

# Una herramienta para verificación formal de especificaciones gráficas orientadas a objetos

Trabajo de grado

Licenciatura en Informática  
Facultad de Informática  
UNLP

**Alumno**

**Javier Andrés Cengia**

**Director**

**Dra. Claudia Pons**



# Indice

<b>Introducción</b>	<b>5</b>
<b>Objetivo</b>	<b>6</b>
<b>1. Evolución del Proceso de Desarrollo del Software</b>	<b>7</b>
1.1. Introducción	7
1.2. Ciclo de vida del software	7
1.2.1. Proceso code-and-fix	8
1.2.2. Proceso Cascada	8
1.2.3. Prototipación	9
1.2.4. Proceso en espiral	9
1.3. Incorporación de los modelos en el proceso de desarrollo	10
1.3.1. Utilidad de los modelos	11
1.3.2. Los modelos en el ciclo de vida del software	11
1.3.3. Cualidades de los modelos	12
1.4. Metodologías de desarrollo de software orientado a objetos	12
1.4.1. Unificación de metodologías y origen de UML	13
1.5. Herramientas CASE	15
1.6. Modelos formales y modelos no-formales. La necesidad de integración	15
1.7. Verificación formal en herramientas de modelado	16
<b>2. La Notación UML</b>	<b>19</b>
2.1. Introducción	19
2.2. Motivación para definir UML	19
2.3. Objetivos	20
2.4. Historia	20
2.5. Organización de UML	21
2.5.1. Mecanismos de extensión	22
2.5.2. Diagramas de casos de uso	23
2.5.3. Diagramas de estructura estática	24
2.5.4. Diagramas de interacción	27
2.5.4.1. Diagramas de secuencia	27
2.5.4.2. Diagramas de Colaboración	28
2.5.5. Diagramas de estados	28
2.5.6. Diagramas de actividades	29
2.5.7. Diagramas de componentes	30
2.5.8. Diagramas de despliegue	30

<b>3. La M&amp;D-theory</b>	<b>31</b>
3.1. Introducción	31
3.1.1. Dicotomía de entidades	31
3.1.2. Relaciones de instanciación	32
3.1.3. Ventajas de la integración	32
3.1.4. Organización del capítulo	33
3.2. Nivel de los Modelos	33
3.2.1. Elementos	33
3.2.2. Evolución	33
3.2.3. Estructura de la Teoría	34
3.2.4. Paquete Foundation	35
3.2.4.1. Paquete Foundation: UML_Data Types	35
3.2.4.2. Paquete Foundation: Core	37
3.2.5. Paquete Behavioral Elements	47
3.2.5.1. Paquete Behavioral Elements: State Machines	47
3.2.6. Paquete Model Management	52
3.3. Nivel de los Datos	54
3.3.1. Elementos	54
3.3.2. Evolución	55
3.4. Integración de ambos niveles: La M&D-theory	61
3.5. La Interpretación semántica de UML	63
3.5.1. El dominio semántico	64
3.6. Función de interpretación semántica	65
3.7. Conclusiones	67
<b>4. Rational Rose</b>	<b>69</b>
4.1. Introducción	69
4.2. Vistas de un modelo en Rational Rose	69
4.3. Ambiente de la aplicación	70
4.3.1. Interfaz gráfica	70
4.4. Especificación de un modelo en Rational Rose	72
4.4.1. Diagrama de clases	73
4.4.1.1. Toolbox del diagrama de clases	73
4.4.1.2. Creación y especificación de elementos de modelado	74
4.4.2. Diagrama de estados	81
4.4.2.1. Toolbox del diagrama de estados	82
4.4.2.2. Creación y especificación de elementos de modelado	82
4.5. Representación textual de un modelo	86
4.5.1. Gramática de un archivo .mdl	87
4.5.1.1. Estructura principal	87

4.5.1.2. Elementos destacados	87
<b>5. Herramienta. Manual de Uso</b>	<b>91</b>
5.1. Arranque de la aplicación	91
5.2. Ventana principal de la aplicación	91
5.2.1. Barra de menús	94
5.2.1.1. Menú File	94
5.2.1.2. Menú Model Level	94
5.2.1.3. Menu Data Level	94
5.2.1.4. Menú Predefined Axioms	95
5.2.1.5. Menú User Axioms	95
5.3. Traducción de un modelo UML	96
5.4. Navegando los elementos del modelo formal	96
5.5. Consultando los elementos del modelo formal	97
5.5.1. Evaluación de una función	98
5.5.2. Evaluación de un predicado	99
5.6. Evolución del modelo formal	100
5.6.1. Ejecución de una acción de modificación	101
5.6.2. Ejecución de una acción de creación	102
5.6.3. Ejecución de una acción de cancelación	104
5.7. Evaluación de propiedades sobre el modelo	105
5.7.1. Evaluación de un axioma estático	106
5.7.2. Evaluación de un axioma estático	107
5.8. Edición de axiomas	110
5.8.1. Editor de axiomas estáticos	110
5.8.2. Editor de axiomas dinámicos	113
5.8.2.1. Editor de precondiciones	113
5.8.2.2. Editor de postcondiciones	114
5.8.3. Utilizando los axiomas editados	115
5.8.3.1. Consulta y evaluación de axiomas	115
5.8.3.2. Manejo de bibliotecas de axiomas	116
5.9. Persistencia de los estados de un modelo formal	116
5.10. Evolución en el nivel de los datos	117
5.10.1. Creación de un objeto -Object-	117
5.10.2. Creación de un valor -DataValue-	118
5.10.3. Creación de una conexión -Link-	119
5.10.4. Creación de un mensaje -Message-	120
5.11. Evolución en ambos niveles	121

<b>6. Herramienta: Diseño</b>	<b>124</b>
6.1. Organización	124
6.2. Paquete Principal	124
6.2.1. Paquete Herramienta	124
6.2.2. Paquete Teoría	126
6.2.2.1. Paquete Signatura	127
6.2.2.2. Paquete Axiomas	128
6.2.2.3. Paquete Lógica Dinámica	129
6.2.2.4. Paquete Lógica de primer orden	131
6.2.2.5. Paquete Formalización de UML	134
6.2.3. Paquete Traducción	139
6.2.4. Paquete Interfaz	140
<b>Conclusiones</b>	<b>142</b>
<b>Referencias Bibliográficas</b>	<b>144</b>

# Introducción

La construcción de un sistema de software debe ser precedida por la construcción de un modelo, tal como ocurre en otros sistemas ingenieriles. El modelo de un sistema es una representación conceptual obtenida a partir de la identificación, clasificación y abstracción de los elementos que constituyen el problema y su posterior organización en una estructura formal. En consecuencia, actúa como una especificación de los requerimientos que el sistema debe satisfacer, proveyendo un medio de comunicación y negociación entre usuarios, analistas y desarrolladores, así como también un documento de referencia durante la corrección de errores y 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. Actualmente todos los métodos de desarrollo de software han adoptado esta filosofía. Lo que varía entre métodos es los tipos de modelos que deben construirse, la forma de crearlos y organizarlos, y el lenguaje en el cual expresarlos.

El modelo del sistema se expresa utilizando un lenguaje de modelado. Un modelo se califica como informal cuando está expresado mediante lenguaje natural, figuras, tablas u otras notaciones. Por otra parte, un modelo es formal cuando la notación empleada es un formalismo, es decir posee una sintaxis y semántica precisamente definidos. También existen estilos de modelado intermedios llamados semiformales, ya que en la práctica los ingenieros de software frecuentemente utilizan notaciones cuya sintaxis y semántica están parcialmente formalizadas.

Los lenguajes gráficos de modelado propuestos por metodologías tales como OOA, OOSA, OMT, Booch's Design Method o RUP; son exitosos debido, principalmente, al uso de construcciones gráficas que transmiten un significado intuitivo. 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 diversos problemas: malas interpretaciones de los modelos, inconsistencia entre los diferentes modelos del sistema, discusiones acerca del significado del lenguaje o conflictos de evolución.

En otro sentido, los lenguajes formales de modelado, tales como Z, VDM, F-Logic o DS-Logic poseen una sintaxis y semántica bien definidas. Sin embargo su uso en la industria es poco frecuente debido 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, centran su esfuerzo sobre el manejo del formalismo en lugar de hacerlo sobre el modelo en sí mismo. 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 sólo debe interactuar con el lenguaje gráfico, pero además cuenta con la base formal provista por el esquema matemático subyacente. Esta propuesta ofrece claras ventajas comparada con la utilización por separado de un lenguaje informal o 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 con estas características será fácilmente aceptado tanto por los ingenieros de software, como por los usuarios.

La propuesta más exitosa para lograr la integración 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. La principal ventaja de esta propuesta reside en 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.

UML es un lenguaje estándar para modelar sistemas orientados a objetos. A partir de su estandarización han surgido activas discusiones acerca de la precisión semántica de sus construcciones. Mientras que el OMG (Object Management Group) fue responsable de la estandarización de UML como notación, su semántica aún es un tema de investigación

Un grupo de investigadores del LIFIA se encuentra trabajando en la producción de un método riguroso de análisis y diseño orientado a objetos, que combine técnicas gráficas con técnicas formales, asegurando que el método resultante sea accesible para los ingenieros de software típicos. Actualmente se ha definido un modelo conceptual basado en lógica dinámica: la M&D-Theory. Este modelo conceptual representa formalmente la información expresada mediante modelos UML. Su definición se basa en un número importante de trabajos teóricos que tratan diferentes partes de UML definiendo formalmente su semántica.

## Objetivo

Una de las claves para el éxito de la formalización 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 deberían usar el formalismo del lenguaje 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.

El objetivo de esta tesis es implementar una herramienta que permita el manejo del modelo conceptual propuesto. Dicha herramienta implementará un método de transformación automático, consistente en un conjunto de reglas para crear un modelo formal a partir de los modelos expresados en UML. La herramienta permitirá además el manejo del modelo formal, la aplicación de mecanismos de chequeo y la evolución en los distintos niveles de modelado.

UML es parcialmente soportado por una herramienta CASE desarrollada por Rational Software Corporation, llamada Rational Rose. Esta aplicación permite crear especificaciones gráficas en UML, y generar un archivo con la representación textual de la especificación. La representación textual permite la manipulación de la especificación para diversos usos. Se espera que la nueva herramienta pueda integrarse dentro de la herramienta CASE de Rational Rose.

Las etapas a cumplir para lograr el objetivo se enumeran a continuación:

- ♦ Implementación del metamodelo de UML en Lógica Dinámica (M&D-Theory) utilizando un lenguaje orientado a objetos (Smalltalk).
- ♦ A partir de un modelo UML, permitir instanciar términos de la M&D-Theory. Para esto se utilizará la representación textual generada por la herramienta Rational Rose.
- ♦ Visualizar los términos formales generados en forma jerárquica. Permitir recorrer, consultar y modificar el árbol.
- ♦ Crear una biblioteca de ‘propiedades estáticas’ de los modelos. Una propiedad estática es una fórmula que expresa una característica que debe estar presente en todo modelo.
- ♦ Permitir evaluar las propiedades estáticas sobre el modelo formal.
- ♦ Incorporar las características dinámicas al modelo formal:
  - Evolución del modelo (modificación de la especificación).
  - Evolución del sistema modelado (envíos de mensajes entre los objetos).
- ♦ Crear una biblioteca de ‘propiedades dinámicas’ de los modelos.
- ♦ Permitir evaluar las propiedades dinámicas sobre el modelo formal, simulando su evolución.
- ♦ Posibilitar a los usuarios la creación de sus propias bibliotecas de propiedades estáticas y dinámicas, y la evaluación de las mismas sobre el modelo formal.



# 1. Evolución del Proceso de Desarrollo del Software

## 1.1. Introducción

La evolución del proceso de desarrollo de software se produce ante la necesidad de construir sistemas de mayor tamaño, más complejos, de mejor calidad y menor costo. En consecuencia, surge la Ingeniería de Software como la aplicación de modelos y formas de la ingeniería tradicional a la práctica de construir sistemas de software, situación que ha condicionado su accionar al tener como base las precisiones y seguridades que en otros ámbitos tiene la ingeniería.

En los comienzos de la Informática, la programación se veía como un “arte”, existían pocos métodos formales y pocas personas los usaban. El programador aprendía mediante el método del ensayo y error. El proceso de desarrollo de un programa sólo involucraba al usuario y a la computadora. Por ejemplo, un matemático escribía un programa para resolver ecuaciones complejas. Con el surgimiento de los lenguajes de alto nivel y la caída de los costos del hardware, la programación de computadoras se popularizó y los usuarios se incrementaron. Ya tenían la posibilidad especificar el problema de manera menos estricta y contratar un programador, aunque existía la posibilidad de que éste no capturara precisamente sus requerimientos y/o ellos no quedaran satisfechos con la solución.

Este problema se acrecentó proporcionalmente al aumento del tamaño y la complejidad del software. No alcanzaba con reunir un grupo de desarrolladores y unificar lo programado por cada uno. A esta altura ya era necesario una permanente comunicación, interacción y coordinación entre los usuarios y programadores del sistema en desarrollo.

Históricamente han surgido varios enfoques que buscan abordar de manera sistemática, la planificación, análisis, diseño e implementación de los proyectos de desarrollo de software, sean estos de gran escala o pequeñas aplicaciones a medida. Cada uno de estos enfoques tiene su raíz en las preconcepciones dominantes en su época y, sobre todo, en la búsqueda incesante de mejoras a los enfoques precedentes.

Como en el resto de las actividades industriales, también en la construcción de software es importante realizar una buena planificación del trabajo, y una buena asignación de recursos a los distintos miembros del equipo de desarrollo.

## 1.2. Ciclo de vida del software

Los sistemas de software se desarrollan en una serie de actividades conocida como ciclo de vida del software. Un proceso de desarrollo de software intenta determinar el orden de las actividades involucradas y los criterios de transición asociadas entre estas actividades. El ciclo de vida también ayuda a los analistas y diseñadores a resolver problemas que surgen durante el desarrollo del sistema. A continuación, se presentan de manera breve algunos de los modelos de ciclo de vida más conocidos y utilizados, indicando sus características principales.

## 1.2.1. Proceso code-and-fix

El proceso básico usado en los primeros días del desarrollo de software, consiste en dos etapas: (1) Escribir algún código. (2) Fijar los problemas en el código.

De esta manera, el orden de los pasos era elaborar algún código primero y pensar sobre los requerimientos, diseño, prueba y mantenimiento a continuación. Este proceso tenía las dificultades de presentar una baja estructuración del código luego de una determinada cantidad de fijaciones. Pese a que se podía desarrollar un software de calidad, existía la posibilidad de que éste tuviera una correspondencia muy pobre con las reales necesidades del usuario y al no existir la conciencia de la necesidad real de pruebas y modificaciones el costo de las sucesivas fijaciones era muy alto. En consecuencia, el proceso de producción de software se convirtió en no predecible y poco controlable. Los desarrolladores no lograban cumplir con los tiempos de entrega, ni con los presupuestos establecidos.

## 1.2.2. Proceso Cascada

El proceso de cascada fue propuesto por Boehm en 1976 y también se lo conoce como “Ciclo de vida Clásico”. Se caracteriza por visualizar el proceso de desarrollo como una fábrica de producción en cadena, en la que cada fase empieza donde terminó la anterior (figura 1). Las etapas del ciclo son las siguientes:

- ♦ *Análisis*: Se realiza la recopilación de requisitos, tanto del sistema como del software. Se documenta todo lo que se ha estudiado y se establece *qué* se va a hacer. Todo esto se debe comentar con el usuario antes de continuar.
- ♦ *Diseño*: Se describe *cómo* el sistema va a satisfacer los requerimientos. Esta etapa a menudo tiene diferentes niveles de detalle. También se debe documentar todo el diseño realizado. En esta fase se establece la calidad del producto.
- ♦ *Implementación*: Es la traducción del diseño a un lenguaje de programación. A veces surgen problemas que obligan a volver al análisis o al diseño.
- ♦ *Prueba*: Se verifica si el software obtenido es consistente con la especificación de requerimientos. Si no cumple lo antedicho se regresa a las etapas anteriores.
- ♦ *Mantenimiento*: Se realizarán los cambios necesarios para mejoras funcionales, reparaciones, aumentos de rendimiento y conversiones demandadas por el usuario.

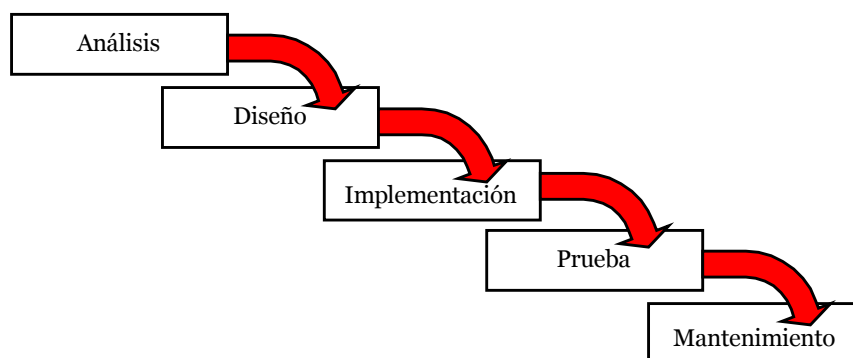


Figura 1.1: Proceso cascada

Este proceso presenta ciertos problemas:

Es difícil seguir la linealidad del ciclo.

Normalmente el usuario no especifica todos los requisitos.

