i) $\rightarrow$attribute(a)$\in$allAttributes(classifier(i)) $\wedge$IsA(classifier(value(a)),type(attribute(a)))
Aspectos dinámicos	Refinamiento del modelo spec por el agregado de una nueva clase: addAttribute : Class x Attribute $\rightarrow$Act $\forall$c:Classifier $\forall$a:Attribute [addAttribute(c,a)] a$\in$attributes(c)	Los objetos de sys reaccionan al recibir mensajes: _._:Object x Message$\rightarrow$ Act $\forall$o:Object $\forall$m:Message $\langle$o.m$\rangle$*true* $\rightarrow$ $\exists$t:Transition t$\in$firingTransitions(behavior(classifier(o),o.m))

Contar con una estructura formal de primer orden, en contraste con una estructura de orden superior, facilita los procedimientos para calcular la validez de las fórmulas. A pesar de que la lógica de primer orden es no-decidible (y por lo tanto también lo es la lógica dinámica de primer orden), los sistemas de computación satisfacen ciertas propiedades (por ejemplo, se interpretan sobre estructuras aritméticas, el estado de un programa en un momento dado queda determinado por un conjunto finito de valores) las cuales permiten calcular la validez de las fórmulas dinámicas en forma finita y efectiva. El problema de decidibilidad de la lógica dinámica ha sido descripto en el capítulo 3.

Semántica conjuntista: La intensión de los lenguajes de especificación es describir un conjunto de sistemas (no un solo sistema) cuyos elementos poseen las propiedades requeridas. La semántica de estos lenguajes es un conjunto de modelos, en lugar de un simple modelo.

Notemos que el dominio semántico que hemos definido para UML es un conjunto potencia (es decir un conjunto de conjuntos de modelos).

Esta semántica conjuntista permite definir interesantes propiedades:

- Una especificación spec es consistente si $Sem(spec) \neq \emptyset$, lo cual significa que una implementación puede ser encontrada para la especificación.
- Composición de especificaciones. Composición significa que varias especificaciones pueden ser compuestas para obtener una nueva especificación. Es importante lograr que la semántica del resultado pueda ser deducida a partir de la semántica de las partes (semántica composicional).
- Una especificación spec' refina a otra especificación spec si $Sem(spec') \subseteq Sem(spec)$, lo cual significa que spec puede ser reemplazada por spec´ sin perdida de información.

Abstracción y completitud del dominio semántico: El dominio semántico debe ser tan abstracto como sea posible, pero incluyendo toda la información esencial. Una especificación gráfica escrita en UML informa al lector acerca de la estructura, la funcionalidad, patrones de interacción, colaboraciones, restricciones, etc. de los componentes del sistema. Pero la especificación además contiene un importante monto de información relacionada con la notación en si, por ejemplo ubicación, color, lo cual típicamente no tiene ninguna influencia sobre la semántica. Esta información no debe estar presente en el dominio semántico contribuyendo a un más compacto y claro dominio semántico. Por otro lado es importante asegurar que toda la información expresada por UML pueda ser representada en el dominio semántico (completitud del dominio semántico). El dominio semántico que hemos definido es abstracto ya que trata la sintaxis abstracta del lenguaje y es incompleto ya que cubre sólo un subconjunto del lenguaje UML. Sin embargo, dado que la teoría posee una estructura jerárquica resulta fácilmente extensible, lo cual permitiría su completación. Es importante destacar que el subconjunto de UML contemplado por la teoría es suficientemente expresivo, dado que la información que puede ser representada por los diagramas no incluídos en la teoría puede también ser representada (desde otro punto de vista) por otro tipo de diagramas presentes en la teoría.

6. Usando el modelo formal

El modelo formal que hemos definido provee las siguientes ventajas:

- Las especificaciones expresadas en el lenguaje gráfico obtienen un significado preciso.
- Conduce a un entendimiento mas profundo de los conceptos expresados mediante la notación, lo cual permite un uso más maduro de la tecnología.
- La construcción del modelo formal nos permitió descubrir ambigüedades e inconsistencias en el lenguaje UML.
- Las distintas vistas de un sistema (class diagrams, statecharts, constraints, etc.) son integradas en un único modelo formal. Esta integración permite definir reglas de compatibilidad entre las vistas, tanto en el nivel sintáctico como en el semántico.
- Mediante mecanismos formales de deducción es posible obtener información no presentada explícitamente en la especificación.
- Técnicas formales de verificación disponibles en el modelo formal pueden usarse para detectar errores en la especificación gráfica.
- La notación matemática permanece oculta tras la notación gráfica, por lo tanto los desarrolladores de software no necesitan conocimientos matemáticos especiales.

En las siguientes sub secciones mostraremos ejemplos concretos de cada uno de estos tópicos.

6.1 Uso maduro de la tecnología OO

Para utilizar adecuadamente las herramientas de modelado es necesario conocer claramente su significado. En general los diagramas estáticos de UML, tales como diagramas de clases y diagramas de relaciones poseen un significado claro y aceptado por los usuarios, mientras que los diagramas dinámicos tal como las máquinas de estados constituyen elementos complejos cuya semántica suele ser mal interpretada por los ingenieros de software. En esta sección usaremos la lógica dinámica para definir formalmente el significado de estos diagramas.

6.1.1 Semántica de las máquinas de estados

La M&D-theory describe la semántica de las máquinas de estado mediante los axiomas que especifican la relación existente entre las acciones de evolución en el nivel de los datos (en particular CallActions) y la máquina de estados asociada con la clase del receptor de la acción. En esta sección describiremos la semántica standard de las maquinas de estados orientadas a objetos, tal como se utiliza en UML (ver [UML 97 (b)]) y luego daremos su formalización usando la M&D-theory.

Definición informal de la semántica de las máquinas de estado

Una máquina de estados permite especificar el comportamiento completo de su contexto (el contexto es la clase a la cual la máquina está especificando). Cuando un objeto (instancia de la clase) recibe un mensaje, la máquina de estados ligada a la clase determina cual es el efecto asociado al mensaje recibido. En la máquina de estados esta información es representada de la siguiente forma: cada transición es disparada por un evento (trigger) el cual generalmente representa la recepción de un determinado mensaje. Además cada transición define una secuencia de acciones (effect) a desplegar como respuesta al evento.

Un evento puede habilitar una o múltiples transiciones cuyo trigger es dicho evento. Puede también ocurrir que ninguna transición se habilite, en este caso el evento no será atendido. Cuando más de una transición se habilita es necesario seleccionar un subconjunto. Luego este subconjunto de transiciones es ejecutado (o disparado) lo cual produce un cambio de estados en la máquina. Esta transformación se denomina *paso*.

El subconjunto de transiciones que se disparan es determinado por la función de selección llamada *firingTransitions* que describiremos luego. El disparo de una transición puede originar nuevos eventos que afecten al objeto corriente o a otros objetos.

Si estas acciones son sincrónicas entonces la transición se congela hasta que los objetos invocados completen sus propias computaciones. Cuando todas las computaciones han terminado el evento se considera atendido y el paso termina. Si las acciones son asincrónicas el paso termina sin esperar a que se completen las computaciones derivadas.

Transiciones conflictivas (descriptas abajo) no se disparan en el mismo paso.

El orden en el cual las transiciones seleccionadas son disparadas no está definido

Informalmente, la semántica de un paso involucra la ejecución de un conjunto maximal de transiciones no-conflictivas a partir de un estado.

Una transición siempre lleva de un estado válido a otro estado válido. Las transiciones que se originan en un estado compuesto, al dispararse provocan la salida del estado compuesto y también de todos sus sub-estados internos. Estas transiciones se denominan transiciones de alto nivel o interrupciones.

Selección de transiciones

Para seleccionar las transiciones que pueden ser disparadas es necesario tener en cuenta básicamente tres temas: habilitación, conflictos y prioridades, tal como se describe a continuación.

Habilitación:

Una transición está habilitada (enabled) si se cumple lo siguiente:

- El trigger se corresponde con el mensaje recibido.
- El estado corriente del receptor del mensaje es el origen (source state) de la transición.
- La guarda (donde los parámetros fueron reemplazados por los argumentos actuales) se satisface.

Conflictos:

En un estado dado, es posible que varias transiciones estén habilitadas. Entonces es necesario determinar cuales de ellas pueden ser disparadas simultáneamente sin contradicciones (o conflictos). Por ejemplo si hay dos transiciones con origen en un estado s, una rotulada con e[c1] y la otra con e[c2], y si ambas guardas [c1] y [c2] son verdaderas, entonces sólo una de estas transiciones puede ser disparada.

Dos transiciones son conflictivas si la intersección de sus conjuntos de estados de origen es no vacía. La intuición es que sólo transiciones concurrentes pueden ser disparadas simultáneamente.

Prioridades:

Las prioridades se usan para solucionar conflictos. Por definición una transición que sale de un subestado tiene prioridad más alta que cualquier otra transición conflictiva que sale de un estado que contiene al subestado.

Si t1 es una transición cuyo origen es s1, y t2 tiene origen s2, entonces:

- si s1 es un sub-estado de s2, entonces t1 tiene prioridad más alta que t2.
- si s1 y s2 no están relacionados jerárquicamente, entonces no hay prioridad definida entre t1 y t2.

Nótese que otras políticas podrían ser definidas, por ejemplo en statecharts clásicos la prioridad es inversa: los estados más externos tienen mayor prioridad que los estados anidados. Sin embargo en el contexto de objetos los estados internos son más especializados que sus ancestros.

El conjunto de transiciones que se dispararán (llamado *firingTransitions*) es el conjunto maximal que satisface las siguientes condiciones:

- Todas las transiciones en el conjunto están habilitadas.
- No existen conflictos dentro del conjunto.
- No existe otra transición fuera del conjunto con prioridad más alta que una transición dentro del conjunto.

Definición formal de la semántica de las máquinas de estado

En la M&D-theory los siguientes axiomas formalizan la semántica de las máquinas de estado descripta en la sección anterior:

[1] $\forall$o:object,m:Message $\langle$o.m$\rangle$*true* $\rightarrow$ firing$\neq\emptyset$

[2] $\forall$o:object $\forall$m:Message

[o.m](currentStates(o)=currentStates(o)-{source(t) | t$\in$firing}$\cup${target(t) | t$\in$firing}

$\wedge\forall$ t$\in$firing $\forall$a$\in$effect(t) sent(a)

$\wedge$ mailBox(o) = mailBox(o) – first(mailBox(o)))

donde firing=firingTransitions(behavior(classifier(o)),m)

Estas fórmulas especifican las pre y post condiciones de una CallAction:

- La pre-condición es la existencia de una transición de estados en la correspondiente máquina de estados. La máquina de estados es el behavior ligado al classifier del objeto *o*. La función firingTransitions retorna el conjunto de transiciones que el trigger denotado por m habilita en dicha máquina de estados.

- La post-condición consiste en que la secuencia de acciones denotada por effect(t) debe ser ejecutada (sólo estamos considerando acciones asincrónicas). Además, luego del disparo el objeto receptor debe cambiar de estado (deja los source states y pasa a los target states). Finalmente, el mensaje que disparó la transición es borrado del mailbox, pues ya ha sido procesado.

El subconjunto de transiciones que se disparan es determinado por la función de selección llamada *firingTransitions*. La función firingTransitions se aplica sobre una máquina de estados y una acción que actúa como trigger y la definimos de la siguiente forma:

firingTransitions: StateMachine, Message $\rightarrow$ Set of Transition

firingTransitions(H,a)=*filter-no-conflicting*(*allEnabled*(H,a))

donde:

allEnabled: StateMachine, Message $\rightarrow$ Set of Transition

allEnabled(H,a) = { t $\mid$ t$\in$allTransitions(H) $\wedge$ *enabled*(t,a)}

Y donde el predicado *enabled* está definido de la siguiente forma:

enabled: Transition, Message $\rightarrow$Boolean

enabled(t,m) $\leftrightarrow$ trigger(t)=call(specification(m))

$\wedge$ source(t)=currentState(receiver(m))

$\wedge$ eval(guard(t)[self / receiver(m), parameters / arguments(m)])=true)

Y donde la función *filter-no-conflicting* está definida como:

filter-no-conflicting: Set of Transition $\rightarrow$ Set of Transition

filter-no-conflicting($\varnothing$) = $\varnothing$

filter-no-conflicting({x}$\cup$S) = {x} $\cup$ *filter-no-conflicting*(S), si para todo y$\in$S se cumple $\neg$*conflict*(x,y)

filter-no-conflicting({x}$\cup$S) = *filter-no-conflicting*(S), si existe y$\in$S tal que *conflict*(x,y) y x$\leq_{priority}$y.

filter-no-conflicting({x}$\cup$S)= *filter-no-conflicting*({x}$\cup$S-{y}), si existe y$\in$S tal que *conflict*(x,y) y y$\leq_{priority}$x.

Y finalmente:

conflict: Transition x Transition $\rightarrow$ Boolean

conflict(t1,t2) = ({source(t1)}$\cup$allSubstates(source(t1)))$\cap$({source(t2)}$\cup$allSubstates(source(t2))) $\neq\varnothing$

$\leq_{priority}$: Transition x Transition $\rightarrow$ Boolean

t1$\leq_{priority}$ t2 = (source(t1)=source(t2) $\vee$ source(t2)$\in$allSubstates(source(t1)))

6.1.2 Refinamientos de Máquinas de Estados

Dado que las máquinas de estado (StateMachines) definen el comportamiento de elementos generalizables, en particular Clases, es necesario caracterizar la relación existente entre las máquinas de estados correspondientes a dos clases relacionadas por generalización. En general esta relación se denomina refinamiento de máquinas de estado (State Machine refinement). Existen distintas relaciones de refinamiento, las cuales describiremos a continuación:

Refinamiento por Subtipo

La política de refinamiento por subtipo se basa en el principio de sustitutibilidad. Una clase y una subclase cumplen este principio cuando los objetos del subtipo pueden ser tratados como si perteneciesen al supertipo, sin riesgo de comportamiento no-admisible. Para lograr esta propiedad en necesario realizar una especialización contravariante en el dominio de los métodos y covariante en el co-dominio. Esto significa que un método especializado tiene un co-dominio mas restringido, pero tiene un dominio más amplio. Este problema ha sido descripto formalmente (ver por ejemplo [Castagna 95], [Abadi and Cardelli 95], [Boyland and Castagna 96]).

Por otra parte el subtipo puede agregar nuevos métodos. En este caso, estados y transiciones son sólo agregados, no eliminados.

El refinamiento se interpreta de la siguiente forma:

- Un estado refinado conserva sus transiciones de entrada y de salida, pudiendo agregar transiciones nuevas. El conjunto de transiciones puede estar refinado. Puede además tener un conjunto mayor de subestados y puede cambiar su propiedad de concurrencia de falso a verdadero.

- Una transición refinada puede ir a un nuevo estado destino, el cual es un subestado del estado especificado en la clase base.

- Una guarda refinada tiene la misma condición pero puede agregar disyunciones. Esto garantiza que las pre-condiciones son más débiles.

- Una secuencia de acciones refinada contiene al menos las mismas acciones, y puede contener acciones adicionales.

La siguientes fórmulas expresan el concepto de refinamiento por subtipo, donde el predicado refinement(H,L) es verdadero cuando la máquina H es un refinamiento de la máquina L:

$\forall$H,L:StateMachine
refinement(H,L) $\leftrightarrow$ refinement(top(H),top(L))

$\forall$c:CompositeState,s:SimpleState
$\neg$refinement(s,c)

$\forall$p:State,s:SimpleState
refinement(p,s) $\leftrightarrow$ name(p)=name(s) $\wedge$ parent(p)=parent(s)

$\forall$p,q:CompositeState
refinement(q,p) $\leftrightarrow$ name(q)=name(p) $\wedge$ parent(q)=parent(p)
$\wedge$ (isConcurrente(p) $\rightarrow$isConcurrente(q))
$\wedge\forall$pi$\in$substates(p) $\exists$qi$\in$substates(q) refinement(qi,pi)
$\wedge\forall$t2$\in$internalTransitions(p) $\exists$t1$\in$internalTransitions(q) refinement(t1,t2)

$\forall$t1,t2:Transition
refinement(t1,t2) $\leftrightarrow$ source(t1)=source(t2) $\wedge$
(target(t1)=target(t2) $\vee$ target(t1)$\in$allSubstates(t2)) $\wedge$
trigger(t1)=trigger(t2) $\wedge$
guard(t2) $\rightarrow$guard(t1) $\wedge$
effect(t2)$\subseteq$ effect(t1)

Refinamiento por Herencia

La idea de esta política es alentar reuso de implementación en lugar de preservar el comportamiento. En este caso es posible agregar nuevas operaciones, y las operaciones existentes pueden ser redefinidas arbitrariamente, pero no eliminadas. El refinamiento se interpreta de la siguiente forma:

·• Un estado refinado conserva sus transiciones, pudiendo agregar transiciones nuevas. El conjunto de transiciones puede estar refinado. Puede además tener un conjunto mayor de subestados y puede cambiar su propiedad de concurrencia de falso a verdadero.

• Una transición refinada puede ir a un nuevo estado destino, pero debe conservar su estado fuente.

• Una guarda refinada puede tener una condición diferente.

• Una secuencia de acciones refinada puede contener otras acciones, pueden haberse eliminado acciones o modificado su secuencia.

La siguientes fórmulas expresan el concepto de refinamiento por herencia:

$\forall$H,L:StateMachine
hrefinement(H,L) $\leftrightarrow$ hrefinement(top(H),top(L))

$\forall$c:CompositeState,s:SimpleState
$\neg$hrefinement(s,c)

$\forall$p:State,s:SimpleState
hrefinement(p,s) $\leftrightarrow$ name(p)=name(s) $\wedge$ parent(p)=parent(s)

$\forall$p,q:CompositeState
hrefinement(q,p) $\leftrightarrow$ name(q)=name(p) $\wedge$ parent(q)=parent(p)
$\wedge$ (isConcurrente(p) $\rightarrow$isConcurrente(q))
$\wedge\forall$pi$\in$substates(p) $\exists$qi$\in$substates(q) hrefinement(qi,pi)
$\wedge\forall$t2$\in$internalTransitions(p) $\exists$t1$\in$internalTransitions(q) hrefinement(t1,t2)

$\forall$t1,t2:Transition
hrefinement(t1,t2) $\leftrightarrow$ source(t1)=source(t2) $\wedge$ trigger(t1)=trigger(t2)

Refinamiento General

En el caso general, estados y transiciones pueden ser agregados o eliminados. Refinamiento se interpreta libremente sin restricciones, es decir que no existen requerimientos formales sobre las propiedades y relaciones de los elementos de la máquina de estados base y la máquina refinada:

• Un estado refinado puede tener distinto conjunto de transiciones de entrada o salida, pudiendo agregar, modificar o eliminar transiciones. Puede además tener un conjunto distinto de sub-estados y puede cambiar su propiedad de concurrencia.

• Una transición refinada puede ir a un nuevo estado destino y puede partir de un nuevo estado fuente.

• Una guarda refinada puede tener una condición diferente.

• Una secuencia de acciones refinada puede contener otras acciones, pueden haberse eliminado acciones o modificado su secuencia.

La siguiente fórmula expresa el concepto de refinamiento general:

$\forall$H,L:StateMachine grefinement(H,L) $\leftrightarrow$ true

6.2 Detección de ambigüedades en UML

Existen conceptos en el lenguaje UML que se prestan a interpretaciones ambiguas. En otros casos, la interpretación no es ambigua pero resulta inconsistente. Mostraremos ejemplos significativos de este problema:

Reglas de simetría

En varios casos es necesario asegurar que la relación existente entre dos elementos de modelado representa la inversa de otra relación, por ejemplo:

- Un feature f es un feature de una clase c si y sólo si el dueño de ese feature es c:
  $\forall$f:Feature,c:Class (f$\in$features(c) $\leftrightarrow$owner(f)=c)

- e es un associationEnd ligado a una clase c si y sólo si el tipo de e es c:

  $\forall$e:AssociationEnd,c:Class (e$\in$associationEnds(c) $\leftrightarrow$type(e)=c)

- Una máquina de estados h define el comportamiento de una clase c si y sólo si el contexto de h es c:

  $\forall$h:StateMachine,c:Class (context(h)=c $\leftrightarrow$ behavior(c)=h)

  lo cual también puede ser expresado de la siguiente forma:

  $\forall$c:Class context(behavior(c))=c

- Una transición pertenece a las transiciones de salida de un estado *s* si y sólo si *s* es el origen de la transición. Lo mismo ocurre con las transiciones de entrada:
  $\forall$t:Transition$\forall$s:State (t$\in$outgoings(s)$\leftrightarrow$ source(t)=s $\wedge$ t$\in$incomings(s)$\leftrightarrow$ target(t)=s)

- En el nivel de los datos también es necesario expresar este tipo de requerimientos. Por ejemplo un linkEnd pertenece a una instancia i si y sólo si la instancia ligada al link es i:
  $\forall$l:LinkEnd,i:Instance (l$\in$linkEnds(i)$\leftrightarrow$ instance(l)=i)

Estos requerimientos no han sido definidos en el metamodelo de UML [UML 97 (b)], sin embargo es razonable que se satisfagan y en general los diseñadores los asumen como verdaderos.

Tipos de datos básicos vs. tipos de datos definidos por el usuario

El metamodelo de UML dentro del paquete Foundation define un sub-paquete llamado Core donde se define la metaclase Classifier con dos especializaciones principales: Class y DataType (ver figura 6.1). La particularidad de DataType es que sus instancias deben ser valores puros cuyas operaciones son funciones, es decir el estado de las instancias de DataType es inmutable. En contraposición a las instancias de Class las cuales son objetos cuyo estado puede modificarse.

Por otra parte, también dentro del paquete Foundation, se define un sub-paquete conteniendo los tipos de datos primitivos que se usan en el metamodelo, tales como Integer, String, Boolean, Multiplicity y tipos enumerativos como AggregationKind (cuyos valores son none, shared and composite), VisibilityKind (cuyos valores son public, protected, private), etc. Estos tipos de datos primitivos se definen como una especialización de la metaclase DataType (ver figura 6.2).

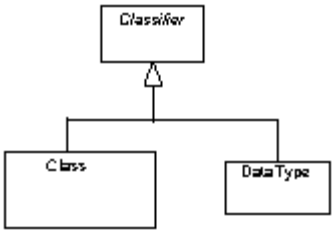


Figura 6.1: parte del package Core de UML.

Esto introduce ambigüedades y circularidades en el metamodelo. Consideremos la siguiente situación: un diseñador define el tipo de dato Integer instanciando la metaclase DataType. ¿Qué relación existe entre este tipo de dato definido por el usuario y la metaclase Integer? Claramente se encuentran a distinto nivel, ya que la primera es una instancia del metamodelo, mientras que la segunda es una metaclase perteneciente al metamodelo. Mas aún esto permitiría instanciar la metaclase Integer, lo cual carece de sentido ya que las instancias de Integer son los números 1,2,3,... y no clases del modelo. Por otro lado, también existe el problema de circularidad, ya que por ejemplo para definir la multiplicidad de Integer el diseñador debe usar Integer.

Como conclusión consideramos que no es correcto definir los tipos de datos primitivos como especializaciones de la metaclase DataType. En la M&D-theory hemos solucionado esta ambigüedad mediante la definición de una metaclase UML_DataType, cuyas especializaciones representan los tipos de datos básicos que se usan en el metamodelo. Esta metaclase no está relacionada con la metaclase DataType. En este contexto el tipo de dato Integer definido por un usuario instanciando la metaclase DataType no guarda ninguna relación con el tipo de datos primitivo Integer, lo cual elimina las mencionadas ambigüedad y circularidad en el metamodelo.

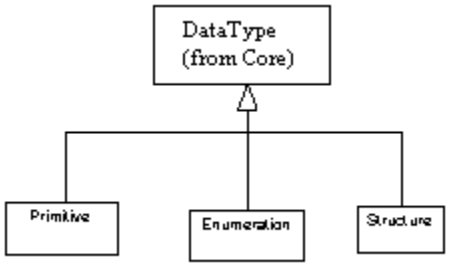


Figura 6.2: parte del package DataType de UML.

Significado de los diagramas de comportamiento

Tal como señalamos en la sección anterior, en general los diagramas estáticos de UML, tales como diagramas de clases y diagramas de relaciones poseen un significado claro y aceptado por los usuarios, mientras que los diagramas dinámicos tal como las máquinas de estados, diagramas de colaboración, diagramas de secuencia, diagramas de casos de uso constituyen elementos complejos cuya semántica suele ser mal interpretada por los ingenieros de software. En la sección anterior hemos mostrado que la lógica M&D puede usarse para definir formalmente el significado de uno de estos diagramas: las máquinas de estados.

6.3 Compatibilidad entre submodelos

Las distintas vistas (o submodelos) de un sistema, tales como diagramas de Clases, Statecharts y Constraints deben ser compatibles tanto a nivel sintáctico como semántico. La M&D-theory hace posible definir reglas de compatibilidad dado que provee un marco formal único donde las diferentes vistas del sistema coexisten. En esta sección presentaremos ejemplos de dichas reglas de compatibilidad:

Clases vs. Máquinas de Estados

Una StateMachine está relacionada con una clase (su context), y sólo puede hacer referencia a elementos de dicha clase. La siguiente formula formaliza este requerimiento:

$\forall$h:StateMachine syntactic-compatible(context(h),h)

donde $\forall$h:StateMachine $\forall$c:Classifier
syntactic-compatible(c,h) $\leftrightarrow \forall t \in$ allTransitions(h) syntactic-compatible(c,t)

y donde
$\forall$t:Transition $\forall$c:Classifier
syntactic-compatible(c,t) $\leftrightarrow$
trigger-compatible(c,trigger(t)) $\wedge$ guard-compatible(c,guard(t)) $\wedge \forall s \in$ effect(t) effect-compatible(c,s)

Donde un evento es trigger-compatible con una clase si dicho evento puede ser recibido por las instancias de la clase (i.e creación, destrucción o recepción de un mensaje), tal como se expresa a continuación:

$\forall$c:Class $\forall$e:Event
(trigger-compatible(c,e) $\leftrightarrow$ e=create $\vee$ e=destroy $\vee$ e=timeOut $\vee \exists$ op $\in$ allOperations(c) e=call(op))

Donde una guarda es guard-compatible con una clase si la expresión booleana de la guarda es sintácticamente compatible con la clase, tal como se expresa a continuación:

guard-compatible(c,g) $\leftrightarrow$ syntactic-compatible(c,expression(g))

Y donde un evento es effect-compatible con una clase si es un evento que puede ser disparado por las instancias de la clase (i.e creación, destrucción o envíos de mensajes a otros objetos dentro del alcance del emisor). Esta condición sólo puede verificarse en tiempo de ejecución ya que requiere evaluar las expresiones que denotan objetos.

Pre/post condiciones vs. Máquinas de Estados

En los diagramas de clases es posible expresar precondiciones y postcondiciones para las operaciones de una clase. Esta es una forma de especificar el comportamiento del sistema. Por otra parte las StateMachine también proveen un medio para especificar comportamiento. Resulta entonces necesario integrar ambas "vistas" garantizando su compatibilidad (o consistencia). La siguiente formula formaliza este requerimiento:

$\forall$<op,s,r,p>:Message (classifier(r)=owner(op) $\rightarrow$
(eval(precondition(op)[self: / r, parameters / p])=true
$\rightarrow$[r.<op,s,r,p>]eval(postcondition(op)[self / r, parameters / p])=true))

Esta fórmula establece que si las precondiciones de una operación op se cumplen antes de la ejecución de la operación, entonces las postcondiciones se cumplen luego de su ejecución. Recordemos que la semántica de la ejecución de la operación op está dada por la máquina de estados ligada a la clase del receptor del mensaje que provoca la ejecución de op (ver axiomas de recepción y procesamiento de mensajes en la clase Object en el capítulo 5).