El usuario no obtiene una versión del producto hasta que finaliza el ciclo.

Sin embargo, su aporte a la ingeniería de software fue muy valioso porque dejó en claro dos aspectos importantes: el proceso de desarrollo de software debe estar sujeto a disciplina, organización y planificación, y la codificación debe posponerse hasta que los objetivos sean bien definidos.

### 1.2.3. Prototipación

Consiste en la creación de una implementación parcial de un sistema, denominada prototipo, con el propósito explícito de aprender sobre los requerimientos del mismo (figura 2). Un prototipo es construido de una manera tan rápida como sea posible para ser entregado a los usuarios, posibilitando que experimenten utilizándolo. Su respuesta, les permite a los desarrolladores verificar los requerimientos, medir costos y beneficios, etc., quienes capturan en la documentación actual de la especificación de requerimientos la nueva información entregada, para el desarrollo del sistema real. El prototipo puede evolucionar hasta el sistema final, o bien puede ser descartado debido a su baja calidad en diseño.

La principal ventaja de este proceso radica en permitir al usuario interactuar con el sistema antes de su finalización, facilitando la localización inconsistencias y otros errores. Entre las desventajas se puede mencionar que su éxito depende de las herramientas de software que se utilicen para construir y modificar el prototipo. Además, el usuario suele confundir un prototipo avanzado con el producto final y presionar para que sea entregado como tal. Información adicional sobre este proceso puede encontrarse en [Gilb 88], [Wong 84] y [Gomaa 81].

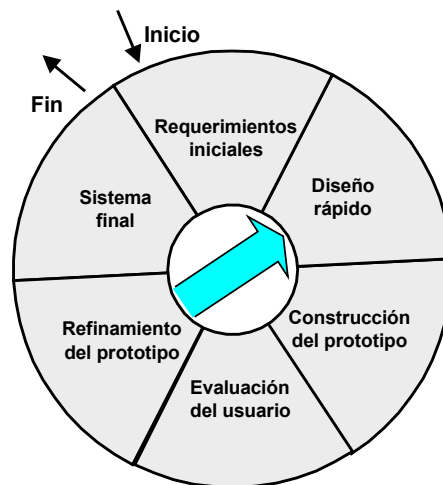


Figura 1.2: Prototipación

### 1.2.4. Proceso en espiral

Fue propuesto por Boehm [Boehm 88] y agrupa las mejores características del proceso en cascada y de la prototipación, agregándole el análisis de riesgos [Boehm, Papaccio 88]. El proceso en espiral es cíclico. Cada ciclo es dirigido por un análisis de riesgos -potenciales problemas a resolver- y se encuentra dividido en cuatro etapas, cada una de las cuales es un cuadrante del diagrama cartesiano que representa el proceso completo. En cada ciclo, se construye una versión del sistema más depurada y completa que la anterior, hasta obtener el producto final. Las tareas que se realizan en cada etapa son:

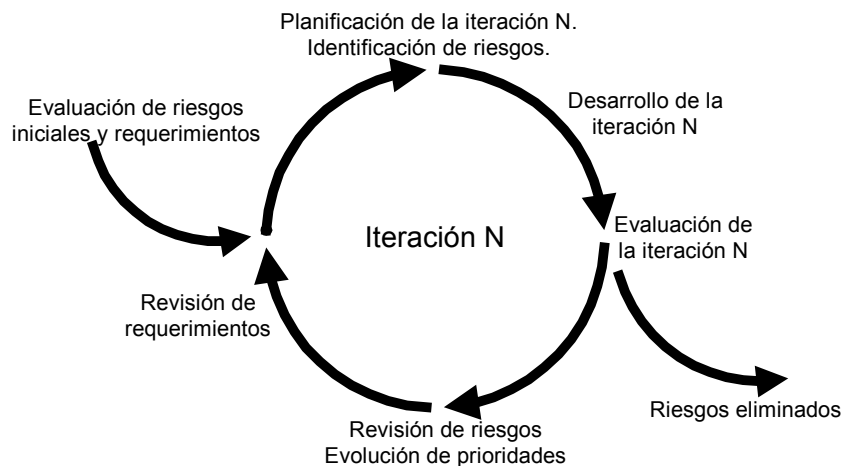
*Primera etapa:* Se identifican los objetivos, alternativas y restricciones a considerar.

*Segunda etapa:* Se evalúan las alternativas y los riesgos potenciales que corre el producto a partir de cada una de ellas. Para evaluar dichos riesgos se utilizarán prototipos y/o simulaciones, dependiendo de lo avanzado del proyecto.

*Tercera etapa:* Consiste en el desarrollo y verificación del producto en el nivel en el que se encuentre, ya sea de diseño solamente, o de implementación y codificación.

*Cuarta etapa:* Consiste en la revisión de los resultados obtenidos y la planificación del siguiente ciclo si ello fuera preciso.

El proceso en espiral es visto como un enfoque más realista para el desarrollo de grandes sistemas de software. Usa un método evolutivo para el desarrollo, y prototipación como una técnica de reducción de riesgos. Se ilustra en la figura 1.3.



**Figura 1.3: Proceso Espiral**

## 1.3. Incorporación de los modelos en el proceso de desarrollo

Los modelos se utilizan en muchas actividades de la vida humana: antes de construir una casa los arquitectos realizan planos, los músicos representan sus composiciones en partituras, los cocineros escriben recetas, etc. Un modelo es la representación simplificada de una realidad compleja, que permite comprenderla y simularla.

A finales de la década del 70 se observó un cambio importante en la filosofía del desarrollo de software, tendiente a solucionar los problemas de los procesos clásicos. Tom DeMarco introdujo el concepto de *ingeniería de software basada en modelos* [DeMarco 79]. En su trabajo, 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 de ingeniería. 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 entre métodos es la clase de modelos que deben construirse, la forma de representarlos, manipularlos, etc.

El punto de partida en el proceso de desarrollo 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.

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. La elección de los modelos tiene una profunda influencia sobre cómo se ataca el problema y se elabora la solución.

### 1.3.1. Utilidad de los modelos

En el proceso de desarrollo del software los modelos se utilizan para los siguientes propósitos:

*Definir las necesidades del usuario.* El propósito principal de la especificación de un producto es definir las necesidades de sus usuarios.

*Como medio de comunicación y negociación* entre usuarios y desarrolladores, y para la toma de decisiones en el equipo de desarrolladores. Permite mostrar al usuario una temprana aproximación al producto final.

*Como documento de referencia durante la corrección de errores.* Luego de introducir modificaciones en el sistema, la especificación es necesaria para chequear que la nueva implementación, está corrigiendo realmente los errores contenidos en la versión previa del producto.

*Como documento de referencia durante la evolución.* 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.

### 1.3.2. Los modelos en el ciclo de vida del software

Durante el ciclo de vida del software diferentes modelos del sistema en construcción son creados, tal como lo ilustra la figura 1.4. Las diferencias entre estos modelos residen en los aspectos del sistema que son contemplados (ningún modelo representa al sistema completo, sino que cada modelo enfatiza 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 cómo 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.

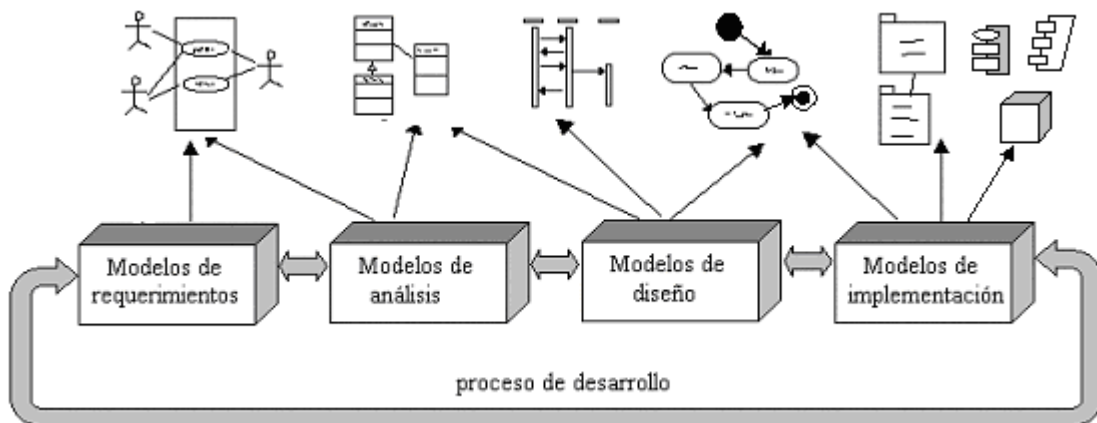


Figura 1.4: Modelos en el proceso de desarrollo

### 1.3.3. Cualidades de los modelos

El modelo de un problema es esencial para describirlo y entenderlo, independientemente del sistema informático que se utilice 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.

La actividad de construcción del modelo es 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.

Las cualidades relevantes para los modelos de análisis son:

*El modelo debe ser claro, no ambiguo y entendible.* Para que el modelo resulte útil debe poseer una semántica precisa. 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. Si el modelo del problema es incompleto, inconsistente, o se presta a interpretaciones erróneas, el resultado será un sistema de software cuya funcionalidad real difiera considerablemente con la esperada por sus usuarios.

*El modelo debe ser consistente.* No debe contener información contradictoria. Dado que un sistema es representado a través de diferentes submodelos relacionados, debería ser posible especificar precisamente cuál es la relación existente entre ellos, de manera que sea posible garantizar la consistencia del modelo compuesto.

*El modelo debe ser completo.* Debe documentar todos los requerimientos necesarios. Generalmente, no es posible lograr un modelo completo desde el inicio del proceso, entonces es importante comenzar con un documento de especificación incompleto y expandirlo a medida que se obtiene más información acerca del dominio del problema.

*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.* En principio el modelo debe ser verificado para asegurar que cumple con las expectativas del usuario. Luego, asumiendo que es correcto, puede usarse como referencia para verificar la corrección de las implementaciones del sistema.

## 1.4. Metodologías de desarrollo de software orientado a objetos

El desarrollo de software tradicional ha tenido un enfoque algorítmico, donde las metodologías se basan en procedimientos y funciones aplicados sobre estructuras de datos. En la actualidad, el desarrollo de software sigue un enfoque orientado a objetos, donde el elemento central del sistema de software es el *objeto*. La simplicidad del paradigma, que sólo cuenta con cinco conceptos fundamentales -los objetos, los mensajes, las clases, la herencia y el polimorfismo- para expresar de manera uniforme el análisis, el diseño y la realización de una aplicación informática, ha incentivado el nacimiento de alrededor de cincuenta metodologías en la primera mitad de la década del 90. Una

descripción detallada del paradigma de programación orientada a objetos puede encontrarse en [Budd 91, Booch 94, Coad and Yourdon 91, Shlaer and Mellor 88].

El inconveniente de esta superpoblación de métodos es que favorece la confusión, de modo que los usuarios se encuentran en una situación de espera que limita los progresos del desarrollo orientado a objetos. Sin embargo, el examen de los métodos dominantes permite extraer un consenso alrededor de ideas comunes. Se pueden mencionar las nociones de clase y asociación descritas por [Rumbaugh et al. 91], partición en subsistemas [Booch 94] y casos de uso en la expresión de las necesidades de interacción entre el usuario y el sistema [Jacobson et al. 92]. Además, los métodos más aceptados se ven reforzados por la experiencia, adoptando las prácticas más apreciadas por los usuarios.

### 1.4.1. Unificación de metodologías y origen de UML

Ante la evidencia de que las diferencias entre los métodos disminuían y que la guerra entre estos no hacía progresar la tecnología de objetos, Jim Rumbaugh y Grady Booch decidieron, a fines de 1994, unificarlos en un método único: The Unified Method. Al año siguiente se agrega Ivar Jacobson, creador de OOSE. El grupo se fijó cuatro objetivos:

- ♦ Representar sistemas completos utilizando conceptos de orientación a objetos;
- ♦ Establecer una relación explícita entre conceptos y artefactos ejecutables;
- ♦ Tener en cuenta factores de escalabilidad, inherentes a sistemas complejos y de misión crítica;
- ♦ Crear un lenguaje de modelado manejable tanto por humanos como por máquinas.

Si bien se alcanzó un rápido consenso sobre conceptos de orientación a objetos, el punto de divergencia estaba en la representación gráfica para los diferentes elementos de modelado. Luego de sucesivas versiones del método y sus respectivos feedbacks, se establecen nuevas prioridades en el esfuerzo de unificación. En primer lugar se orienta hacia la definición de un lenguaje universal para el modelado de objetos y, eventualmente, luego hacia la estandarización del proceso de desarrollo orientado a objetos. De esta manera nace The Unified Modeling Language for Object-Oriented Development (UML) [UML 97].

En consecuencia, la notación UML fue concebida para servir de lenguaje de modelado de objetos, independientemente de la metodología utilizada para el desarrollo de un sistema orientado a objetos. UML puede substituir -sin pérdida de información- a las notaciones de los métodos de Booch, OMT y OOSE (Objectory). En la figura 5 pueden verse las demás notaciones que aportaron conceptos a UML.

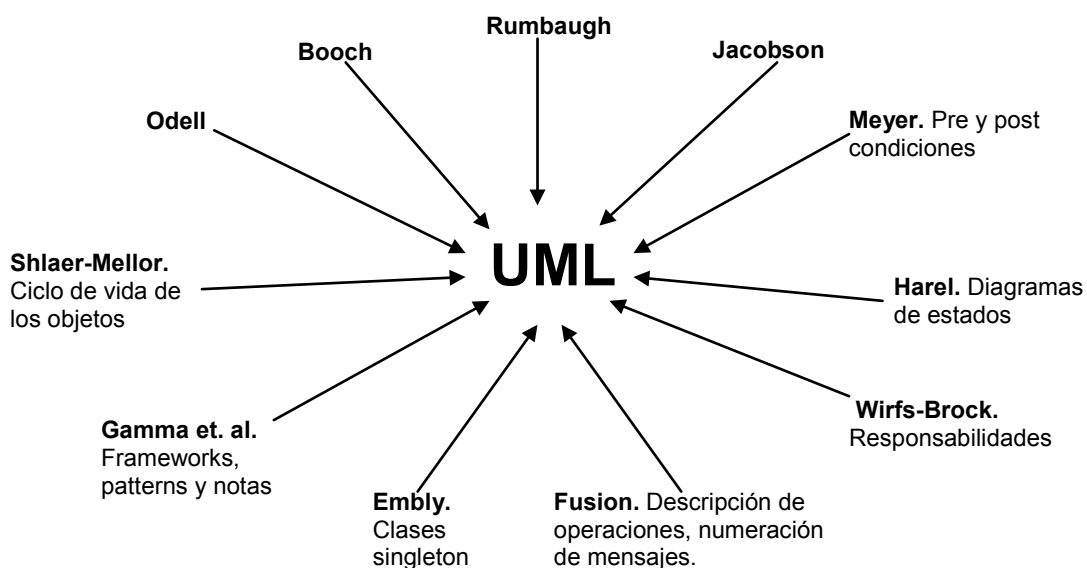


Figura 1.5: Contribuciones a UML

En cuanto a la metodología utilizada en el proceso de desarrollo, a fines del año 1998, los creadores de UML publican una versión unificada bajo la denominación de "*Proceso Unificado de Desarrollo*" [Jacobson et al. 1999]. Tiempo después se denominaría luego RUP (Rational Unified Process).

Sus principales características son:

- ♦ *Guiado por casos de uso*: todas las actividades, desde la especificación hasta el mantenimiento, utilizan como base los casos de uso.
- ♦ *Centrado en la arquitectura*. La arquitectura es una vista del diseño completo con las características más importantes resaltadas, dejando los detalles de lado. Debe satisfacer las necesidades expresadas en los casos de uso, tener en cuenta las evoluciones y cumplir las restricciones de realización. La arquitectura debe ser simple y comprensible intuitivamente.
- ♦ *Iterativo e incremental*: Se recomienda dividir el proceso en ciclos para hacerlo más manejable. Para cada ciclo se establecen fases de referencia, cada una de las cuales debe ser considerada como un miniproyecto cuyo núcleo fundamental está constituido por una o más iteraciones de las actividades básicas de cualquier proceso de desarrollo.

Las fases de cada ciclo de desarrollo del Proceso unificado son:

**Inicio**: especificación de la visión del producto final. Identificación y prioridad de riesgos.

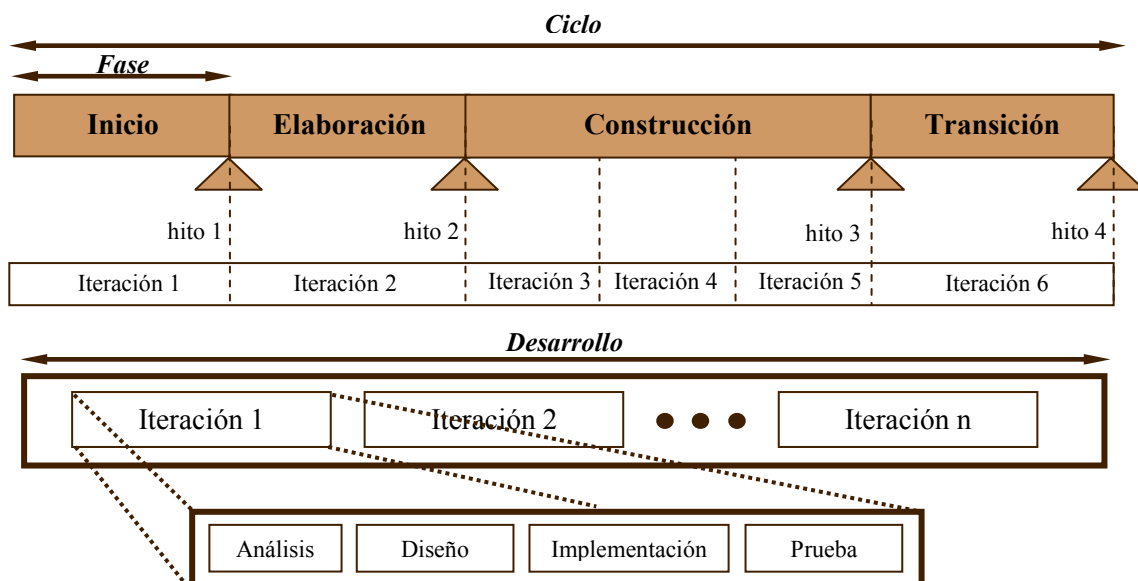
**Elaboración**: especificación de la mayoría de los casos de uso. Diseño de la arquitectura del sistema. Planificación de actividades. Estimación de recursos necesarios.

**Construcción**: Evolución de la visión en un producto, denominado versión beta, listo para ser entregado a la comunidad de usuarios.

**Transición**: Usuarios experimentados prueban el producto e informan las deficiencias. Corrección e incorporación de mejoras sugeridas. Entrenamiento de usuarios.

Las actividades de cada iteración comprenden: el análisis de requerimientos, el diseño, la implementación y las pruebas.

Un *hito* en el proceso unificado es un momento de tiempo en el ciclo donde se evalúan los objetivos logrados y se pueden tomar decisiones críticas para continuar con la siguiente fase. Con la conclusión de cada ciclo se obtienen una versión del producto. La figura 6 ilustra las etapas del proceso en diferentes dimensiones.



**Figura 1.6: Ciclo de vida del RUP**



## 1.5. Herramientas CASE

Por mucho tiempo, el único soporte del que disponían los desarrolladores de software estaba limitado a los tradicionales editores de texto para la codificación, y los compiladores del lenguaje respectivo.

Debido a esta escasez de herramientas adecuadas para el desarrollo de sistemas, surgió la lógica necesidad de crear sistemas que se pudieran utilizar como verdaderas herramientas de soporte en la construcción de software. De este modo nace la Ingeniería de Software Asistida por Computadora (Computer-Aided Software Engineering). Popularmente conocidas como herramientas CASE, están destinadas a apoyar una o más técnicas dentro del un proceso de desarrollo de software.

*Rational Rose* es la herramienta CASE que comercializan los desarrolladores de UML y que soporta en forma parcial la especificación de UML. Esta herramienta propone la utilización de vistas estáticas y dinámicas del modelo lógico y físico del sistema para capturar su proceso de análisis y diseño. Permite crear y refinar vistas hasta obtener un modelo completo que represente el dominio del problema.

Las características más destacadas de esta herramienta son:

*Desarrollo iterativo*: soporta un proceso de desarrollo iterativo controlado, donde el desarrollo se lleva a cabo en una secuencia de iteraciones.

*Trabajo en grupo*: permite que haya varias personas trabajando al mismo tiempo en el proceso iterativo controlado. Cada desarrollador opera en un espacio de trabajo privado que contiene el modelo completo y tiene un control exclusivo sobre la propagación de los cambios en dicho espacio de trabajo.

*Generación de código*: es posible generar código en distintos lenguajes de programación a partir de un diseño en UML.

*Ingeniería inversa*: proporciona mecanismos que permiten, a partir del código de un programa, obtener información sobre su diseño.

## 1.6. Modelos formales y modelos no-formales. La necesidad de integración

El modelo es una descripción abstracta de un sistema y se construye utilizando un lenguaje de modelado. Dependiendo del lenguaje utilizado, los modelos se pueden clasificar como informales ó formales. Los modelos informales son expresados utilizando lenguaje natural, figuras, tablas u otras notaciones gráficas. Se habla de modelos formales cuando la notación empleada es un formalismo, es decir, posee una sintaxis y semántica precisamente definidos, por ejemplo fórmulas matemáticas ó distintos tipos de lógica. Incluso existen estilos de modelado intermedios llamados semiformales, 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 los usados en las metodologías 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], y la notación Unified Modeling Language (UML) [UML 99, 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 como:

Malas interpretaciones de los modelos: la interpretación del modelo que realiza el usuario no coincide con la interpretación que realizó su creador.

Inconsistencia entre los diferentes modelos del sistema: la relación existente entre los submodelos, 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.

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 posibles interpretaciones.

En lo que respecta a 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]- la principal característica es que poseen sintaxis y semántica precisamente definidas. Sin embargo, su uso en la industria es poco frecuente. Esto se debe a la complejidad de sus formalismos matemáticos, difíciles de entender y comunicar. En la mayoría de los casos los expertos en el dominio del sistema que deciden utilizar una notación formal, concentran su esfuerzo en el manejo del formalismo en lugar de hacerlo sobre el modelo del problema. Esto conduce a la creación de modelos formales que no representan 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. La propuesta descrita en el capítulo 3 de esta tesis [Pons 00] ofrece claras ventajas; tanto por sobre el uso de un lenguaje informal, como por sobre el uso de un lenguaje formal, ya que les permite a los desarrolladores de software 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 propuesta más exitosa para lograr la integración consiste en definir formalmente la semántica de un lenguaje de modelado conocido y aceptado por la comunidad -ver por ejemplo [Evans et al. 99; Breu et al 97; UML 99; Evans et al 98; Kim and Carrington 99; Moreira y Clark 94; France et al 97; Waldoke et al 98; Wieringa and Broersen 98; Lano and Biccaregui 98]-. Sus principales componentes son reglas para asociar estructuras sintácticas del lenguaje de modelado con elementos en un dominio semántico formalmente definido.

La principal ventaja de esta propuesta reside en 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.

## **1.7. Verificación formal en herramientas de modelado**

El número de herramientas de modelado que soportan UML se ha incrementado notablemente en los últimos años, y son ampliamente utilizadas en la ingeniería de software orientada a objetos. Sin embargo, ha quedado rezagado el uso de métodos formales de ingeniería y especificación del software, excepto en proyectos de misión crítica. En parte se debe a que la especificación de UML no incluye ninguna aproximación a los métodos formales de ingeniería de software.

La definición de UML incluye la especificación de OCL (Object Constraint Language) [OCL 97], que consiste de una lógica simple para expresar restricciones e invariantes sobre elementos de

modelado. Sin embargo, ninguna de las herramientas comerciales o en desarrollo proveen soporte para OCL, mas allá de permitir la especificación de restricciones. Por el momento, la excepción a la regla es el trabajo de la Technische Universität Dresden, e IBM Labs (Inglaterra y Holanda), quienes han desarrollado un parser de OCL que ha sido integrado con el proyecto Argo/UML [Argo].

Por otro lado, se está desarrollando una herramienta de especificación de requerimientos denominada Ted Pearson, con el propósito de integrarla a las existentes que soportan UML, y poder asistir en la verificación de los requerimientos del sistema luego de su diseño.



# 2. La Notación UML

## 2.1. Introducción

UML (Unified Modeling Language) es un lenguaje gráfico para especificar, construir, visualizar y documentar los componentes de un sistema de software orientado a objetos. UML agrupa las prácticas ingenieriles más exitosas en el modelado de sistemas complejos. Está basado en los mejores conceptos de los métodos Booch, OMT y OOSE, entre otros. Conforman un lenguaje único, común y ampliamente aceptado por los usuarios de métodos antecesores.

UML está focalizado en estandarizar un lenguaje de modelado, y no un proceso 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, pudiendo extenderlos, si es necesario, bajo determinadas reglas.

## 2.2. Motivación para definir UML

El desarrollo de un modelo previo a la realización ó renovación de un sistema de software complejo, es tan importante como un anteproyecto para la construcción de un edificio. Los buenos modelos son esenciales para la comunicación entre equipos de trabajo y para asegurar validez desde el punto de vista de la arquitectura. Se construyen modelos porque es imposible comprender un sistema complejo en su totalidad.

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 incluyen desarrollo basado en componentes, programación visual, patrones de diseño y ambientes integrados de desarrollo. También se buscan técnicas para manejar la complejidad de los sistemas a medida que aumentan su alcance y escala. Debido a que la complejidad varía según el dominio de aplicación y la fase del desarrollo, se debe poner énfasis en crear una técnica que pueda contemplar adecuadamente la gran variedad de complejidades arquitectónicas a través de todas las fases del desarrollo, y en distintos tipos de dominios.

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 modelado estándar que permita expresar la información de manera clara y precisa y que sea conocido y aceptado por la comunidad. Antes de la aparición de UML, la mayoría de los lenguajes de modelado 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, en la mayoría de los casos sólo aparentes, entre los lenguajes de modelado de métodos previos. También se ha logrado unificar las perspectivas entre muchos diferentes tipos de sistemas, fases de desarrollo y conceptos internos. El eterno costo de utilizar y soportar muchos lenguajes de modelado motivó a muchas compañías, que utilizan o producen tecnología de objetos, a fomentar el desarrollo de UML.

## 2.3. Objetivos

Las metas principales que condujeron al diseño de UML son las siguientes:

- ❑ Proveer a los usuarios de un lenguaje de modelado expresivo y listo para usar, del modo 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 modelado; UML estandariza dichos conceptos.
- ❑ Proveer mecanismos de extensibilidad y especialización de los conceptos básicos. Se espera que UML deba ser personalizado para dominios específicos. A la vez no se quiere que los conceptos básicos sean redefinidos o reimplementados para cada área de aplicación. Las adaptaciones de UML serán las mínimas que se requieran para su adaptación al 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 modelado. Los usuarios necesitan un formalismo que los ayude a entender el lenguaje. Debe ser preciso y cercano al usuario. UML define formalmente el modelo estático usando un meta-modelo 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, frameworks, design patterns 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.

## 2.4. Historia

El desarrollo de UML se inició en Octubre de 1994 cuando Grady Booch y Jim Rumbaugh de *Rational Software Corporation* [Rational] comenzaron a trabajar en la unificación completa de sus métodos, Booch y OMT respectivamente. Como resultado, un borrador público es presentado en Octubre de 1995, denominado The Unified Method 0.8. A mediados de 1995 *Ivar Jacobson* y su compañía *Objectory* se incorporaron a *Rational*, aportando el método OOSE. Ahora el proceso unificado se denominaba Rational Objectory Process. Los esfuerzos de Booch, Rumbaugh, y Jacobson resultaron en la publicación de los documentos UML 0.9 y 0.91, en Junio y Octubre de 1996, respectivamente. Durante todo 1996 hubo un creciente feedback con la comunidad en general. Los comentarios fueron atendidos, sin embargo era necesaria una atención más focalizada.

Durante 1996, varias organizaciones expresaron que UML era estratégico para sus negocios y formaron un consorcio con Rational para lograr una sinergia hacia la nueva versión de UML. En Enero de 1997, UML 1.0 se presenta al Object Management Group (OMG) para su estandarización. OMG es una corporación independiente y sin fines de lucro, que aboga por el desarrollo de especificaciones para la industria del software que sean técnicamente excelentes, comercialmente viables e independientes del vendedor. En 1997 más compañías se unen a al consorcio, para contribuir con sus ideas en la revisión de UML. En Septiembre de ese año, la publicación de UML 1.1 es aprobada por el OMG, convirtiéndose en la notación estándar para el modelado de sistemas orientados a objetos.

Existe actualmente una *Revision Task Force* (RTF) responsable de la generación de revisiones menores de la especificación UML 1.1. La primera RTF de UML terminó su revisión en Junio de 1999 con el draft final UML 1.3. Los miembros de la segunda RTF están trabajando en lo que sería la siguiente revisión (UML 1.4). Además de estas revisiones menores, también se está trabajando en una revisión mayor que concluiría con la publicación de UML 2.0. La figura 2.1 ilustra el camino de UML.

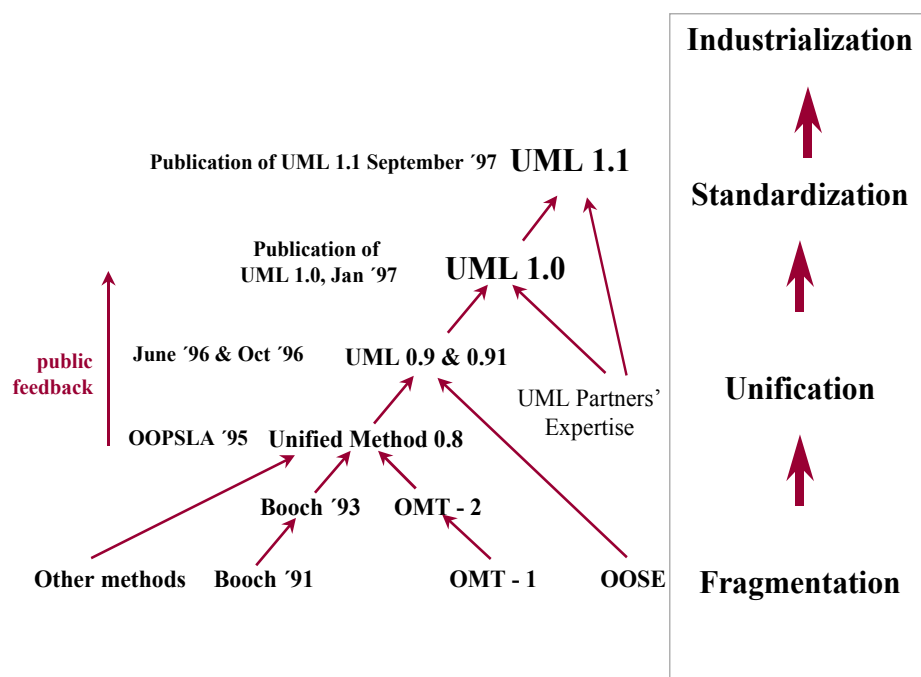


Figura 2.1: Evolución de UML

## 2.5. Organización de UML

La elección de los modelos y diagramas a crear en la especificación del sistema, tiene un profundo impacto sobre la manera de abordar y resolver el problema. 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 algún detalle de un elemento en particular, pero no necesariamente todos.

Un modelo se puede ver de una forma estática o de una forma dinámica. Un diagrama que representa una parte del modelo estático, describe la estructura de parte del sistema. Si representa una parte del modelo dinámico, describe el comportamiento de parte del sistema.

Para la representación de las vistas de un modelo, UML define los siguientes diagramas:

#### Modelo Estático:

Diagramas de Casos de Uso

Diagramas de Estructura estática:

Diagrama de clases

Diagrama de Objetos

Diagramas de Implementación:

Diagramas de Componentes

Diagramas de Despliegue

#### Modelo Dinámico:

Diagramas de Estados

Diagramas de Actividad

Diagramas de Interacción:

Diagramas de Secuencia

Diagramas de Colaboración

#### Gestión del modelo:

Diagramas de Clase (paquetes)

Mecanismos de extensión (forman parte de cualquier diagrama)

## 2.5.1. Mecanismos de extensión

Son mecanismos de propósito general que pueden ser aplicados a cualquier elemento de modelado. Constituyen recursos para extender el lenguaje. Los más importantes se detallan a continuación:

### **Stereotype**

Permite crear nuevos tipos de elementos de modelado basados en los elementos que forman el metamodelo UML. En consecuencia, un estereotipo será un nuevo tipo de elemento de modelado que extiende la semántica del metamodelo pero no modifica la estructura de los tipos o clases preexistentes. Algunos estereotipos están predefinidos en UML, otros pueden ser definidos por el usuario. Gráficamente, un estereotipo se representa como un nombre entre comillas.

«nombreEstereotipo»

### **Constraint**

Son condiciones o restricciones, relacionadas a un elemento o a un conjunto de ellos. Tienen significado semántico ya que representan condiciones que deben ser cumplidas por las instancias del modelo. UML no especifica una sintaxis particular para las constraints, las cuales pueden expresarse en lenguaje natural, expresiones matemáticas, pseudocódigo, etc. Se representan mediante un *comment* próximo al elemento condicionado ó como un texto entre paréntesis (figura 2.2).



## Comment

Es un texto ligado a uno o más elementos de modelado. Es una *constraint* escrita en lenguaje informal. Un comentario se representa con texto dentro de un símbolo *note* que es ligado a un elemento del modelo con una línea de punta.

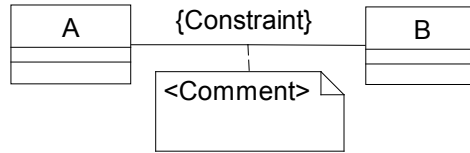


Figura 2.2: Constraints y Comments

## 2.5.2. Diagramas de casos de uso

Los diagramas de casos de uso son diseñados con el fin de representar la información del análisis de requerimientos. Los casos de uso se describen bajo la forma de acciones y reacciones el comportamiento de un sistema desde el punto de vista del usuario. Permiten definir los límites del sistema y las relaciones entre el sistema y su entorno. En general, los casos de uso representan una secuencia cualquiera de transacciones llevadas a cabo cuando un usuario utiliza el sistema.

Un *actor* es el rol que determinado usuario juega con respecto al sistema. Es importante destacar el uso de la palabra *rol*, pues con esto se especifica que un *actor* no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema. Un caso de uso es una secuencia de interacciones entre un sistema y algún agente externo que usa alguno de sus servicios. Es iniciado por un actor o invocado desde otro caso de uso.