Generalizaciones vs. otros elementos

La semántica de la relación de generalización (Generalization) hace que la presencia de un diagrama de generalización ejerza fuerte impacto sobre otros elementos de modelado. Por ejemplo si dos clases están relacionadas por generalización entonces el comportamiento de sus instancias (definido mediante máquinas de estado) debe cumplir ciertos requisitos: si c_1 es una subclase de c_2 entonces el comportamiento definido para c_1 debe ser un refinamiento del comportamiento definido para c_2. Esto queda expresado mediante la siguiente fórmula:

$\forall c_1,c_2$:Classifier ($IsA(c_1,c_2) \rightarrow$ refinement(behavior(c_1) , behavior(c_2)))

Una situación similar se presenta con respecto a los Constraints definidos en dos clases relacionadas por generalización: si c_1 es una subclase de c_2 entonces los Constraints que se definen sobre c_1 deben ser consistentes con los Constraints existentes en c_2, tal como se expresa en la siguiente fórmula:

$\forall c_1,c_2$:Classifier ($IsA(c_1,c_2) \rightarrow$ consistent(constraints(c_1) $\cup$constraints(c_2)))

Constraints vs. otros elementos

Dado que un concepto puede ser definido mediante más de un sub-modelo y dado que en cada submodelo es posible definir ciertos constraints, resulta necesario asegurar que la combinación de todos los constraints sea consistente, es decir:

$\forall$m:ModelElement consistent(allConstraints(m))

Además un constraint asociado a un elemento de modelado debe ser sintácticamente compatible, es decir debe hacer referencia a elementos dentro de un alcance determinado:

$\forall$m:ModelElement $\forall c \in$allConstraints(m) syntactic-compatible(m,c)

6.4 Deducción de información no explícita

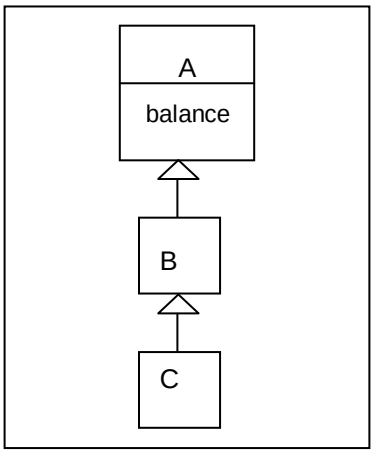
Uno de los elementos fundamentales de toda lógica es su aparato deductivo que consiste en una colección de reglas que pueden ser aplicadas sobre cierta información inicial para derivar información adicional, en una forma puramente mecánica.

Muchos tipos diferentes de sistemas deductivos han sido propuestos; uno de ellos es el estilo de deducción llamado Hilbert-style consistente en un conjunto de axiomas o sentencias del lenguaje que se asumen como verdaderas *a priori* y reglas de inferencia de la forma

$$\frac{\varphi 1\ \varphi 2\ \ldots\ \varphi n}{\psi}$$

A partir de las cuales nuevos teoremas pueden ser derivados. Las sentencias sobre la línea son llamadas *premisas* y la sentencia debajo de la línea es la *conclusión*.

En el capítulo 3 mostramos un sistema deductivo para la Lógica Dinámica que fue definido por el profesor Harel y sus colegas en [Harel et al.99]. En esta sección mostraremos un ejemplo de deducción. Sea spec el siguiente modelo UML:



Llamaremos ϕ_{INST} a la fórmula de instanciación de spec, es decir:

ϕ_{INST} = $\exists$a,b,c:Classifier

(name(a)=A $\wedge$ name(b)=B $\wedge$ name(c)=C

$\wedge\exists$f:Feature (f$\in$features(a) $\wedge$name(f)=balance) *[premisa1]*

$\wedge$ $\exists$g1,g2:Generalization supertype(g1)=a $\wedge$ subtype(g1)=b

$\wedge$ supertype(g2)=b $\wedge$ subtype(g2)=c)

Y sea $\phi_{UML'}$ el subconjunto de los axiomas de la M&D-theory (ver capítulo 5) que usaremos como premisas en esta deducción:

$\phi_{UML'}$ = $\forall$c:Classifier, $\forall$x:GeneralizableElement, $\forall$f:Feature

x$\in$supertypes(c) $\leftrightarrow$ x$\in$map supertype generalizations(c)

x$\in$allSupertypes(c) $\leftrightarrow$ x$\in$supertypes(c) $\cup$ ($\cup_{ci\in supertypes(c)}$ allSupertypes(ci))

f$\in$allFeatures(c) $\leftrightarrow$ $\exists$c´$\in$allSupertypes(c) f$\in$features(c´) *[premisa 2]*

***Teorema*:**

A partir de las premisas ϕ_{INST} y $\phi_{UML'}$ es posible deducir la siguiente conclusión, la cual constituye información no explícita en el modelo inicial: ”la clase C posse un atributo heredado llamado balance”.

***Demostración*:**

A partir de estas premisas fácilmente se deduce que:

$\exists$a,c:Classifier

(name(a)=A $\wedge$ name(c)=C $\wedge$ a$\in$allSupertypes(c))

ya que

a$\in$allSupertypes(c) $\leftrightarrow$ a$\in$supertypes(c) $\cup$ ($\cup_{ci\in supertypes(c)}$ allSupertypes(ci))

$\leftrightarrow$ a$\in$ {b}$\cup$allSupertypes(b)

$\leftrightarrow$ a$\in$allSupertypes(b)
$\leftrightarrow$ a$\in$supertypes(b) $\cup$ ($\cup_{ci\in supertypes(b)}$ allSupertypes(ci))
$\leftrightarrow$ a$\in${a}$\cup$allSupertypes(a)
$\leftrightarrow$ ***true***

Luego, reuniendo esta información con la *[premisa 1]* se tiene:

$\exists$a,c:Classifier

(name(a)=A $\wedge$name(c)=C$\wedge$a$\in$allSupertypes(c)$\wedge\exists$f:Feature(f$\in$features(a) $\wedge$name(f)=balance))

Dado que se comprobó la existencia de la clase A perteneciente a allSupertypes de la clase C y se comprobó que A posee un feature llamado balance es posible usar la *[premisa 2]* para deducir:

$\exists$c:Classifier

(name(c)=C $\wedge$ $\exists$f:Feature (f$\in$allFeatures(c) $\wedge$name(f)=balance))

Es decir que existe una clase llamada C, la cual posse un atributo heredado llamado balance. □

6.5 Verificación de la correctitud del sistema

El lenguaje formal que hemos definido permite expresar reglas de buena formación tanto del modelo como de los datos del sistema. Dado un sistema particular es posible determinar si cumple con estas reglas. A continuación mostramos algunas reglas de buena formación (las cuales ya han sido presentadas en el capítulo 5) y luego damos un ejemplo donde se prueba la no-correctitud de un sistema es decir la violación de alguna regla de buena formación.

6.5.1 Reglas de buena formación del modelo

[1] Dentro de un Classifier los nombres de los atributos son únicos:

$\forall$c:Classifier $\forall$f,g$\in$attributes(c) name(f) = name(g) $\rightarrow$ f=g

[2] Las operaciones de un Classifier tienen signaturas distintas dos a dos:
$\forall$c:Classifier $\forall$f,g$\in$operations(c) hasSameSignature(f,g)) $\rightarrow$f = g

[3] Herencia circular no está permitida:
$\forall c_1,c_2$:Classifier *IsA*(c_1,c_2) $\wedge$*IsA*(c_2 ,c_1) $\rightarrow$ $c_2 = c_1$

[4] La raiz de una jerarquía de herencia no puede generalizarse:
$\forall$g:Generalization $\neg$isRoot(subtype(g))

[5] La transición inicial del nivel más alto de una máquina de estados debe tener una acción de creación como trigger. En los demás caso, las transiciones iniciales no tienen trigger.
$\forall$t:Transition
kind(source(t))=#initial $\rightarrow$
(trigger(t)=nullElement $\vee$ (isTop(parent(source(t))) $\wedge$ trigger(t)=create))

[6] source y target de una transitions deben tener el mismo estado padre:
$\forall$t:Transition parent(source(t)) =parent(target(t))

6.5.2 Reglas de buena formación de los datos

[1] Los AttributeLinks de las instancias se corresponden con las declaraciones en su Classifier:
$\forall$i:Instance $\forall$l:AttributeLink(l$\in$slots(i) $\leftrightarrow$attribute(l)$\in$allAttributes(classifier(i)))

[2] Los valores de los atributos de las instancias respetan el tipo definido en su Classifier:
$\forall$a:AttributeLink IsA(classifier(value(a)) , type(attribute(a)))

[3] Una instancia no puede pertenecer por composición a mas de una instancia compuesta:
$\forall$i:Instance
$\exists$e1,e2$\in$oppositeLinkEnds(i)((aggregation(associationEnd(e1))=#composite $\wedge$
aggregation(associationEnd(e2))=#composite) $\rightarrow$ e1=e2)

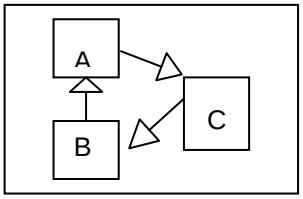
[4] Todas las instancias deben satisfacer los Constraints definidos en su clase:
$\forall$i:Instance $\forall$c$\in$allConstraints(classifier(i)) (eval(c)[self:=i] = true)

[5] la operación invocada por un mensaje debe pertenecer a la interfaz del receptor del mensaje:
specification(m)$\in$operations(classifier(receiver(m)))

6.5.3 Ejemplos de Verificación

Ejemplo 1: ***Errores de diseño***

Sea spec el siguiente modelo UML. Puede observarse que presenta un problema de herencia circular. Es posible demostrar que sem(spec)=$\varnothing$, es decir que spec es inconsistente con las reglas de buena formación definidas en la teoría. En particular la fórmula de instanciación del modelo es inconsistente con la regla de buena formación de la jerarquía de herencia (la regla número [3]).



Llamaremos φ a la regla de buena formación, es decir:

$\varphi = \forall c_1,c_2$:Classifier $IsA(c_1,c_2) \wedge IsA(c_2,c_1) \rightarrow c_2 = c_1$

donde el predicado IsA involucra las siguientes definiciones (ver capítulo 5):

$\forall$c,c1:Classifier, $\forall$x:GeneralizableElement, $\forall$f:Feature

x$\in$supertypes(c) $\leftrightarrow$ x$\in$map supertype generalizations(c)

x$\in$allSupertypes(c) $\leftrightarrow$ x$\in$supertypes(c) $\cup$ ($\cup_{ci\in supertypes(c)}$ allSupertypes(ci))

IsA(c,c1) $\leftrightarrow$ c=c1 $\vee$ c1$\in$allSupertypes(c)

Y llamaremos ϕ_{INST} a la fórmula de instanciación del modelo, es decir:

ϕ_{INST} = $\exists$a,b,c:Classifier

(name(a)=A $\wedge$ name(b)=B $\wedge$ name(c)=C

$\wedge$ $\exists$g1,g2,g3:Generalization supertype(g1)=a $\wedge$ subtype(g1)=b $\wedge$ supertype(g2)=c

$\wedge$ subtype(g2)=a $\wedge$ supertype(g3)=b $\wedge$ subype(g3)=c

$\wedge$g1$\in$specializations(a) $\wedge$ g1$\in$generalizations(b) $\wedge$ g2$\in$specializations(c)

$\wedge$ g2$\in$generalizations(a) $\wedge$ g3$\in$specializations(b) $\wedge$ g3$\in$generalizations(c))

Teorema: el modelo spec es inconsistente con la regla de buena formación, es decir:

($\varphi \wedge \phi_{INST}$) $\rightarrow$ ***false***

Demostración:

A partir de ϕ_{INST} puede deducirse que $\exists$a,b:Classifier a$\neq$b $\wedge$ *IsA*(a,b) $\wedge$ *IsA*(b,a), de la siguiente forma:

Sean a,b,c los classifiers definidos mediante el modelo spec,

supertypes(b) = map supertype generalizations(b) = map supertype {g1} ={supertype(g1)} = {a}

supertypes(a) = map supertype generalizations(a) = map supertype {g2} ={supertype(g2)} = {c}

supertypes(c)= map supertype generalizations(c) = map supertype {g3} ={supertype(g3)} = {b}

Por lo tanto:

$IsA(b,a) \leftrightarrow b=a \vee a \in allSupertypes(b)$
$\leftrightarrow a \in allSupertypes(b)$
$\leftrightarrow a \in supertypes(b) \cup (\cup_{ci \in supertypes(b)} allSupertypes(ci))$

$\leftrightarrow a \in \{a\} \cup allSupertypes(a)$
$\leftrightarrow$ ***true***

y también:

$IsA(a,b) \leftrightarrow a=b \vee b \in allSupertypes(a)$
$\leftrightarrow b \in allSupertypes(a)$
$\leftrightarrow b \in supertypes(a) \cup (\cup_{ci \in supertypes(a)} allSupertypes(ci))$

$\leftrightarrow b \in \{c\} \cup allSupertypes(c)$

$\leftrightarrow b \in allSupertypes(c)$
$\leftrightarrow b \in supertypes(c) \cup (\cup_{ci \in supertypes(c)} allSupertypes(ci))$

$\leftrightarrow b \in \{b\} \cup allSupertypes(b)$

$\leftrightarrow$ ***true***

Por lo tanto existen dos classifiers distintos, a y b, tales que *IsA*(a,b) y también *IsA*(b,a), lo cual es la negación de φ, es decir $\phi_{INST} \rightarrow \neg\varphi$. Luego es directo demostrar que $\varphi \wedge \phi_{INST} \rightarrow \varphi \wedge \neg\varphi$. Lo cual es equivalente a $\varphi \wedge \phi_{INST} \rightarrow$ ***false***. □

Ejemplo 2: ***Valores no válidos***

Consideremos la especificación (llamada spec) del sistema bancario del capítulo 5. Sea $U=(S^U, w_o, m_U)$ un Kripke-frame que es un modelo para la especificación, es decir $U \in sem(spec)$.

Es posible demostrar que cualquier programa que fuerce asignaciones de valores no válidos (por ejemplo que no respeten los tipos definidos en los Classifiers) no termina sobre este modelo. Por ejemplo, sea w un estado cualquiera de U, es bastante directo probar el siguiente teorema:

***Teorema*:**

$U,w \models \forall o$:Object (name(classifier(o))=BankAccount $\rightarrow$ [update(o,balance,"astring")]***false***)

es decir que no existe ningún w' tal que $w' \in S^U$ y $(w,w') \in m_U$(update(o,balance, "astring"))

Demostración:
Dado que el frame U es un modelo para la especificación UML del sistema bancario, las siguientes fórmulas son válidas en todos los estados de U:

[1] Fórmulas de instanciación:
$\exists c_1$:Class name(c_1)=BankAccount
$\wedge \exists$a:Attribute(a$\in$attributes(c1) $\wedge$ name(a)=balance type(a)=Real)

[2] Fórmula que especifica el efecto de la acción update:
$\forall$o:Object,n:Name,v:Instance [update(o,a,v)]value(o,a)=v

[3] Definición de la función que retorna el valor de un atributo de una instancia:

$\forall$i:Instance,n:Name,v:Instance (value(i,n)=v $\leftrightarrow$ $\exists$l$\in$slots(i) value(l)=v $\wedge$ name(attribute(l))=n)

[4] Regla de buena formación de los datos : los valores de los atributos de las instancias respetan el tipo definido en su Classifier:
$\forall$a:AttributeLink IsA(classifier(value(a)) , type(attribute(a)))

Vamos a suponer que

U,w $\models$ $\exists$o:Object (name(classifier(o))=BankAccount $\wedge$ $\langle$update(o,balance,"astring"$\rangle$***true***)

es decir que existe algún w´ tal que w´ $\in S^U$ y (w,w´) $\in m_U$(update(o,balance, "astring")) y llegaremos a una contradicción, lo cual dejará demostrada la validez del teorema:

Por la fórmula [2] tenemos que
U,w´ $\models$ $\exists$o:Object (name(classifier(o))=BankAccount $\wedge$ value(o,balance)="astring")

Por la definición [3] tenemos que
U,w´ $\models$ $\exists$o:Object (name(classifier(o))=BankAccount $\wedge$ $\exists$l$\in$slots(o) value(l)="astring" $\wedge$ name(attribute(l))=balance)

Por el axioma de instanciación [1] sabemos que
U,w´ $\models$ $\exists$o:Object (name(classifier(o))=BankAccount
$\wedge$ $\exists$l$\in$slots(o) value(l)="astring" $\wedge$ name(attribute(l))=balance $\wedge$ type(attribute(l))=Real)

Es decir que
U,w´ $\models$ $\exists$l:AttributeLink (value(l)="astring" $\wedge$ type(attribute(l))=Real)

Y dado que classifier("astring") es String, y dado que $\neg$*IsA*(String,Real), tenemos que:
U,w´ $\models$ $\exists$a:AttributeLink $\neg$*IsA*(classifier(value(a)), type(attribute(a)))

Lo cual contradice la regla [4] de buena formación que expresa que:
U,w´ $\models$ $\forall$a:AttributeLink *IsA*(classifier(value(a)), type(attribute(a)))
□

6.6 Conclusiones

Para utilizar adecuadamente las herramientas de modelado es necesario conocer claramente su significado. En general los diagramas estáticos de UML, tales como diagramas de clases y diagramas de relaciones poseen un significado claro y aceptado por los usuarios, mientras que los diagramas dinámicos tal como las máquinas de estados constituyen elementos complejos cuya semántica suele ser mal interpretada por los ingenieros de software. En este capítulo hemos usado la lógica para definir formalmente el significado de algunos de estos diagramas.

La construcción del modelo formal nos permitió descubrir ambigüedades e inconsistencias en el lenguaje UML. Hemos mostrado ejemplos significativos de este problema.

Por otra parte, las distintas vistas de un sistema (class diagrams, statecharts, constraints, etc.) son integradas en un único modelo formal. Esta integración permite definir reglas de compatibilidad entre las vistas, tanto en el nivel sintáctico como en el semántico, dado que provee un marco formal único donde las diferentes vistas del sistema coexisten. Hemos mostrado ejemplos de dichas reglas de compatibilidad.

Con respecto a las herramientas formales, uno de los elementos fundamentales de toda lógica es su aparato deductivo que consiste en una colección de reglas que pueden ser aplicadas sobre cierta información inicial para derivar información adicional, en una forma puramente mecánica. Mediante mecanismos formales de deducción hemos mostrado que es posible obtener información no presentada explícitamente en una especificación. Además, el lenguaje formal que hemos definido permite expresar reglas de buena formación tanto del modelo como de los datos del sistema. Dado un sistema particular técnicas formales de verificación disponibles en el modelo formal pueden usarse para determinar si cumple con estas reglas. Hemos mostrado un ejemplo donde se prueba la no-correctitud de un sistema es decir la violación de alguna regla de buena formación.

Finalmente, la teoría lógica subyacente hace que el lenguaje gráfico se convierte en un lenguaje formal y por lo tanto las especificaciones escritas utilizando el lenguaje gráfico pueden ser formalmente analizadas para detectar contradicciones y ambigüedades tempranamente en el proceso de desarrollo del software. Una de las claves para el éxito de esta propuesta reside en ocultar la notación matemática tanto como sea posible tras la notación gráfica. Por ejemplo, debería ser posible utilizar la semántica formal para desarrollar herramientas CASE. Sólo los desarrolladores del lenguaje deberían usar el formalismo para construir las herramientas CASE y justificar su correctitud, mientras que los desarrolladores de software de aplicación podrían manejar los modelos gráficos sin necesidad de conocer el formalismo matemático subyacente.

7. Formalizando evolución

7.1 Introducción

Una técnica aceptada y ampliamente usada en ingeniería de software es la combinación de diferentes modelos (o vistas) para la descripción de un sistema de software. El beneficio primario de esta técnica consiste en que cada modelo se concentra sobre determinados aspectos relacionados, ignorando los restantes. El uso de diferentes modelos clarifica las diferentes características del sistema, pero al mismo tiempo es necesario considerar que estos modelos no son independientes entre sí y que su semántica tiene aspectos superpuestos. Por lo tanto, las relaciones existentes entre los modelos deben ser precisamente definidas, de manera que sea posible determinar si las diferentes vistas del sistema son semánticamente compatibles y si satisfacer los requerimientos de integración.

La compatibilidad entre las distintas vistas debe preservarse aún si el sistema cambia (o evoluciona). En nuestros días los sistemas de software deben ser flexibles y adaptables, dado que los requerimientos del mercado poseen una naturaleza altamente dinámica. Como consecuencia de este dinamismo, en los últimos años ha surgido el concepto de *Modelo Dinámico*. Un sistema con un modelo dinámico posee un modelo explícito que interpreta en tiempo de ejecución. Esto permite que las modificaciones sobre el modelo sean inmediatamente adoptadas por el sistema. Existen numerosas formas en las cuales un modelo puede ser adaptado (o modificado). La técnica de modificación incremental consiste en derivar un nuevo modelo a partir de un modelo original mediante la aplicación de operaciones de modificación. Cada operación realiza adaptaciones específicas sobre el modelo.

Las modificaciones realizadas sobre un modelo pueden afectar a los demás modelos que están relacionados con él. Por ejemplo ciertas propiedades del modelo original asumidas como válidas por los demás modelos pueden ya no cumplirse. Esta clase de problemas ha sido denominada conflictos de evolución (evolution conflicts) y ha sido analizada por diferentes investigadores (ver por ejemplo [Steyaert et al. 96, Lucas 97, Mens et al. 98]).

Por lo tanto, resulta necesario contar con mecanismos de evolución que garanticen la consistencia del sistema a través del proceso de evolución. Para lograr este objetivo, los mecanismos de evolución deben proveer los siguientes servicios:

- Identificación de modificaciones primitivas.
- Definición formal de las operaciones de evolución, incluyendo condiciones de aplicabilidad y propagación de cambios.
- Considerar que el Modelo puede evolucionar en tiempo de ejecución (es decir, contemplar *Modelos Dinámicos).*
- Identificación de situaciones conflictivas originadas por la aplicación de operaciones de evolución.
- Identificación de patrones de evolución (es decir grupos de modificaciones primitivas que se presentan juntas frecuentemente).

En este capítulo usaremos la M&D-logic como un mecanismo de evolución con semántica formal. En la sección 7.2 describimos el metamodelo dinámico formalizando operaciones de evolución primitivas sobre modelos UML. En la sección 7.3 analizamos el problema de los conflictos de evolución y definimos reglas para detectar automáticamente la presencia de tales conflictos. La sección 7.4 contiene un resumen de trabajos relacionados. Finalmente, en la sección 7.5 presentamos nuestras conclusiones.

7.2 La teoría dinámica

Dicotomía de entidades

Recordemos que la M&D-theory en utiliza un dominio semántico que integra los dos niveles inferiores de la arquitectura de las notaciones de modelado (es decir el nivel del modelo y el nivel de los datos), permitiendo de esta manera representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden.

Las entidades descriptas por la M&D-theory se clasifican en dos conjuntos disjuntos:

- Entidades de modelado
- Entidades modeladas

Las entidades de modelado se corresponden con construcciones (sintácticas) correctas del lenguaje UML, tales como clases (Class) y a máquinas de estados (StateMachine). En contraste, las entidades modeladas, tales como objetos (Object) y conexiones (Link) representan los datos del sistema modelado.

Dicotomía de Evolución

Paralelamente a la clasificación de las entidades existe una clasificación de las acciones de evolución que se aplican sobre dichas entidades:

- evolución del modelo
- evolución de los datos

La especificación (o modelo) de un sistema puede evolucionar por diversas razones, a lo largo de su ciclo de vida. Una de las formas más comunes de evolución es la que involucra cambios estructurales, tales como agregar nuevas clases de objetos, agregar o eliminar atributos de clases existentes, modificar la jerarquía de herencia, etc. En el otro extremo encontramos formas de evolución que modifican no sólo la estructura del sistema, sino que modifican el comportamiento especificado para los objetos. Los cambios de comportamiento se reflejan en las modificaciones realizadas sobre diagramas de comportamiento, tales como máquinas de estado.

Por otra parte, en el nivel de los datos la evolución es provocada por tres clases diferentes de acciones: *modification* (en general callActions que representan la recepción de un mensaje, lo cual causa que una operación sea invocada en el objeto receptor; la ejecución de una operación puede causar modificaciones en el estado interno del receptor así como también el envío de mensajes a otros objetos); *creation* (estas acciones provocan la creación de una nueva instancia de una clase); *cancellation* (estas acciones provocan que instancia deje de existir en el sistema).

Acciones de evolución

Formalizaremos a las acciones de evolución como acciones en la lógica dinámica M&D. Es decir que cada acción de evolución se representa mediante un término de sort Action. Esto nos permitirá expresar y analizar propiedades de las acciones.

Además dado que la M&D-logic es order-sorted (es decir, existe una relación de orden parcial entre los sorts), utilizaremos esta característica para definir una jerarquía de acciones cuya raíz es el sort Action. De esta forma obtendremos una especificación jerárquica de las acciones de evolución; por ejemplo: la sentencia ModelEvolution$\leq$Action implica que si se ha demostrado que la propiedad P se cumple para todo α de tipo Action, entonces dicha propiedad P también se cumple para todo β de tipo ModelEvolution. Pero, por otra parte es posible especificar propiedades (más específicas)

que sean válidas para acciones de tipo ModelEvolution y que no sean válidas para todas las acciones de tipo Action.

7.2.1 Granularidad de las acciones de evolución

Las acciones de evolución se clasifican en dos categorías (ver figura 7.1):

- PrimitiveActions (acciones primitivas)
- CompositeActions (acciones compuestas)

Las acciones primitivas representan modificaciones atómicas sobre un modelo, mientras que las acciones compuestas representan grupos de modificaciones que se aplican conjuntamente. Una acción compuesta representa una transición ininterrumpible que puede violar temporariamente la consistencia del sistema, sin embargo luego de su finalización todas las inconsistencias generadas deben haber sido subsanadas.

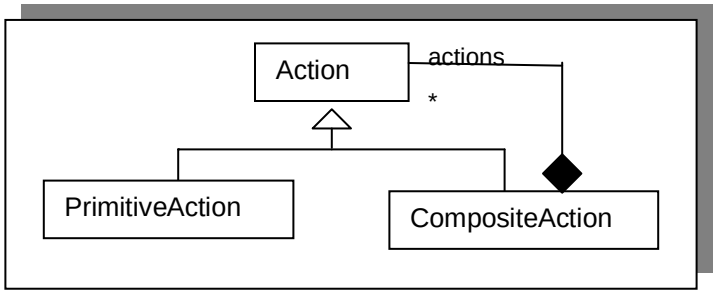


Figura 7.1: Jerarquía de acciones

7.2.2 Clasificación de acciones de evolución primitivas

De acuerdo con la dicotomía existente entre las entidades del sistema, proponemos una jerarquía de acciones formada por dos grupos de acciones primitivas.

- **model evolution** (acciones que modifican el modelo)
- **data evolution** (acciones que modifican los datos)

Por otra parte, realizamos una subdivisión de las acciones primitivas teniendo en cuenta la clase de modificación que realizan. Por ejemplo, algunas acciones agregan información, mientras que otras acciones eliminan información. De acuerdo con este criterio definimos tres grupos de acciones primitivas:

- **Creation Actions** (acciones que agregan un nuevo elemento, ya sea en el modelo o en los datos)
- **Cancellation Actions** (acciones que eliminan un elemento existente, ya sea del modelo o de los datos)
- **Modification Actions** (acciones que modifican un elemento, ya sea del modelo o de los datos)

La figura 7.2 contiene la jerarquía de acciones primitivas. Notemos que las dos ramas de la generalización no son disjuntas y por lo tanto a partir su combinación es posible obtener seis subclases de acciones primitivas; estas son: ModelEvolution-Creation, ModelEvolution-Cancellation, ModelEvolution-Modification, DataEvolution-Creation, DataEvolution-Cancellation, DataEvolution-Modification.

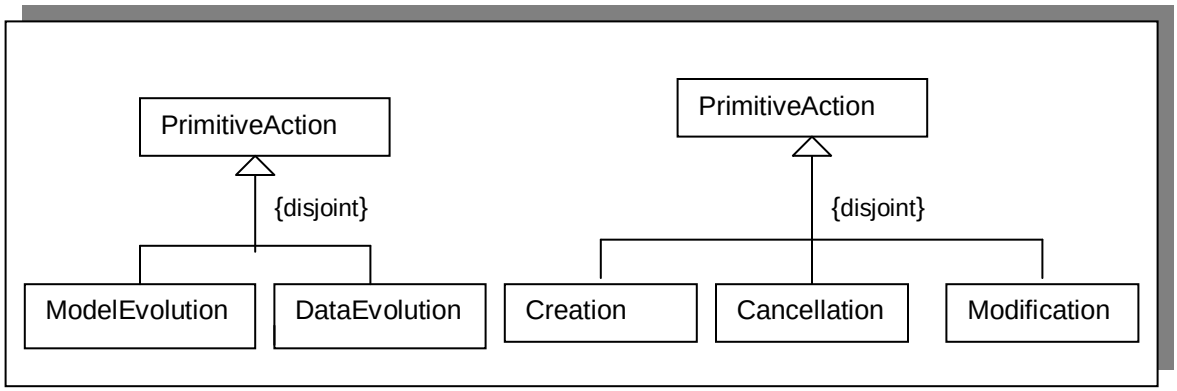


Figura 7.2: jerarquía de acciones primitivas

Es importante destacar las diferencias entre una acción primitiva aplicada sobre una entidad del nivel del modelo y una acción primitiva aplicada sobre una entidad del nivel de los datos. Por ejemplo:

- Mientras que creación en el nivel de los datos (DataEvolution-Creation) significa la creación de un nuevo ítem de datos u objeto (e.g. Peter) como instancia de algún elemento en el nivel del modelo (e.g. Employee), una creación en el nivel del modelo (ModelEvolution-Creation) significa la definición de una nueva vista (o sub-modelo) del sistema (e.g. una nueva Clase) como instancia de una entidad en el metamodelo (e.g. la metaclase Class).
- Las modificaciones sobre el nivel del modelo (ModelEvolution-Modification) significan modificaciones sobre la especificación del sistema, mientras que las modificaciones sobre el nivel de los datos (DataEvolution-Modification) constituyen el comportamiento dinámico del sistema durante su ejecución.