Existen tres tipos de relación entre los elementos de estos diagramas. Una relación de comunicación indica a participación del actor en un caso de uso. Una relación de uso expresa que la tarea descrita en un caso de uso utiliza el comportamiento especificado en otro caso de uso. Una relación de extensión implica que un caso de uso extiende el comportamiento de un caso de uso base.

Este tipo de diagramas (figura 2.3) es representado con un grafo de actores, un conjunto de *use cases* que pueden ser encerrados por un delimitador del sistema, asociaciones de comunicación entre los actores y los use cases, y generalizaciones entre los use cases. Un *actor* puede ser mostrado como un rectángulo de clase con el estereotipo «actor», o más comúnmente, con un icono estándar representando una persona y el nombre del rol debajo. Un caso de uso se muestra como un elipse conteniendo el nombre del caso de uso expresado desde el punto de vista del actor.

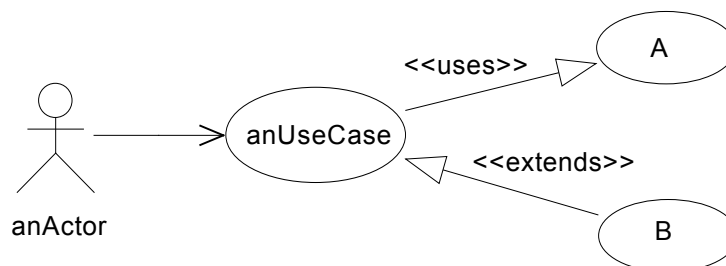


Figura 2.3: Diagrama de casos de usos

### 2.5.3. Diagramas de estructura estática

Este tipo de diagramas comprende a los diagramas de clases y a los diagramas de objetos. Los diagramas de clases muestran la estructura estática del modelo, en términos de clases y tipos, su estructura interna, y las relaciones con otros elementos. Son colecciones de elementos declarativos (estáticos) y sus relaciones, conectadas como un grafo. Los diagramas de clases pueden ser organizados en paquetes.

Los diagramas de objetos modelan las instancias de elementos contenidos en los diagramas de clases, incluyendo valores de tipos. Muestran un conjunto de objetos, sus estados y sus relaciones en un momento concreto. Se utilizan para visualizar, especificar, construir y documentar la existencia de ciertas instancias en el sistema, junto a las relaciones entre ellas.

#### Class

Es un descriptor para un conjunto de objetos con similar estructura, comportamiento y relaciones. Su nombre debe ser único dentro del package al cual pertenece. Cada clase tiene un conjunto de propiedades, entre las que se destacan los atributos y las operaciones. Se representa con un rectángulo (Figura 2.4) con tres compartimentos separados por líneas horizontales: el superior contiene el nombre y otras propiedades generales de la clase, incluyendo estereotipos; el del medio contiene la lista de atributos y el inferior contiene la lista de operaciones.

#### Attribute

Es una característica estructural de una class. Cada Attribute tiene un tipo asociado, un valor inicial, una visibilidad y puede especificar si está definido a nivel de Clase o de Instancia. Un atributo se representa con texto y la sintaxis por defecto es la siguiente:

```
visibility name : type = initialValue {propertyString}
```

Donde *type* es el tipo del atributo con nombre *name*. Además, puede especificarse un valor inicial y un conjunto de propiedades del atributo. La visibilidad de un atributo se especifica como: public (+), protected (#), o private (-).

#### Operation

Especifica una característica de comportamiento de una *class*. Para cada operación se indica si es polimórfica, si es abstracta y si modifica o no el estado del sistema. Cada operación tiene parámetros, un tipo asociado, valores iniciales, una visibilidad y si está definida a nivel *clase* o *instancia*. La implementación puede ser especificada explícitamente proveyendo un valor para el cuerpo (típicamente en un lenguaje de programación cuyo alcance está fuera de UML) o implícitamente derivada a partir de los diagramas de comportamiento. La sintaxis de una operación en UML es:

```
visibility name (parametersList) : returnType {propertyString}
```

Donde cada uno de los parámetros en *parametersList* se denota igual que un atributo. El resto de la notación es similar a la de un atributo.

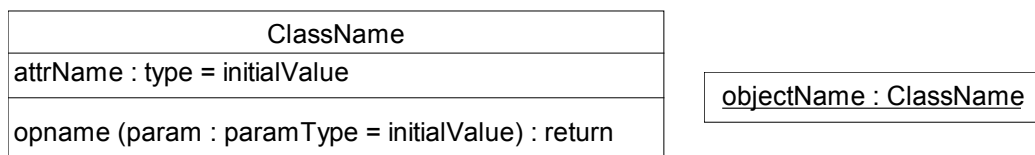


Figura 2.4: Clase y objeto

## Object

Representa una instancia particular de una clase. Un objeto se representa como un rectángulo con dos compartimentos. El compartimento superior muestra el nombre de la instancia y su clase, separados con “.” y subrayados. El compartimento inferior muestra una lista de los atributos del objeto y sus respectivos valores.

`objectName : className`                      `attributeName : type = value`

## Package

Los paquetes ofrecen un mecanismo general para la partición de los modelos y la agrupación de los elementos de modelado. Son unidades de organización jerárquica de uso general en los modelos de UML. Un *package* puede contener elementos ordinarios de modelado y otros packages sin límites de anidamiento, pero un elemento pertenece solo a un package. También definen propiedades de alcance y visibilidad de los elementos. Un *package* se representa mediante rectángulo grande (figura 2.5) con uno más pequeño en el vértice superior izquierdo. El nombre puede estar en cualquiera de los rectángulos, dependiendo de si los elementos contenidos son mostrados en el interior del package. Relaciones de dependencia entre packages, pueden ser mostradas cuando al menos hay una dependencia entre elementos contenidos en dichos packages.

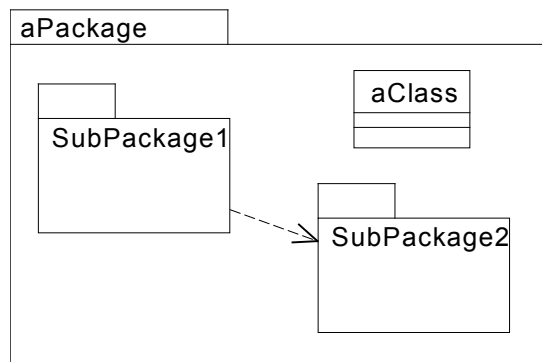


Figura 2.5: Paquetes

## Association

Es una relación estructural entre clases de objetos. En la mayoría de los casos son binarias. Las asociaciones binarias se representan con una línea sólida que une las clases involucradas. Las asociaciones entre más de tres clases se representan con un rombo, desde donde salen líneas sólidas hacia cada clase involucrada. El nombre de la asociación puede presentarse cercano a la línea.

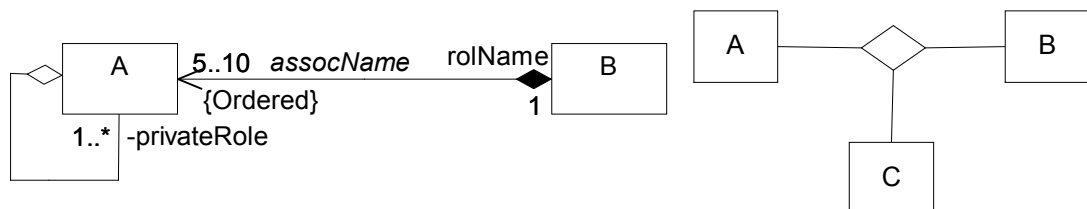


Figura 2.6: Asociación binaria y n-aria

## AssociationEnd

Representa el papel que cumple una clase en la asociación. Es parte de una asociación, no de la clase, y no se puede separar. Cada asociación tiene un rol por cada clase que conecta. Un rol se puede representar con varios adornos cercanos o ligados a la unión de la asociación con la clase. Las propiedades de un rol que pueden representarse son:

*Nombre*: un texto que indica el rol que juega la clase en la relación.

*Navegabilidad*: se representa con una flecha e indica que la navegación hacia la clase ligada es posible.

*Multiplicidad*: indica la cantidad de instancias de la clase que participan en la asociación. Se representa con rangos de números enteros positivos, por ejemplo: \* (de cero a muchos), 0..1 (cero ó uno), 1..\* (de uno a muchos), etc.

*Orden*: si la multiplicidad es mayor a 1, las instancias de las clases involucradas en la asociación pueden estar ordenadas. Se representa con el texto `{ordered}`.

*Agregación*: especifica la relación parte/todo, entre el agregado (todo) y una parte que lo compone. Se representa con un diamante entre el agregado (todo) y la asociación. Un diamante vacío indica agregación (la existencia de la parte no depende del todo). Un diamante lleno especifica la versión fuerte de agregación, denominada composición (la existencia de la parte depende del todo).

*Calificador*: es un conjunto de atributos cuyos valores, junto a un objeto de la clase de partida, sirven para particionar el conjunto de instancias de la clase destino relacionadas a través de la asociación. Se representa con un rectángulo pequeño entre el final de la asociación y la clase fuente. Los atributos se muestran dentro del rectángulo.

*Visibilidad*: ídem visibilidad de atributos.

*Variabilidad*: indica la forma en que la conexión es modificada. `{none}` los links pueden ser agregados o eliminados sin restricciones. `{frozen}` indica que ningún link puede ser agregado, ni eliminado luego de que el objeto fue creado e inicializado. `{addOnly}` indica que pueden ser agregados links adicionales, pero no pueden ser eliminados ni modificados.

## AssociationClass

Es una asociación que tiene propiedades de clase (o una clase que tiene propiedades de asociación). Se representa con una línea punteada que une el símbolo de clase con el camino de la asociación. El nombre de la clase y el de la asociación son redundantes y deberían ser los mismos.

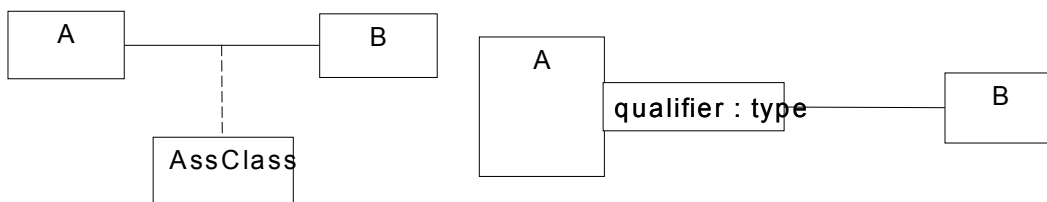
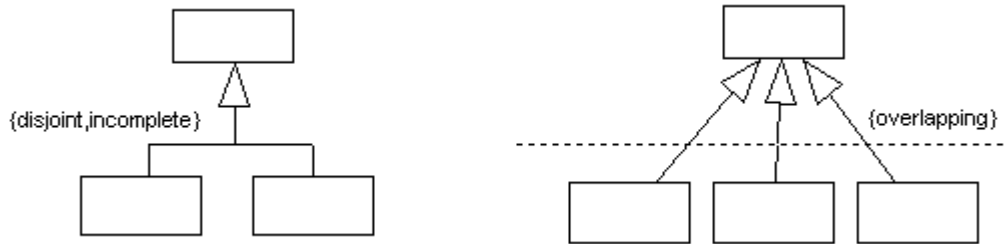


Figura 2.7: AssociationClass y Qualifier

## Generalization

Es la relación taxonómica entre un elemento más general y uno más específico, el cual es completamente consistente con el primer elemento y que agrega información adicional. Un elemento (clase o paquete) puede heredar de uno (herencia simple) o varios (herencia múltiple) superelementos. Se representa con un camino de línea sólida desde el elemento más específico (subclase) hasta el elemento más general (superclase), con un triángulo vacío entre el final del camino y el elemento más general. Ver figura 2.8.

Cada generalización puede tener un discriminador para especificar el nombre de una partición para los subtipos. Se representa con un texto cercano al camino de generalización. También pueden usarse restricciones semánticas predefinidas entre las subclases. Se representan con una lista de palabras clave entre llaves cercanas al triángulo de la relación ó sobre una línea punteada (si se quiere abarcar varias generalizaciones). Las siguientes restricciones son predefinidas: `{overlapping}` un descendiente de una clase A puede descender de más de una subclase de A. `{disjoint}` un descendiente de una clase A no puede descender de más de una subclase de A. `{complete}` todas las subclases han sido especificadas y no se esperan subclases adicionales. `{incomplete}` se sabe que solo algunas subclases han sido especificadas y faltan otras.



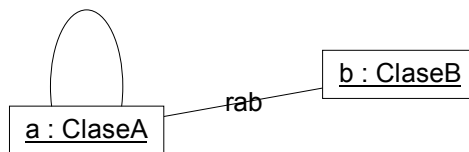
**Figura 2.8: Generalizaciones restringidas**

### Dependency

Indica una relación semántica entre dos (o más) elementos de modelado. Especifica que un cambio en el elemento destino o independiente puede requerir un cambio en el elemento fuente o dependiente de la relación de dependencia. Se representa con una flecha punteada desde el elemento dependiente hasta el elemento independiente. Ver figura 2.5.

### Link

Es una tupla, generalmente un par, de referencias a objetos. Representa una instancia de una asociación. Un link binario se muestra como un camino entre dos objetos, incluso puede mostrarse como un *loop* en el caso de asociaciones reflexivas. Pueden mostrarse nombres de roles en los extremos del link y el nombre subrayado de la asociación junto al camino. Un link no tiene nombre de instancia, pero puede identificarse con los objetos que relaciona. Un link n-ario es mostrado con un diamante desde donde parten los caminos hacia cada uno de los objetos participantes. Ver figura 2.9.



**Figura 2.9: Links**

## 2.5.4. Diagramas de interacción

Un 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 tipos de diagramas de interacción: diagramas de colaboración y diagramas de secuencia.

### 2.5.4.1. Diagramas de secuencia

Un diagrama de secuencia especifica las interacciones entre objetos desde un punto de vista temporal. Muestra los objetos que se encuentran en un escenario y la secuencia de mensajes intercambiados a través del tiempo para llevar a cabo la funcionalidad descrita. Esta descripción es importante para documentar y validar los casos de uso, trasladándolos al nivel de mensajes de los objetos existentes. También sirven para detallar el uso de los mensajes de las clases diseñadas, en el contexto de una operación. Sus elementos principales son *Object lifeline*, *Activation* y *Message*. En un diagrama de secuencia los objetos se representan como una línea vertical punteada con un símbolo del objeto encabezando la línea y con rectángulos a través de la línea principal que denotan la ejecución de operaciones (activaciones). Mientras que los envíos de mensajes entre objetos se denotan mediante flechas sólidas desde la línea de vida del objeto que emite el mensaje hasta la línea de vida del objeto que lo ejecuta. La figura 2.10 ilustra un diagrama de este tipo.

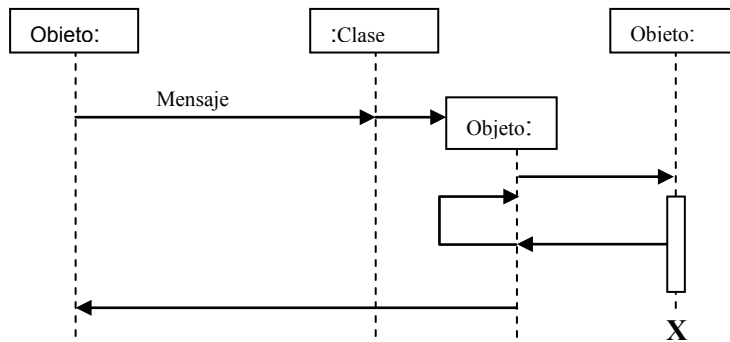


Figura 2.10: Diagramas de secuencia

### 2.5.4.2. Diagramas de Colaboración

Un diagrama de colaboración muestra simultáneamente las interacciones de un conjunto de objetos y las relaciones estructurales que permiten estas interacciones. No se muestra el tiempo como una dimensión separada, aunque se puede indicar el orden del flujo de mensajes usando una secuencia de números. Un diagrama de colaboración se representa con un grafo de referencias a objetos y links ligados a flujos de mensajes. Sus elementos principales son: *Collaboration*, *Message Flow*, *Message Flow*, *Interaction*, *Active Object* y *Multiobject*. La figura 2.11 ilustra un diagrama de este tipo.

Los diagramas de secuencia proporcionan una forma de ver el escenario en un orden temporal (qué pasa primero, qué pasa después), por lo que resultan útiles en las primeras fases de análisis. Los diagramas de colaboración proporcionan la representación principal de un escenario, ya que las colaboraciones se organizan en torno a los enlaces de unos objetos con otros y se utilizan más frecuentemente en la fase de diseño, cuando se están buscando las relaciones.

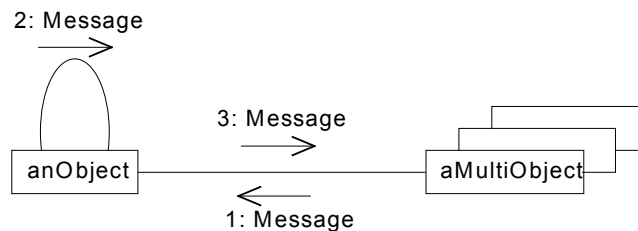


Figura 2.11: Diagrama de Colaboración

### 2.5.5. Diagramas de estados

Un diagrama de estados representa una máquina de estados. Muestra la secuencia de estados por los que un objeto o una interacción pasa durante su existencia en respuesta a los estímulos recibidos, junto con sus respuestas y acciones. Un estado es una condición durante la vida de un objeto, de forma que cuando dicha condición se satisface se lleva a cabo alguna acción o se espera por un evento. El estado de un objeto se puede caracterizar por el valor de uno o varios de los atributos de su clase.