Aún luego de esta sub-clasificación la diversidad de acciones primitivas en cada grupo es amplia. Por lo tanto hemos considerado apropiado continuar refinando la jerarquía de acciones primitivas. Así hemos clasificado a las acciones de evolución del modelo de acuerdo con la clase de elemento de modelado que está siendo modificado. De esta forma hemos definido tres nuevos grupos:

- **Structural Actions** representan la evolución de elementos que modelan la estructura del sistema, tales como clases, asociaciones o generalizaciones.
- **Behavioral Actions** representan la evolución de elementos que modelan el comportamiento del sistema, tales como máquinas de estados.
- **Constraint Actions** representan la evolución de las reglas (o constraints) que restringen al sistema.

La ub-clasificación de las acciones de evolución del modelo se muestra en la figura 7.3. Por otra parte hemos considerado que no es necesario refinar la jerarquía de acciones de evolución de los datos (es decir que no hemos definido subclases de DataEvolution).

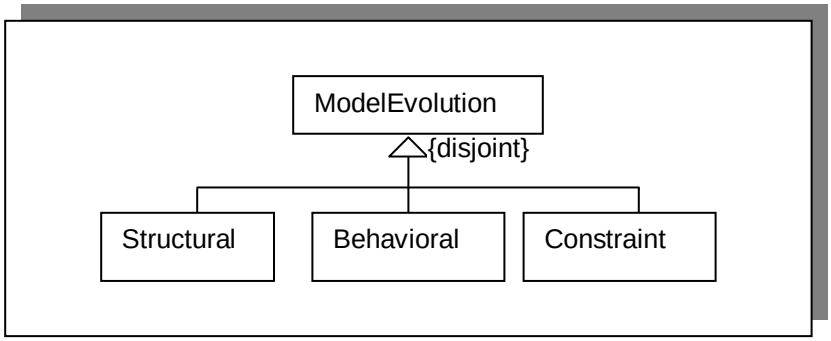


Figura 7.3*:* jerarquía de Model Evolution Actions

A partir de las jerarquías definidas podemos obtener doce subclases de acciones. La siguiente tabla contiene estas subclases:

		Creation	Cancellation	Modification
Model Evolution	Structural	Structural-Creation	Structural-Cancellation	Structural-Modification
	Behavioral	Behavioral- Creation	Behavioral-Cancellation	Behavioral-Modification
	Constraint	Constraint- Creation	Constraint-Cancellation	Constraint-Modification
DataEvolution		Data-Creation	Data-Cancellation	Data-Modification

La tabla de la figura 7.4 muestra las principales acciones incluidas en cada grupo. La signatura y la especificación básica de cada una de estas acciones se encuentra en el capítulo 5, como parte de la definición de la M&D-theory. Sin embargo, por razones de claridad, en dicho capítulo no hemos incluido la especificación completa de las operaciones, la cual será presentada luego.

	Creation	Cancellation	Modification
Structural	-adding a new Subpackage, Classifier or Relationship to a Package addSubpackage addClassifier: addAssociation add Generalization -adding new features to a Classifier, addFeature -adding new associacionEnds to an association, addConnection	- deleting an existing Subpackage, Classifier or Relationship from a Package deleteSubpackage deleteClassifier delete Association delete Generalization -deleting an existing feature from a Classifier, deleteFeature -deleting an existing associacionEnds from an association, deleteConnection	Renaming a model Element setName -modifying the characteristics of an existing association. setAggregation setChangeable setMultiplicity setTargetScope setQualifier setOrdered setNavigable -modify an existing feature of a classifier setOwnerScope setVisibility

	Creation	Cancellation	Modification
			-modify an existing attribute of a classifier setInitialValue setChangeable setMultiplicity setTargetScope -modify the signature of an existing operation. addParameter deleteParameter - add or delete a referenced element into a Package addReferencedElement deleteReferencedElement
Behaviorral	-Agregar una StateMachine en el modelo asociándola a una clase. setBehavior	-Deleting a StateMachine form the model cancellBehavior: Class	-Modifying the behavioral characteristics of an operation. setPrecondition setPostcondition setImplementation -Modifying an existing StateMachine from the model addSubState deleteSubState addInternalTransition deleteInternalTransition - modificando transitions setTrigger setGuard setEffect
Constraint	Attaching a new constraint to a model element addConstraint	Detaching an existing constraint from a model element deleteConstraint	Modifying an existing constraint from the model setBody
Data	Creation of objects newObject newLink	Destruction of objects and connections destroy	Modification of objects: local invocations and call actions update -.-

Figura 7.4: principales acciones de evolución

7.2.3 Especificación de las acciones de evolución primitivas

En la M&D-theory, cada acción de evolución se define mediante dos fórmulas:

- **Precondiciones necesarias** para describir las condiciones de aplicabilidad de la acción. La fórmula □(⟨*op*⟩*true* →*cond*) indica que durante toda la vida del sistema, la acción *op* es aplicable sólo si la condición *cond* es verdadera.
- **Postcondiciones suficientes** para describir el efecto (efecto directo y propagación de cambios) de la acción. La fórmula □([*op*]*cond*) establece que luego de la aplicación de la acción *op* la condición *cond* es verdadera.

Estas fórmulas se construyen sobre términos de sort Element (representando elementos de modelado) así como también sobre términos de sort Data (representando los datos del sistema) permitiendo de esta manera expresar:

- Propagación de cambios Intra-nivel: como una modificación realizada sobre una entidad de datos impacta sobre los demás datos, y como una modificación realizada sobre un modelo impacta sobre los demás modelos.

- Propagación de cambios Inter-nivel: como una modificación realizada sobre una entidad de datos impacta sobre sus modelos, y como una modificación realizada sobre un modelo impacta sobre los datos modelados.

La especificación de cada acción de evolución consta de tres cláusulas:

- ***Precondiciones***. Son las condiciones bajo las cuales la acción es aplicable.
- ***Efecto***. La acción de evolución se aplica sobre una entidad en particular que llamaremos source. Como resultado de la acción el source sufre modificaciones. Estas modificaciones son el efecto directo de la acción.
- ***Propagación***. Además del efecto directo, otras modificaciones (generalmente sobre elementos relacionados al source) pueden ser necesarias con el objetivo de mantener la consistencia del sistema (asegurar el cumplimiento de los axiomas de buena formación o well-formedness axioms).

Una precondición que siempre se asume como válida es la buena formación de los elementos, es decir, se asume que el modelo sobre el cual se aplica la acción, así como también los parámetros de la acción satisfacen los axiomas de buena formación definidos en la especificación $Spec_{UML}$. Es posible demostrar que las acciones de evolución preservan la buena formación de los elementos. Es decir que si se parte de un sistema bien formado y se le aplica una acción de evolución, siempre se obtendrá un modelo bien formados como resultado.

El formato de la especificación es el siguiente

Action act

Precondition τ

Effect γ

Propagation δ

Y representa a la siguiente fórmula dinámica:

$\Box$ (($\langle$act$\rangle$*true* $\rightarrow\tau$) $\wedge$ ([act] ($\gamma\wedge\delta$)))

7.2.3.1 Ejemplos de especificación de acciones primitivas

En esta sección introducimos la especificación de las principales acciones de evolución primitivas. Las fórmulas se asumen cuantificadas por:

$\forall$p,q:Package, c:Classifier, r:Relationship, g:Generalization, a:Association, f:Feature, f1:Attribute, f2:Operation, e:AssociationEnd, h:StateMachine, i:Constraint, m:ModelElement, t:Transition, s1,s2:State

Structural Creation

Action addSubPackage(p,q)
Precondition
[1] The new subpackage should not exist previously. (as a consequence: q$\notin$allContents(p))
Exists(p) $\wedge$ $\neg$Exists(q)
[2] The new package must be empty
contents(q)=$\varnothing$
[3] in a Package the Package names are unique
$\forall$q1:Package (q1$\in$contents(p) $\rightarrow$name(q_1) $\neq$ name(q))
Effect
Exists(q) $\wedge$ q$\in$ownedElement(p) $\wedge$ package(q)=p
Propagation
no propagation.

Action addClassifier(p,c)
Precondition
[1]The source Package exists and the new Classifier does not exist (as a consequence, c$\notin$allContents(p))
Exists(p) $\wedge$ $\neg$Exists(c)
[2] in a Package the Classifier names are unique
$\forall c_1$: Classifier ($c_1\in$contents(p) $\rightarrow$ name(c_1) $\neq$ name(c))
[3] the new Classifier does not participate in any relationship.
AssociationEnds(c)=$\varnothing$ $\wedge$ generalizations(c)=$\varnothing$ $\wedge$ specializations(c)=$\varnothing$
[4] The type of the attributes must be included in the Package.
$\forall$f$\in$allAttributes(c) type(f)$\in$allContents(p)
[5] The type of the Parameters must be included in the Package .
$\forall$f$\in$allOperations(c) $\forall$m$\in$parameters(f) type(m)$\in$allContents(p)
Effect
Exists(c) $\wedge$c$\in$ownedElement (p) $\wedge$ package(c)=p
Propagation

[1]Life dependency
$\forall f \in features(c) Exists(f)$

Action addAssociation(p,a)
Precondition
[1]The source Package exists and the new Relationship does not exist (as a consequence, $a \notin allContents(p)$ y $\forall c \in allConnectedElements(a)\ a \notin allAssociations(c)$)
$Exists(p) \wedge \neg Exists(a)$
[2] all elements connected by the new relationship must be included in the Package.
$\forall c \in allConnectedElements(a)\ c \in allContents(p)$
[3] in a Package the association names are unique
$\forall a_1$: Association ($a_1 \in contents(p) \rightarrow name(a_1) \neq name(a)$)
[4]No opposite AssociationEnds may have the same rol-name within a Classifier
$\forall c \in allConnectedElements(a)\ (\forall e1 \in allOppositeAssociationEnds(c)\ \forall e2 \in connections(a)\ name(e1) \neq name(e2)$)
Effect
$Exists(a) \wedge a \in ownedElement(p) \wedge package(a)=p$
Propagation
[1]Life-dependency.
$\forall e \in connections(a)\ Exists(e)$
[2]The association is connected to the classifiers
$\forall e \in connections(a)\ e \in associationEnds(type(e))$

Action addGeneralization(p,g)
Precondition
[1] The source Package exists and the new Relationship does not exist (as a consequence, $g \notin allContents(p)$)
$Exists(p) \wedge \neg Exists(g)$
[2] all elements connected by the new relationship must be included in the Package.
$supertype(g) \in allContents(p) \wedge subtype(g) \in allContents(p)$
[3] A root cannot have any Generalizations.
$\neg isRoot(subtype(g))$
[4] No GeneralizableElement which is a leaf can have a subtype
$\neg isLeaf(supertype(g))$
[5] Circular inheritance.
$IsA(supertype(g), subtype(g)) \rightarrow supertype(g) = subtype(g)$
[6] multiple inheritance.
$\forall c$:Classifier ($IsA(subtype(g),c) \rightarrow$
$\forall f,g$:Feature($(f \in allFeatures(supertype(g)) \wedge g \in allFeatures(c) \wedge name(f)=name(g)) \rightarrow f=g$))
[7] Constraint consistency
$consistent(allConstraints(subtype(g)) \cup allConstraints(supertype(g))) \wedge$
$\forall c \in subtypes(subtype(g))\ consistent(allConstraints(c) \cup allConstraints(supertype(g)))$
[8] Behavioral consistency
$refinement(behavior(subtype(g)), behavior(supertype(g)))$
Effect
$Exists(g) \wedge g \in ownedElement(p) \wedge package(g)=p$
Propagation
[1] The new generalization is linked to the generalizable elements
$g \in specialization(supertype(g)) \wedge g \in generalizations(subtype(g))$

Action addFeature(c,f1)
Precondition

[1] The source Classifier exists and the new Feature does not exist (as a consequence, f1∉attributes(c)).
Exists(c) ∧ ¬Exists(f1)
[2]No Features may have the same name within a Classifier
∀g∈attributes(c) name(f1)≠name(g)
[3] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd.
∀e∈oppositeAssociationEnds(c) name(f1)≠name(e)
[4] The connected type should be included in the Package of the Classifier.
type(f1)∈allContents(package(c))
Effect
Exists(f1) ∧ f1∈features(c) ∧ owner(f1)=c
Propagation
[1] A new slot for the attribute must be added into the existing instances of c.
∀i:Instance((Exists(i)∧classifier(i)=c)→(∃l:AttributeLink (atribute(l)=f1 ∧ value(l)=eval(initialValue(f1)) ∧ slots(i)=slots(i)∪{l})))

Action addFeature(c,f2)
Precondition
[1] The source Classifier exists and the new Operation does not exist (as a consequence f2∉operations(c)).
Exists(c) ∧ ¬Exists(f2)
[2]No Operations may have the same signature within a Classifier
∀g∈operations(c) ¬hasSameSignature(f2,g)
[3] The type of the Parameters should be included in the Package of the Classifier.
∀p∈parameters(f2) type(p)∈allContents(package(c))
[4] pre and post conditions of the operation are the expression *true.*
precondition(f2)=*true* ∧ postcondition(f2)=*true*
Effect
Exists(f2) ∧ f2∈features(c) ∧ owner(f2)=c
Propagation
[1] life dependency
∀p∈parameters(f2)Exists(p)
[2]There is no structural impact on the instances of c, but their behavior is modified indirectly.

Action addConnection(a,e)
Precondition
[1] The source Association exists and the new Connection does not exist (as a consequence, e∉allConnections(a)).
Exists(a) ∧ ¬Exists(e)
[2]No opposite AssociationEnds may have the same rol-name within a Classifier
∀c∈allConnectedElements(a) (∀e1∈allOppositeAssociationEnds(c) name(e1) ≠name(e))
[3] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd.
∀c∈allConnectedElements(a) (∀f∈allAtributes(c) name(f)≠name(e))
[4] At most one AssociationEnd may be an aggregation or composition.
aggregation(e)<>#none →∀e1∈allConnections(a) aggregation(e1)=#none
[5] The connected type should be included in the Package of the Association.
type(e)∈allContents(package(a))
Effect
Exists(e) ∧ connections(a)=connections(a)∪{e} ∧association(e)=a ∧ e∈associationEnds(type(e))
Propagation
[1] A new linkEnd must be added into the existing Links of the Association.

∀k:Link ((Exists(k)∧association(k)=a) →(∃l:LinkEnd (associationEnd(l)=e ∧ instance(l)=nullElement ∧ linkRoles(k)=linkRoles(k)∪{l})))

Structural Cancellation

Action deleteSubpackage(p,q)
Precondition
[1] The element must exist in the Package.
q∈ownedElements(p)
[2]∀q1:Package Exists(q1)→q∉referencedElements(q1)
[3] The package must be empty
contents(q)=∅
Effect
¬Exists(q) ∧q∉ownedElement (p)
Propagation
no propagation.

Action deleteClassifier (p,c)
Precondition
[1] The element must exist in the Package.
c∈ownedElements(p)
[2] the Classifier does not participate in any relationship.
AssociationEnds(c)=∅ ∧ generalizations(c)=∅ ∧ specializations(c)=∅
[3] there is no instance of the classifier
∀i:Instance (Exists(i) →classifier(i)≠c)
[4] The deleted classifier cannot be the type of an attribute of other class in the package.
∀c1∈allContents(p) ((∃f∈attributes(c1)type(f)=c) →c1=c)
[5] The deleted classifier cannot be the type of a parameter of other class in the package.
∀c1∈allContents(p) ((∃b∈operations(c1) ∃p∈parameters(b) type(p)=c) →c1=c)
[6] The deleted classifier cannot be referenced from other StatesMachines.
∀h:StateMachine(c∈referencedElements(h) → context(h)=c)
Effect
¬Exists(c) ∧c∉ownedElement(p)
Propagation
[1] life dependency of the Features.
∀f∈allFeatures(c) ¬Exists(f)
[2] StateMachine release
Let h be behavior(c) before applying the action,
¬Exists(h) ∧ h∉ownedElements(p)

Action deleteAssociation(p,a)
Precondition
[1] The element must exist in the Package.
a∈ownedElements(p)
[2] there is no instance of the Association
∀k:Link (Exists(k) →association(k)≠a)
[3] The deleted association cannot be referenced from other elements in the package.
∀m∈allContents(p) a∉referencedElements(m)
Effect
¬Exists(a) ∧a∉ownedElement(p)
Propagation
[1]All the AssociationEnds of the Association are deleted.

$\forall e \in connections(a)\ \neg Exists(e)$
[2] Classifiers are disconnected
$\forall e \in connections(a)\ e \notin associationEnds(type(e))$

Action deleteGeneralization(p,g)
Precondition
[1] The element must exist in the Package.
$g \in ownedElements(p)$
[2] inherited elements are not referenced from the subtypes.
$\forall c \in (subtypes(subtype(g)) \cup \{subtype(g)\})\ \forall f \in allFeatures(supertype(g))\ f \notin referencedElements(c)$
Effect
$\neg Exists(g) \wedge g \notin ownedElement(p)$
Propagation
[1] The deleted generalization is disconnected from the generalizable elements
$g \notin specialization(supertype(g)) \wedge g \notin generalizations(subtype(g))$

Action deleteFeature(c,f1)
Precondition
[1] The Feature must exist in the Classifier.
$f1 \in attributes(c)$
[2] The deleted Feature cannot be referenced from other elements in the package.
$\forall m \in allContents(p)\ f1 \notin referencedElements(m)$
Effect
$\neg Exists(f) \wedge f \notin features(c)$
Propagation
[1] The corresponding slot must be deleted from all the existing instances of c.
$\forall i{:}Instance\ ((Exists(i) \wedge classifier(i)=c) \rightarrow (\exists l{:}AttributeLink\ (atribute(l)=f1 \wedge slots(i)=slots(i)-\{l\})\ )\ )$

Action deleteConnection(a,e)
Precondition
[1] The Connection must exist in the Association.
$e \in connections(a)$
[2] The deleted AssociationEnd cannot be referenced from other elements in the package.
$\forall m \in allContents(package(a))\ e \notin referencedElements(m)$
Effect
$\neg Exists(e) \wedge e \notin connections(a) \wedge e \notin associationEnds(type(e))$
Propagation
[1] A corresponding linkEnd must be deleted from the existing instances of the Association.
$\forall k{:}Link\ (\ (Exists(k) \wedge association(k)=a) \rightarrow$
$(\exists l{:}LinkEnd\ (associationEnd(l)=e \wedge linkRoles(k)=linkRoles(k)-\{l\})\ )\ )$

Behavioral Creation

Action setBehavior (c,h)
Precondition
[1] The source Classifier exists and the new StateMachine does not exist.
$Exists(c) \wedge \neg Exists(h)$
[2] They are syntactically compatible, that is to say only features of the Classifier are referenced in the state Machine
syntactic-compatible(c,h)
[3] Behavioral correctness: the state machine satisfies the pre and post conditions of the corresponding operations. [behavior(c):=h] means term substitution in the context of the formula.

(∀o∈instances(c) ∀<op,s,o,p>:Message
eval(precondition(op)[self:=o, parameters:=p])=true
→[r.<op,s,o,p>]eval(postcondition(op)[self:=o, parameters:=p])=true) [behavior(c):=h]
[4] Behavioral correctness: all the operations specified by the StateMachine should preserve the constraints of c.
(∀i∈allConstraints(c) ∀o∈instances(c) ∀m:Message
eval(i[self:=o])=true →[o.m]eval(i[self:=o])=true) [behavior(c):=h]
Effect
Exists(h) ∧ package(h)=package(c) ∧ h∈ownedElements(package(c)) ∧ context(h)=c ∧ behavior(c)=h
Propagation
There is no structural propagation on instances of c, but their behavior is affected indirectly.

Behavioral Cancellation

Action cancellBehavior(c)
Precondition
[1] The source Classifier and its StateMachine must exist.
Exists(c) ∧ Exists(behavior(h))
Effect
Let h be behavior(c) before the application of the action.
¬Exists(h) ∧ h∉ownedElements(package(c)) ∧ context(h)=nullElement ∧ behavior(c)=nullElement
Propagation
There is no structural propagation on instances of c, but their behavior is affected indirectly.

Constraint Creation

Action addConstraint(m,i)
Precondition
[1] The source ModelElement exists and the new Constraint does not exist.
Exists(m) ∧ ¬Exists(i)
[2] the Constraint and the constrained Element are syntactically compatible
syntactic-compatible(m,i)
[3] Constraint consistency
consistent(allConstraints(m)∪{i}) ∧
∀c:GeneralizableElement (*IsA*(c,m) →consistent(constraints(c)∪{i})
[4] If m is a Classifier, all its instances should satisfy the new Constraint.
∀o∈instances(m) eval(i)[self:=o] = true
[5]If m is a Classifier, all its operations should preserve the new constraint.
∀o∈instances(m) ∀m∈Message
(eval(i[self:=o]) = true) →[o.m] eval(i[self:=o]) = true)
Effect
Exists(i) ∧ i∈constraints(m) ∧ m∈constrainedElements(i)
Propagation
There is not direct propagation. But there is indirect propagation because the new constraint is inherited by all the subtypes of m. Also the behavior of instances of m may be affected by the constraint.

Constraint Cancellation

Action deleteConstraint(m,i)
Precondition
[1] The source ModelElement and its Constraint must exist.
Exists(m) $\wedge$ i$\in$constraints(m)
Effect
$\neg$Exists(i) $\wedge$ i$\notin$constraints(m)
Propagation
There is not direct propagation.

Data Creation

Action newObject(c,o)
Precondition
[1]Exists(c) $\wedge$ $\neg$Exists(o)
Effect
[1]The instance exists
Exists(o) $\wedge$ classifier(o)=c
[2] the slots of the new instances matches the declarations in the Classifier.
$\forall$l:AttributeLink(l$\in$slots(o) $\leftrightarrow$attribute(l)$\in$allAttributes(c))
[3] The new instance does not participate in any relationship
linkEnds(o)=$\varnothing$
[4] Initial values of attributes
$\forall$l$\in$slots(o) (value(l)=eval(initialValue(atribute(l))))
[5] Satisfaction of Constraints.
$\forall$i$\in$allConstraints(c) (eval(i)[self:=o] = true)
[6] mailbox(o)= $\varnothing$
Propagation
There is not direct propagation.

Action newLink(a,k)
Precondition
[1]The Association exists in the model
Exists(a) $\wedge$ $\neg$Exists(k)
[2] all elements connected by the new relationship must exist.
$\forall$i$\in$connectedElements(k) Exists(i)
[3] The set of LinkEnds must match the set of AssociationEnds of the Association.
$\forall$e:AssociationEnd (e$\in$map associationEnds linkRoles(k) $\leftrightarrow$ e$\in$allConnections(a))
[4] There are not two Links of the same Association which connects the same set of Instances in the same way.
$\forall$k1:Link (association(k1)=a $\rightarrow$connectedElements(k1)$\neq$connectedElements(k))
[5] The type of the Instance must match the type of the AssociationEnd
$\forall$l$\in$linkRoles(k) (type(associationEnd(l))=classifier(instance(l)))
[6] An Instance may not belong by composition to more than one composite Instance.
$\forall$i$\in$connectedElements(k) (($\exists$e$\in$oppositeLinkEnds(i) aggregation(associationEnd(e))= #composite)
$\rightarrow$
($\exists$e1$\in$linkRoles(k) aggregation(associationEnd(e1))= #composite $\rightarrow$ instance(e1)=i))
[7] Satisfaction of Constraints.

∀c∈allConstraints(a) (eval(c)[self:=k] = true)
Effect
[1]The link exists
Exists(k) ∧ association(k)=a
Propagation
[1]The new link is connected to the instances
∀l∈linkRoles(k) l∈linkEnds(instance(l))

Data Cancellation

Action destroy(o)
Precondition
[1] The object exists
Exists(o)
Effect
[1]¬Exists(o)
[2] The associations of the deleted object are modified
∀l∈linkEnds(o) instance(l)=nullElement
Propagation
[1] all the parts of a composite objects are destroyed
∀i∈allParts(o) ¬Exists(i)
[2] the association of the deleted objects are modified.
∀i∈allParts(o) (∀l∈linkEnds(i) instance(l)=nullElement)

Action destroy(k)
Precondition
[1]The link exists
Exists(k)
Effect
¬Exists(k)
Propagation
[1] instances are disconnected.
∀l∈linkRoles(k) l∉linkEnds(instance(l))

7.3 Conflictos de evolución

Todo mecanismo de evolución debe garantizar que la consistencia del sistema de software sea preservada durante su evolución. Para lograr este objetivo es necesario poseer un entendimiento preciso de los efectos directos e indirectos (o propagación) producidos por cada modificación. De esta forma podrá garantizarse la detección sistemática de inconsistencias, evitando así su ocurrencia.

En esta sección mostramos un ejemplo de conflicto de evolución. Consideraremos dos modificaciones arbitrarias que no causan problemas cuando son aplicadas separadamente, pero que pueden originar conflictos cuando son integradas (es decir aplicadas conjuntamente).

Luego analizamos como la M&D-logic nos permite identificar situaciones conflictivas originadas por la aplicación de acciones de evolución, de la siguiente forma:

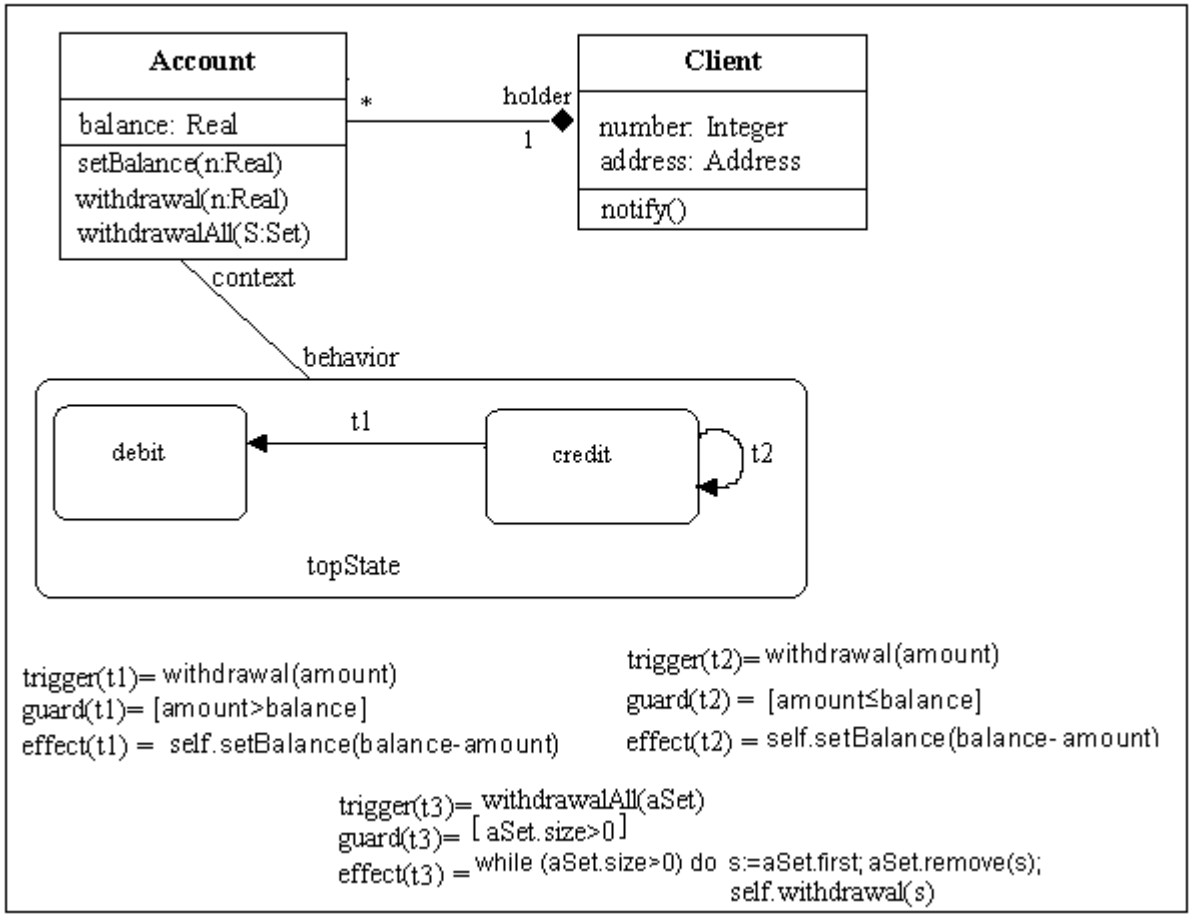
Sea,

- M la especificación de un sistema orientado a objetos expresada en UML.
- $Spec_M$ la especificación expresada en lógica dinámica que interpreta formalmente a M, es decir ***translation***(M)=$Spec_M$
- U un modelo para $Spec_M$, es decir U es un sistema de transición entre mundos posibles perteneciente al conjunto ***semantics***($Spec_M$).
- a,b son dos acciones de evolución que no causan problemas cuando son aplicadas separadamente sobre M.

La combinación de ambas acciones origina conflicto si el predicado Conflict(a,b) se satisface en U, es decir U |=Conflict(a,b) . Para esto, en las siguiente sección definiremos adecuadamente el predicado Conflict para cada subclase de acciones de evolución.

7.3.1 Un ejemplo de conflicto

Figura 7.5 : Modelo del sistema bancario



Consideremos un sistema bancario muy simplificado. Este sistema consta de dos clases: Account y Client. La clase Account define una operación *withdrawal* que permite extraer dinero de la cuenta y

una operación *withdrawalAll* que permite realizar un conjunto de extracciones conjuntamente. Cada cliente posee un conjunto de cuentas. La figura 7.5 contiene la especificación expresada en UML de este sistema.

Consideremos dos modificaciones sobre este modelo:

Evolución 1: *the NotifyingAccount*

La clase *Account* fue modificada de manera que sus instancias envíen el mensaje *notify* al su titular (holder) siempre que se realice una extracción superior al monto de la cuenta.

La acción de evolución que fue aplicada sobre el modelo original para obtener la nueva versión fue:

setEffect(t1,s), dode s es la secuencia ⟨setBalance(balance-amount) ; holder.notify() ⟩.

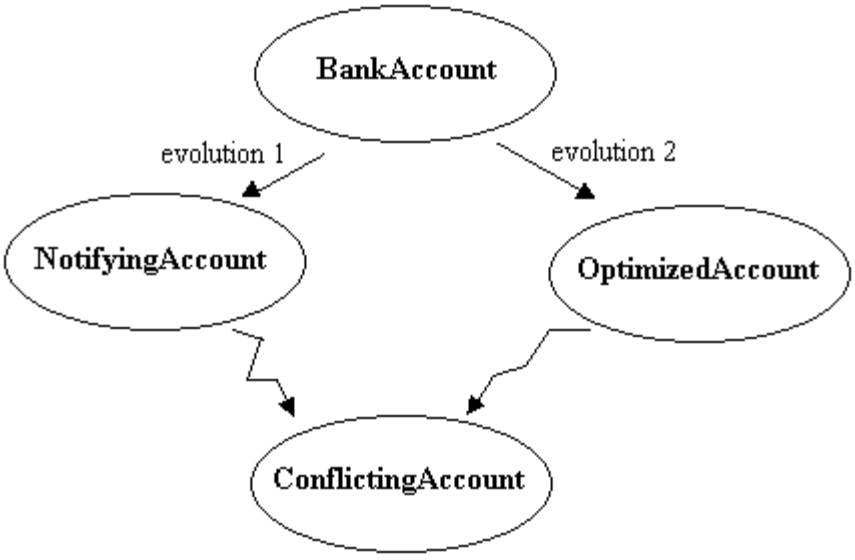
Evolución 2: *the OptimizedAccount*

Una versión optimizada de *Account* fue creada. En esta nueva versión *withdrawalAll* realiza todos las extracciones directamente en lugar de invocar al mensaje *withdrawal* para que las realice.

La acción de evolución que fue aplicada sobre el modelo original para obtener esta nueva versión fue:

setEffect(t3,s) donde s es la secuencia <while (aSet.size>0) do s:=aSet.first; aSet.remove(s); setBalance(balance-s)>.

Figura 7.6 : conflictos de evolución



Integrando ambas evoluciones: *the ConflictingAccount*

Las modificaciones presentadas previamente no causan problemas cuando son aplicadas separadamente, sin embargo cuando ambas son integradas puede ocurrir un

comportamiento no esperado (y tal vez erróneo), ya que los titulares de las cuentas no siempre serán notificados cuando el saldo de sus cuentas sea negativo.

El problema surge porque la acción 1 está modificando el efecto de un mensaje (en este caso withdrawal) mientras que la acción 2 está eliminando la invocación de dicho mensaje dentro de otro mensaje (en este caso withdrawalAll). La figura 7.6 ilustra esta situación conflictiva.

El ejemplo presentado muestra un caso de conflicto bastante sutil; existen por supuesto otros conflictos mucho más directos y obvios, como por ejemplo:

evolución 1: elimina la clase Account del modelo.

evolución 2: agrega un nuevo atributo a la clase Account

7.3.2 Identificación y clasificación de conflictos

Consideraremos todos los posibles conflictos que pueden presentarse entre dos acciones de evolución primitivas. De acuerdo con nuestra jerarquía de acciones primitivas tenemos 12 diferentes clases de acciones primitivas, lo cual nos llevará a considerar 144 (es decir 12^2) diferentes combinaciones de acciones. La matriz de la figura 7.7 muestra esta combinación de acciones. En las siguientes secciones analizaremos algunas de las sub-matrices en particular.

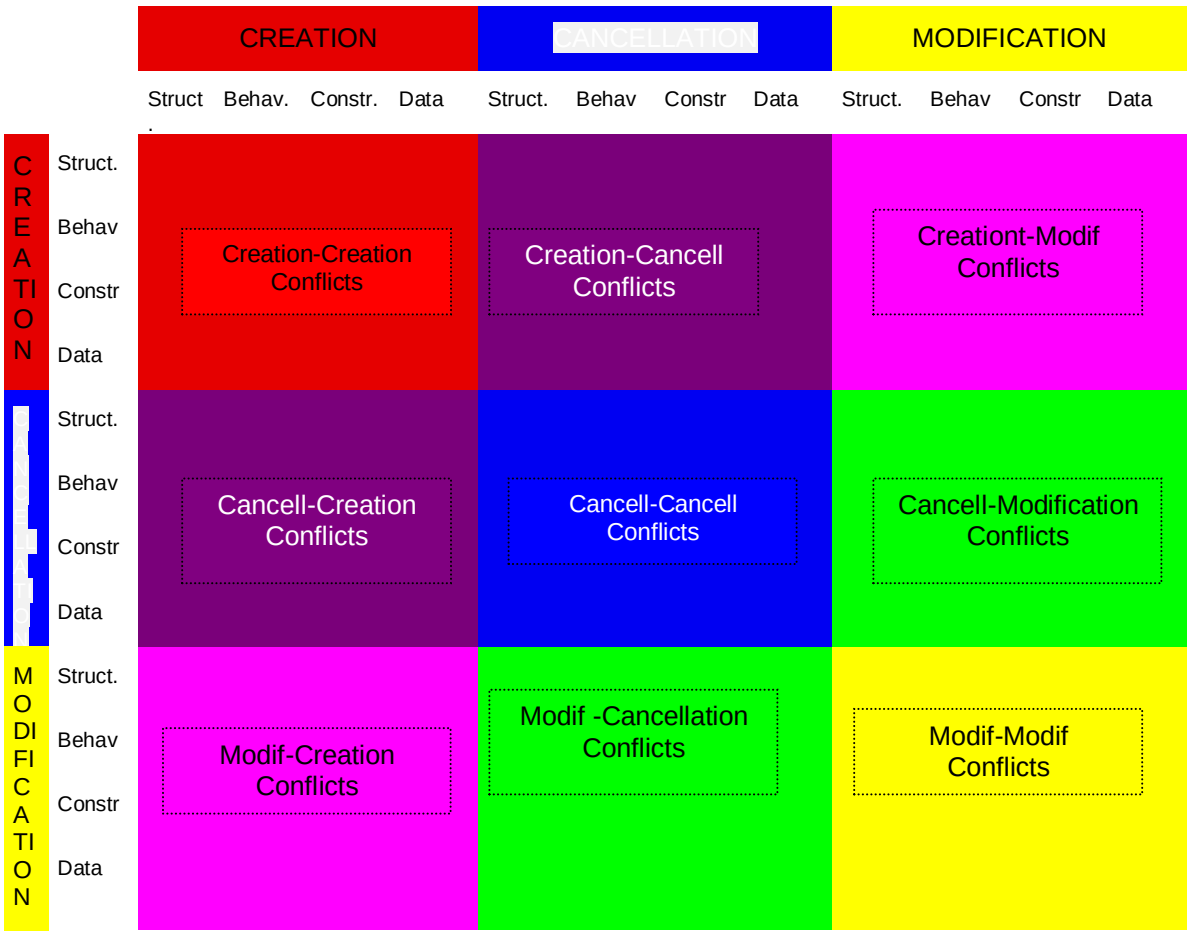


Figura 7.7: matriz de conflictos

7.3.2.1 Sub-matriz de conflictos Creation vs. Creation

Esta sub-matriz muestra los conflictos que se originan como consecuencia de la combinación de dos acciones de Creación que no causan problemas cuando son aplicadas en forma separada. La matriz es simétrica (es decir T[i,j]=T[j,i]), ya que la clase de conflicto es la misma independientemente del orden en el cual las acciones son aplicadas.

		CREATION			
CREATION		**Structural**	**Behavioral**	**Constraint**	**Data**
	Structural	1	2	3	4
	Behavioral	2	5	6	7
	Constraint	3	6	8	9
	Data	4	7	9	10

En las siguientes secciones analizaremos cada clase de conflicto en particular:

1 Structural Creation vs. Structural Creation

	addSubpackage(p1,q1)	addClassifier(p1,c1)	addAssociation(p1,r1)	addGeneralization(p1,r1)	addFeature(c1,f1)	addConnection(a1,e1)
addSubpackage(p,q)	conflict 1	no conflict	no conflict	no conflict	no conflict	no conflict
addClassifier(p,c)		conflict 2	no conflict	no conflict	no conflict	no conflict
addAssociation(p,r)			conflict 3	no conflict	no conflict	no conflict
addGeneralization(p,r)				conflict 4	conflict 5	conflict 6
addFeature(c,f)					conflict 7	conflict 8
addConnection(a,e)						conflict 9

No hay conflicto si la primera acción crea un nuevo Classifier, Package, Association, Feature o AssociationEnd, ya que ese nuevo elemento recién creado no puede ser referenciado por la segunda acción. Sólo puede existir conflicto cuando ambas acciones crean elementos del mismo tipo y con el mismo nombre, de la siguiente forma:

[Conflict 1] Los nombres de dos sub-packages se repiten dentro del mismo Package.

Conflict(addSubpackage (p,q), addSubpackage (p1,q1))$\leftrightarrow$ p=p1 $\wedge$ name(q)=name(q1) $\wedge$ q$\neq$q1

[Conflict 2] Los nombres de dos Classifiers se repiten dentro del mismo Package

Conflict(addClassifier(p,c), addClassifier(p1,c1))$\leftrightarrow$ p=p1 $\wedge$ name(c)=name(c1) $\wedge$ c$\neq$c1

[Conflict 3] Los nombres de dos Association se repiten dentro del mismo Package.

Conflict(addAssociation(p,r), addAssociation(p1,r1))↔ p=p1 ∧ name(r)=name(r1) ∧ r≠r1

[Conflict 7] Los nombres de dos Features se repiten dentro del mismo Classifier.

Conflict(addFeature(c,f), addFeature(c1,f1))↔ c=c1 ∧ name(f)=name(f1) ∧ f≠f1

[Conflict 8] Un Attribute y un AssociationEnd opuesto tienen el mismo nombre dentro del mismo Classifier.

Conflict(addFeature(c,f), addConnection(a,e))↔ a∈allConnections(c) ∧ name(f)=name(e)

[Conflict 9] (a) Hay dos AssociationEnds con el mismo nombre dentro de una Association. (b)Más de un AssociationEnd es aggregation o composition. (c)En un Classifier, dos AssociationEnds opuestos tienen el mismo rol-name.

Conflict(addConnection(a,e), addConnection(a1,e1))↔

(a=a1 ∧ name(e)=name(e1) ∧ e≠e1)

∨ (a=a1 ∧ aggregation(e)<>#none∧aggregation(e1)<>#none ∧ e≠e1)

(c) ∨ ∃c:Classifier (a∈allConnections(c)∧a1∈allConnections(c)∧a≠a1∧name(e)=name(e1))

[Conflict 4]

Sean r y r1 las generalizations que están siendo agregadas en el modelo. Sean super1=supertype(r), super2=supertype(r1), sub1=subtype(r), sub2=subtype(r1)

Conflict(addGeneralization(p,r), addGeneralization(p1,r1)) ↔

(a) IsA(super2,sub1) ∧ IsA(super1,sub2) ∧ ¬(super1=sub1=super2=sub2)

(b) ∨ (IsA(super1,sub2) ∧ ¬consistent(allConstraints(sub1)∪ allConstraints(super2)))

∨ (IsA(super2,sub1) ∧ ¬consistent(allConstraints(sub2)∪allConstraints(super1)))

(c) ∨ (IsA(super1,sub2) ∧ ¬refinement(behavior(sub1), behavior(super2)))

∨ (IsA(super2,sub1) ∧¬refinement(behavior(sub2), behavior(super1)))

(d) ∨ (sub1=sub2 ∧ super1≠super2 ∧

∃f,g:Feature(f∈allFeatures(super1)∧ g∈allFeatures(super2) ∧ f≠g ∧ name(f)=name(g)))

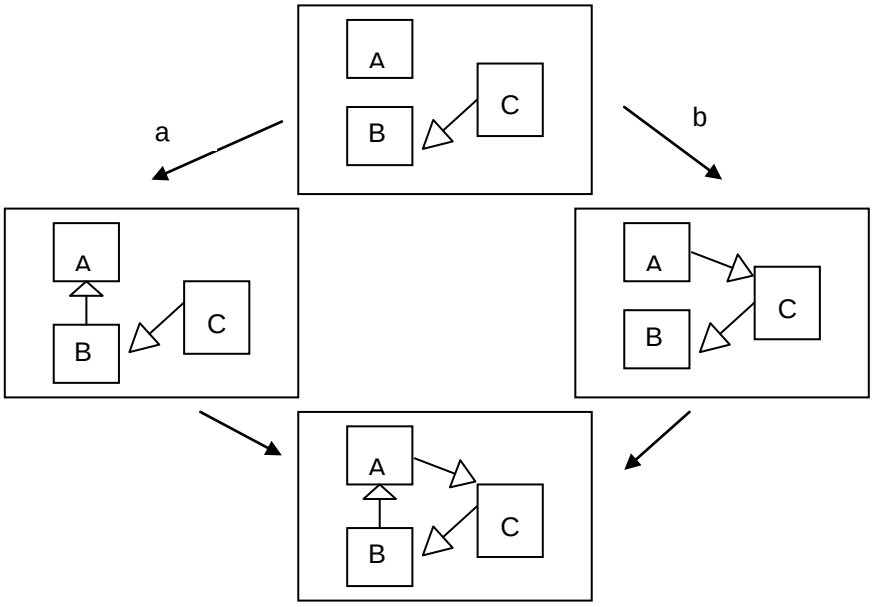
Las fórmulas de arriba especifican:

(a) Generalización cíclica, como se muestra en la figura.

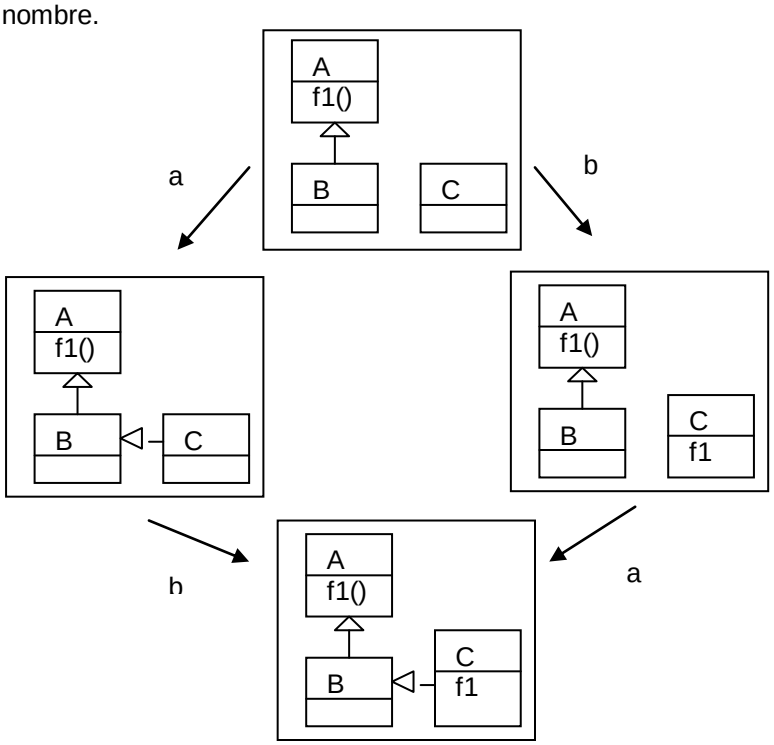
(b) Refinamiento inconsistente de constraints.

(c) Refinamiento inconsistente de behavior.

(d) Conflicto de nombres ocacionado por herencia múltiple.



[Conflict 5] La acción a agrega un feature f1 al Classifier C. Luego de aplicar la acción b, C ha heredado un feature f1 (ver figura). Existe un problema de redefinición de features. Esta situación no es un conflicto ya que la redefinición de features está permitida. Sin embargo podría representar un conflicto de "intensiones" y por lo tanto debe señalarse como *warning*. Una situación similar se presenta cuando el feature insertado es heredado por una clase que ya tenía un feature con el mismo nombre.



¬Conflict(addGeneralization(p,r), addFeature(c,f))

Warning(addGeneralization(p,r), addFeature(c,f)) ↔

subtype(r)=c ∧ (∃f1∈allFeatures(supertype(r)) name(f1)=name(f))

∨ (supertype(r)=c ∧ (∃f1∈allFeatures(subtype(r)) name(f1)=name(f)))

[Conflict 6] Este conflicto es similar al descripto anteriormente.

¬Conflict(addGeneralization(p,r), addConnection(a,e))

Warning(addGeneralization(p,r), addConnection(a,e)) ↔

(a∈allConnections(subtype(r))∧(∃e1∈allOppositeAssociationEnds(supertype(r))name(e1)=name(e))

∨

(a∈allConnections(supertype(r))∧(∃e1∈allOppositeAssociationEnds(subtype(r))name(e1)=name(e)))

2 Structural Creation vs. Behavioral Creation

	setBehavior(c1,h)
addSubpackage(p,q)	no conflict
addClassifier(p,c)	no conflict
addAssociation(p,r)	no conflict
addGeneralization(p,r)	conflict 1
addFeature(c,f)	no conflict
addConnection(a,e)	no conflict

En general, no se presentan conflictos entre acciones de StructuralCreation y acciones de BehavioralCreation, ya que los elementos estructurales que son insertados no pueden estar referenciados en la Behavioral creation debido a que son nuevos elementos es decir que no existen en el modelo sobre el cual la acción es aplicada. Por ejemplo, si la acción estructural agrega un feature f a un classifier c y la acción de comportamiento asigna una máquina de estados h a la misma clase c, sabemos que el feature f no ocurre h ya que f no existía previamente. Por lo tanto la presencia de conflictos se define de la siguiente forma:

¬Conflict(addSubpackage(p,q) , setBehavior(c1,h))

¬Conflict(addClassifier(p,c) , setBehavior(c1,h))

¬Conflict(addAssociation(p,r) , setBehavior(c1,h))

¬Conflict(addFeature(c,f) , setBehavior(c1,h))

¬Conflict(addConnection(a,e), setBehavior(c1,h))

[Conflict1] La única fuente de conflicto es la creación de una Generalización que produzca inconsistencias entre los comportamientos heredados. Puede existir conflicto si una acción asigna una máquina de estados h a una clase c y la otra acción crea una generalización que hace que c herede otra máquina h´. Como consecuencia de la combinación de ambas acciones ocurre un refinamiento de comportamiento y ambas máquinas h y h´ deben satisfacer la relación de

refinamiento. Una situación similar se presenta cuando la nueva máquina de estados es heredada por una clase que ya poseía una máquina de estados.

Conflict(addGeneralization(p,r), setBehavior(c,h)) $\leftrightarrow$

isA(c,subtype(r)) $\wedge$ $\neg$refinement(h , behavior(supertype(r)))

$\vee$ isA(supertype(r),c) $\wedge$ $\neg$refinement(behavior(subtype(r)), h)

3 Structural Creation vs. Constraint Creation

	addConstraint(m,i)
addSubpackage(p,q)	no conflict
addClassifier(p,c)	no conflict
addAssociation(p,r)	no conflict
addGeneralization(p,r)	conflict 1
addFeature(c,f)	no conflict
addConnection(a,e)	no conflict

En general no existe conflicto entre una acción Structural Creation y una acción Constraint Creation, ya que los elementos que están siendo agregados por la acción estructural no pueden ser referenciados por el nuevo constraint. Como consecuencia, el predicado Conflict se define de la siguiente forma:

$\neg$Conflict(addSubpackage(p,q) , addConstraint(m,i))

$\neg$Conflict(addClassifier(p,c) , addConstraint(m,i))

$\neg$Conflict(addAssociation(p,r) , addConstraint(m,i))

$\neg$Conflict(addFeature(c,f) , addConstraint(m,i))

$\neg$Conflict(addConnection(a,e), addConstraint(m,i))

[Conflict1] La única fuente de conflicto es la creación de una Generalization. Puede existir conflicto si la acción *a* agrega un nuevo constraint *i* a un classifier c y luego de aplicar a acción *b*, c hereda otros constraints. Como consecuencia de la combinación de ambas acciones se produce un refinamiento de constraints, el cual debe cumplir ciertas condiciones de consistencia. Una situación similar ocurre cuando el constraint insertado es heredado por una clase que ya poseía otros constraints.

Conflict(addGeneralization(p,r), addConstraint(m,i)) $\leftrightarrow$

isA(m,subtype(r)) $\wedge$ $\neg$consistent({i} $\cup$ constraints(supertype(r)))

$\vee$ isA(supertype(r), m) $\wedge$ $\neg$consistent(constraints(subtype(r)) $\cup${i})

4 Structural Creation vs. Data Creation

	newObject(c1,o)	newLink(a1,k)
addSubpackage(p,q)	no conflict	no conflict
addClassifier(p,c)	no conflict	no conflict
addAssociation(p,r)	no conflict	no conflict
addGeneralization(p,r)	conflict 1	conflict 2

addFeature(c,f)	conflict 3	no conflict
addConnection(a,e)	no conflict	conflict 4

No se produce conflicto entre las siguientes acciones:

¬Conflict(addSubpackage(p,q) newObject(c1,o))

¬Conflict(addClassifier(p,c) , newObject(c1,o))

¬Conflict(addAssociation(p,r) , newObject(c1,o))

¬Conflict(addSubpackage(p,q) newLink(a1,k))

¬Conflict(addClassifier(p,c) , newLink(a1,k))

¬Conflict(addAssociation(p,r) , newLink(a1,k))

¬Conflict(addConnection(a,e), newObject(c1,o))

¬Conflict(addFeature(c,f) , newLink(a1,k))

[Conflict 1] Como consecuencia de la generalización la clase c1 puede heredar nuevos atributos los cuales no tendrán los correspondientes slots en la instancia *o*.

Conflict(addGeneralization(p,r) , newObject(c1,o)) ↔ IsA(c1, subtype(r))

[Conflict 2] Como consecuencia de la generalización la asociación a1 puede heredar nuevos LinkEnds los cuales no tendrán los correspondientes linkRoles en el link *k*.

Conflict(addGeneralization(p,r) , newLink(a1,k)) ↔ IsA(a1, subtype(r))

[Conflict 3] La instancia *o* no posee un slot para el nuevo atributo f. Si el feature creado es una operación entonces no se produce este conflicto estructural, sin embargo el comportamiento de la nueva instancia será diferente del esperado. Esto será señalado como un *warning*.

Conflict(addFeature(c,f) , newObject(c1,o)) ↔ c=c1

[Conflict 4] El link *k* no posee el correspondiente linkRole para el nuevo LinkEnds *e*.
Conflict(addConnection(a,e), newLink(a1,k)) ↔ a=a1

5 Behavioral Creation vs. Behavioral Creation

	setBehavior(c1,h1)
setBehavior(c,h)	conflict

Existen distintos tipos de conflictos y situaciones potencialmente conflictivas señaladas como *warning*:

Conflict(setBehavior(c,h), setBehavior(c1,h1)) ↔

[1](c=c1 ∧ h≠h1)

[2]∨ (c≠c1 ∧ IsA (c,c1) ∧ ¬refinement(h, h1))

[3]∨ (c≠c1 ∧ IsA (c1,c) ∧ ¬refinement(h1, h))

Warning(setBehavior(c,h), setBehavior(c1,h1)) ↔ c≠c1 ∧

(specifiedOperations(h)∩referencedElements(h1)≠∅ ∨

specifiedOperations(h1)∩referencedElements(h)≠∅)

[Conflict 1] Existe conflicto cuando ambas acciones asignan una máquina de estados diferente a la misma clase.

[Conflicts 2 and 3] Existe conflicto cuando ambas clases están relacionadas por generalización y las máquinas de estados no satisfacen la relación de refinamiento.

[Conflict 4] Si la intersección entre los elementos especificados y referenciados no es vacía, se produce un conflicto de captura, ya que una acción está usando una operación (por ejemplo dentro de una cláusula de efecto de la nueva máquina de estados) mientras que la otra acción está modificando el comportamiento de tal operación.

6 Behavioral Creation vs. Constraint Creation

	addConstraint(m,i)
setBehavior(c,h)	conflict

[Conflict] Estas acciones son conflictivas cuando se aplican sobre el mismo elemento de modelado y no son consistentes entre ellas (por ejemplo el nuevo behavior podría no respetar los constraints ligados al elemento).

Conflict(setBehavior(c,h), addConstraint(m,i)) $\leftrightarrow$

c=m $\wedge$ [setBehavior(c,h)]($\exists$o$\in$instances(c) $\exists$g:Message (eval(i[self/o]) $\wedge$ $\neg$[o.m]eval(i[self/o])))

7 Behavioral Creation vs. Data Creation

	newObject(c1,o)	newLink(a,k)
setBehavior(c,h)	conflict	no conflict

No existe conflicto entre estas acciones, sin embargo el comportamiento de la nueva instancia podría ser distinto al esperado y por lo tanto esto se señala como un *warning*.

$\neg$Conflict(setBehavior(c,h), newObject(c1,o))

$\neg$Conflict(setBehavior(c,h), newLink(a,k))

Warning(setBehavior(c,h), newObject(c1,o))$\leftrightarrow$ c=c1

8 Constraint Creation vs. Constraint Creation

	addConstraint(m1,i1)
addConstraint(m,i)	conflict

[Conflict] Ambas acciones están asignando un nuevo constraint al mismo elemento. Existirá conflicto cuando dichos constraints no sean consistentes entre sí.