Una máquina de estados es ligada a una clase o a un método. Los estados se representan con símbolos de estado y las transiciones con flechas que conectan dichos símbolos (figura 2.12). Los autómatas utilizados por UML, para representar las máquinas de estados, son determinísticos. Esto significa que no describe comportamientos ambiguos. Siempre se debe especificar un estado inicial y se pueden describir varios estados finales. Los elementos que se destacan en estos diagramas se describen a continuación:

## State

Es una situación en la vida de un objeto, o interacción, durante la cual se satisface alguna condición, se realiza una acción o se espera un evento. Un estado normal se representa con un rectángulo de esquinas redondeadas. El estado inicial se representa con un círculo lleno y los estados finales se muestran con un círculo lleno y una circunferencia de mayor diámetro alrededor. Un estado puede tener tres compartimientos: el superior para el nombre del estado; el intermedio para el valor característico de los atributos del objeto en ese estado y el inferior para las actividades internas del objeto. En el compartimento de actividad interna se colocan acciones de la forma:

```
eventName / actionExpression
```

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 y las transiciones internas.

## Event

Es una ocurrencia que puede causar la transición de un estado a otro de un objeto. Los tipos de ocurrencia pueden ser: una condición booleana que cambia su valor; la recepción de una señal de otro objeto en el modelo; la recepción de un mensaje, el paso de cierto período de tiempo, etc. Se representa con el siguiente formato:

```
eventName(parametersList)
```

## Simple Transition

Es una relación entre dos estados que indica que un objeto en el primer estado puede entrar al segundo estado y ejecutar ciertas operaciones cuando un evento ocurre y si ciertas condiciones son satisfechas. Se representa como una línea sólida entre dos estados, que puede venir acompañada de un texto con el siguiente formato:

```
eventSignature [guardCondition] / actionExpression ^sendClause
```

donde:

*eventSignature* es el nombre del evento que origina la transición;

*guardCondition* son las condiciones adicionales al evento necesarias para que la transición ocurra;

*actionExpression* es un mensaje al mismo u otro objeto, que se ejecuta como resultado de la transición y el cambio de estado;

*sendClause* son acciones adicionales que se ejecutan con el cambio de estado, por ejemplo, el envío de eventos a otros objetos.

## Complex Transition

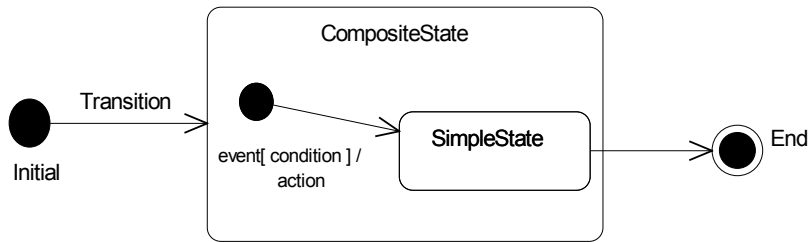
Puede tener múltiples estados fuente y/o múltiples estados destino. Se representa como una línea vertical desde la cual ingresan o salen varias líneas de transición de estado.

## Internal Transition

Representa una transición entre subestados que permanece en el mismo estado compuesto. Se representa gráficamente dentro del estado compuesto.

## 2.5.6. Diagramas de actividades

Un diagrama de actividades es una variante de los diagramas de estados, organizado respecto a las acciones y principalmente destinado a representar el comportamiento interno de un método o de un caso de uso. Representa el estado de la ejecución de un mecanismo bajo la forma de un desarrollo de etapas agrupadas secuencialmente en ramas paralelas de flujo de control. Esta clase de diagramas es adecuada para representar programas concurrentes.



**Figura 2.12: Máquinas de Estados**

Cada actividad representa una etapa particular en la ejecución de la operación y se representa con un rectángulo redondeado de mayor base que el utilizado para estados. Las actividades se conectan con transiciones automáticas, representadas por flechas. Cuando una actividad termina se desencadena la transición y comienza la siguiente actividad. Una actividad no posee transiciones internas. Es posible representar sincronización entre flujos de control con condiciones mutuamente exclusivas. Información más detallada sobre estos diagramas se puede encontrar en [UML 99].

## 2.5.7. Diagramas de componentes

Un diagrama de componentes muestra las dependencias entre los componentes del software. Un componente es un fragmento de código ya sea código fuente, binario o ejecutable. Las relaciones de dependencia se utilizan para indicar que un componente se refiere a los servicios ofrecidos por otro componente. Una dependencia se representa con una flecha punteada desde el usuario al proveedor y puede especializarse con un estereotipo.

Desde el punto de vista de estos diagramas se consideran los requisitos relacionados con la facilidad de desarrollo, la gestión del *software*, el reuso, y las restricciones impuestas por los lenguajes de programación y las herramientas utilizadas en el desarrollo. Los elementos de modelado dentro de un diagrama de componentes serán componentes y paquetes. En cuanto a los componentes, sólo aparecen tipos de componentes, ya que las instancias específicas de cada tipo se encuentran en el diagrama de despliegue.

Un paquete en un diagrama de componentes representa una división física del sistema de software. Los paquetes se organizan en una jerarquía de capas, teniendo cada una de ellas una interfaz bien definida. Un ejemplo típico de jerarquía de capas es: Interfaz de usuario; Paquetes específicos de la aplicación; Paquetes reusables; Mecanismos claves; y Paquetes *hardware* y del sistema operativo. Más información puede encontrarse en [UML 99].

## 2.5.8. Diagramas de despliegue

Los diagramas de despliegue muestran las relaciones físicas entre los componentes de *hardware* y de *software* en el sistema final, es decir, la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes de *software* (procesos y objetos) que se ejecutan en ellos. Los diagramas de despliegue se utilizan para razonar sobre la topología de procesadores y dispositivos sobre los que se ejecuta el software.

Un diagrama de despliegue se representa con un grafo de nodos unidos por asociaciones de comunicación. Un nodo es un objeto físico en tiempo de ejecución que representa un recurso computacional, generalmente con memoria y capacidad de procesamiento. Pueden ser presentados como tipos y como instancias (nombre subrayado). Un nodo se representa con un cubo de tres dimensiones. La naturaleza del nodo puede precisarse con un estereotipo. Información detallada en [UML 99].



# 3. La M&D-theory

## 3.1. Introducción

En este capítulo se presenta una descripción resumida de la M&D-theory que define formalmente la semántica de UML. Dicha teoría fue desarrollada por Claudia Pons en su tesis doctoral [Pons 00]. La idea básica de esta 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.

### 3.1.1. Dicotomía de entidades

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

- Entidades de modelado
- Entidades modeladas

Esta dicotomía puede observarse en la figura 3.1. Las entidades de modelado se corresponden con construcciones sintácticas correctas del lenguaje UML, tales como clases (Class), máquinas de estados (StateMachine), etc. En contraste, las entidades modeladas, tales como objetos (Object) ó conexiones (Link) representan los datos del sistema modelado.

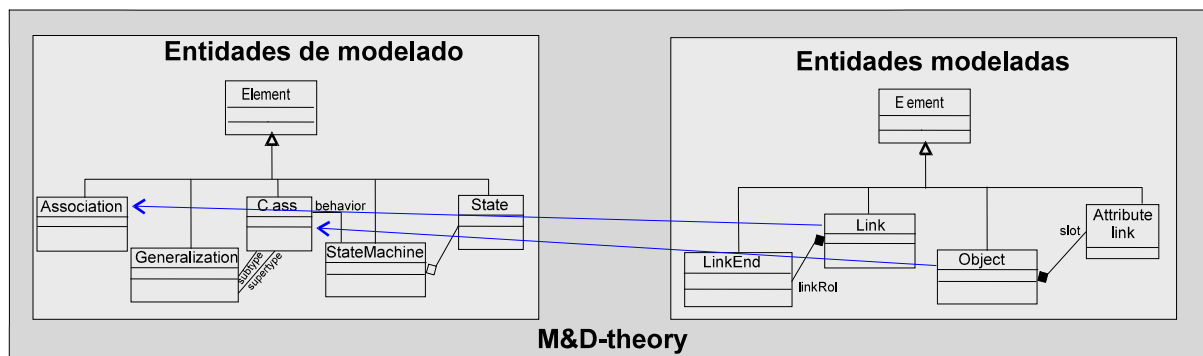


Figura 3.1: Dicotomía de entidades

Estas entidades se relacionan de distintas formas:

*Entre entidades de modelado.* Por ejemplo las clases están relacionadas con las máquinas de estado 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.

*Entre 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.

*Entidades de modelado 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 son instancias de una Clase, mientras que las conexiones son instancias de una asociación (Association).

### 3.1.2. 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 3.2 las flechas llenas que conectan entidades entre diagramas alineados horizontalmente representan algunas instancias 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 3.2 las flechas punteadas que conectan diagramas alineados verticalmente representan algunas instancias inter-nivel. Por ejemplo, BankAccount es instancia de Class, holder es instancia de Association, C1 es instancia de Object, S2 es instancia de Object.

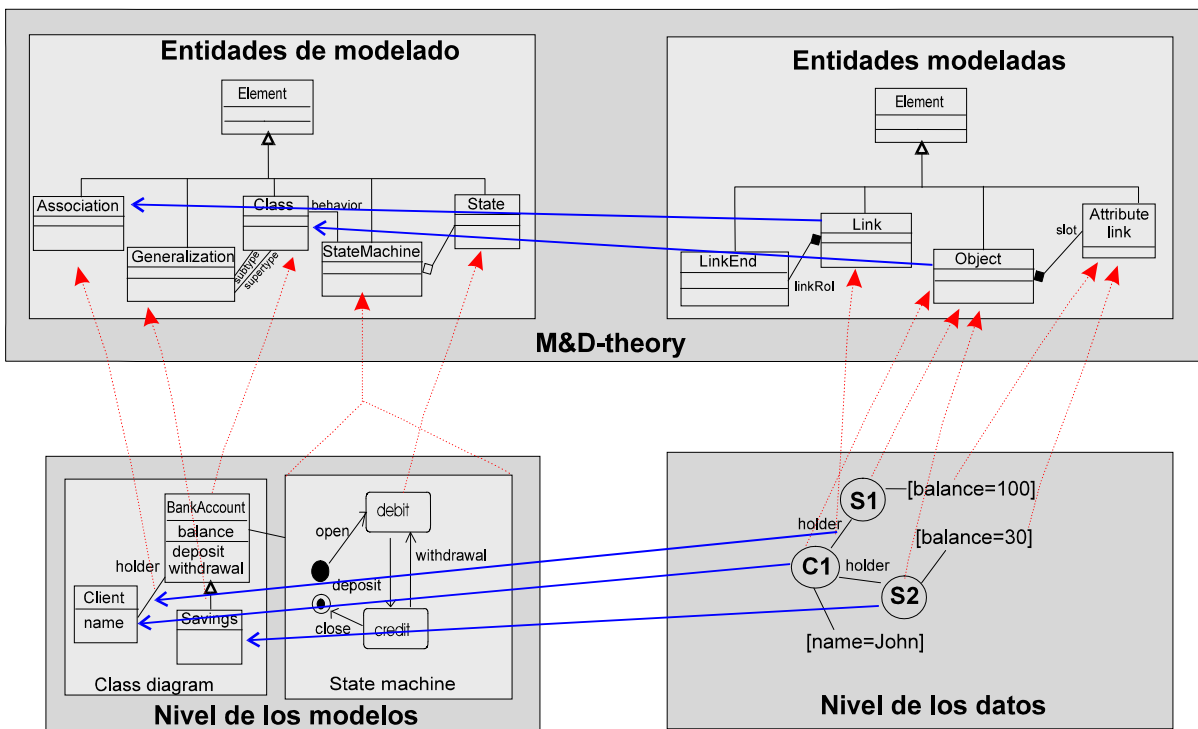


Figura 3.2: Relaciones de instanciación

### 3.1.3. Ventajas de la integración

En la descripción de un lenguaje existen dimensiones ortogonales a la sintaxis y la semántica, son los aspectos estáticos y los aspectos dinámicos. La diferenciación entre semántica estática y semántica dinámica es ampliamente conocida y aceptada. Mientras que la primera caracteriza propiedades estáticas (invariantes en el tiempo) de las interpretaciones del lenguaje, la segunda describe la evolución o comportamiento de dichas interpretaciones.

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 esta propuesta proporciona a cada una de las cuatro dimensiones:

	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.

### 3.1.4. Organización del capítulo

Este capítulo está organizado de la siguiente manera: en la sección 3.2 se describe la formalización  $(\Sigma_{UML}, \phi_{UML})$  de las entidades de modelado. En la sección 3.3 se presenta la formalización  $(\Sigma_{SYS}, \phi_{SYS})$  de las entidades modeladas. La sección 3.4 contiene consideraciones acerca de la integración de ambos niveles. Luego, en la sección 3.5 se describe la función de interpretación que asocia las construcciones de UML con los elementos en el dominio semántico. Por último, la sección 3.6 contiene las principales conclusiones.

## 3.2. Nivel de los Modelos

### 3.2.1. Elementos

En el lenguaje UML, los diagramas de clases modelan los aspectos estructurales del sistema. Estos diagramas incluyen gráficos para representar clases y relaciones entre ellas, tales como generalizaciones (Generalization), agregaciones y asociaciones. Por otra parte, la parte dinámica del sistema es modelada mediante diagramas de colaboración y máquinas de estados, los cuales describen el comportamiento de un grupo de instancias en términos de envíos de mensajes y el dinamismo interno de los objetos en términos de transiciones entre estados respectivamente.

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.

### 3.2.2. 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: agregar nuevas clases de objetos, agregar o eliminar atributos de clases existentes, modificar la jerarquía de herencia, etc. Existen también formas de evolución que no sólo modifican la estructura del sistema, sino que lo hacen con el comportamiento especificado para los objetos. Los cambios de comportamiento se reflejan en las modificaciones realizadas sobre diagramas de comportamiento.

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

### 3.2.3. 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. Gran parte de la información contenida en cada paquete puede ser comprendida y analizada independientemente. Sin embargo es necesario prestar atención a las relaciones existentes entre los diferentes paquetes. La figura 2.3 muestra los paquetes de mas alto nivel en los cuales la teoría se encuentra dividida. Sus nombres son: Foundation, BehavioralElements y ModelManagement. La descripción de cada paquete está integrada por las siguientes secciones:

- **Sintaxis Abstracta.** La sintaxis abstracta es representada mediante diagramas UML mostrando las metaclases y sus relaciones contenidas en el paquete. Estos diagramas muestran además algunas reglas de buena formación, tales como requerimientos de multiplicidad.
- **Descripción Informal.** Presenta una breve descripción informal, utilizando lenguaje natural. Para cada metaclase se enumeran sus atributos y asociaciones con una breve explicación de su significado.
- **Especificación en Lógica Dinámica.** Especificación formal de las metaclases usando un lenguaje formal basado en *Dynamic Logic* (DL). Esta especificación consta de una signatura  $\Sigma_{UML} = ((S_{UML}, \leq), F_{UML}, P_{UML}, A_{UML})$  y una fórmula  $\phi_{UML}$  sobre  $\Sigma_{UML}$ . 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  $\phi_{UML}$  es la unión de dos conjuntos disjuntos de fórmulas;  $\phi_S$  y  $\phi_D$ , estáticas y 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), y 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.

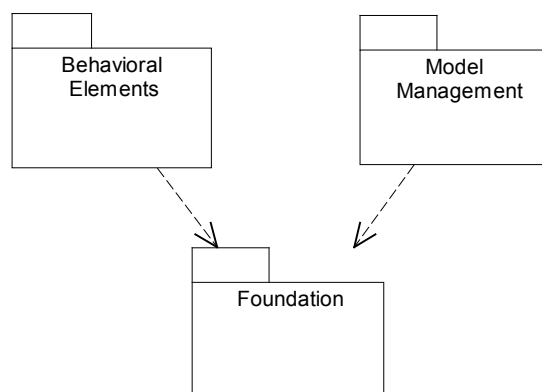


Figura 3.3: Paquetes del nivel superior

### 3.2.4. Paquete Foundation

El paquete Foundation define la infraestructura de UML. Se descompone en dos subpaquetes: Core y UML\_Data Types (figura 3.4). El paquete UML\_Data Types define los tipos de datos básicos de la teoría, tales como Boolean, Integer, etc. El paquete Core especifica los principales elementos para modelar la estructura de un sistema, tales como Class, DataType, Generalization, Association, Attribute, Operation, etc.

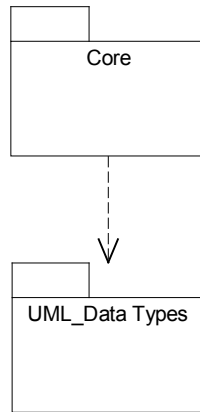


Figura 3.4: Paquete Foundation

#### 3.2.4.1. Paquete Foundation: UML\_Data Types

##### Sintaxis Abstracta

La figura 3.5 muestra la descripción gráfica de la sintaxis abstracta de los elementos que componen el paquete UML\_Data Types. Es importante destacar la diferencia entre estos tipos de datos básicos y los tipos de datos utilizados por los desarrolladores de sistemas. Los tipos del paquete UML\_DataTypes son necesarios para definir el lenguaje en sí, mientras que los desarrolladores, utilizando el lenguaje, pueden crear otros tipos de datos, instanciando la metaclassa DataType del paquete Core.

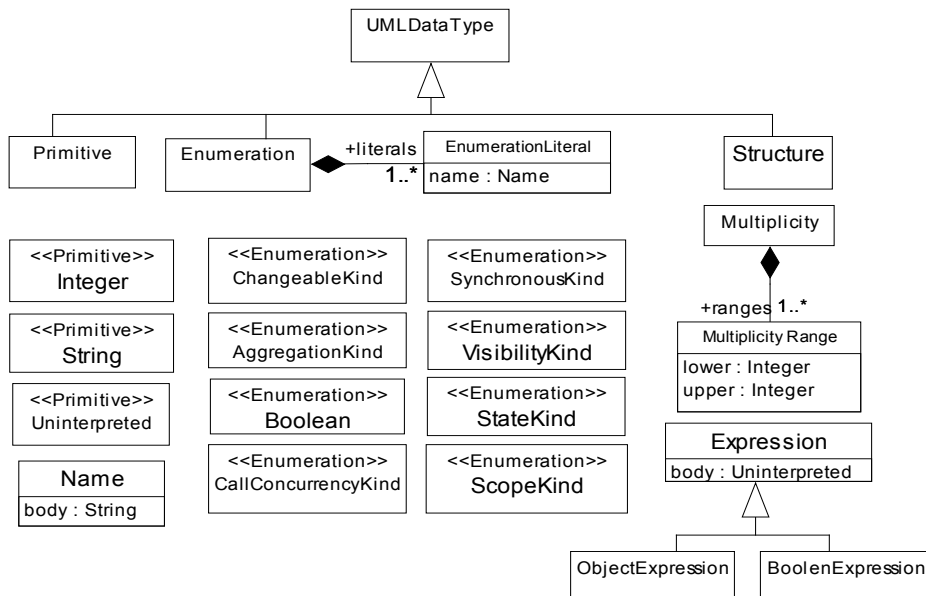


Figura 3.5: Paquete UML\_DataTypes

## 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. A continuación se presenta una descripción informal de los sorts más importantes:

### UMLDataType

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

### AggregationKind

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

### CallConcurrencyKind

Define una enumeración cuyos valores son: *sequential* (indica que las invocaciones a la operación deben ser secuenciales) y *concurrent* (indica que múltiples invocaciones de la operación pueden ocurrir simultáneamente).

### 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.

### 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.

### 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

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

### SynchronousKind

Define una enumeración cuyos valores son *synchronous* y *asynchronous*.

### Name

Define un átomo que es usado para nombrar ModelElements. Cada nombre tiene una representación como String.

### Multiplicity

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

### MultiplicityRange

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

### Expression

Define una expresión que puede evaluarse en un contexto. La función *referencedElements(e)* retorna el conjunto de elementos 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 (expresiones de camino válidas).

### 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.

## ObjectExpression

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

### 3.2.4.2. Paquete Foundation: Core

El paquete Core define las estructuras básicas necesarias para la construcción de modelos de objetos. Las metaclasses 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 metaclasses que son abstractas. Las clases abstractas como ModelElement, GeneralizableElement y Classifier, no son instanciables y se utilizan para organizar el metamodelo, compartir conceptos y estructuras.

Este paquete 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 siguientes dos figuras muestran la sintaxis abstracta, expresada en notación gráfica, de los elementos que componen el paquete Core. La figura 3.6 muestra los principales elementos que se usan para modelar la estructura de un sistema, mientras que la figura 3.7 muestra los elementos que permiten representar relaciones.

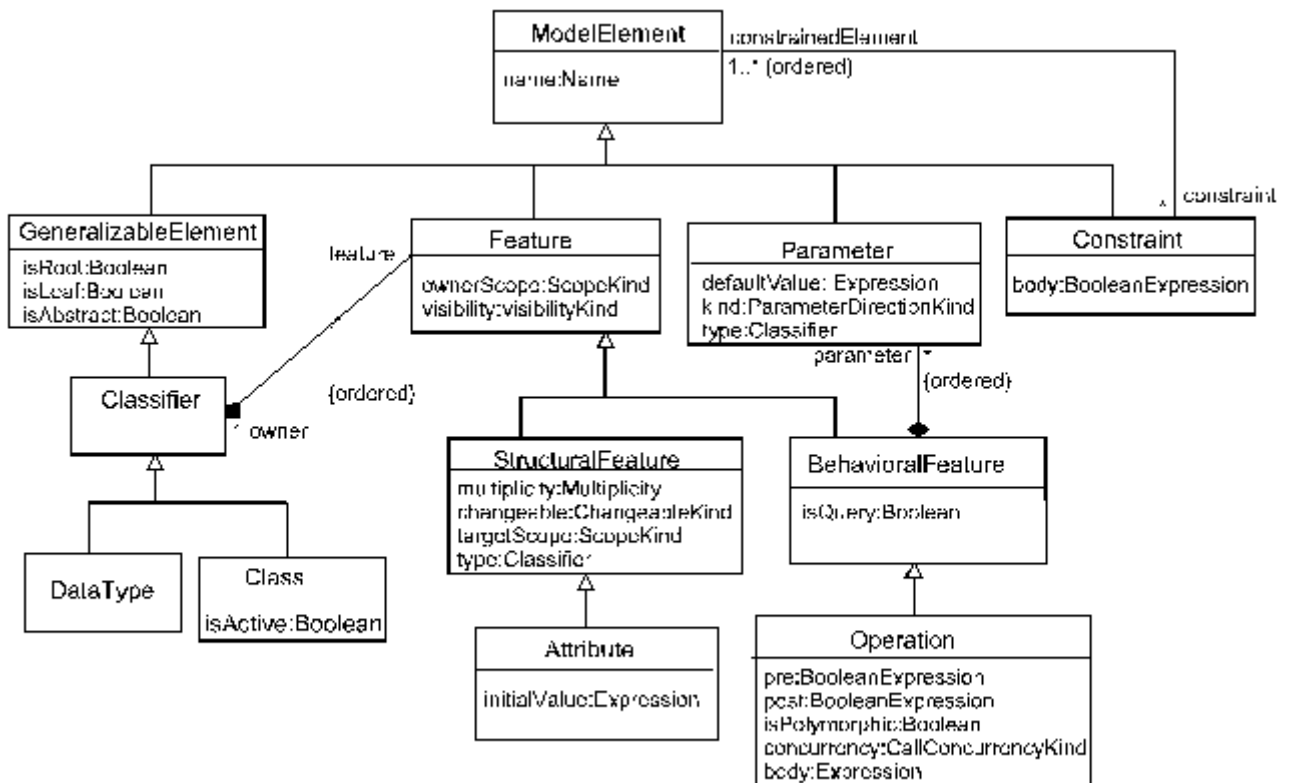


Figura 3.6: Paquete Core. Estructura principal

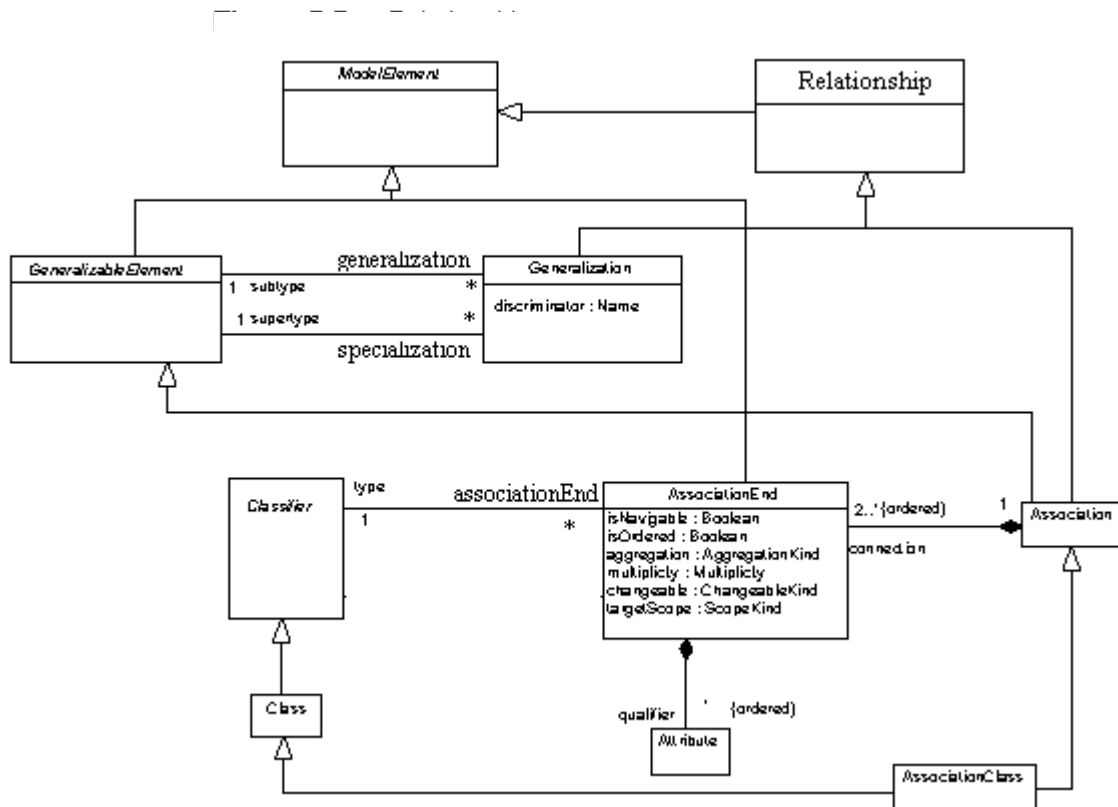


Figura 3.7: Paquete Core. Relaciones

## 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. A continuación se describen algunos sorts de importancia, con el fin de ejemplificar.

### Association

Define una relación semántica entre Classifiers (generalmente clases). Contiene al menos dos AssociationEnds, cada una conectada con un Classifier y representa la conexión entre este y la Association. Cada AssociationEnd define un conjunto de propiedades que caracterizan a la relación.

#### Asociaciones

**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 modelo: 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.

### Attribute

Un atributo 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.

#### Atributos

**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

Se refiere a una propiedad dinámica de los elementos, tal como una operación en un Classifier. BehavioralFeature es una metaclass abstracta.

### Atributos

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 de esta lista.

### Acciones

addParameter : agrega un nuevo parámetro al Behavioral Feature.

deleteParameter : elimina un parámetro del Behavioral Feature.

## Class

Es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones y relaciones. Una clase puede ser abstracta, indicando que no pueden crearse instancias directas de ella.

### Atributos

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 activa.

deactivate : 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 metaclass abstracta.

### Asociaciones

features : 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.

## Feature

Declara una característica estructural o de comportamiento un Classifier o de sus instancias. Feature es una metaclass abstracta.

### Atributos

ownerScope : sus posibles valores son: *instance* (el Feature corresponde a las instancias) y *classifier* (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 deleteFeature).

## GeneralizableElement

Representa elementos generalizables, es decir que pueden participar en relaciones de generalización. En una jerarquía de elementos generalizables cada elemento hereda todas las propiedades definidas en sus ancestros. GeneralizableElement es una metaclass abstracta.

### Atributos

isAbstract : *true* indica que el GeneralizableElement es una declaración incompleta (abstracta) y no es instanciable, *false* indica que es completa (concreta).

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 ancestro inmediato del GeneralizableElement.

specializations: conjunto de Generalization cuyo subtipo es descendiente inmediato del GeneralizableElement.

## 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.

### Atributos

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 subtipo.

subtype : designa al GeneralizableElement que es una versión especializada del supertipo.

## ModelElement

Es una entidad del Modelo. ModelElement es una metaclass abstracta que sirve como base para todas las metaclasses de modelado de UML, las cuales sus subclases directas o indirectas.

### Atributos

name : un *Name* que identifica al ModelElement el cual debe ser único dentro de un contexto.

### Asociaciones

constraints : conjunto de Constraints que afectan al elemento.

package : 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 mismo).

### Acciones

setName : asigna un nuevo nombre al ModelElement.

addConstraint : agrega un Constraint al conjunto constraints.

deleteConstraint : elimina 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

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.

### Atributos

concurrency : especifica el comportamiento de la instancia que recibe otro llamado mientras se ejecuta esta operación.

isPolymorphic : si es *true* puede ser redefinida en las subclases; si es *false* entonces es heredada sin cambios por todas las subclases.

precondition : describe las precondiciones para ejecutar la Operation, es una expresión booleana (BooleanExpression) 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.

### 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.

## StructuralFeature

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

### Atributos

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* (indica que elementos adicionales pueden ser agregados al conjunto, pero ninguno puede ser eliminado).

multiplicity : la cantidad posible de valores que puede almacenar el atributo para cada instancia.

### Asociaciones

type : indica el tipo que deben tener los valores del atributo.

### Acciones

setChangeable : asigna un valor de ChangeableKind al StructuralFeature.

setMultiplicity : asigna un valor de Multiplicity al StructuralFeature.

setTargetScope : asigna un valor de ScopeKind al StructuralFeature.

## Especificación en Lógica Dinámica

Specification of Association	
Sorts	Association
Taxonomy	Association $\leq$ GeneralizableElement Association $\leq$ Relationship
Updatable functions	
	connections: Association $\rightarrow$ Seq of AssociationEnd <i>Additional functions</i> allConnections: Association $\rightarrow$ Seq of AssociationEnd
Updatable predicates	
Actions	
	addConnection: Association x AssociationEnd $\rightarrow$ ModelEvolution deleteConnection: Association x AssociationEnd $\rightarrow$ ModelEvolution
Axioms	$\forall a$ : Association $\forall e, e1, e2$ : AssociationEnd
Static axioms	<p><i>axioms for additional functions:</i></p> <p>[1] allConnections returns the set of all AssociationEnds of the Association itself and all its inherited AssociationEnds.</p> <p>allConnections(a) = connections(a) <math>\cup</math> ( <math>\cup_{s \in \text{supertypes}(a)}</math> allConnections(s) )</p> <p>[2] connectedElements(a) returns a sequence containing all the classifiers connected by the association.</p> <p>connectedElements(a) = map type connections(a)</p> <p>[3] allConnectedElements(a) returns a sequence containing all the classifiers connected by the association itself and by its supertypes.</p> <p>allConnectedElements(a) = map type allConnections(a)</p> <p><i>well-formedness axioms:</i></p> <p>[1] The AssociationEnds must have a unique name within the Association. <math>\forall e1, e2</math>: allConnections(a) name(e1)=name(e2) <math>\rightarrow</math> e1=e2</p> <p>[2] At most one AssociationEnd may be an aggregation or composition. <math>\forall e1, e2</math>: allConnections(a) aggregation(e1) <math>\neq</math> #none <math>\wedge</math> aggregation(e2) <math>\neq</math> #none <math>\rightarrow</math> e1 = e2</p> <p>[3] If an Association has 3 or more AssociationEnds then no AssociationEnd may be an aggregation or composition. size(allConnections(a)) &gt; 2 <math>\rightarrow</math> (<math>\forall e</math>: allConnections(a) aggregation(e) = #none)</p> <p>[4] An Association must have 2 or more AssociationEnds size(allConnections(a)) &gt; 1</p> <p>[5] life dependency exists(a) <math>\leftrightarrow</math> (<math>\forall e</math>: connections(a) exists(e) )</p> <p>[6] symmetry <math>e \in \text{connections}(a) \leftrightarrow \text{association}(e)=a</math></p>
Dynamic axioms	
	<p><math>\langle \text{addConnection}(a, e) \rangle \text{true} \rightarrow e \notin \text{allConnections}(a)</math>  <math>[\text{addConnection}(a, e)] \text{exists}(e) \wedge e = \text{last}(\text{connections}(a)) \wedge \text{association}(e)=a \wedge e \in \text{associationEnds}(\text{type}(e))</math></p>

	$\langle \text{deleteConnection}(a,e) \rangle \text{true} \rightarrow e \in \text{connections}(a)$ $[\text{deleteConnection}(a,e)] (\neg \text{exists}(e) \wedge e \notin \text{connections}(a) ) \wedge e \notin \text{associationEnds}(\text{type}(e))$
End specification of Association	

### Specification of Attribute

Sorts	Attribute
Taxonomy	Attribute $\leq$ StructuralFeature
Updatable functions	
	initialValue: Attribute $\rightarrow$ Expression
Updatable predicates	
Actions	
	setInitialValue: Attribute x Expression $\rightarrow$ ModelEvolution
Axioms	$\forall a$ : Attribute $\forall e$ : Expression
Static axioms	
Dynamic axioms	
	[setInitialValue(a,e)] initialValue(a)=e
End specification of Attribute	