Conflict(addConstraint(m,i), addConstraint(m1,i1)) ↔ m=m1 ∧ ¬consistent({i,i1})

9 Constraint Creation vs. Data Creation

	newObject(c1,o)	newLink(a,k)
addConstraint(m,i)	conflict 1	conflict 2

[Conflict 1] Existe conflicto si la nueva instancia no satisface el nuevo constraint:

Conflict(addConstraint(m,i), newObject(c1,o)) ↔ (m=c1 ∧ eval(i[self:=o])=false)

[Conflict 2] Existe conflicto si la nueva instancia no satisface el nuevo constraint:

Conflict(addConstraint(m,i), newLink(a,k)) ↔ (m=a ∧ eval(i[self:=k])=false)

10 Data Creation vs. Data Creation

	newObject(c1,o)	newLink(a,k)
newObject(c1,o)	no conflict	no conflict
newLink(a,k)	no conflict	conflict 1

No existe conflicto entre esta clase de acciones de evolución:

∀a1a2:DataCreation ¬Conflict(a1,a2)

7.4 Trabajos relacionados

- La mayoría de los trabajos sobre evolución de especificación trata el problema de evolución estructural, por ejemplo modificar la jerarquía de herencia o agregar una nueva clase (ver por ejemplo los trabajos de [Bergstein 97, Kesim and Sergot 96, Bertino et al. 98]), pero no tratan el problema de evolución de comportamiento, como por ejemplo cambiar la forma en que un objeto reacciona al recibir un determinado mensaje.
- Acerca del problema de consistencia de librerías de clases en evolución, Mira Mezini en [Mezini 97] divide el problema en conflictos de evolución horizontales y verticales. Conflictos horizontales ocurren cuando cambios en la clase base invalidan a las subclases, mientras que conflictos verticales ocurren cuando la clase base es especializada por una subclase en una forma conflictiva (no prevista por los diseñadores). Mezini propone un sistema automático que mantiene la consistencia de la librería, a partir de propiedades definidas por los diseñadores.
- El mecanismo de Contratos de Reuso (Reuse Contracts) [Steyaert et al. 96, Lucas 97] permite especificar componente de software. Un contrato de reuso define un conjunto de participantes interrelacionados. El mecanismo además provee un conjunto de operadores de reuso que constituyen la única forma de modificar un contrato. De esta forma es posible analizar la relación entre diferentes versiones de un componente que ha ido evolucionando. En [Mens et al. 98] se trata presenta una adaptación de la idea de contratos de reuso con el objetivo de formalizar reuso y evolución de modelos UML; se provee una caracterización precisa de reuso en UML, permitiendo detectar automáticamente varios conflictos de reuso. En general la técnica de Contratos de Reuso cubre evolución estructural pero sólo trata un grupo limitado de casos de evolución de comportamiento.

- Con respecto a los trabajos dedicados a proveer semántica formal al lenguaje UML, mientras que la sintaxis abstracta y la semántica estática (reglas de buena formación) del lenguaje han sido cubiertas adecuadamente, la semántica dinámica (tanto del modelo como de los datos) no ha sido tratada en forma precisa. Por ejemplo:
  - El lenguaje UML ha sido descripto básicamente en dos documentos: UML11-notation [UML 97(a)] y UML11-semantics [UML 97 (b)]. Estos documentos dan una noción clara de la sintaxis de UML, pero no logran definir precisamente la semántica del lenguaje. El documento llamado "semantics" en realidad sólo define reglas de buena formación de los modelos (es decir restricciones sobre la sintaxis) y explica la semántica de las construcciones UML de una forma poco precisa y en muchos casos contradictoria, usando lenguaje natural. Las reglas de buena formación son estáticas y no se trata el tema de evolución de modelos.
  - El "PUML group" Precise UML group [Evans et al. 98] ha construido un modelo semántico preciso del lenguaje UML. El modelo incluye la descripción de la sintaxis abstracta de UML usando el lenguaje formal Z y la definición formal de una función que interpreta las construcciones sintácticas en un dominio semántico formalmente definido. Sin embargo este modelo no describe evolución.
  - Lano y Biccaregui en [Lano and Biccaregui 98] proponen una semántica axiomática para la notación UML, usando teorías estructuradas en lógica temporal. Transformaciones sobre modelos UML, tales como agregar una clase o relación son representadas en el formalismo mediante extensiones de teorías.

7.5 Conclusiones

La mayoría de los trabajos sobre evolución de especificación trata el problema de evolución estructural, por ejemplo modificar la jerarquía de herencia o agregar una nueva clase, pero no tratan el problema de evolución de comportamiento, como por ejemplo cambiar la forma en que un objeto reacciona al recibir un determinado mensaje. El mecanismo de evolución propuesto en ese capítulo cubre ambos tipos de evolución.

En la M&D-theory, cada acción de evolución se define mediante dos fórmulas:

- Precondiciones necesarias para describir las condiciones de aplicabilidad de la acción.
- Postcondiciones suficientes para describir el efecto (efecto directo y propagación de cambios) de la acción.

Estas fórmulas se construyen sobre términos de sort Element (representando elementos de modelado) así como también sobre términos de sort Data (representando los datos del sistema) permitiendo de esta manera expresar:

- Propagación de cambios Intra-nivel: como una modificación realizada sobre una entidad de datos impacta sobre los demás datos, y como una modificación realizada sobre un modelo impacta sobre los demás modelos.
- Propagación de cambios Inter-nivel: como una modificación realizada sobre una entidad de datos impacta sobre sus modelos, y como una modificación realizada sobre un modelo impacta sobre los datos modelados.

Animando el sistema de transición que representa la semántica de la M&D-logic es posible analizar como reacciona el sistema a determinados cambios en su modelo (tanto cambios estructurales como cambios en la definición del comportamiento de los objetos). Además es posible observar como se combina la evolución del modelo con la evolución de los datos modelados. La M&D-logic permite expresar reglas de consistencia entre diferentes diagramas UML y entre estos diagramas y

los datos modelados. Luego, utilizando el mecanismo de deducción de la lógica es posible validar estas reglas a través de la evolución, garantizando la consistencia del sistema.

En resumen, los principales aportes del mecanismo de evolución propuesto son:

- Identifica un grupo completo de modificaciones primitivas.
- Define formalmente las condiciones de aplicabilidad de cada operación, es decir las condiciones bajo las cuales la aplicación de la operación es semánticamente correcta.
- Identifica y especifica la propagación de cambios provocada por la aplicación de cada operación.
- Considera *Modelos Dinámicos* ya que contempla la evolución combinada del modelo y los datos.
- Identifica situaciones conflictivas originadas por la aplicación de operaciones de evolución y permite detectar automáticamente la presencia de tales conflictos.

Además la M&D-logic provee un marco para definir transformaciones no-primitivas de modelos, tales como refinamientos y patrones de evolución, permitiendo analizar semánticamente tales transformaciones. Por ejemplo, dados dos modelos M y M´ obtenidos uno a partir del otro mediante la aplicación de un grupo de acciones primitivas, es posible demostrar la relación existente entre sus semánticas es decir, la relación entre los conjuntos sem(M´) y sem(M). El estado actual de nuestro trabajo no incluye este tipo de análisis pero creemos que constituye un área valiosa de ser investigada.

8. Midiendo calidad de modelos orientados a objetos

En el campo de la ingeniería de software aún se necesita un entendimiento más profundo y claro de cuales son las propiedades deseables y no-deseables de un diseño orientado a objetos. Además de identificar las propiedades también es necesario estudiar sus efectos sobre la calidad del sistema a desarrollar (tanto a corto como a largo plazo). Las propiedades deseables deben representar aquellas características que finalmente llevan a obtener un producto de software más eficiente, mantenible y extensible.

Los puntos centrales del problema consistente en medir la calidad de los sistemas orientados a objetos son:

- Es necesario determinar cuales son las propiedades deseables y no-deseables de un diseño.
- Debe existir una definición formal de estas propiedades.
- Es necesario proveer un mecanismo, formal y efectivo, para detectar (y cuantificar) la presencia de estas propiedades.

Calidad de un diseño es algo difícil de evaluar ya que es un concepto complejo integrado por diferentes aspectos. Sin embargo la comunidad informática ha logrado reconocer un conjunto de propiedades que caracterizan a un buen diseño orientado a objetos. Las propiedades de acoplamiento, cohesión, complejidad y tamaño han sido usadas extensivamente para caracterizar la calidad de diseños estructurados tradicionales. Estas propiedades también son importantes en los diseños orientados a objetos, pero sin embargo las métricas tradicionales no son aplicables sobre éstos (ver por ejemplo [Chidamber and Kemerer 94], [Tegarden et al.92] y [Wilde and Huitt 92]) debido a la presencia de nociones fundamentalmente diferentes, tales como herencia y polimorfismo, los cuales son inherentes al paradigma de orientación a objetos. Estos nuevos conceptos resultan vitales para la construcción de productos de software reusables, flexibles y adaptables y no pueden ser ignorados por las métricas de calidad. Esto ha llevado a la definición de nuevas métricas para medir productos de software orientados a objetos (ver por ejemplo [Chen and Lu 93], [Kim et al.94] y [Li and Henry 93]).

Existen numerosas propuestas que miden eficazmente la s propiedades tradicionales, como por ejemplo [Poulin 97, Briand et al. 97, Price and Demurjian 97, Benlarbi 97], mientras que menor esfuerzo ha sido dirigido a las nuevas propiedades inherentes de los objetos: [Moore 96, Bansiya 97, Bansiya et al.99(a), Bansiya et al.99(b)].

Otro problema adicional consiste en que muchas de las métricas disponibles actualmente pueden ser aplicadas sólo luego de que el producto está terminado o casi terminado, ya que extraen datos de la implementación. Esto hace que los problemas se detecten demasiado tarde. Es deseable contar con una herramienta que tome información proveniente de las primeras etapas del proceso de desarrollo (fases de análisis de requerimientos), esto le daría a los desarrolladores la oportunidad de evaluar y mejorar la calidad del producto tempranamente en el proceso de desarrollo.

Nuestra postura consiste en que ambas clases de propiedades, las tradicionales y las nuevas propiedades orientadas a objetos, deben ser consideradas conjuntamente para medir la calidad de un diseño orientado a objetos. Sin embargo creemos que es necesario prestar especial atención al concepto de *polimorfismo*, ya que debe ser considerado el punto clave para determinar la calidad de un diseño orientado a objetos. Adicionalmente la información referente a polimorfismo puede ser extraída tempranamente a partir de los modelos de análisis y diseño, tal como veremos a continuación.

En este capítulo definiremos una nueva métrica para medir la calidad de un diseño orientado a objetos. Esta métrica se aplica sobre el modelo del sistema escrito en UML, lo cual permite un análisis temprano de la calidad del sistema. Nuestra intención no es definir un mecanismo de evaluación de calidad completo (en el sentido que contemple todas las propiedades del sistema) sino que sólo nos ocuparemos de caracterizar *polimorfismo*. Luego, la medida de polimorfismo debe ser combinada con las medidas de las restantes propiedades (tales como acoplamiento, cohesión, entropía, etc.) con el objetivo de determinar la calidad total del sistema. Sin embargo esta tarea de combinación de métricas está fuera del alcance del presente trabajo.

8.1 Definición formal de polimorfismo

Daremos una definición rigurosa de la propiedad de polimorfismo usando la M&D-logic. Los conceptos básicos de polimorfismo han sido tomados de [Woolf 97].

Sea *M* la especificación de un sistema orientado a objetos expresada en UML. Sea $Spec_M$ la especificación expresada en lógica dinámica que interpreta formalmente a *M*, es decir ***translation***(*M*)=$Spec_M$ Y sea U un modelo para $Spec_M$, es decir U es un sistema de transición entre mundos posibles perteneciente al conjunto ***semantics***($Spec_M$).

- **Definición 1: atributos polimórficos**

Dos *atributos son polimórficos* cuando tienen el mismo nombre y el mismo tipo. Formalmente, sea a un nombre de atributo y sean C_1 y C_2 dos clases. El atributo a es polimórfico para las clases C_1 y C_2 en el modelo U, si se satisface la siguiente fórmula:

$U \models Polymorphic(a,C_1,C_2)$

Donde el predicado Polymorphic está definido de la siguiente forma:

$\forall a:Name\ \forall C_1,C_2:Class$
$Polymorphic(a,C_1,C_2) \leftrightarrow \exists a_1,a_2:Attribute\ (\ a_1 \in attributes(C_1) \wedge a_2 \in attributes(C_2)$
$\wedge\ name(a_1)=a \wedge name(a_2)=a \wedge\ \ type(a_1)=type(a_2)\ )$

- **Definición 2: métodos polimórficos**

Dos *métodos son polimórficos* cuando tienen el mismo nombre y la misma signatura (tipos de los parámetros y tipo del resultado) y además el mismo efecto (es decir, cambian el estado del receptor en la misma forma y desencadenan los mismos mensajes a otros objetos en el sistema). Formalmente, sea m un nombre de método y sean C_1 y C_2 dos clases. El método m es polimórfico para las clases C_1 y C_2 en el modelo U, si se satisface la siguiente fórmula:

$U \models Polymorphic(m,C_1,C_2)$

Donde el predicado Polymorphic está definido de la siguiente forma:

$\forall a:Name\ \forall C_1,C_2:Class$
$Polymorphic(m,C_1,C_2) \leftrightarrow \exists m_1,m_2:Operation\ (\ m_1 \in operations(C_1) \wedge m_2 \in operations(C_2) \wedge$
$name(m_1)=m \wedge name(m_2)=m \wedge$
$hasSameSignature(m_1,m_2) \wedge hasSameBehavior(m_1,m_2)\ )$

Donde el predicado hasSameSignature determina si dos métodos tienen la misma signatura (es decir, nombre y parámetros de entrada y salida) y está definido en la M&D-theory de la siguiente forma:

∀b,b´:BehavioralFeatures
hasSameSignature(b,b') ↔ (name(b)=name(b') ∧ areEquivalent(parameters(b), parameters(b')))

Donde areEquivalent se define sobre dos listas de parámetros de la siguiente forma:

areEquivalent(∅,∅)=true
areEquivalent(p-ps,∅)=false
areEquivalent(∅,p-ps)=false
areEquivalent(p1-ps,p2-ps´)=equivalent(p1,p2)∧areEquivalent(ps,ps´)

Y finalmente el predicado equivalent se aplica sobre dos parámetros individuales:

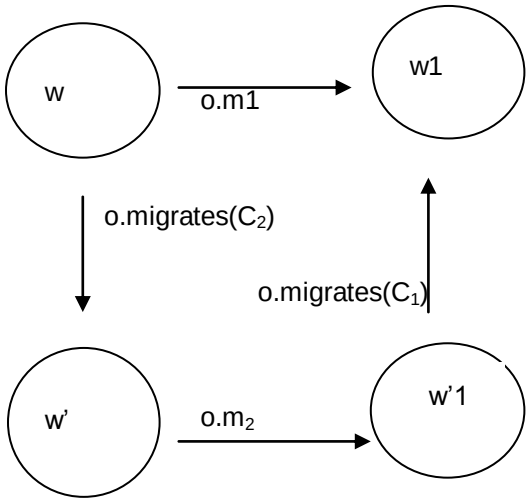
∀p1,p2:Parameter equivalent(p1,p2) ↔ defaultValue(p1)=defaultValue(p2) ∧
kind(p1)=kind(p2) ∧ type(p1)=type(p2)

Dos operaciones satisfacen el predicado hasSameBehavior(m_1,m_2) si son indistinguibles, es decir que para cualquier objeto *o*, los efectos de aplicar *o*.m_1 son iguales a los efectos de aplicar *o*.m_2. Para que esto ocurra es necesario que para todo mundo posible w en U y para todo objeto *o* tal que classifier(*o*)=C_1,

si $(w,w_1) \in m_U(o.m_1)$

entonces existen w',w'$_1$ en S^U tal que:

$(w,w´) \in m_U(o.\text{migrates}(C_2))$
$(w´,w´_1) \in m_U(o.m_2)$
$(w´_1,w_1) \in m_U(o.\text{ migrates}(C_1))$



Es decir que el siguiente diagrama conmuta, donde la acción *o*.migrates(C) representa migración de clase, (formalmente [*o*.migrates(C)]classifier(*o*)=C).

Corolario: el método m es polimórfico para las clases C_1 y C_2, si y sólo si el siguiente esquema es válido:

$\forall o \in \text{instances}(C_1)\ [o.m]\phi \leftrightarrow [o.\text{migrates}(C_2)]\ [o.m]\ [o.\text{migrates}(C_1)]\ \phi$

- **Definición 3: clases polimórficas**

Dos *clases son polimórficas* si todos sus atributos y métodos son polimórficos. Dos objetos pertenecientes a clases polimórficas se denominan objetos polimórficos. El mecanismo de ligadura dinámica permite sustituir objetos polimórficos en tiempo de ejecución sin riesgo de problemas de comportamiento ya que ambos objetos son indistinguibles. Este concepto de sustitutabilidad es un elemento clave de la programación orientada a objetos. Formalmente, sean C_1 y C_2 dos clases. Las clases C_1 y C_2 son polimórficas en el modelo U, si se satisface la siguiente fórmula:

U |= Polymorphic(C_1,C_2)

Donde el predicado Polymorphic está definido de la siguiente forma:

$\forall C_1,C_2$: Class Polymorphic(C_1,C_2) $\leftrightarrow$
interface(C_1)=interface (C_2) $\wedge$ $\forall n \in$ interface(C_1) polymorphic(n,C_1 ,C_2)

Donde $n \in$ interface(C) $\leftrightarrow$ $\exists f \in$ allFeatures(C) name(f)=n

- **Definición 4: jerarquías polimórficas**

El concepto de clases polimórficas resulta demasiado fuerte, ya que difícilmente en una jerarquía existan dos clases donde la totalidad de sus métodos sean polimórficos. Por lo tanto se ha definido un concepto más flexible denominado *núcleo polimórfico* de una jerarquía de clases (conocido también por el nombre de *core interface*). El *núcleo* o *core* es un conjunto de métodos polimórficos que es compartido por varias clases. Una jerarquía de clases es polimórfica cuando tiene un núcleo polimórfico.

Las jerarquías polimórficas presentan importantes ventajas ya que simplifican la definición de sus clientes. Si un cliente usa solamente features pertenecientes al núcleo de la jerarquía, éste podrá sustituir una instancia de una clase por una instancia de otra clase en la jerarquía. Esta sustitución no producirá problemas ya que todas las instancias se comportan equivalentemente para el subconjunto de features en el núcleo.

Formalmente, sea H un conjunto de clases organizadas jerárquicamente. La *jerarquía* H es *polimórfica* en el modelo U si existe una clase 'core' (la cual no necesariamente debe pertenecer a la jerarquía). Esto es expresado mediante la siguiente fórmula:

U |= Polymorphic(H)

Donde el predicado Polymorphic está definido de la siguiente forma:

$\forall$H:Set of Class Polymorphic(H) $\leftrightarrow$ $\exists$C:Class isCore(C,H)

Donde:
$\forall$ H:Set of Class $\forall$C:Class isCore(C,H) $\leftrightarrow$
(interface(C)$\neq\emptyset \wedge \forall S \in$H(interface(C)$\subseteq$interface(S) $\wedge$ $\forall n \in$interface(C) polymorphic(n,C,S)))

Puede observarse, a partir de la definición, que para una jerarquía H el conjunto {C | isCore(C,H)} puede tener cardinalidad mayor que uno. En general estaremos interesados en el maximal de este conjunto, es decir una clase C con mayor cantidad de atributos y métodos.

8.2 Una métrica para cuantificar polimorfismo

En la sección anterior hemos presentado una definición rigurosa de polimorfismo en el marco de la M&D-theory. Basándonos en dicha definición propondremos una nueva métrica de polimorfismo. La definición formal provee un mecanismo objetivo y preciso para detectar la presencia de *polimorfismo* en un diseño orientado a objetos. La métrica dará una técnica para cuantificar el polimorfismo. Luego, la medida de polimorfismo debe ser combinada con las medidas de las restantes propiedades (tales como acoplamiento, cohesión, entropía, etc.) con el objetivo de determinar la calidad total del sistema. Sin embargo esta tarea de combinación de métricas está fuera del alcance del presente trabajo.

Definimos la siguiente función sobre un modelo de diseño S (donde S es un ModelElement de tipo Model, es decir un Package conteniendo la definición completa del sistema):

- hierarchies(S) retorna la colección conteniendo todas las jerarquías no-triviales disjuntas definidas en el modelo S.

Y definimos las siguientes funciones sobre una jerarquía (árbol) de clases h:

- classes(h) retorna un conjunto conteniendo todas las clases que pertenecen a la jerarquía h.
- methods(h) retorna un conjunto conteniendo todos los métodos definidos en las clases de h:

  methods(h)= $\bigoplus_{c \in classes(h)}$ interface(c), donde $\bigoplus$ representa unión de bags (con repeticiones).
- core(h) retorna el mayor núcleo de la jerarquía h.
- width(h) retorna el ancho de una jerarquía h y se define de la siguiente forma:

  width(h) = #(interface(core(h))) , donde el símbolo # representa cardinalidad de conjuntos.
- children(h) retorna el conjunto de hijos de h.

Sea S un Modelo UML, la medida de polimorfismo de S se obtiene promediando las medidas de polimorfismo de todas las jerarquías disjuntas de S, de la siguiente forma:

***polymorphism_metric*:** System $\rightarrow$ 0..1

$$\textbf{\textit{polymorphism\_metric}}(S) = \frac{\sum_{h \in hierarchies(S)} polymorphism\_measure(h)}{\#hierarchies(S)}$$

Donde la función polymorphism_measure está definida de la siguiente forma:

polymorphism_measure: Hierarchy $\rightarrow$ (0..1)

polymorphism_measure(h) = polymorphic_methods(h) / #(methods(h))

polymorphic-methods(h)= width(h) * (#classes(h) –1)

+ $\sum_{hi \in children(h)}$ polymorphic-methods(hi–core(h))

Donde (hi – core(h)) es la jerarquía que se obtiene eliminando de las clases de hi a todos los métodos pertenecientes a core(h).

Destaquemos que el rango de la función ***polymorphism_metric*** es el intervalo 0...1 donde el número cero representa ausencia de polimorfismo y el número 1 representa la máxima medida de polimorfismo (es decir todos los métodos son polimórficos). Cuando la métrica es aplicada sobre una jerarquía trivial (compuesta por sólo una clase) se obtiene el número cero.

8.3 Ejemplos

8.3.1 Identificando polimorfismo

La jerarquía de colecciones ha sido definida en varios lenguajes orientados a objetos. La figura 8.1 muestra parte de dicha jerarquía en el lenguaje Smalltalk (ver [Lalonde et al.]). Utilizando las definiciones formales dadas en la sección anterior, identificaremos la presencia de polimorfismo en estas clases.

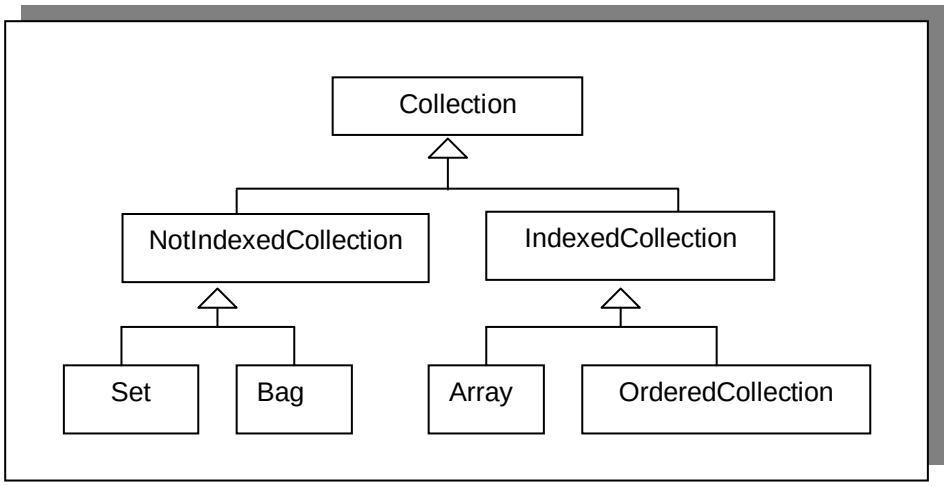


Figura 8.1: Jerarquía de colecciones

- **Atributos polimórficos**

Las clases de esta jerarquía poseen un atributo llamado size que contiene un número entero representando la cantidad de elementos contenidos en la colección. Este atributo es polimórfico para todas las clases. Como ejemplo veremos que se satisface la definición para las clases Set y Bag.

Tesis: el atributo size es polimórfico para las clases Set y Bag,

Polymorphic(size,Set,Bag)

Hipótesis: La jerarquía contiene dos clases, sean Bag y Set, con un atributo llamado *size* de tipo Integer, y por lo tanto:

$\exists a_1,a_2$:Attribute ($a_1 \in$ attributes(Set) $\wedge$ $a_2 \in$ attributes(Bag)

$\wedge$ name(a_1)=size $\wedge$ name(a_2)=size $\wedge$ type(a_1)=type(a_2))

Demostración: aplicando modus ponens a la definición del predicado Polymorphic y a la hipóteis, se tiene que:

Polymorphic(size,Set,Bag) □

- **Métodos polimórficos**

El método *remove* es un método polimórfico en algunas clases de esta jerarquía ya que tiene la misma signatura y además el mismo efecto. Como ejemplo demostraremos que *remove* es polimórfico para las clases Set y OrderedCollection, donde elimina todas las ocurrencias de un elemento que llega como parámetro (en el caso de Set sólo puede existir una ocurrencia del elemento, mientras que en una OrderedCollection el elemento puede aparecer varias veces).

Tesis:

remove es polimórfico para las clases Set y OrderedCollection,

Polymorphic(*remove*, Set,OrderedCollection)

Hipótesis:

En la jerarquía de Collection existe una operación removeInSet perteneciente a operations(Set) y existe una operación removeInCol perteneciente a operations(OrderedCollection), tales que:

name(removeInSet)=*remove*
parameters(removeInSet)= <p1>
defaultValue(p1)=nullElement
kind(p1)=in
type(p1)=Object
$\forall$s,x:Object (classifier(s)=Set $\rightarrow$ [s.*remove*(x)]x$\notin$s)

name(removeInCol)=*remove*
parameters(removeInCol)= <p2>
defaultValue(p2)= nullElement
kind(p2) = in
type(p2)=Object
$\forall$c,x:Object
(classifier(c)=OrderedCollection $\rightarrow$ [c.*remove*(x)]occurrences(c,x)=0)

Demostración:

Primero probaremos dos lemas:

***Lema 1*:** ambas operaciones tienen la misma signatura:

hasSameSignature(removeInSet,removeInCol)

Demostración:

Por hipóteis tenemos que:

[1]defaultValue(p1)=defaultValue(p2) $\wedge$ kind(p1)=kind(p2) $\wedge$ type(p1)=type(p2)

Aplicando modus ponens en la definición del predicado equivalent se obtiene:

[2]equivalent (p1,p2)

Ahora aplicando la defínción del predicado areEquivalent se obtiene:

[3]areEquivalent(parameters(removeInSet), parameters(removeInCol)))

ya que cada lista de parámetros contiene un solo elemento y ya se mostró que son equivalentes.

Luego por [3] y por la hipótesis de que name(removeInCol)=remove y name(removeInSet)=remove y aplicando modus ponens en la definición del predicado hasSameSignature, se obtiene:

[4]hasSameSignature(removeInSet,removeInCol) □

***Lema 2*:** ambas operaciones tienen el mismo comportamiento

hasSameBehavior(removeInSet,removeInCol)

Demostración:

Por hipótesis tenemos que:

[*hipótesis 1*] $\forall$s,x:Object (classifier(s)=Set $\rightarrow$ [s.*remove*(x)]x$\notin$s)

[*hipótesis 2*] $\forall$c,x:Object

(classifier(c)=OrderedCollection $\rightarrow$ [c.*remove*(x)]occurrences(c,x)=0)

Debemos mostrar que para cualquier fórmula ϕ se cumple lo siguiente:

$\forall$s$\in$instances(Set)[s.remove(x)]$\phi\leftrightarrow$[s.migrates(OrderedCollection)] [s.remove(x)] [s.migrates(Set)]ϕ

y además:

$\forall$s$\in$instances(OrderedCollection)[s.remove(x)]$\phi \leftrightarrow$

[s.migrates(Set)] [s.remove(x)] [s.migrates(OrderedCollection)]ϕ

Dado que la semántica de estas especificaciones es de mínimo cambio, solo será necesario analizar las postcondiciones del mensaje *remove* en Set y las postcondiciones del mensaje *remove* en OrderedCollection, es decir:

[1] $\forall$s$\in$instances(Set) [s.remove(x)] x$\notin$s $\leftrightarrow$

[s.migrates(OrderedCollection)] [s.remove(x)] [s.migrates(Set)]x$\notin$s

[2] $\forall$c$\in$instances(OrderedCollection) [c.remove(x)] occurrences(c,x)=0 $\leftrightarrow$

[c.migrates(Set)] [c.remove(x)] [c.migrates(OrderedCollection)] occurrences(c,x)=0

La demostración es trivial ya que la fórmula occurrences(s,x)=0 es equivalente a la fórmula x$\notin$s y la fórmula occurrences(s,x)>0 es equivalente a la fórmula x$\in$s, tanto para Set como para OrderedCollection. □

Ahora estamos en condiciones de demostrar el teorema. Recordemos que la definición del predicado Polymorphic es la siguiente:

[1]Polymorphic(m,C_1,C_2) $\leftrightarrow$ $\exists m_1,m_2$:Operation ($m_1\in$operations(C_1) $\wedge$ $m_2\in$operations(C_2) $\wedge$

name(m_1)=m $\wedge$ name(m_2)=m $\wedge$

hasSameSignature(m_1,m_2) $\wedge$ hasSameBehavior(m_1,m_2))

[2] Por hipótesis tenemos que existen dos operaciones removeInSet perteneciente a operations(Set) y removeInCol perteneciente a operations(OrderedCollection), cuyo nombre coincide y es *remove*.