### Specification of BehavioralFeature

Sorts	BehavioralFeature
Taxonomy	BehavioralFeature $\leq$ Feature
Updatable functions	
	parameters: BehavioralFeature $\rightarrow$ Seq of Parameter
Updatable predicates	
	isQuery: BehavioralFeature <i>additional predicates</i> hasSameSignature: BehavioralFeature x BehavioralFeature
Actions	
	addParameter: BehavioralFeature x Parameter $\rightarrow$ ModelEvolution deleteParameter: BehavioralFeature x Parameter $\rightarrow$ ModelEvolution
Axioms	$\forall b$ : BehavioralFeature $\forall p, p1, p2$ : Parameter
Static axioms	
	<i>axioms for additional functions and predicates</i> [1] The additional predicate hasSameSignature checks if two behavioralFeatures have the same signature. hasSameSignature(b,b1) $\leftrightarrow$ (name(b)=name(b1) $\wedge$ areEquivalent(parameters(b),parameters(b1))) areEquivalent( $\emptyset$ , $\emptyset$ )=true areEquivalent(p-ps, $\emptyset$ )=false areEquivalent( $\emptyset$ ,p-ps)=false areEquivalent(p1-ps,p2-ps')=equivalent(p1,p2) $\wedge$ areEquivalent(ps,ps') <i>well-formedness axioms</i> [1] All Parameters should have a unique name. $\forall p1, p2$ : parameters(b) name(p1)=name(p2) $\rightarrow$ p1= p2 [2] The type of the Parameters should be included in the Package of the Classifier. $\forall p$ : parameters(b) type(p) $\in$ allContents(package(owner(b))) [7] life dependency Exists(b) $\leftrightarrow$ ( $\forall p$ :parameters(b) exists(p) )
Dynamic axioms	
	$\langle \text{addParameter}(b,p) \rangle \text{true} \rightarrow p \notin \text{parameters}(b)$ [addParameter (b,p)] Exists(p) $\wedge$ p $\in$ parameters(b) $\langle \text{deleteParameter}(b,p) \rangle \text{true} \rightarrow p \in \text{parameters}(b)$ [deleteParameter(b,p)] $\neg \text{exists}(p) \wedge p \notin \text{parameters}(b)$
End specification of BehavioralFeature	

Specification of Class	
Sorts	Class
Taxonomy	Class $\leq$ Classifier
Updatable functions	
	behavior: Class $\rightarrow$ StateMachine
Updatable predicates	
	isActive: Class
Actions	
	activate: Class $\rightarrow$ ModelEvolution deactivate: Class $\rightarrow$ ModelEvolution
Axioms	$\forall c: \text{Class}$
Static axioms	
	[1] If a Class is concrete, all the Operations in the full descriptor of the Class should have a realizing Method. $\neg \text{isAbstract}(c) \rightarrow (\forall p: \text{allOperations}(c) \text{ body}(p) \neq \text{nullElement})$ [2] symmetry $\text{context}(\text{behavior}(c)) = c$
Dynamic axioms	
	[activate(c)] isActive(c) [deactivate(c)] $\neg \text{isActive}(c)$
End specification of Class	

Specification of Classifier	
Sorts	Classifier
Taxonomy	Classifier $\leq$ GeneralizableElement
Updatable functions	
	features: Classifier $\rightarrow$ Seq of Feature associationEnds: Classifier $\rightarrow$ Set of AssociationEnd <i>Additional functions</i> associations: Classifier $\rightarrow$ Set of Association oppositeAssociationEnds: Classifier $\rightarrow$ Set of AssociationEnd allFeatures : Classifier $\rightarrow$ Set of Feature allOperations : Classifier $\rightarrow$ Set of Operation allAttributes : Classifier $\rightarrow$ Set of Attribute allAssociationEnds: Classifier $\rightarrow$ Set of AssociationEnd allAssociations: Classifier $\rightarrow$ Set of Association allOppositeAssociationEnds: Classifier $\rightarrow$ Set of AssociationEnd
Updatable predicates	
	<i>additional predicates</i> directPartOf: Classifier x Classifier partOf: Classifier x Classifier
Actions	
	addFeature: Classifier x Feature $\rightarrow$ ModelEvolution deleteFeature: Classifier x Feature $\rightarrow$ ModelEvolution
Axioms	$\forall c: \text{Classifier} \quad \forall f, f_1, f_2: \text{Feature} \quad \forall e: \text{AssociationEnd}$
Static axioms	
	<i>axioms for additional functions</i> [1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features. $\text{allFeatures}(c) = \text{features}(c) \cup (\cup_{ci \in \text{supertypes}(c)} \text{allFeatures}(ci))$ [2] The operation allOperations results in a Set containing all Operations of the Classifier itself and all its inherited Operations. $\forall o: \text{Operation} \quad o \in \text{allOperations}(c) \leftrightarrow o \in \text{allFeatures}(c)$ [3] The operation allAttributes results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes. $\forall t: \text{Attribute} \quad t \in \text{allAttributes}(c) \leftrightarrow t \in \text{allFeatures}(c)$ [4] The operation allAssociationEnds results in a Set containing all AssociationEnds of the Classifier itself and all its inherited AssociationEnds. $\text{allAssociationEnds}(c) = \text{associationEnds}(c) \cup (\cup_{ci \in \text{supertypes}(c)} \text{allAssociationEnds}(ci))$ [5] allAssociations returns the associations in which the classifier participates. $\text{allAssociations}(c) = \text{map association allAssociationEnds}(c)$

	<p>[6] oppositeAssociationEnds returns the opposite AssociationEnds of the Classifier itself.  <math>\text{oppositeAssociationEnds}(c) = \{e \in (\bigcup_{a \in \text{associations}(c)} \text{allConnections}(a)) / \text{type}(e) \neq c\}</math></p> <p>[7] allOppositeAssociationEnds returns the opposite AssociationEnds of the Classifier itself and all its inherited opposite AssociationEnds.  <math>\text{allOppositeAssociationEnds}(c) = \text{oppositeAssociationEnds}(c) \cup (\bigcup_{ci \in \text{supertypes}(c)} \text{allOppositeAssociationEnds}(ci))</math></p> <p><i>axioms for additional predicates</i></p> <p>[1] the <i>directPartOf</i> predicate: The set of Association with associationsEnds that are aggregation or composition defines a relation which we will call <i>directPartOf</i>. <i>directPartOf</i>(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 .  <math>\text{directPartOf}(c1,c2) \leftrightarrow (\exists e: \text{AssociationEnd} (e \in \text{oppositeAssociationEnds}(c1) \wedge \text{aggregation}(e) \neq \text{none} \wedge \text{type}(e) = c2))</math></p> <p>[2] the <i>partOf</i> predicate: <i>partOf</i> is the transitive closure of <i>directPartOf</i>.  <math>\text{partOf} = \text{directPartOf}^*</math></p> <p><i>Well-formedness axioms</i></p> <p>[1] No Attributes may have the same name within a Classifier  <math>\forall f,g: \text{attributes}(c) (\text{name}(f) = \text{name}(g) \rightarrow f = g)</math></p> <p>[2] No Operations may have the same signature in a Classifier.  <math>\forall f,g: \text{operations}(c) (\text{hasSameSignature}(f,g) \rightarrow f = g)</math></p> <p>[3] No opposite AssociationEnds may have the same rol-name within a Classifier  <math>\forall f,g: \text{oppositeAssociationEnds}(c) (\text{name}(f) = \text{name}(g) \rightarrow f = g)</math></p> <p>[4] The name of an Attribute cannot be the same as the name of an opposite AssociationEnd.  <math>\forall f: \text{oppositeAssociationEnds}(c) \forall g: \text{allAttributes}(c) \text{name}(f) \neq \text{name}(g)</math></p> <p>[5] 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  <math>(\text{partOf}(c1,c2) \rightarrow \neg \text{isA}(c1,c2)) \wedge ((\text{partOf}(c1,c2) \wedge \text{isA}(c2,c1)) \rightarrow (\exists c3: \text{Classifier}(\text{isA}(c3,c1) \wedge c3 \neq c2 \wedge \neg \text{partOf}(c2,c3))))</math></p> <p>[6] life dependency  <math>\text{exists}(c) \leftrightarrow (\forall f: \text{features}(c) \text{exists}(f))</math></p> <p>[7] symmetry  <math>f \in \text{features}(c) \leftrightarrow \text{owner}(f) = c</math>  <math>e \in \text{associationEnds}(c) \leftrightarrow \text{type}(e) = c</math></p>
Dynamic axioms	
	$\langle \text{addFeature}(c,f) \rangle \text{true} \rightarrow f \notin \text{allFeatures}(c)$ $[\text{addFeature}(c,f)] \text{exists}(f) \wedge f \in \text{features}(c) \wedge \text{owner}(f) = c$  $\langle \text{deleteFeature}(c,f) \rangle \text{true} \rightarrow f \in \text{features}(c)$ $[\text{deleteFeature}(c,f)] \neg \text{exists}(f) \wedge f \notin \text{features}(c)$

End specification of Classifier

### Specification of Element and NullElement

Sorts	Element, NullElement
Taxonomy	
NonUpdatable functions	
	nullElement: $\rightarrow \text{NullElement}$

End specification of Element and NullElement

### Specification of Feature

Sorts	Feature
Taxonomy	Feature $\leq$ ModelElement
Updatable functions	
	owner: Feature $\rightarrow$ Classifier ownerScope: Feature $\rightarrow$ ScopeKind visibility: Feature $\rightarrow$ VisibilityKind
Updatable predicates	
Actions	
	setOwnerScope: Feature x ScopeKind $\rightarrow$ ModelEvolution setVisibility: Feature x VisibilityKind $\rightarrow$ ModelEvolution
Axioms	$\forall f: \text{Feature} \forall c: \text{Classifier} \forall s: \text{ScopeKind} \forall v: \text{VisibilityKind}$
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 $\leq$ ModelElement
Updatable functions	
	generalizations: GeneralizableElement $\rightarrow$ Set of Generalization specializations: GeneralizableElement $\rightarrow$ Set of Generalization <i>additional functions</i> supertypes: GeneralizableElement $\rightarrow$ Set of GeneralizableElement subtypes: GeneralizableElement $\rightarrow$ Set of GeneralizableElement allSupertypes : GeneralizableElement $\rightarrow$ Set of GeneralizableElement allConstraints: GeneralizableElement $\rightarrow$ Set of Constraint
Updatable predicates	
	isAbstract: GeneralizableElement isLeaf: GeneralizableElement isRoot: GeneralizableElement isA : GeneralizableElement x GeneralizableElement
Actions	
Axioms	$\forall c_1, c_2, c: \text{GeneralizableElement}$
Static axioms	
	<i>axioms for additional functions</i> [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) $\cup$ ( $\cup_{ci \in \text{supertypes}(c)} \text{allSupertypes}(ci)$ ) [4] allConstraints returns the constraints of the model element itself and all its inherited constraints allConstraints(c)=constraints(c) $\cup$ ( $\cup_{ci \in \text{supertypes}(c)} \text{allConstraints}(ci)$ ) [5] Definition of isA predicate isA(c,c1) $\leftrightarrow c=c1 \vee c1 \in \text{allSupertypes}(c)$ <i>well-formedness axioms</i> [1] Circular inheritance is not allowed. isA(c1,c2) $\wedge$ isA(c2,c1) $\rightarrow c_2 = c_1$ [2] Multiple inheritance does not lead to name conflict (isA(c1,c2) $\wedge$ isA(c1,c3) $\wedge$ c3 $\neq$ c2) $\rightarrow$ ( $\forall f: \text{Feature} (f \in \text{allFeatures}(c2) \wedge f \in \text{allFeatures}(c3)$ $\rightarrow \exists c: \text{Classifier} (\text{IsA}(c2,c) \wedge \text{IsA}(c3,c) \wedge f \in \text{features}(c))$ ) ) [3] Constraint consistency. isA(c1,c2) $\rightarrow$ consistent(constraints(c1) $\cup$ constraints(c2)) [4] Behavioral consistency. isA(c1,c2) $\rightarrow$ refinement(behavior(c1),behavior(c2))
Dynamic axioms	

End specification of GeneralizableElement

### Specification of Generalization

Sorts	Generalization
Taxonomy	Generalization $\leq$ Relationship
Updatable functions	
	discriminator: Generalization $\rightarrow$ Name supertype: Generalization $\rightarrow$ GeneralizableElement subtype : Generalization $\rightarrow$ GeneralizableElement
Updatable predicates	

Actions	
Axioms	$\forall g$ : Generalization
Static axioms	<p><i>axioms for additional functions</i></p> <p>[1] connectedElements(g) returns a sequence containing the two GeneralizableElements connected by the generalization. connectedElements(g) = {supertype(g), subtype(g)}</p> <p>[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)</p> <p><i>well-formedness axioms</i></p> <p>[1] A root cannot have any Generalizations. ¬isRoot( subtype(g) )</p> <p>[2] No GeneralizableElement which is a leaf can have a subtype ¬isLeaf( supertype(g) )</p> <p>[3] A GeneralizableElement may only be a subclass of GeneralizableElement of the same kind. sameKind(subtype(g), supertype(g) )</p>
Dynamic axioms	

End specification of Generalization

### Specification of ModelElement

Sorts	ModelElement
Taxonomy	ModelElement $\leq$ Element
Updatable functions	<p>name: ModelElement <math>\rightarrow</math> Name constraints: ModelElement <math>\rightarrow</math> Set of Constraint package: ModelElement <math>\rightarrow</math> (Package + NullElement)</p> <p><i>Additional functions</i> packages: ModelElement <math>\rightarrow</math> Set of Package</p>
Updatable predicates	sameKind: ModelElement x ModelElement
Actions	<p>setName: ModelElement x Name <math>\rightarrow</math> ModelEvolution addConstraint: ModelElement x Constraint <math>\rightarrow</math> ModelEvolution deleteConstraint: ModelElement x Constraint <math>\rightarrow</math> ModelEvolution</p>
Axioms	$\forall m$ : ModelElement $\forall c$ : Constraint $\forall p$ : Package
Static axioms	<p><i>axioms for additional functions</i></p> <p>[1] The operation packages results in all the Packages to which a ModelElement belongs. packages = { package(m) } <math>\cup</math> allSurroundingPackages(package(m))</p> <p><i>well-formedness axioms</i></p> <p>[1] Constraint consistency consistent(allConstraints(m))</p> <p>[2] <math>\forall c</math>: allConstraints(m) syntacticCompatible(m,c)</p>
Dynamic axioms	<p>[setName(m,n)] name(m) = n &lt;addConstraint(m,c)&gt;true <math>\rightarrow</math> consistent({ c } <math>\cup</math> allConstraints(m)) [addConstraint(m,c)] exists(c) <math>\wedge</math> c <math>\in</math> constraints(m) <math>\wedge</math> m <math>\in</math> constrainedElements(c) &lt;deleteConstraint(m,c)&gt;true <math>\rightarrow</math> c <math>\in</math> constraints(m) [deleteConstraint(m,c)] ¬exists(c) <math>\wedge</math> c <math>\notin</math> constraints(m)</p>

End specification of ModelElement

### Specification of Operation

Sorts	Operation
Taxonomy	Operation $\leq$ BehavioralFeature
Updatable functions	<p>concurrency: Operation <math>\rightarrow</math> CallConcurrencyKind precondition: Operation <math>\rightarrow</math> BooleanExpression postcondition: Operation <math>\rightarrow</math> BooleanExpression</p>



Updatable predicates	
	isPolymorphic: Operation
Actions	
	setConcurrency: Operation x CallConcurrencyKind → ModelEvolution setPrecondition: Operation x BooleanExpression → ModelEvolution setPostcondition: Operation x BooleanExpression → ModelEvolution
Axioms	$\forall o: \text{Operation} \forall c: \text{CallConcurrencyKind} \forall b: \text{BooleanExpression}$
Static axioms	
	[1] Consistency between preconditions and constraints. consistent( $\{\text{precondition}(o)\} \cup \text{allConstraints}(\text{owner}(o))$ ) [2] Consistency between postconditions and constraints. consistent( $\{\text{postcondition}(o)\} \cup \text{allConstraints}(\text{owner}(o))$ )
Dynamic axioms	
	[setConcurrency(o,c)] concurrency(o)=c [setPrecondition(o,b)] precondition(o)=b [setPostcondition(o,b)] postcondition(o)=b
End specification of Operation	

<b>Specification of StructuralFeature</b>	
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	$\forall e: \text{StructuralFeature} \forall c: \text{ChangeableKind} \forall m: \text{Multiplicity} \forall s: \text{ScopeKind}$
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	

### 3.2.5. Package Behavioral Elements

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

Este trabajo sólo trata el subpaquete State Machines. Los dos restantes no son considerados y su inclusión en la M&D-theory constituye parte de planes de trabajo futuro. Sin embargo, es importante destacar que este hecho no le resta poder expresivo al lenguaje de modelado UML, debido a que toda la información representada mediante estos diagramas puede también ser expresada mediante otros diagramas presentes en la teoría.

#### 3.2.5.1. Package 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 que 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 3.8 se muestra, usando notación gráfica, la sintaxis abstracta de los elementos del paquete State Machines.