[3]Por el lema 1 tenemos que hasSameSignature(removeInSet,removeInCol)

[4]Por el lema 2 tenemos que hasSameBehavior(removeInSet,removeInCol)

Luego, aplicando modus ponens entre la conjunción de [2], [3] y[4] y la defínción [1], se obtiene:

Polymorphic(*remove*,Set,OrderedCollection) □

- **Métodos no polimórficos**

Consideremos la siguiente variante en la definición de *remove*:

[*1*] $\forall$s,x:Object (classifier(s)=Set $\rightarrow$ [s.*remove*(x)]x$\notin$s)

[*2*] $\forall$c,x:Object

(classifier(c)=OrderedCollection$\rightarrow$ (n=occurrences(c,x)$\wedge$n>0 $\rightarrow$[c.*remove*(x)]occurrences(c,x)=n-1))

Ahora el mensaje c.*remove*(x) borra sólo una ocurrencia de x dentro de la colección. Este cambio no será percibido por los Sets dado que no contienen repeticiones.

Con la nueva definición los métodos ya no son polimórficos. Es posible demostrar la verdad de la primera equivalencia:

[1] $\forall$s$\in$instances(Set) [s.remove(x)] x$\notin$s $\leftrightarrow$

[s.migrates(OrderedCollection)] [s.remove(x)] [s.migrates(Set)]x$\notin$s

Pero la segunda equivalencia falla, ya que si por ejemplo tenemos una OrderedCollection c y un elemento x tal que occurrences(c,x)=3, entonces n=3 y n>0 y no se cumple la siguiente equivalencia:

[c.*remove*(x)]occurrences(c,x)=n-1 $\not\leftrightarrow$

[c.migrates(Set)] [c.remove(x)] [c.migrates(OrderedCollection)] occurrences(c,x)=n-1

Ya que luego de la acción c.*remove*(x) tenemos que occurrences(c,x)=2, mientras que luego de la migración de c hacia la clase Set y la aplicación de la operación remove tenemos que occurrences(c,x)=0 ya que de acuerdo con el comportamiento de la operación remove sobre Set se tiene que x$\notin$c.

8.3.2 Aplicando la métrica

Describimos tres diferentes variantes de un diseño orientado a objetos expresado en UML. Para simplificar la presentación los métodos y los atributos polimórficos han sido detectados y llevados hacia arriba en la jerarquía de clases. Por lo tanto la clase raíz de la jerarquía actúa como núcleo.

Análisis de la jerarquía de la figura 1:

polymorphism_measure(h_A) = polymorphic_methods(h_A) / #(methods(h_A))

= polymorphic_methods(h_A) / 15 [como #(methods(h_A)) =15]

= (width(h_A)*(#(classes(h_A)-1)

+ $\sum_{hi \in children(h_A)}$ polymorphic-methods(h_i – core(h_A))) /15 [por definición de polymorphic_methods(h_A)]

= (4 * 2 + $\sum_{hi \in children(h_A)}$ polymorphic-methods(h_i – core(h_A))) /15 [porque width(h_A)=4, #(classes(h_A)=3]

= (8 + 0) / 15 = 0.53 [porque children de h_A son jerarquías triviales]

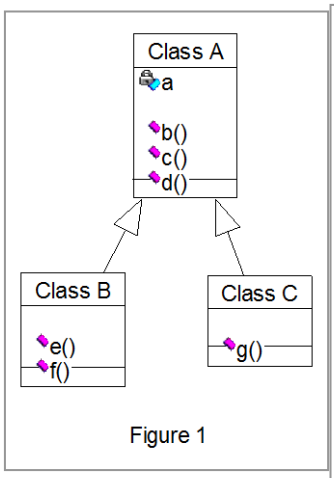


Figure 1

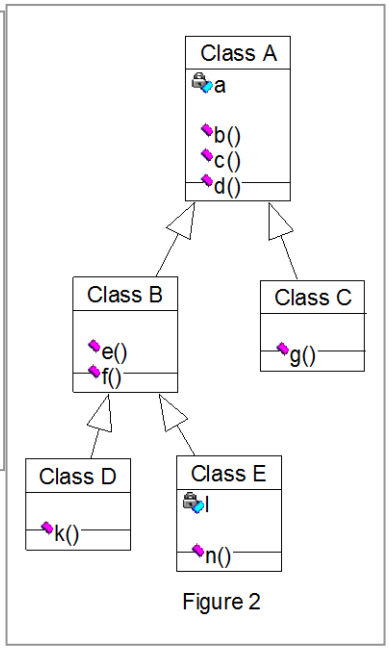


Figure 2

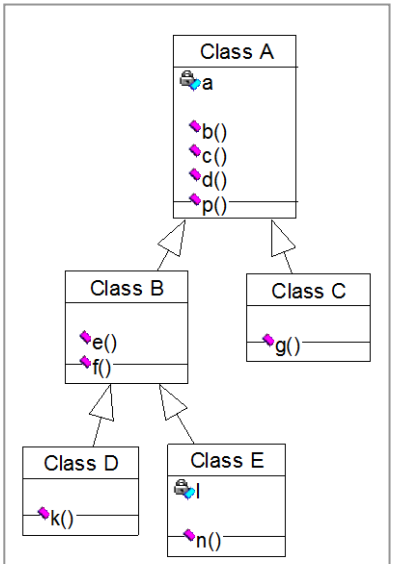


Figure 3

Análisis de la jerarquía de la figura 2:

polymorphism_measure(h_A) = polymorphic_methods(h_A) / #(methods(h_A))

= polymorphic_methods(h_A) / 30

= (width(h_A)*(#(classes(h_A)-1)

+ $\sum_{hi \in children(h_A)}$ polymorphic-methods(h_i – core(h_A))) /30

y como width(h_A)=4, #classes(h_A)=5, h_A tiene una jerarquía no trivial, h_B.

= (4 * 4 + polymorphic-methods(h_B – core(h_A))) /30

= (16 + (width(h_B)*#(classes(h_B) + 0) / 30

= (16 + 2*2) / 30 = 20/30 = 0.66

Análisis de la jerarquía de la figura 3:

polymorphism_measure(h_A) = polymorphic_methods(h_A) / #(methods(h_A))

= polymorphic_methods(h_A) / 35

= (width(h_A)*(#(classes(h_A)-1)

+ $\sum_{hi \in children(h_A)}$ polymorphic-methods(h_i – core(h_A))) /35

y como width(h_A)=5, #(classes(h_A)=5, h_A tiene una jerarquía no trivial, h_B.

= (5 * 4 + polymorphic-methods(h_B – core(h_A))) /35

= (20 + (width(h_B)*(#(classes(h_B)-1) + 0) / 35

= (20 + 2*2) / 35 = 24/35 = 0.68

Comparación:

Los resultados obtenidos han sido 0.53 para el diseño número 1, 0.66 para el número 2 y 0.68 para el número 3. Puede observarse que el polimorfismo, dado por la cantidad de métodos compartidos por las subclases, es notablemente mayor en la tercera versión, lo que hace que su métrica de polimorfismo sea mayor que las métricas de las restantes versiones.

8.4 Conclusiones

La calidad de un diseño es algo difícil de evaluar ya que es un concepto abstracto y complejo integrado por diferentes aspectos. Sin embargo la comunidad informática ha logrado reconocer un conjunto de propiedades que caracterizan a un buen diseño orientado a objetos.

Nuestra postura consiste en que ambas clases de propiedades, las tradicionales (por ej. acoplamiento y cohesión) y las nuevas propiedades orientadas a objetos (por ej. polimorfismo), deben ser consideradas conjuntamente para medir la calidad de un diseño orientado a objetos. Sin embargo creemos que es necesario prestar especial atención al concepto de *polimorfismo*, ya que debe ser considerado el punto clave para determinar la calidad de un diseño orientado a objetos.

En este capítulo hemos presentado una definición rigurosa de polimorfismo en el marco de la M&D-theory y basándonos en dicha definición hemos propuesto una nueva métrica de polimorfismo. La definición formal provee un mecanismo objetivo y preciso para detectar y cuantificar la presencia de *polimorfismo* en un diseño orientado a objetos.

Dado que la métrica se aplica sobre el modelo de diseño del sistema, el cual puede evolucionar, resulta necesario re-aplicar la métrica cada vez que el modelo es modificado. La métrica propuesta sólo ha sido aplicada sobre pequeños ejemplos, aún es necesario realizar su validación, es decir aplicarla sobre sistemas reales de mediana a gran envergadura y analizar sus resultados.

9. Formalizando re-uso

9.1 Componentes reusables

Durante el proceso de construcción de un nuevo sistema de software es importante contar con la posibilidad de re-utilizar componentes pre-existentes, evitando de esta manera realizar el mismo trabajo una y otra vez. Esto claramente conduce a economizar tiempo, recursos humanos y financieros. Para que un componente sea reusable (es decir factible de ser utilizado en varios sistemas) es necesario contar con:

- la especificación de la estructura y comportamiento del componente.
- un mecanismo para identificar aquellos componentes que pueden ser incorporados (re-usados) en un nuevo sistema en desarrollo. Por ejemplo, si ϕ es la especificación de un componente y γ es la especificación de un módulo requerido por el sistema en construcción, entonces es necesario poder determinar si ϕ satisface (o conforma) a γ.

Numerosos trabajos han sido desarrollados alrededor del tema de los componentes reusables. En este capítulo nos concentraremos en la definición de Contratos dada por Helm y Holland en [Helm and Holland 90]. Un contrato define el comportamiento de un grupo de participantes interrelacionados, identificando: quienes son los participantes del contrato y cuales son sus obligaciones contractuales. Las obligaciones contractuales consisten en:

- Obligaciones de tipo (interfaz externa del participante).
- Obligaciones causales (en respuesta a cada mensaje el participante debe ejecutar ciertas acciones y debe garantizar el cumplimiento de ciertas condiciones).
- Invariantes (condiciones que deben cumplirse en todo momento).

Mediante las obligaciones de tipo, los contratos capturan las dependencias estructurales entre los participantes (incluyendo interfaces); mientras que mediante las obligaciones causales los contratos especifican como el comportamiento de un participante se relaciona con el comportamiento de los demás participantes.

Como ejemplo, consideremos el contrato integrado por un objeto (que llamaremos *subject o model*) que posee ciertos datos y una colección de objetos (que llamaremos views), los cuales representan los datos gráficamente, por ej. un histograma, una tabla y un diagrama de torta(ver figura 9.1). Estos objetos cooperan de manera tal que cada *view* siempre refleja el valor corriente de su *subject*.

9.1.1 Especificación formal del contrato

La figura 9.2 muestra este contrato expresado en el lenguaje de alto nivel provisto por Helm y Hollan. En la figura 9.3 presentamos este mismo contrato expresado en lógica dinámica M&D. La fórmula rotulada con [1] establece las obligaciones de tipo del participante llamado Subject, en la fórmula [2] se expresan las obligaciones de tipo del participante View. La fórmula [3] establece la existencia de una asociación entre ambos participantes (esta también constituye una obligación de tipo). La fórmula [4] contiene las obligaciones causales y finalmente la fórmula [6] contiene el invariante.

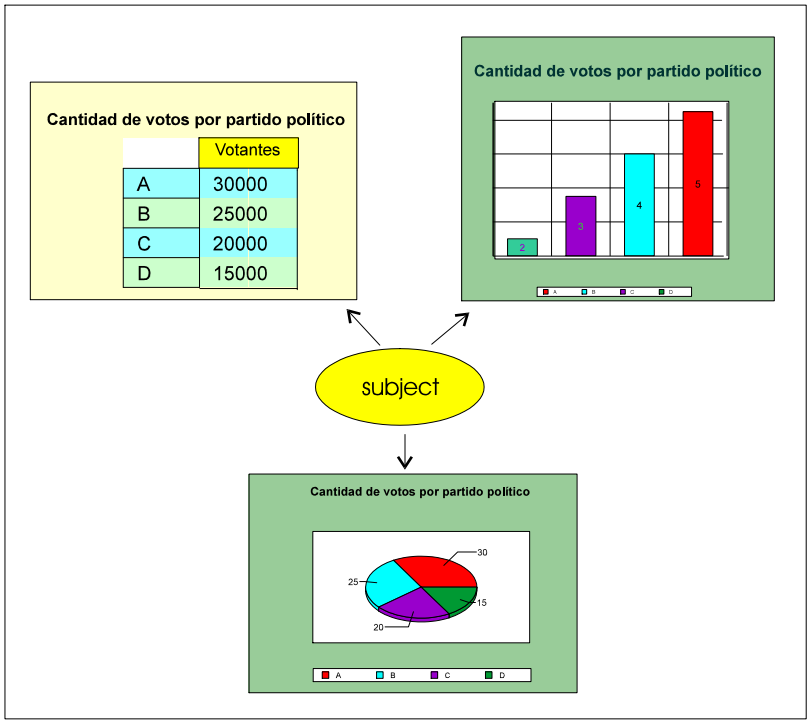


Figura 9.1: dependencias entre el subject y sus views

```
contract SubjectView

  Subject supports [
        data: Data
        setData(val:Data)       ⊗Δdata {data=val};notify()
        notify()                ⊗ ⟨∀v:v∈views: v.update()⟩
        attachView(v:View)      ⊗ {v∈views}
        detachView(v:View)      ⊗ {v∉views}
        ]
  Views:Set(View) where each View supports [
        update()                ⊗ {view reflects subject.data}
        setSubject(s:Subject)   ⊗ {subject=s)}
        ]
  invariant
        subject.setData(val)    ⊗ ⟨∀v∈views: v reflects subject.data⟩

end contract
```

Figura 9.2: contrato SubjectView extraído de [Helm and Holland 90]

$$
\begin{array}{ll}
 & \exists sv,:Class \\
 & \quad (\\
[1] & \quad name(s)=Subject \\
 & \quad \wedge \exists a:Attribute \\
 & \quad\quad (a \in attributes(s) \wedge\ name(a)=data \wedge type(a)=Data) \\
 & \quad \wedge \exists p_1,p_2,p_3,p_4:Operation \\
 & \quad\quad (\{p_1,p_2,p_3,p_4\} \subseteq operations(s) \wedge\ name(p_1)=attachView \wedge name(p_2)=dettachView \wedge \\
 & \quad\quad name(p_3)=setData \wedge name(p_4)=notify \\
 & \quad\quad \wedge\ parameters(p_1)=.. \wedge parameters(p_2)=..\wedge parameters(p_3)=..\wedge parameters(p_4)=..) \\
 & \quad \wedge \\
[2] & \quad name(v)=View \\
 & \quad \wedge \exists p_1,p_2:Operation \\
 & \quad (\{p_1,p_2\} \subseteq operations(v) \wedge\ name(p_1)=setSubject \wedge name(p_2)=update \\
 & \quad\quad \wedge\ parameters(p_1)=.. \wedge parameters(p_2)=..) \\
 & \quad \wedge \\
[3] & \quad \exists a:Association \\
 & \quad\quad (\exists e1,e2:AssociationEnd\ (connections(a)=\{e1,e2\} \\
 & \quad\quad \wedge\ name(e1)=subject \wedge type(e1)=s \wedge multiplicity(e1)=0..1 \\
 & \quad\quad \wedge aggregation(e1)=\#aggregation \\
 & \quad\quad \wedge\ name(e2)=view \wedge type(e2)=v \wedge multiplicity(e1)=0..n \wedge aggregation(e2)=\#none)) \\
 & \quad \wedge \\
[4] & \quad \forall si \in instances(s)\ \forall vi \in instances(v) \\
 & \quad\quad (\ [si.attachView(vi)]vi \in value(si,views) \\
 & \quad\quad\quad \wedge [si.dettachView(vi)]vi \notin value(si,views) \\
 & \quad\quad\quad \wedge [si.setData(n)]\ (value(si,data)=n \wedge Enabled(si.notify)) \\
 & \quad\quad\quad \wedge [si.notify] \forall vi \in value(si,views)\ Enabled(vi.update) \\
 & \quad\quad\quad \wedge \\
 & \quad\quad\quad [vi.setSubject(si)]value(vi,subject)=si \\
 & \quad\quad\quad \wedge [vi.update]reflects(vi,value(vi,subject)) \\
 & \quad \wedge \\
[5] & \quad vi \in value(si,views) \rightarrow reflects(vi,si)\ ) \\
 & \quad\quad) \\
 & \quad)
\end{array}
$$

Figura 9.3: contrato SubjectView expresado en M&D-logic

9.1.2 Satisfacción de las obligaciones contractuales

Hemos señalado que se necesita un mecanismo para identificar aquellos componentes que pueden ser re-usados en un sistema en desarrollo. Por ejemplo, podría ser necesario identificar aquellos componentes que satisfagan un determinado contrato. Sea κ la especificación de un componente y sea Φ un contrato, entonces es necesario poder determinar si κ satisface (o conforma) a Φ. Para proveer cierta flexibilidad sintáctica Helm y Holland han definido el concepto de "conformance declarations", que consiste en un conjunto de asociaciones de la forma $a \rightarrow b$ para indicar que el identificador *a* del componente se asocia con el identificador *b* del contrato.

Nuestra lógica permite representar estas ideas en forma elegante, por ejemplo "conformance declarations" se expresan mediante sustituciones de términos. Por otra parte el aparato deductivo

de la lógica ofrece un mecanismo formal para determinar si un componente cumple con un contrato.

Veamos un ejemplo.

Sean:

- *M* la especificación de un sistema orientado a objetos expresada en UML.
- $Spec_M$ la especificación expresada en lógica dinámica que interpreta formalmente a *M*, es decir ***translation***(*M*)=$Spec_{M\&D.}$
- U un modelo para $Spec_M$, es decir U $\in$***semantics***($Spec_M$).
- *SpecificSubject* y *SpecificView* dos clases presentes en el modelo *M*, con las siguientes declaraciones (i.e conformance declarations): {*SpecificSubject*$\rightarrow$*Subject*, *SpecificView*$\rightarrow$*View*}
- Φ_{SV} la fórmula que expresa el contrato (ver figura 9.3).

Decimos que *M* conforma el contrato Φ_{SV} si y sólo si satisface las obligaciones de tipo, obligaciones causales e invariantes requeridos por el contrato. Es decir que la fórmula Φ_{SV} (bajo sustitución de nombres de clases) es verdadera en los modelos de *M*:

$$U \models \Phi_{SV} \text{ [Subject/SpecificSubject, view/SpecificView]}$$

9.1.3 Refinamiento e inclusión de contratos

La M&D_logic nos da la posibilidad de expresar formalmente los conceptos de refinamiento e inclusión de contratos.

Refinamiento

Un *contrato refinado* define obligaciones más especializadas que las del contrato base. Sean Φ_1 y Φ_2 dos M&D-fórmulas expresando los contratos C_1 y C_2 respectivamente.

C_1 es un refinamiento de C_2 si y sólo si $\Phi_1 \rightarrow \Phi_2$

o trasladándonos al nivel semántico, podemos decir que todo modelo del contrato refinado Φ_1 es también un modelo para el contrato base Φ_2:

C_1 es un refinamiento de C_2 si y sólo si $Sem(\Phi_1) \subseteq Sem(\Phi_2)$

Inclusión

Frecuentemente el comportamiento de un sistema complejo puede ser expresado mediante la composición de sub-sistemas más simples. Análogamente, un contrato puede estar definido en términos de un grupo de sub-contratos. Sean $\Phi,\Phi_1,...,\Phi_n$ M&D-fórmulas expresando los contratos $C,C_1,...,C_n$ respectivamente. El contrato C es la composición de $C_1,...,C_n$ si y sólo si:

$$\Phi \leftrightarrow \Phi_1 \wedge ... \wedge \Phi_n$$

9.2 Patrones de Diseño

En torno a la necesidad de re-utilizar parte del trabajo realizado previamente para facilitar el desarrllo de un nuevo sistema, surgió una idea de re-uso más conceptual y abstracta que la de los

los componentes reusables. Esta idea consistió en re-usar la experiencia de los diseñadores mediante la identificación de patrones de diseño. Un patrón es un diseño particular que se repite en determinadas situaciones y que ha sido reconocido como “buen diseño”, es decir que conduce a obtener sistemas más flexibles y elegantes y consecuentemente más re-usables. Un patron de diseño [Gamma et al 94] consta de cuatro elementos escenciales:

- el *nombre* del patrón, que se usa para identificarlo.
- el *problema*, que describe la situación en la cual el patrón es aplicable.
- la *solución*, que describe los elementos que integran el patrón, sus relaciones, responsabilidades y colaboraciones.
- las *consecuencias*, que son los resultados (costos y veneficios) de aplicar el patrón.

Existen catálogos como por ejemplo [Gamma et al 94] donde se describen numerosos patrones de diseño utilizando lenguaje natural complementado con lenguajes gráficos como UML. Sería deseable contar con una descripción más formal de los patrones que permitiera lo siguiente:

- los diseñadores lograrían un entendimiento mas preciso de los elementos de cada patrón, logrando de esta forma durante el desarrollo de un nuevo sistema un uso mas seguro y efectivo de los patrones adecuados .
- durante la tarea de re-ingeniería sería posible detectar formalmente la presencia de determinados patrones en los de viejos sistemas

Con el objetivo de mostrar que la M&D-logic puede asistir en la tarea de expresar y reconocer patrones de diseño, en esta sección describiremos el patrón conocido como Cadena de Responsabilidad (Chain of Responsibility) y luego lo expresaremos formalmente en el lenguaje lógico.

Descripción:

La idea de este patrón de diseño es lograr un desacople entre el emisor de un pedido y el receptor de tal pedido. Esto se logra permitiendo que más de un objeto sea capaz de atender el pedido, de esta forma el receptor no está fijado de antemano, sino que existe una cadena de objetos que potencialmente recibirán el pedido y el receptor real se decide durante la ejecución. Entonces, el primer objeto en la cadena recibe el pedido y o bien lo atiende o bien lo transfiere al siguiente objeto en la cadena, de esta forma el pedido va avanzando a través de la cadena hasta ser atendido. El objeto que hace el pedido no sabe explícitamente cual objeto lo atenderá.

Estructura:

Este patrón define una jerarquía de clases cuya raiz es la clase abstracta Handler. Toda instancia de la clase Handler está relacionada (a través de la asociación successor) con otra instancia de la misma clase. Existen además otras clases (concretas) que representan los distintas clases de objetos que integran la cadena. Cada una de estas clases especializa el mensaje handleRequest() que representa la atención del pedido. Además existe la clase Client cuyas instancias serán las emisoras de los pedidos. Un cliente está conectado al primer handler en la cadena.

El diagrama UML de la figura 9.4 muestra la estructura del patrón.

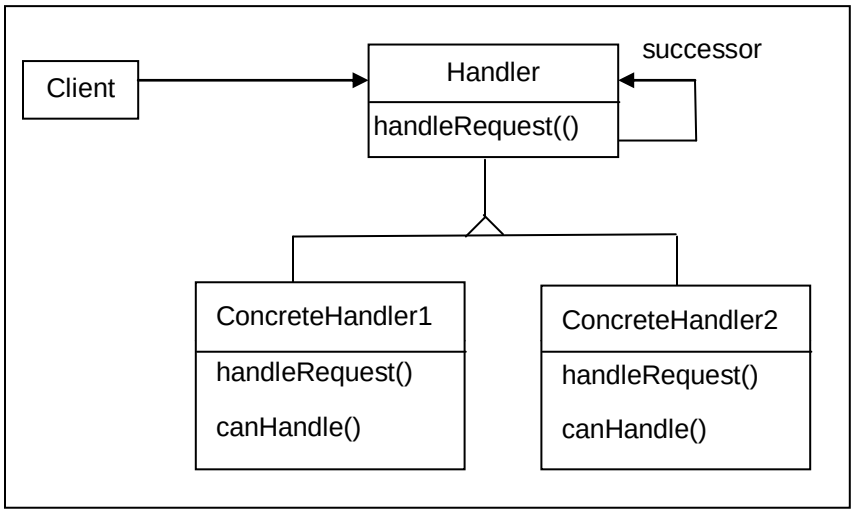


Figura 9.4: patrón de diseño Cadena de responsabilidad

Colaboraciones:

Cuando un cliente emite un pedido , este pedido se propaga a lo largo de la cadena hasta que una instancia de algún ConcreteHandler$_i$ asume la responsabilidad de atenderlo.

9.2.1 Especificación formal del patrón

El patrón Cadena de responsabilidad es formalizado a través de la siguiente fórmula que llamaremos Φ_{Chain},

$\exists$h:Class (

[1] name(h)=Handler

[2] $\wedge$ $\exists$a:Association
(∃e1,e2:AssociationEnd (connections(a)={e1,e2}
$\wedge$ name(e1)=successor $\wedge$ type(e1)=h $\wedge$ multiplicity(e1)=0..1 $\wedge$aggregation(e1)=#none
$\wedge$ type(e2)=h $\wedge$ multiplicity(e1)=0..1 $\wedge$ aggregation(e2)=#none))

[3] $\wedge$ $\exists p_1,p_2 \in$operations(h) $\exists$q,r,s:Parameter
(name(p_1)= canHandle $\wedge$ parameters(p_1)=$\langle$q,r$\rangle$
$\wedge$ type(q)=Request $\wedge$ kind(q)=#in $\wedge$ type(r)=Boolean $\wedge$ kind(r)=#return $\wedge$
name(p_2)=handleRequest $\wedge$ parameters(p_2)=$\langle$s$\rangle$ $\wedge$ type(s)=Request$\wedge$ kind(s)=#in)

[4] $\wedge$ $\exists$g:Generalization supertype(g)=h

[5] $\wedge$ $\forall$h'$\in$allSubtypes(h) ($\forall$o$\in$instances(h´)

[o.handleRequest(r)] (Enabled(succ.handleRequest(r)) $\leftrightarrow$ [o.canHandle(r,x)]x=false))

)

donde succ=value(o,successor)

La fórmula [1] indica el nombre de la clase sobre la que se aplicará el patrón. Las fórmulas [2] y [3] establecen obligaciones estructurales (nombres y tipos de asociaciones y operaciones). La fórmula [4] establece la obligación de que la clase Handler esté especializada. La fórmula [5] expresa las colaboraciones entre los objetos. Esta fórmula dinámica expresa que si *o* es el objeto que actualmente ha recibido el pedido, luego de la ejecución de la acción *o*.handleRequest(r), el pedido será delegado a su sucesor en la cadena si y sólo si dicho objeto *o* no ha podido atenderlo.

9.2.2 Reconocimiento del patrón

Sean:

- *M* la especificación de un sistema orientado a objetos expresada en UML.
- $Spec_M$ la especificación expresada en lógica dinámica que interpreta formalmente a *M*, es decir ***translation***(*M*)=$Spec_M$.
- U un modelo para $Spec_M$, es decir U $\in$***semantics***($Spec_M$).
- *SpecificHandler* una clase presente en el modelo *M*, con la siguientes conformance declaration: { *SpecificHandler*$\rightarrow$ *Handler* }
- Φ_{Chain} la fórmula que expresa el patrón.

Decimos que *M* contiene al patrón Φ_{Chain} (o bien que patrón Chain of Responsability está presente en el diseño *M*) si y sólo si satisfacen los requerimientos de estructura y colaboraciones establecidos en el patrón. Es decir que la fórmula Φ_{Chain} (bajo sustitución de nombres de clases) es verdadera en los modelos de *M*:

$$U \models \Phi_{Chain}[Handler / SpecificHandler]$$

9.3 Conclusiones

Durante el proceso de construcción de un nuevo sistema de software es importante contar con la posibilidad de re-utilizar componentes pre-existentes, evitando de esta manera realizar el mismo trabajo una y otra vez. Los *contratos* son una herramienta de especificación cuyo objetivo es favorecer la re-utilización correcta de componentes.

Hemos señalado que se necesita un mecanismo para identificar aquellos componentes que pueden ser re-usados en un nuevo sistema. Por ejemplo, podría ser necesario identificar los componentes que satisfacen un determinado contrato. En esta dirección, hemos mostrado que tanto componentes como contratos pueden ser expresados mediante una M&D-fórmula. De esta forma, el aparato deductivo de la lógica ofrece un mecanismo formal para determinar si un componente cumple con un contrato. Además, esta representación nos brinda la posibilidad de expresar formalmente los conceptos de refinamiento e inclusión de contratos.

Por otra parte hemos mostrado que la M&D-logic permite expresar patrones de diseño y consecuentemente permite identificar la presencia de determinados patrones dentro de un diseño a través de deducciones lógicas.

10. Resumen y conclusiones

10.1 El proceso de desarrollo de software basado en modelos

El punto de partida en el proceso de desarrollo de software es la construcción de un modelo, el cual actúa como una especificación precisa de los requerimientos que el sistema debe satisfacer. Un modelo del sistema consiste en una conceptualización del dominio del problema. El modelo se focaliza sobre el mundo real: identificando, clasificando y abstrayendo los elementos que constituyen el problema y organizándolos en una estructura formal.

El modelo del sistema se usa básicamente para los siguientes propósitos:

- *Para definir las necesidades del usuario*.
- *Como un medio de comunicación y negociación* entre los usuarios y los desarrolladores y entre los distintos desarrolladores entre sí.
- *Como un documento de referencia durante la corrección de errores* en el producto.
- *Como un documento de referencia durante la evolución* del producto.

Se ha observado que la construcción de modelos es una técnica muy efectiva para detectar y resolver discrepancias entre los divergentes puntos de vista de los usuarios acerca de sus requerimientos, brindando así bases firmes para las siguientes etapas del proceso de desarrollo.

Durante el proceso de desarrollo de software diferentes modelos del sistema en construcción son creados. Las diferencias entre estos modelos residen en los aspectos del sistema que son contemplados (ningún modelo representa al sistema completo, sino que cada modelo hace énfasis en una parte del sistema) y en el grado de abstracción (en las primeras etapas del ciclo de vida se construyen modelos más abstractos, que luego son sustituidos y/o complementados por modelos más concretos). Los distintos modelos están relacionados entre sí en dos direcciones: horizontal y vertical. Cada plano vertical está formado por un grupo de submodelos que conforman una visión completa del sistema a un cierto nivel de abstracción (o en un cierto punto de su ciclo de vida). Las relaciones horizontales representan evolución de un modelo a través del proceso de desarrollo, por ejemplo la relación entre un modelo de análisis y un modelo de diseño.

El modelo de un problema constituye la base fundamental de información sobre la que interactúan los expertos en el dominio del problema, los analistas y los desarrolladores de software. El modelo es esencial para describir y entender el problema, independientemente de cualquier posible sistema informático que se use para su automatización. Por lo tanto es de fundamental importancia que exprese la esencia del problema en forma clara y precisa. Por otra parte, los modelos son el resultado de una actividad compleja y creativa y por lo tanto son propensos a contener errores, omisiones e inconsistencias. Su verificación es muy importante, ya que los errores en la etapa de modelado tienen un costoso impacto sobre las siguientes etapas del proceso de desarrollo de software.

10.2 Modelos formales vs. modelos no-formales

El modelo del sistema se construye utilizando un lenguaje de modelado (que puede variar desde lenguaje natural o diagramas hasta formulas matemáticas). Los modelos informales son expresados utilizando lenguaje natural, figuras, tablas u otras notaciones. Hablamos de modelos formales cuando la notación empleada es un formalismo, es decir posee una sintaxis y semántica (significado) precisamente definidos. Existen estilos de modelado intermedios llamados semi-

formales, ya que en la práctica los ingenieros de software frecuentemente usan una notación cuya sintaxis y semántica están sólo parcialmente formalizadas.

El éxito de los lenguajes gráficos de modelado se basa principalmente en el uso de construcciones gráficas que transmiten un significado intuitivo; por ejemplo un cuadrado representa un objeto, una línea uniendo dos cuadrados representa una relación entre ambos objetos. Estos lenguajes resultan atractivos para los usuarios ya que aparentemente son fáciles de entender y aplicar. Sin embargo, la falta de precisión en la definición de su semántica puede originar problemas, por ejemplo:

- Malas interpretaciones de los modelos.
- Inconsistencia entre los diferentes modelos del sistema.
- Discusiones acerca del significado del lenguaje.

Por otro lado, los lenguajes formales de modelado poseen una sintaxis y semántica bien definidas. Sin embargo su uso en la industria es poco frecuente. Esto se debe a la complejidad de sus formalismos matemáticos que son difíciles de entender y comunicar.

La necesidad de integrar lenguajes gráficos, cercanos a las necesidades del dominio de aplicación con técnicas formales de análisis y verificación puede satisfacerse combinando ambos tipos de lenguaje. La idea básica para obtener una combinación útil consiste en ocultar los formalismos matemáticos detrás de la notación gráfica. De esta manera el usuario solo debe interactuar con el lenguaje gráfico, pero puede contar con la base formal provista por el esquema matemático subyacente. Esta propuesta ofrece claras ventajas sobre el uso de un lenguaje informal así como también sobre el uso de un lenguaje formal, ya que permite que los desarrolladores de software puedan crear modelos formales sin necesidad de poseer un conocimiento profundo acerca del formalismo que los sustenta. Un lenguaje que posea estas características será fácilmente aceptado tanto por parte de los ingenieros de software, como por parte de los usuarios.

10.3 Combinando técnicas de modelado formales con técnicas no-formales

En el capítulo 4 hemos descripto las diferentes propuestas para la integración de lenguajes de modelado gráficos con lenguajes de modelado formales:

- suplemento
- extensión
- interfaz
- semántica

Hemos destacado que la técnica consistente en definir formalmente la semántica de un lenguaje de modelado conocido y aceptado por la comunidad constituye la propuesta más adecuada, dado que permite que el lenguaje gráfico se convierta en un lenguaje formal y por lo tanto las especificaciones escritas utilizando el lenguaje gráfico pueden ser formalmente analizadas para detectar contradicciones y ambigüedades tempranamente en el proceso de desarrollo del software. Una de las claves para el éxito de esta propuesta reside en ocultar la notación matemática tanto como sea posible tras la notación gráfica. Por ejemplo, debería ser posible utilizar la semántica formal para desarrollar herramientas CASE. Sólo los desarrolladores del lenguaje deberían usar el formalismo para construir las herramientas CASE y justificar su correctitud, mientras que los desarrolladores de software de aplicación podrían manejar los modelos gráficos sin necesidad de conocer el formalismo matemático subyacente. Entre los numerosos trabajos teóricos que tratan

diferentes partes de UML definiendo formalmente su semántica hemos seleccionado los más representativos y los hemos descripto y clasificado en dos grupos: formalizaciones basadas en el modelo y formalizaciones basadas en el metamodelo.

Las principales ventajas y desventajas de cada enfoque son:

- El enfoque basado en el modelo es apropiado para la especificación de información inherente al dominio de la aplicación; permitiendo detectar inconsistencias y errores en las especificaciones escritas en el lenguaje de modelado. Sin embargo no permite expresar reglas de consistencia entre diferentes elementos de modelado, por ejemplo, propiedades acerca de la estructura del sistema, tales como relaciones entre clases, no pueden ser expresadas. Tampoco permite representar evolución del modelo en un formalismo de primer orden.
- El enfoque basado en el metamodelo permite especificar reglas de consistencia entre diferentes elementos de modelado (por ejemplo entre clases y maquinas de estado). También permite detectar inconsistencia y errores en la definición del lenguaje de modelado en sí y provee un marco formal claro y simple para expresar evolución y refinamiento de modelos. Su principal desventaja reside en que es difícil representar y analizar la información correspondiente al dominio de las aplicaciones descriptas con el lenguaje.

10.4 Nuestra propuesta de integración: la M&D-theory

En el capítulo 5 hemos presentado una nueva propuesta para definir formalmente la semántica de UML. La idea básica de esta nueva formalización consiste en utilizar un dominio semántico que integra los dos niveles inferiores de la arquitectura de las notaciones de modelado (es decir el nivel del modelo y el nivel de los datos), permitiendo de esta manera representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden, que llamamos M&D-theory.

Las entidades descriptas por la M&D-theory se clasifican en dos conjuntos disjuntos:

- Entidades de modelado
- Entidades modeladas

Las entidades de modelado se corresponden con construcciones (sintácticas) correctas del lenguaje UML, tales como clases (Class) y a máquinas de estados (StateMachine). En contraste, las entidades modeladas, tales como objetos (Object) y conexiones (Link) representan los datos del sistema modelado.

Nivel de los Modelos:

La especificación de este nivel consta de una signatura Σ_{UML}**= ((**S_{UML}**,≤), **F_{UML}**, **P_{UML}**, **A_{UML}**)** y una fórmula ϕ_{UML} sobre Σ_{UML}. Los elementos del álgebra inicial denotada por la especificación son elementos de modelado, tales como clases, asociaciones y máquinas de estados. La relación de transición entre posibles mundos representa modificaciones sobre la especificación del sistema, por ejemplo el agregado de una nueva clase, la modificación de una clase existente, etc. La fórmula ϕ_{UML} es la conjunción de dos conjuntos disjuntos de fórmulas, ϕ_S y ϕ_D de fórmulas estáticas y fórmulas dinámicas respectivamente. El primer conjunto consiste en fórmulas no modales que deben satisfacerse en todos los estados posibles del sistema (son invariantes o propiedades estáticas o reglas de buena formación de los modelos). Estas reglas se usan para realizar análisis de la estructura del sistema y reportar posibles errores en su diseño. Mientras que el segundo

conjunto consiste en fórmulas modales que definen la semántica de las acciones, es decir de la evolución de los modelos.

Nivel de los Datos:

La especificación formal de los elementos en el nivel de los datos consta de una signatura Σ_{SYS}**= (** ($\mathbf{S}_{SYS,}\leq$)**,** $\mathbf{F}_{SYS}$**,** $\mathbf{P}_{SYS}$, $\mathbf{A}_{SYS}$**)** y una fórmula γ_{SYS} sobre Σ_{SYS}. Los elementos del álgebra inicial denotada por la especificación son los datos del sistema y sus relaciones, tales como objetos, conexiones entre objetos, mensajes, etc. La relación de transición entre posibles mundos representa evolución de los datos, por ejemplo cambios en los valores de los atributos de los objetos. La fórmula ϕ_{SYS} es la conjunción de dos conjuntos disjuntos de fórmulas, γ_S y γ_D de fórmulas estáticas y fórmulas dinámicas respectivamente. El primer conjunto consiste en fórmulas no modales que deben satisfacerse en todos los estados posibles del sistema (son invariantes o propiedades estáticas o reglas de buena formación de los objetos). Mientras que el segundo conjunto consiste en fórmulas modales que definen la semántica de las acciones, es decir de la evolución de los datos.

La M&D-theory:

La M&D-theory (*Model&Data theory)* es una teoría dinámica de primer orden, formada por una signatura (la cual define el lenguaje de la teoría) y un conjunto de axiomas:

$$\text{M\&D-theory}=(\Sigma_{M\&D}, \phi_{M\&D})$$

La signatura de la teoría, $\Sigma_{M\&D}$=((**S**, ≤), **F, P, A**), es una signatura en lógica dinámica con las siguientes características especiales:

- La signatura$\Sigma_{M\&D}$ incluye a la signatura Σ_{UML}. Esto significa, por ejemplo que existe un conjunto distinguido de sorts $\mathbf{S}_{UML}\subseteq\mathbf{S}$ y un conjunto distinguido de funciones $\mathbf{F}_{UML}\subseteq\mathbf{F}$. Estos símbolos representan a los elementos de modelado, tales como Class y StateMachine. Usualmente son llamados metaclases o metasorts.

- La signatura $\Sigma_{M\&D}$ incluye a la signatura Σ_{SYS}. Esto significa, por ejemplo que existe un conjunto distinguido de sorts $\mathbf{S}_{SYS}\subseteq\mathbf{S}$. Estos símbolos representan a los objetos del sistema y sus relaciones, tales como Object y Message.

Por otra parte los axiomas de la teoría, están integrados por distintos grupos de axiomas. Es decir, $\phi_{M\&D}$ es la conjunción de tres fórmulas: $\phi_{M\&D} = \phi_{UML} \wedge \gamma_{SYS} \wedge \phi_{JOINT}$. Primeramente, ϕ_{UML} es la fórmula sobre Σ_{UML} que define las características de los elementos de modelado (esta fórmula se obtiene mediante la conjunción de todos los axiomas del nivel de los modelos). Luego, γ_{SYS} es la fórmula construida sobre Σ_{sys} que describe las características de los elementos modelados (esta fórmula se obtiene mediante la conjunción de todos los axiomas del nivel de los datos). Finalmente, ϕ_{JOINT} es una fórmula construida utilizando el lenguaje integrado *M&D* y por lo tanto puede expresar tanto propiedades de los modelos, como propiedades de los datos, como propiedades relacionando ambos niveles.

Interpretación semántica de UML

Las principales componentes de la interpretación semántica de UML son reglas para asociar estructuras sintácticas del lenguaje de modelado con elementos en un dominio semántico formalmente definido.

El dominio semántico donde las construcciones de UML son interpretadas está formado por sistemas de transición de la forma $U=(S^U,w_o,m_U)$. Notemos que el dominio de las interpretaciones es un álgebra heterogénea cuyos elementos incluyen tanto datos (por ejemplo objetos) como meta-datos (por ejemplo clases).

El conjunto de relaciones de transición entre mundos está particionado en dos conjuntos disjuntos:

- Un conjunto de transiciones que representan modificaciones sobre la especificación del sistema (evolución de los meta-datos).
- Un conjunto de transiciones que representan modificaciones sobre los datos del sistema (evolución de los datos).

Las relaciones entre los conceptos sintácticos de UML y los conceptos del dominio semántico se establecen a través de la función de interpretación semántica **Sem**.

Sem: ConstruccionesUML $\rightarrow$ DominioSemántico

la interpretación semántica de UML se obtiene en dos pasos:

1- interpretación (o traducción) del lenguaje UML en una teoría M&D.

2- interpretación semántica de la teoría M&D.

Es decir,

ConstruccionesUML $\rightarrow^{translation}$ **M&D-theory** $\rightarrow^{semantics}$ **Dominio-semántico**

La función ***translation*** asocia cada modelo UML bien formado con su correspondiente M&D-theory. Esta función está determinada por medio de un conjunto de reglas que permiten crear una M&D-theory a partir de los distintos submodelos relacionados que componen un modelo UML. Las reglas trabajan sobre una teoría básica. Las reglas van enriqueciendo progresivamente esta teoría, agregando dos clases de axiomas específicos:

- de instanciación ϕ_{INST} (describen cuales elementos de modelado son usados en este modelo)
- y de completación ϕ_{COMP} (describen características especiales del sistema modelado)

10.5 Evaluando la M&D-theory

La principal diferencia entre nuestra propuesta y las restantes propuestas discutidas anteriormente reside en que la integración de entidades de modelado y entidades modeladas dentro de la misma teoría permite representar los aspectos estáticos y dinámicos tanto del modelo como del sistema modelado dentro de un marco formal de primer orden. la siguiente tabla esquematiza el tratamiento que nuestra propuesta da a cada una de las cuatro dimensiones discutidas en el capítulo anterior:

	Modelo	Sistema modelado
Aspectos estáticos	Axiomas de primer orden sobre entidades de modelado	Axiomas de primer orden sobre entidades modeladas
Aspectos dinámicos	Acciones y axiomas modales sobre entidades de modelado.	Acciones y axiomas modales sobre entidades modeladas

Contar con una estructura formal de primer orden, en contraste con una estructura de orden superior, facilita los procedimientos para calcular la validez de las fórmulas. A pesar de que la lógica de primer orden es no-decidible (y por lo tanto también lo es la lógica dinámica de primer orden), los sistemas de computación satisfacen ciertas propiedades (por ejemplo, se interpretan sobre estructuras aritméticas, el estado de un programa en un momento dado queda determinado por un conjunto finito de valores) las cuales permiten calcular la validez de las fórmulas dinámicas en forma finita y efectiva.

Es importante destacar que la sintaxis abstracta y la semántica estática de UML han sido tratadas satisfactoriamente en distintos trabajos, sin embargo la semántica dinámica no ha sido descripta en forma precisa, por ejemplo el metamodelo de UML [UML 97 (b)] da una noción precisa la sintaxis abstracta del lenguaje, pero no trata adecuadamente su semántica ya que las reglas de buena formación son sólo restricciones sintácticas y de ninguna manera “condiciones semánticas” como suele creerse. Además el manual de UML no considera aspectos dinámicos del modelo (evolución o refinamientos) y expresa su semántica (el significado de las construcciones de UML) usando simplemente lenguaje natural.

Ventajas standard del modelo formal:

El modelo formal que hemos definido provee las siguientes ventajas:

- Las especificaciones expresadas en el lenguaje gráfico obtienen un significado preciso.
- Conduce a un entendimiento mas profundo de los conceptos expresados mediante la notación, lo cual permite un uso más maduro de la tecnología.
- La construcción del modelo formal nos permitió descubrir ambigüedades e inconsistencias en el lenguaje UML.
- Las distintas vistas de un sistema (class diagrams, statecharts, constraints, etc.) son integradas en un único modelo formal. Esta integración permite definir reglas de compatibilidad entre las vistas, tanto en el nivel sintáctico como en el semántico.
- Mediante mecanismos formales de deducción es posible obtener información no presentada explícitamente en la especificación.
- Técnicas formales de verificación disponibles en el modelo formal pueden usarse para detectar errores en la especificación gráfica.
- La notación matemática permanece oculta tras la notación gráfica, por lo tanto los desarrolladores de software no necesitan conocimientos matemáticos especiales.

La M&D-logic además incorpora ciertas características que no han sido cubiertas completamente por las propuestas anteriores, tales como modificabilidad (o evolución de modelos), métricas de calidad de modelos y reusabilidad. Estos temas han sido tratados en detalle en los capítulos 7,8 y 9 respectivamente. A continuación presentamos un resumen de cada uno de ellos:

Formalizando evolución:

La mayoría de los trabajos sobre evolución de especificación trata el problema de evolución estructural, por ejemplo modificar la jerarquía de herencia o agregar una nueva clase, pero no tratan el problema de evolución de comportamiento, como por ejemplo cambiar la forma en que un objeto reacciona al recibir un determinado mensaje. El mecanismo de evolución propuesto en esta tesis cubre ambos tipos de evolución.

Animando el sistema de transición que representa la semántica de la M&D-logic es posible analizar como reacciona el sistema a determinados cambios en su modelo (tanto cambios estructurales

como cambios en la definición del comportamiento de los objetos). Además es posible observar como se combina la evolución del modelo con la evolución de los datos modelados. La M&D-logic permite expresar reglas de consistencia entre diferentes diagramas UML y entre estos diagramas y los datos modelados. Luego, utilizando el mecanismo de deducción de la lógica es posible validar estas reglas a través de la evolución, garantizando la consistencia del sistema.

En resumen, los principales aportes del mecanismo de evolución propuesto son:

- Identifica un grupo completo de modificaciones primitivas.
- Define formalmente las condiciones de aplicabilidad de cada operación, es decir las condiciones bajo las cuales la aplicación de la operación es semánticamente correcta.
- Identifica y especifica la propagación de cambios provocada por la aplicación de cada operación.
- Considera *Modelos Dinámicos* ya que contempla la evolución combinada del modelo y los datos.
- Identifica situaciones conflictivas originadas por la aplicación de operaciones de evolución y permite detectar automáticamente la presencia de tales conflictos.

Además la M&D-logic provee un marco para definir transformaciones no-primitivas de modelos, tales como refinamientos y patrones de evolución, permitiendo analizar semánticamente tales transformaciones.

Formalizando calidad:

La calidad de un diseño es algo difícil de evaluar ya que es un concepto abstracto y complejo integrado por diferentes aspectos. Sin embargo la comunidad informática ha logrado reconocer un conjunto de propiedades que caracterizan a un buen diseño orientado a objetos.

Nuestra postura consiste en que ambas clases de propiedades, las tradicionales (por ej. acoplamiento y cohesión) y las nuevas propiedades orientadas a objetos (por ej. polimorfismo), deben ser consideradas conjuntamente para medir la calidad de un diseño orientado a objetos. Sin embargo creemos que es necesario prestar especial atención al concepto de *polimorfismo*, ya que debe ser considerado el punto clave para determinar la calidad de un diseño orientado a objetos.

En el capítulo 8 hemos presentado una definición rigurosa de polimorfismo en el marco de la M&D-theory y basándonos en dicha definición hemos propuesto una nueva métrica de polimorfismo. La definición formal provee un mecanismo objetivo y preciso para detectar y cuantificar la presencia de *polimorfismo* en un diseño orientado a objetos.

Formalizando reuso:

Durante el proceso de construcción de un nuevo sistema de software es importante contar con la posibilidad de re-utilizar componentes pre-existentes, evitando de esta manera realizar el mismo trabajo una y otra vez. Los *contratos* son una herramienta de especificación cuyo objetivo es favorecer la re-utilización correcta de componentes.

Hemos señalado que se necesita un mecanismo para identificar aquellos componentes que pueden ser re-usados en un nuevo sistema. Por ejemplo, podría ser necesario identificar los componentes que satisfacen un determinado contrato. En esta dirección, hemos mostrado que tanto componentes como contratos pueden ser expresados mediante una M&D-fórmula. De esta forma, el aparato deductivo de la lógica ofrece un mecanismo formal para determinar si un componente cumple con un contrato. Además, esta representación nos brinda la posibilidad de expresar formalmente los conceptos de refinamiento e inclusión de contratos.

Por otra parte hemos mostrado que la M&D-logic permite expresar patrones de diseño y consecuentemente permite identificar la presencia de determinados patrones dentro de un diseño a través de deducciones lógicas.

10.6 Trabajos relacionados

En esta tesis hemos definido una teoría en lógica dinámica que formaliza el proceso de modelado de sistemas orientados a objetos. Primeramente comentaremos los trabajos relacionados directamente con la teoría formal y luego nos referiremos específicamente a la integración entre el modelo formal y las técnicas de modelado no-formales.

10.6.1 Formalizando objetos mediante la lógica

Existen numerosas propuestas para establecer formalmente la semántica de los conceptos de la programación orientada a objetos, usando lógica. Básicamente se han seguido dos direcciones:

- extender las estructuras de la programación lógica con el concepto de objeto estructurado (ver por ejemplo [Abiteboul and Grumbach 88] y [Zaniolo 84]).

- dearrollar una nueva lógica que soporte los conceptos de la orientación a objetos. (ver por ejemplo [Beeri 89], [Chen and Warren 89], [Kifer and Lausen 89], [Maier 86]).

Debido a las limitaciones que presenta el concepto de objeto estructurado, en los últimos tiempos los mayores esfuerzos se han concentrado en la segunda dirección. Así, varias lógicas de objetos han sido propuestas, tales como O-logic[Maier 86], C-logic[Chen and Warren 89], F-logic[Kifer and Lausen 89] y DOL[Wieringa et al.94]. Estas lógicas contemplan identidad de objetos, clasificación y herencia como conceptos primitivos. Mediante estas lógicas también se han definido modelos formales para sistemas de objetos A continuación describimos algunos ejemplos:

- Un grupo de investigación de la universidad de Valencia [Carsí et al. 97] ha defino formalmente el concepto de 'metaclase' utilizando Transaction-Frame Logic. La TF-Logic es una lógica de orden superior, resultante de la unión de dos lógicas: la Frame Logic (F-Logic) para representar los aspectos estructurales del modelo y la Transaction Logic (T-Logic) para representar el comportamiento del sistema. El esquema conceptual definido en el mencionado trabajo es similar al propuesto en esta tesis en el sentido que permite analizar formalmente la evolución del sistema tanto en el nivel de las instancias, como en el nivel de las clases.

- En [Kesim and Sergot 96] se define una variante del cálculo de eventos [Kowalski and Sergot 86] para describir información temporal en bases de datos orientadas a objetos y razonar sobre su evolución. Un objeto es visto como una colección de fórmulas atómicas, además el cálculo soporta nociones de identidad de objetos, clasificación y herencia. Se utiliza programación en lógica de primer orden clásica aumentada con falla y negación. El cálculo permite representar evolución de los objetos en el tiempo: creación y borrado de objetos, modificación de su estado interno y también permite representar cambios estructurales en la definición de las clases.

- En [Kemper et al.92] se desarrolla un modelo formal para bases de datos orientadas a objetos. La formalización se basa sobre lógica dinámica: combinando álgebras many-sorted con estados y transiciones de estados. Las álgebras son utilizadas para modelar los aspectos estáticos del modelo (la parte estructural y la evaluación de expresiones) y las transiciones de estado se usan para representar el comportamiento dinámico de la base de objetos (producido por modificaciones de la base, es decir creación, borrado y modificación de los objetos). Se define el lenguaje de manipulación de la base de objetos con expresiones (o consultas) para observar el contenido de la base y con comandos (tales como asignación a variables, modificación de atributos de objetos, creación y destrucción de objetos) para modificar la base.

Sin embargo el artículo no representa el comportamiento complejo de los objetos, es decir transiciones de estado producidas por el cambio de estado de un objeto en respuesta a la recepción de un mensaje.

- En [Fiadeiro and Maibaum 95] y [Fiadeiro and Maibaum 96] los autores introducen principios de modularización (en particular los propuestos por las metodologias de diseño orientadas a objetos) dentro de la especificación de sistemas reactivos. Este trabajo integra la flexibilidad de las técnicas de diseño orientadas a objetos con lógica temporal y con técnicas algebraicas para modularización de especificaciones. La idea central consiste en adoptar teorías (en lugar de fórmulas) como bloques primitivos para especificar componentes, usando interpretaciones entre teorías (theory morphisms) como un medio para especificar interconexiones entre componentes.

- En [Bahsoun et al.93] se propone un modelo de programación orientada a objetos basado sobre agentes que colaboran concurrentemente. La semántica formal de un agente es una teoría en lógica temporal de acciones (TLA) y un sistema de agentes es formalizado por la conjunción de las especificaciones individuales de cada agente que lo constituye más algunos axiomas que representan la estructura de comunicación. Una clase describe una colección de agentes con el mismo comportamiento. Un agente es una instancia de alguna clase. Es posible definir una clase a partir de otra clase usando herencia (se definen reglas sintácticas para obtener la especificación implícita de la subclase, es decir Superclase + incremento = Subclase). Se define un lenguaje, similar a un lenguaje de programación de alto nivel, para especificar agentes. Esta formalización permite verificar propiedades de sistemas de agentes.

- También nos interesa citar algunos de nuestros trabajos previos en el área de formalización de conceptos de la programación orientada a objetos tales como [Tau et al. 95] donde se define la semántica denotacional de un lenguaje de especificación de bases de objetos, [Pons 95] donde se define un modelo formal para explicar conceptos centrales del paradigma utilizando álgebras universales y [Pons et al. 93] donde se formalizan determinadas formas de evolución de especificaciones orientadas a objetos.

10.6.2 Integrando lenguajes de modelado

En el capítulo cuatro hemos discutido y comparado las diferentes soluciones que han sido propuestas al problema de integrar lenguajes de modelado gráficos, cercanos a las necesidades del dominio de aplicación con lenguajes de modelado formales, tales como la lógica.

Hemos identificado básicamente cuatro propuestas diferentes para realizar la integración:

- **Suplemento**. Buenos ejemplos de esta propuesta son Syntropy [Cook and Daniels 94] y los trabajos de Lano [Goldsack and Kent 96] y Weber [Weber 96] los cuales proponen utilizar la notación formal Z[Spivey 92] para enriquecer notaciones semi-formales.

- **Extensión**. Los ejemplos más relevantes de esta propuesta son las extensiones del lenguaje Z, como por ejemplo Z++[Lano 91] y Object-Z[Duke et al. 91]. También corresponden a este grupo los lenguajes TROLL [Jungclaus et al. 96] OOZE [Alencar and Goguen 91] y MAUDE [Meseguer and Winkler 91] inspirados sobre lenguajes de especificaciones algebraicas.

- **Interfaz**. Algunos ejemplos de esta propuesta son las interfaces gráficas provistas por los lenguajes formales OASIS [Pastor and Ramos 95] y LTL [Reggio and Larosa 97].

- **Semántica** (ver la sección siguiente)

10.6.3 Formalizando UML

Dado que la propuesta de dar semántica a un lenguaje conocido y aceptado resulta la más adecuada y a partir de la estandarización del lenguaje gráfico de modelado Unified Modeling Language (UML) han surgido activas discusiones acerca de la precisión semántica de sus construcciones. Mientras que el OMG fue responsable por la estandarización de UML como notación, la semántica de UML aún es un tema de investigación. Existe un número importante de trabajos teóricos (ver por ejemplo [UML-conference 98]) que tratan diferentes partes de UML definiendo formalmente su semántica. Sin embargo todavía resta un largo camino por recorrer. En particular, es difícil comparar los resultados de los respectivos artículos y es aun más difícil combinar dichos resultados con el objetivo de obtener una semántica standard para UML. Esta dificultad surge especialmente porque los diferentes trabajos utilizan diferentes métodos (o lenguajes) formales, o cubren un subconjunto de la notación o asumen una subclase particular de sistemas a ser especificados. Sin embargo, surge una clara clasificación de las propuestas en dos grupos:

- Formalizaciones basadas en el modelo (ver por ejemplo [Moreira and Clark 94, France et al. 97(a), Goldsack and Kent 96, Waldoke et al. 98, Wieringa and Broersen 98, Lano and Biccaregui 98]).
- Formalizaciones basadas en el metamodelo (ver por ejemplo [France et al. 97 (b), Breu et al. 97, UML 97(b), Evans et al. 98]).

El 'foco' de la formalización constituye el elemento discriminatorio entre ambos grupos. Las formalizaciones del primer grupo centran su atención sobre el nivel del modelo, mientras que las formalizaciones del segundo grupo lo hacen sobre el nivel del metamodelo. Ambos grupos han sido descriptos y comparados detalladamente en el capítulo 4.

10.7 Líneas de trabajo futuro

A continuación consideramos las principales líneas de investigación que se derivan de nuestro trabajo. Básicamente, consideramos dos direcciones: por un lado, extensiones que lleven a una mayor consolidación de los fundamentos teóricos y formales; y por otro lado, extensiones de cada una de las propuestas de uso del modelo formal.

Extensión de la teoría: La teoría que hemos definido es incompleta ya que cubre sólo un subconjunto del lenguaje UML. Sin embargo, dado que la teoría posee una estructura jerárquica resulta fácilmente extensible. Actualmente, nos encontramos trabajando en la extensión de la teoría mediante la definición de otros diagramas complejos, tales como diagramas de colaboración y diagramas de casos de uso.

Análisis de las propiedades de la lógica: Hemos demostrado que la teoría es consistente y que el sistema deductivo es correcto. Sin embargo no hemos realizado aún estudios profundos acerca de la completitud de la teoría con respecto a la clase de modelos min-max.

Mediante mecanismos formales de deducción hemos mostrado que es posible obtener información no presentada explícitamente en una especificación. Pero aún nos resta estudiar las limitaciones del sistema deductivo, así como el análisis de algoritmos eficientes para realizar las pruebas.

Otro tema que nos proponemos estudiar es la composición de especificaciones (o teorías). Composición significa que varias teorías pueden ser compuestas para obtener una nueva teoría. Es importante lograr que la semántica del resultado pueda ser deducida a partir de la semántica de las partes (semántica composicional). También nos interesa la relación de refinamiento entre teorías y como estos conceptos se reflejan en UML.

Implementación de la función de interpretación semántica: La función *translation* asocia cada modelo UML bien formado con su correspondiente M&D-theory. Esta función está determinada por medio de un conjunto de reglas que permiten crear una M&D-theory a partir de los distintos submodelos relacionados que componen un modelo UML. Las reglas trabajan sobre una teoría básica que van enriqueciendo progresivamente, agregando axiomas específicos.

Sería importante contar con una implementación de esta función. En esta dirección hemos realizado algunas experiencias previas usando la herramienta Rational Rose para construir modelos y luego, a partir de la representación (en forma de archivo de texto) generada por la herramienta, hemos elaborado una teoría formal (ver Waldoke et al. 98). En esta experiencia utilizamos Object-Z como lenguaje formal y generamos una teoría orientada al modelo. Nos proponemos realizar una implementación similar utilizando la M&D-theory.

También nos interesa estudiar una función inversa a *translation* que permita reconstituir el modelo UML a partir de una teoría en lógica dinámica.

Ampliación de la métrica de calidad: Hemos definido una nueva métrica para medir la calidad de un diseño orientado a objetos, la cual no contempla todas las propiedades del sistema, sino que sólo se ocupa de caracterizar *polimorfismo*. Luego, la medida de polimorfismo podría ser combinada con las medidas de las restantes propiedades (tales como acoplamiento, cohesión, entropía, etc.) con el objetivo de determinar la calidad total del sistema. Proponemos esta tarea de combinación de métricas como una posible extensión de nuestro trabajo.

Experiencias con patrones de diseño: Por otra parte hemos mostrado que la M&D-logic permite expresar patrones de diseño y consecuentemente permite identificar la presencia de determinados patrones dentro de un diseño a través de deducciones lógicas. Nos interesa aplicar la lógica para expresar formalmente una amplia gama de patrones de diseño y así poder evaluar las ventajas de contar con tal especificación formal (por ejemplo, lograr un entendimiento mas preciso de los elementos de cada patrón, posibles usos en re-ingeniería, etc.).

Definición de patrones de evolución: El mecanismo de evolución propuesto considera sólo formas de evolución primitivas, sin embargo se provee un marco para definir transformaciones no-primitivas de modelos, tales como patrones de evolución, permitiendo analizar semánticamente tales transformaciones.

Proponemos las siguientes tareas futuras:

- Identificar modificaciones no-primitivas o patrones.
- Definir formalmente las condiciones de aplicabilidad de cada patrón.
- Identificar y especificar la propagación de cambios provocada por la aplicación de cada patrón.
- Identificar situaciones conflictivas originadas por la aplicación de patrones de evolución y permitir detectar automáticamente la presencia de tales conflictos.

La definición de los patrones debería realizarse sobre la base de la definición formal de las operaciones primitivas. Por ejemplo, las condiciones de aplicabilidad de un patrón deberían corresponderse con las condiciones de aplicabilidad de las acciones primitivas involucradas en el patrón.

Referencias bibliográficas

[Abadi and Cardelli 95] Martin Abadi and Luca Cardelli, A Theory of Objects, Monographs in Computer Science, Springer, 1996.

[Abiteboul and Grumbach 88] S.Abiteboul and S.Grumbach, COL: a logic-based language for complex objects, In Extending Database Technology EDBT´88 proceedings, Italy, 1988.

[Alencar and Goguen 91] A.Alencar and J.Goguen, OOZE: an object-oriented Z environment, In P.America editor, ECOOP'91 Proceedings, Lecture Notes in Computer Science vol.512. Springer-Verlag, July 1991.

[Bansiya 97] J. Bansiya, Assessing quality of object-oriented designs using a hierarchical approach, OOPSLA'97 Workshop#12 on Object-oriented design quality. Atlanta, USA, October 1997.

[Bansiya et al. 99 (a)] J. Bansiya, C.Davis, L. Etzkorn and W.Li, An entropy-based complexity measure for object oriented designs, Theory and Practice of Object Oriented Systems, 5(2), 1999.

[Bansiya et al. 99 (b)] J. Bansiya, L. Etzkorn, C.Davis and W.Li, A class cohesion metric for object oriented design, Journal of Object Oriented Programming, January 1999.

[Bahsoun et al. 93] Jean Bahsoun, S.Merz and C.Servieres, "A framework for programming and formalizing concurrent objects, ACM Software engineering notes Vo.18, No.5, 1993.

[Beeri 89]C Beeri, Formal models for object oriented databases, In procs.of First Int.Conf.on Deductive and Object Oriented Databases, Kyoto, Japan, 1989.

[Ben-Ari et al.83] M.Ben-Ari, A.Pnuelli and Z.Manna, "The temporal logic of branching time", Acta Informatica 20, 1983.

[Benlarbi 97] S.Benlarbi, Object-oriented design metrics for early quality prediction, OOPSLA'97 Workshop#12 on Object-oriented design quality. Atlanta, USA, October 1997.

[Bergstein 97] Paul Bergstein, Maintenance of object-oriented systems during structural evolution, Theory and Practice of Object Systems, V.3,N.3, John Wiley &Sons, Inc, 1997.

[Bertino et al.98] E.Bertino, E.Ferrari, G.Guerrini and I.Merlo, Extending the ODMG Object Model with time, proceedings of ECOOP'98, Lecture Notes in Computer Science 1445, July 1998.

[Boehm 88] B.Boehm, A espiral model of software development and enhancement, IEEE Computer, 21(5), May 1988.

[Boehm 89] B.Boehm, Tutorial on software risk management, IEEE Computer Society Press, 1989.

[Boehm and Papaccio 88] B.Boehm and P.papaccio, Understanding and controlling software costs, IEEE Transactions on Software Engineering, 14(10), October 1988.

[Bonner and Kifer 95] A.Bonner, M.Kifer, Transaction Logic Programming, Technical Report CSRI-323, November 1995.

[Booch 94] G.Booch, Object Oriented Analysis and Design with Applications, Second Edition, Addison-Wesley Publishing Company, Inc, 1994.

[Boyland and Castagna 96] J.Boyland and G.Castagna, Type-safe compilation of covariant specialization: a practical case, proceedings 10th European Conference on Object-Oriented Programming (ECOOP'96), Springer.

[Breu et al. 1997] R.Breu, U.Hinkel, C. Hofmann, C.Klein, B.Paech, B.Rumpe and V.Thurner. Towards a formalization of the unified modeling language. In ECOOP'97 proceedings, LNCS 1241, Springer, June 1997.

[Briand et al. 97] L.Briand, P.Devanbu and W.Melo. An investigation into coupling measures for C++. In International Conference of Software Engineering (ICSE'97), Boston, USA, May 1997.

[Budd 91] Timothy Budd, An introduction to object-oriented programming, Addison-Wesley Publishing Company, 1991.

[Carsí et al. 97] Implementación de la metaclase en un entorno de prototipación automática basado en Oasis, J.Carsí, I.Ramos, B.Bonet, F.Jaén, M.Penadés, Reporte técnico, Departamento de Sistemes Informatics i Computació, Universitat Politecnica de Valencia, 1998.

[Castagna 95] G.Castagna, Covariance and Contravariance: conflict without a cause. ACM Transactions on Programming Languages and Systems, 17(3), 1995.

[Clarke y Emerson 81] E.Clarke ,E.Emerson, "Design and Synthesis of synchronisation skeleton unsing branching time temporal logic ", in Logic Programs, Lecture Notes in Computer Science 131, 1981

[Coad and Yourdon 91] P.Coad and E.Yourdon, "Object Oriented Analysis", Yourdon Press, Englewood Cliffs,NJ, 1991.

[Cook and Daniels 94] S.Cook and J.Daniels, Let's get formal, Journal of Object-Oriented Programming(JOOP), July-August 1994.

[Chellas 80] Brian Chellas.,"Modal Logic. An Introduction", Cambridge University Press, 1980.

[Chen and Lu 93] J.Chen and J.Lu, A new metric for object oriented design, Information and Software Technology, 35. 1993.

[Chen and Warren 89] W.Chen and D.Warren, C-Logic of complex objects, In procs.ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1989.

[Chidamber and Kemerer 94] S.Chidamber and C.Kemerer, A metric suite for object oriented design, IEEE Transaction on Software Engineering, 20. 1994.

[DeMarco79] Tom DeMarco, Structured Analysis and System Specification. Englewood Cliffs, NJ:Prentice Hall, 1979.

[Duke et al. 91] R.Duke, P.King, G.rose and G.Smith. The Object-Z specification language. In T.Korson, V.Vaishnavi and B.Meyer, editors, Technology of Object-Oriented Languages and Systems:TOOLS 5. Prentice Hall, 1991.

[Emerson and Halpern 85] E.Emerson and J.Halpern, "Decision procedures and expressiveness in temporal logic of branching time, Computer and Systems Sciences,30, 1985.

[Emerson 90] E.emerson, Temporal and modal logic, in Handbook of theoretical computer science, volume B: formal models and semantics, Elsevier 1990.

[Enderton 72] H.Enderton, A mathematical introduction to logic, Academic Press Inc., 1972.

[Evans et al 98] Andy Evans, Robert France, Kevin Lano and Bernhard Rumpe, Developing the UML as a formal modeling notation. In Muller and Bezivin editors, UML'98 Beyond the notation, International workshop, France, 1998.

[Fiadeiro and Maibaum 95] J.Fiadeiro and T.Maibaum, "Interconnecting formalisms: supporting modularity, reuse and incrementality", In Proc.3rd Symposium on Foundations of Software Engineering, ACM Press, 1995.

[Fiaderio and Maibaum 96] J.Fiadeiro and T.Maibaum, in "Formal Methods and Object Technology", Chapter 9, S.Goldsack and S.Kent (editors), Springer-Verlag, 1996.

[France et al. 97(a)] R.France, J.Bruel and M.Larrondo-Petrie. An integrated object-oriented and formal modeling environment, Journal of Object Oriented Programming (JOOP), 1997.

[France et al. 97(b)] R.France, E.Evans and K.Lano, The UML as a formal modeling notation, In Kilov, Rumpe and Simmons editors, OOPSLA'97 Workshop on Object-oriented Semantics, TUM-I9737, Institut Fur Informatik, Technische Universitat Munchen.

[Gamma et al. 94] Gamma, Helm, Johnson and Vlissides, Design Patterns, Addison –Wesley Publishing Company, 1994.

[Geisler et al.98]Dimensions and Dichotomy in Metamodeling, Robert Geisler, Marcus Klar and Claudia Pons, proc. of Third BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, September 1998, Series in Computing, Springer-Verlag.

[Ghezzi et al. 91] C.Ghezzi, M.Jazayeri, D.Mandrioli, Fundamentals of software engineering, Prentice-Hall International Editions, Englewood Cliffs, NJ 1991.

[Gilb 88] T. Gilb, Principles of Software Engineering Management, Addison Wesley, Reading, 1988.

[Goguen and Meseguer 92] J.Goguen and J.Meseguer, Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science, 1992.

[Goldblatt 92] Robert Goldblatt, "Logics of Time and Computation", Second edition, CSLI Lecture Notes No.7, 1992.

[Goldsack and Kent 96] "Formal Methods and Object Technology", Chapter 3: LOTOS in the Object-oriented analysis process. Editors S.J. Goldsack, S.J.H. Kent. Serie FACIT, Springer-Verlag Berlin Heidelberg New York, 1996.

[Gomaa and Scott 81] H.Gomaa and D.Scott, Prototyping as a tool in the specification of user requirements, proceedings 5th International Conference on Software Engineering, San Diego, March 1981.

[Hamilton 81] A.G.Hamilton, Logic for mathematicians. Cambridge University Press, ISBN:0-521-29291-3. 1981

[Harel et al.99] David Harel, Dexter Kozen and Jerzy Tiuryn. Dynamic Logic. Libro en preparación próximo a publicarse.

[Harel et al.77] D.Harel, R.Meyer and V.Pratt, Computability and completeness in logics of programs, In Proc.9th Symposium in Theory of Computing, ACM, 1977.

[Harel and Kozen 84] D.Harel and D.Kozen, A programming language for the inductive sets and applications. Information and Control 63(1-2), 1984.

[Helm and Holland 90] R Helm and I Holland, Contracts: specifying behavioral compositions in object-oriented systems, Proceedings of OOPSLA'90. Oct 1990.

[Jacobson et al 92] I.Jacobson, M.Christerson, P.Jonsson and G.Overgaard, Object Oriented Software Engineering, Addison Wesley, 4th edition, 1992.

[Jungclaus et al. 96] Ralf Jungclaus,G.Saake,T.Hartmann,C.Sernadas,"TROLL- a language for o-o specifications of information systems", ACM Transactions on IS, vol.14 no.2. April 96.

[Jones 90] C B Jones, Systematic software construction using VDM. Prentice Hall, 1990.

[Kemper et al.92] A.Kemper, G.Moerkotte and K.Peithner, "Object orientation axiomatised by dynamic logic", 1992.

[Kesim and Sergot 96] F.Kesim and M.Sergot. A logic programming framework for modeling temporal objects, IEEE Transactions on knowledge and data engineering, vol.8,no.5, October 1996.

[Kiczales and Lamping 92] G. Kiczales, J. Lamping Issues in the Design and specification of Class Libraries. In Proceedings OOPSLA´92, ACM SIGPLAN Notices,pag. 435-451, October 1992.

[Kifer and Lausen 90] M.Kifer and G.Lausen, “F-Logic: a higher order language for reasoning about objects, inheritance and scheme. in proceedings of the ACM SIGMOD symposium on principles of database systems,SIGMOD RECORD, Vol.18, No.6, June 1990.

[Kifer 95] M.Kifer, Deductive and object data languages: a quest for integration, Dood 1996.

[Kim et al 94] E.Kim, O.Chang,S.Kusumoto, T.Kikuno, Analysis of metrics for object oriented program complexity, Procs. 18th Annual International Computer Software and applications Conference, COMPSAC´94.

[Klar and Geisler 97] Marcus Klar and Robert Geisler, "A metamodel for object-oriented systems", Fraunhofer Institut fur Software und Systemtechnik ISST, Berlin. Technical Report.

[Kowalski and Sergot 86]R.Kowalski and M.Sergot, A logic-based calculus of events, In New Generation Computing, vol 4, 1986.

[Lalonde 94] Wilf. Lalonde, Discovering Smalltalk. Addison Wesley. 1994.

[Lamport 94] L.Lamport, “The Temporal Logic of Actions”, ACM Transactions on programming languages and systems, 16(3), 1994.

[Lano 91] K.Lano. Z++, An object-oriented extension to Z. In John Nicholls, editor, Z user workshop, Oxford 1990, Workshops in Computing, Springer Verlag, 1991.

[Lano and Biccaregui 98] Formalizing the UML in Structured Temporal Theories, Kevin Lano, Jean Biccaregui, Second ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische Universitat Munchen.

[Li and Henry 93] W.Li and S.Henry, Object oriented metrics that predict maintainability, The Journal of Systems and Software, 23.

[Lucas 97] Carine Lucas, “Documenting Reuse and evolution with reuse contracts”, PhD Dissertation, Programming Technology Lab, Vrije Universiteit Brussel, September 1997.

[Maier 86] D.Maier, A logic for objects, In procs. Workshop Foundations of Deductive Databases and Logic Programming, Washington, 1986.

[Manna and Pnuelli 92] Z.Manna and A.Pnueli, ““The temporal Logic of Reactive and Concurrent Systems”, Springer Verlag, 1992.

[Mens et al.98] T.Mens, C.Lucas and P.Steyaert, Giving presice semantics to evolution in the UML, In PSMT Workshop on Precise Semantics for Software Modeling Techniques, Ed: M.Broy, D.Coleman, T.Maibaum, B.Rumpe, Technische Universitat Munchen, Report TUM-I9803, April 1998.

[Meseguer and Winkler 92] J.Meseguer, T.Winkler. “Parallel Programming in Maude”. Proceedings of Research Directions in High Level Parallel Programming Languages. Springer Verlag 1992.

[Meyer 92] B.Meyer, Advances in object oriented software engineering. Chapter 1 “Design by contract”. Prentice Hall, 1992.]

[Mezini 97] M.Mezini, Maintaining the consistency of class libraries during their evolution. In proceedings OOPSLA’97, ACM Sigplan notices, pag.1-21, October 1997.

[Moore 96] Ivan Moore, Automatic inheritance hierarchy restructuring and method refactoring, in proceedings of OOPSLA’96, ACM Sigplan, vol.31,no.10, October 1996.

[Moreira and Clark 94] A.Moreira,and R. Clark. “Combining Object_Oriented Analysis and Formal Description Techniques”, In 8th European Conference on Object Oriented Programming, Proceedings. LNCS 821. 1994.

[OCL 97] The Object Constraint Language (OCL) – version 1.1, September 1997. Part of [UML 97].

[Odell95] James Odell. Meta-modeling. In OOPSLA'95 Workshop on Metamodeling in OO, October 1995

[Overgaard 98] Gunnar Overgaard, A formal approach to relationships in the UML, Workshop on Precise Semantics of Modeling Notations, International Conference on Software Engineering ICSE'98, Japan, April 1998.

[Parnas and Clements 86] D.Parnas and P.Clements, A Rational Design Process: how and why to fake it. IEEE Transactions of Software Engineering, 12(2), February 1986.

[Pastor and Ramos 95] O. Pastor, I.Ramos, "Oasis 2.2 : A Class-Definition Language to Model Information System Using an Object-Oriented Approach". SPUPV-95.788, Universitat P. de Valencia.

[Pnuelli 81] A.Pnuelli, "The temporal Semantics of Concurrent Programs", Theoretical Computer Science 13, 1981.

[Pons et al.99] C. Pons, G.Baum and M.Felder, Foundations of Object-oriented modeling notations in a dynamic logic framework, C.Pons, G.Baum, M.Felder, In Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher, 1999.

[Pons 99] El proceso de desarrollo de software basado en modelos. C.Pons, proceedings of CACIC'99, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina. Nov 1999.

[Pons and Kutsche 99] Model Evolution and System Evolution, Claudia Pons, Ralf-D Kutsche, proceedings of CACIC'99, Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina. Nov 1999.

[Pons and Giandini 99] C.Pons, R.Giandini, Precise semantics of model evolution, proceedings of IDEAS´99, San Jose, Costa Rica, 24-26 March 1999.

[Pons et al. 98] C. Pons, G.Baum and M.Felder, Integrating object-oriented model with object-oriented meta-model into a single formalism, Second ECOOP Workshop on Precise Behavioral Semantics, European Conference on Object-oriented Programming, Brussels, Belgium, Ed: H.Kilov, B.Rumpe, Technische Universitat Munchen, Report TUM-I9813, ECOOP'98 Workshop Readers, Lectures Notes in Computer Science. July 1998.

[Pons et al.98] C. Pons, G.Baum and M.Felder, A dynamic Logic Model for the Formal Foundation of Object-oriented Analysis and Design, XVIII International Conference of the Chilean Computer Science Society (SCCC'98), IEEE Computer Science Press, Chile, November 1998.

[Pons et al.97] C.Pons, R.Giandini, G.Baum, Una herramienta para verificación formal de especificaciones gráficas de objetos, Actas de II Jornadas de Ingeniería de Software JIS'97, editadas por O.Diaz y P.Lopistéguy, San Sebastián, España, Septiembre 1997.

[Pons et al.97] Claudia Pons, A User Friendly Algebraic Proposal for a Formal Verification of Object-Oriented Diagrams, Paper presented at the OOPSLA'97 Doctoral Symposium, Atlanta, USA, October 1997.

[Pons 95] Claudia Pons, Formal Semantics for Object-Oriented Systems. Proceedings of 7th International Conference on Software Engineering and Knowledge Engineering (SEKE'95), May 1995, Maryland, USA

[Pons et al.93] An Analysis of Basis Schema Evolution Operations. C.Pons, C.Smith, C.Tau, Ana Monteiro, Gabriel Baum. Proceedings of 1993 Workshop on Information Technologies and Systems (WITS'93). University of Maryland. USA, November 1993.

[Poulin 97] J.Poulin, Measuring Software Reuse- Principles and Practices and Economical Models, Addison Wesley, 1997.

[Popkorn 94] S.Popkorn, “First steps in Modal Logic”, Cambridge University Press, 1994.

[Pratt 76] V.Pratt. “Semantical Considerations on Floyd-Hoare Logic”, Proc. 17th IEEE Symposium on Foundations of Computer Science. 1976.

[Price and Demurjian 97] M.Price and S.Demurjian, Analysing and measuring reusability in object-oriented design, in proceedings OOPSLA’97, Atlanta, USA, 1997.

[Reggio and Larosa 97] g.Reggio and M.Larosa, A graphic notation for formal specification of dynamic systems, proceedings of FME’97 4th International symposium of formal methods Europe, Lecture Notes in Computer Science 1313, Springer.

[Rumbaugh et al. 91] J.Rumbaugh, M.Blaha, W.Premerlani, “Object Oriented Modeling and Design”, Prentice Hall, 1991.

[Rumpe 96] Bernhard Rumpe, Ph.D thesis, Technische Universitat Munchen Report, Germany, 1996.

[Shlaer and Mellor 88] S.Shlaer and J.Mellor, Object Oriented Systems Analysis: Modeling the World in Data, Yourdon Press Computing Series, Yourdon Press, Englewood Cliffs, NJ, 1988.

[Smith 94] G.Smith, A Logic for Object-Z, Technical Report 94-48, Department of Computer Science, University of Queensland, Australia, December 1994.

[Spivey 92] M.Spivey. The Z notation: a reference manual. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.

[Spruit et al. 93] P.Spruit, R.Wieringa and J.-J.Ch.Meyer. Dynamic Database Logic: the first order case. In U.W.Lipeck and B.Thalheim, editors, Modelling Database Dynamics, pages 103-120, Springer, 1993.

[Steyaert et al. 96] P.Steyaert, C.Lucas, K.Mens and T.D’Hondt. Reuse Contracts: Managing the evolution of reusable assets. In proceedings of OOPSLA’96, New York, Oct 1996.

[Tau et al.95] C.Tau, C.Smith, C.Pons, A.Monteiro and G.Baum, Formally Speaking About Schemas, Bases, Classes and Objects", Proceedings of 4th International Conference on Database Systems for Advanced Applications (DASFAA'95), april 1995, Singapore. Edited by S-C Moon & H Ikeda. World Scientific Publisher.

[Tansel et al. 93] Tansel,A., Clifford,J, Gadia,S. , Jajodia,S., Segev, A., and Snodgrass, R., Temporal Databases: Theory, Design and Implementation. Database Systems and Applications Series, Benjamin/ Cummings Publisher. 1993.

[Tegarden et al. 92] D.Tegarden, S.Sheetz and D.Monarchi, Effectiveness of traditional software metrics for object-oriented systems. 25th Annual Conference of System Science, Maui,HI. 1992.

[UML 97] The Unified Modeling Language (UML) Specification – Version 1.1, September 1997. Joint Submission to the Object Management Group (OMG), ver http://www.omg.org.

[UML 97(a)] UML Notation Guide, Version 1.1, September 1997. Part of [UML 97]

[UML 97(b)] UML Semantics, Version 1.1, September 1997. Part of [UML 97].

[UML 97(c)] UML Extension for Objectory Process for S.E. 1.1, September 1997. Part of [UML 97].

[UML 98 (a)] The UML User Guide, Booch, Rumbaugh and Jacobson. Addison Wesley Longman, Inc, 1998.

[UML 98(b)] The UML Reference Manual, Rumbaugh, Jacobson and Booch. Addison Wesley Longman, Inc, 1998.

[UML-conference 98] Proceedings of the UML´98 conference, Mulhouse, France, June 1998.

Lecture Notes in Computer Science, Springer-Verlag.

[UML-conference 99] <<UML>>´99 - The Unified Modeling Language. Beyond the Standard. Proceedings of the UML´99 conference, Colorado, USA, October 1999. Lecture Notes in Computer Science 1723, Springer.

[UML 99] The Unified Modeling Language (UML) Specification – Version 1.3, July 1999. Especificación de UML revisada por el OMG, http://www.rational.com/uml/index.jtmpl

[UML RTF 99] home page for the OMG Unified Modeling Language Revision Task Force (UML RTF). http://uml.shl.com/

[Waldoke et al. 98] S.Waldoke, C. Pons, C.Paz Mezzano and M. Felder, A Formal Approach to Practical Object Oriented Analysis and Design, Proceedings of Argentinean Symposium on Object Orientation, ed: SADIO (Sociedad Argentina de Informática e Inv. Operativa), Buenos Aires, 1998.

[Weber 96] M. Weber. "Combining Statecharts and Z for the Design of Safety-Critical Control Systems", Proceedings of Third International Symposium of FME'96. Oxford, March 1996.

[Wieringa et al.94] R.Wieringa, W.de Jonge and P.Spruit, "Roles and dynamic subclasses: a modal logic approach", In ECOOP'94 Proceedings, Springer-Verlag, 1994.

[Wieringa and Broersen 98] R.Wieringa and J.Broersen, Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams, In PSMT Workshop on Precise Semantics for Software Modeling Techniques, Ed: M.Broy, D.Coleman, T.Maibaum, B.Rumpe, Technische Universitat Munchen, Report TUM-I9803, April 1998.

[Wilde and Huitt 92] N.Wilde and R.Huitt, Maintenance support of object-oriented programs, IEEE Transactions on Software Engineering, 18. 1992.

[Wittgenstein 33] Ludwig Wittgenstein. Tractatus Logico-Philosophicus, second corrected reprint, New York,: Harcourt, Brace and Company; London: Kegan Paul, Trench, Trubner & Co.Ltd., 1933.

[Wong 84] C.Wong, A successful software development, IEEE Transactions on Software Engineering, 10(6), November 1984.

[Woolf 97] B.Woolf, Polymorphic hierarchy. The Smalltalk Report. January 1997.

[Zaniolo 84] C.Zaniolo, The representation and deductive retrieval of complex objects, In procs. of Very Large Databases VLDB´85, Sweden, 1985.

EVALUACIÓN

Esta tesis fue evaluada por el siguiente tribunal:

Prof. Dr. Dino Mandrioli
Dipartimento di Elettronica e Informazione
Politecnico de Milano
Italy

Prof. Dr. Guillermo Simari
Universidad Nacional del Sur
Bahía Blanca
Argentina

Prof. Dr. Juan Victor Echag ü e
Buenos Aires
Argentina

Prof. Dr. Bernhard Möller
Institut f ü r Informatik
Universität Augsburg
Germany

A continuación se presentan los reportes emitidos por los miembros del tribunal.