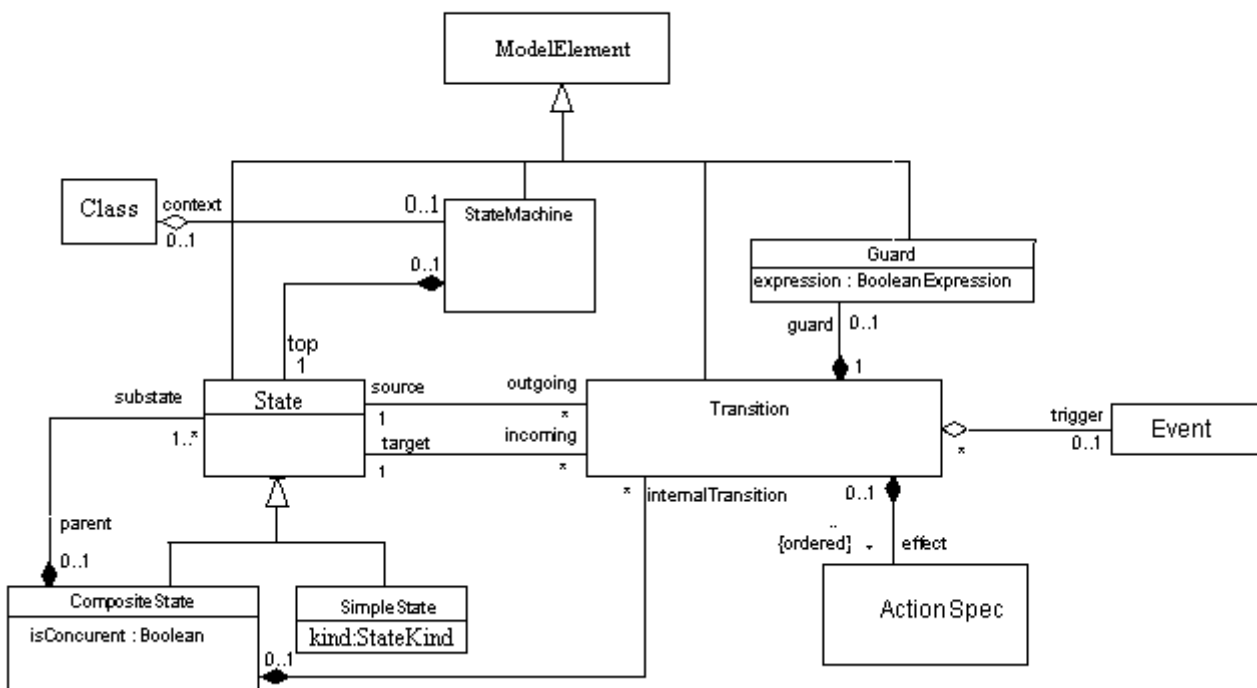


Figura 3.8: Paquete State Machines

### 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. A continuación se muestran sólo algunos para ejemplificar.

#### Event

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

Se consideran 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

Expresión booleana que puede asociarse a una transición para especificar las condiciones bajo las cuales estará habilitada o deshabilitada. La guarda se evalúa cuando se produce un evento, y 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.

## State

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

### Atributos

isRegion : 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 ingresan al estado.

## StateMachine

Una StateMachine está integrada por estados conectados por transiciones. Generalmente, se utiliza para especificar el comportamiento de las instancias de un Classifier. La StateMachine es dueña en forma directa, únicamente, del estado compuesto más externo (top state), los demás estados se encuentran anidados en los respectivos subestados. 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 : es la clase cuyo comportamiento es especificado por la StateMachine. UML permite especificar comportamiento de un o mas Classifiers o BehavioralFeatures, pero este trabajo se limita a representar sólo el de una clase.

top : designa el estado compuesto más externo de la StateMachine. Los demás estados están contenidos, directa o indirectamente, por este estado.

### 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.

cancelBehavior : 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 : guarda 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 de la transición.

## Especificación en Lógica Dinámica

Specification of Event	
Sorts	Event, TimeEvent, ChangeEvent, CreationEvent, CallEvent, DestructionEvent
Taxonomy	Event $\leq$ ModelElement CreationEvent $\leq$ Event CallEvent $\leq$ Event DestructionEvent $\leq$ Event TimeEvent $\leq$ Event ChangeEvent $\leq$ Event
NonUpdatable functions	
	<i>creator functions</i> create: $\rightarrow$ CreationEvent destroy: $\rightarrow$ DestructionEvent call: Operation $\rightarrow$ CallEvent timeOut: $\rightarrow$ TimeEvent changed: $\rightarrow$ ChangeEvent <i>additional functions</i> referencedElements: Event $\rightarrow$ Set of ModelElement
Axioms	
	referencedElements(create) = $\emptyset$ $\wedge$ referencedElements(destroy) = $\emptyset$ referencedElements(changed) = $\emptyset$ $\wedge$ referencedElements(timeOut) = $\emptyset$ referencedElements(call(op)) = {op}
End specification of Event	

Specification of Guard	
Sorts	Guard
Taxonomy	Guard $\leq$ ModelElement
Updatable functions	
	expression: Guard $\rightarrow$ BooleanExpression <i>additional functions</i> referencedElements: Guard $\rightarrow$ Set of ModelElement
Updatable predicates	
	guardCompatible: Class x Guard
Actions	
	setExpression: Guard x BooleanExpression $\rightarrow$ ModelEvolution
Axioms	
	$\forall g$ : Guard $\forall e$ : BooleanExpression
Static axioms	
	<i>axioms for additional functions and predicates</i> referencedElements(g) = referencedElements(expression(g)) guardCompatible(c,g) $\leftrightarrow$ syntactic-compatible(c,expression(g))
Dynamic axioms	
	[setExpression(g,e)] expression(g)=e
End specification of Guard	

Specification of State	
Sorts	State
Taxonomy	State $\leq$ ModelElement
Updatable functions	
	parent : State $\rightarrow$ CompositeState outgoings: State $\rightarrow$ Set of Transition incomings: State $\rightarrow$ Set of Transition
Updatable predicates	
	isRegion: State
Actions	

Axioms	$\forall s: \text{State}$
Static axioms	
	<i>axioms for additional predicates</i> [1] $\text{isRegion}(s) \leftrightarrow \text{parent}(s) \neq \text{nullElement}$
Dynamic axioms	
End specification of State	

<b>Specification of StateMachine</b>	
Sorts	StateMachine
Taxonomy	StateMachine $\leq$ ModelElement
Updatable functions	
	context: StateMachine $\rightarrow$ Class top: StateMachine $\rightarrow$ CompositeState <i>additional functions</i> allStates: StateMachine $\rightarrow$ Set of State allTransitions: StateMachine $\rightarrow$ Set of Transition specifiedOperations: StateMachine $\rightarrow$ Set of Operation referencedElements: StateMachine $\rightarrow$ Set of ModelElement
Updatable predicates	
	refinement: StateMachine x StateMachine syntacticCompatible: Class x StateMachine
Actions	
	setBehavior: Class x StateMachine $\rightarrow$ ModelEvolution cancellBehavior: Class $\rightarrow$ ModelEvolution
Axioms	$\forall h: \text{StateMachine} \forall c: \text{Classifier}$
Static axioms	
	<i>axioms for additional functions</i> [1] allStates returns all the nested states of the state machine. $\text{allStates}(h) = \{\text{top}(h)\} \cup \text{allSubStates}(\text{top}(h))$ [2] allTransitions returns all the transitions contained into the top state and its nested substates. $\text{allTransitions}(h) = \text{allInternalTransitions}(\text{top}(h))$ [3] the predicate syntacticCompatible is true if only features of the class are used within the state machine. $\text{syntacticCompatible}(c, h) \leftrightarrow (\forall t \in \text{allTransitions}(h) \text{ syntacticCompatible}(c, t))$ [4] $\forall \text{op}: \text{Operation} \text{ op} \in \text{specifiedOperations}(h) \leftrightarrow (\exists t \in \text{allTransitions}(h) \text{ trigger}(t) = \text{call}(\text{op}))$ [5] $\text{referencedElements}(h) = \cup_{t \in \text{allTransitions}(h)} \text{referencedElements}(t)$ [6] the predicate refinement(h1, h2) is true if StateMachine h1 is a refinement of StateMachine h2.  <i>Well-formedness axioms</i> [1] A top state cannot have parent $\text{parent}(\text{top}(h)) = \text{nullElement}$ [2] The top state cannot be the source or target of a transition $\text{isEmpty}(\text{outgoings}(\text{top}(h))) \wedge \text{isEmpty}(\text{incomings}(\text{top}(h)))$ [3] The top state must have (one or more) final state. $\exists s: \text{SimpleState } s \in \text{subStates}(\text{top}(h)) \wedge \text{kind}(s) = \# \text{final}$ [4] source and target states must belong to the state machine $\forall t: \text{Transition } t \in \text{allTransitions}(h) \rightarrow \text{source}(t) \in \text{allStates}(h) \wedge \text{target}(t) \in \text{allStates}(h)$ [5] symmetry. $\forall t: \text{Transition } \forall s: \text{State } t \in \text{outgoings}(s) \leftrightarrow \text{source}(t) = s \wedge t \in \text{incomings}(s) \leftrightarrow \text{target}(t) = s$ [6] compatibility between views: only features of its context class can be used within a state machine. $\text{syntacticCompatible}(\text{context}(h), h)$ [7] life dependency $\text{exists}(h) \leftrightarrow \text{exists}(\text{top}(h))$
Dynamic axioms	
	$[\text{setBehavior}(c, h)] (\text{exists}(h) \wedge \text{package}(h) = \text{package}(c) \wedge h \in \text{ownedElements}(\text{package}(c)) \wedge \text{context}(h) = c \wedge \text{behavior}(c) = h )$ $h = \text{behavior}(c) \rightarrow [\text{cancellBehavior}(c)] (\neg \text{exists}(h) \wedge h \in \text{ownedElements}(\text{package}(c)) \wedge \text{behavior}(c) = \text{nullElement} )$
End specification of StateMachine	

Specification of Transition	
Sorts	Transition
Taxonomy	Transition $\leq$ ModelElement
Updatable functions	
	trigger: Transition $\rightarrow$ Event + NullElement guard: Transition $\rightarrow$ Guard effect: Transition $\rightarrow$ Seq of ActionSpec source: Transition $\rightarrow$ State target: Transition $\rightarrow$ State <i>additional functions</i> referencedElements: Transition $\rightarrow$ Set of ModelElements
Updatable predicates	
	syntacticCompatible: Classifier x Transition
Actions	
	setTrigger: Transition x Event $\rightarrow$ ModelEvolution setGuard: Transition x Guard $\rightarrow$ ModelEvolution setEffect: Transition x Seq of ActionSpec $\rightarrow$ ModelEvolution
Axioms	$\forall t$ : Transition $\forall e$ : Event $\forall g$ : Guard $\forall a$ : ActionSpec
Static axioms	
	<i>Axioms for additional functions</i> [1] $\text{referencedElements}(t) = \text{referencedElements}(\text{trigger}(t)) \cup \text{referencedElements}(\text{guard}(t)) \cup (\bigcup_{a \in \text{effect}(t)} \text{referencedElements}(a))$ [2] $\forall c$ : Classifier $\text{syntactic-compatible}(c, t) \leftrightarrow \text{triggerCompatible}(c, \text{trigger}(t)) \wedge \text{guardCompatible}(c, \text{guard}(t)) \wedge (\forall a: \text{effect}(t) \text{ effectCompatible}(c, a))$ [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) $\forall c$ : Class $\forall e$ : Event $\text{triggerCompatible}(c, e) \leftrightarrow e = \text{create} \vee e = \text{destroy} \vee e = \text{timeOut} \vee (\exists op \in \text{allOperations}(c) e = \text{call}(op))$ [3.2] an actionSpec is effectCompatible 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 ) <i>Well-formedness axioms</i> [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. $\text{kind}(\text{source}(t)) = \#initial \wedge \text{isTop}(\text{parent}(\text{source}(t))) \rightarrow \text{trigger}(t) = \text{create}$ $\text{kind}(\text{source}(t)) = \#initial \wedge \neg \text{isTop}(\text{parent}(\text{source}(t))) \rightarrow \text{trigger}(t) = \text{nullElement}$ [2] source and target of the transitions have the same parent. $\text{parent}(\text{source}(t)) = \text{parent}(\text{target}(t))$
Dynamic axioms	
	[setTrigger(t,e)] trigger(t)=e [setGuard(t,g)] guard(t)=g [setEffect(t,s)] effect(t)=s

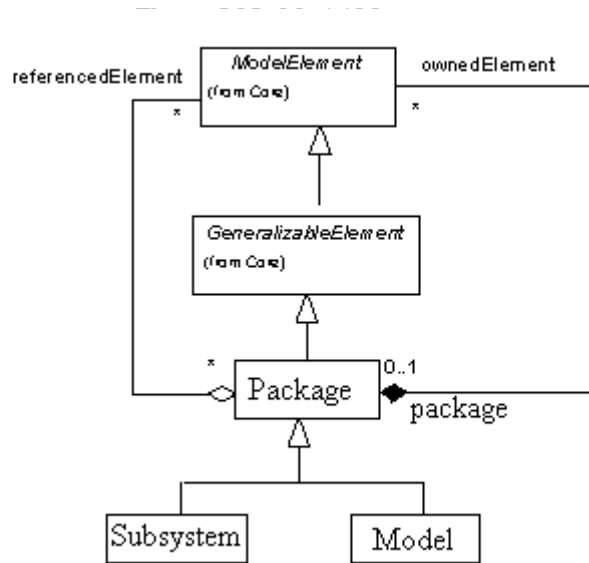
End specification of Transition

### 3.2.6. Package Model Management

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. Como se ha 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 3.9 muestra, usando notación gráfica, la sintaxis abstracta de los elementos del paquete Model Management.



**Figura 3.9: Paquete Model Management**

### Descripción informal

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

### Especificación en Lógica Dinámica

Specification of Package	
Sorts	Package
Taxonomy	Package ≤ GeneralizableElement
Updatable functions	
	referencedElements: Package → Set of ModelElement ownedElements: Package → Set of ModelElement <i>additional functions</i> 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	$\forall p: \text{Package}$
Static axioms	
	<i>axioms for additional functions</i> [1] The operation contents results in a set containing all ModelElements owned or referenced by the Package. $\text{contents}(p) = \text{ownedElements}(p) \cup \text{referencedElements}(p)$ [2] The operation allContents results in a Set containing all ModelElements contained by the Package itself and all its inherited elements. $\text{allContents}(p) = \text{contents}(p) \cup (\cup_{s \in \text{superTypes}(c)} \text{allContents}(s))$ [3] The operation allSurroundingPackages results in a Set containing all surrounding Packages. $\text{allSurroundingPackages}(p) = \{\text{package}(p)\} \cup \text{allSurroundingPackages}(\text{package}(p))$ <i>well-formedness axioms</i> [1] In a Package the Classifier names and the Package names are unique $\forall c_1, c_2: \text{Classifier } c_1 \in \text{contents}(p) \wedge c_2 \in \text{contents}(p) \wedge \text{name}(c_1) = \text{name}(c_2) \rightarrow c_1 = c_2$ $\forall p_1, p_2: \text{Package } p_1 \in \text{contents}(p) \wedge p_2 \in \text{contents}(p) \wedge \text{name}(p_1) = \text{name}(p_2) \rightarrow p_1 = p_2$ [2] In a Package the association names are unique $\forall a_1, a_2: \text{Association } a_1 \in \text{contents}(p) \wedge a_2 \in \text{contents}(p) \wedge \text{name}(a_1) = \text{name}(a_2) \rightarrow a_1 = a_2$ [3] The supertype must be included in the Package of the GeneralizableElement. $\forall g: \text{Generalization } \text{supertype}(g) \in \text{allContents}(\text{package}(\text{subtype}(g)))$ [4] The connected type should be included in the current Package $\forall f: \text{StructuralFeature } \text{type}(f) \in \text{allContents}(\text{package}(\text{owner}(f)))$ [5] The connected Classifiers of the AssociationEnds must be included in the Package of the Association. $\forall a: \text{Association } a \in \text{ownedElements}(p) \rightarrow (\forall r: \text{allConnections}(a) \text{ type}(r) \in \text{allContents}(p))$ [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). $\forall s: \text{StateMachine } \text{context}(s) \in \text{allContents}(\text{package}(s))$ $\forall c: \text{Classifier } \text{behavior}(c) \in \text{allContents}(\text{package}(c))$ [7] life dependency $\text{exists}(p) \leftrightarrow (\forall e \in \text{ownedElements}(p) \text{ exists}(e))$
Dynamic axioms	
	[addReferencedElement(p,e)] $e \in \text{referencedElements}(p)$ [deleteReferencedElement(p,e)] $e \notin \text{referencedElements}(p)$ [addSubpackage(p,e)] $\text{exists}(e) \wedge e \in \text{ownedElement}(p) \wedge \text{package}(e) = p$ [addClassifier(p,e)] $\text{exists}(e) \wedge e \in \text{ownedElement}(p) \wedge \text{package}(e) = p$ [addRelationship(p,e)] $\text{exists}(e) \wedge e \in \text{ownedElement}(p) \wedge \text{package}(e) = p$ [deleteSubpackage(p,e)] $\neg \text{exists}(e) \wedge e \notin \text{ownedElement}(p)$ [deleteClassifier(p,e)] $\neg \text{exists}(e) \wedge e \notin \text{ownedElement}(p)$ [deleteRelationship(p,e)] $\neg \text{exists}(e) \wedge e \notin \text{ownedElement}(p)$

End specification of Package

## 3.3. Nivel de los Datos

### 3.3.1. 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 links y se comunican a través de mensajes que son almacenados en espacios semipúblicos 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.



### 3.3.2. Evolución

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

- **modification:** 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 (LocalInvocations), o 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 una 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 subsorts: Creation, Cancellation y Modification.

#### Sintaxis Abstracta

La figura 3.10 define, mediante notación gráfica, la sintaxis abstracta de los elementos en el nivel de los datos, tales como instancias (Instance), conexiones (Link) y mensajes (Message).

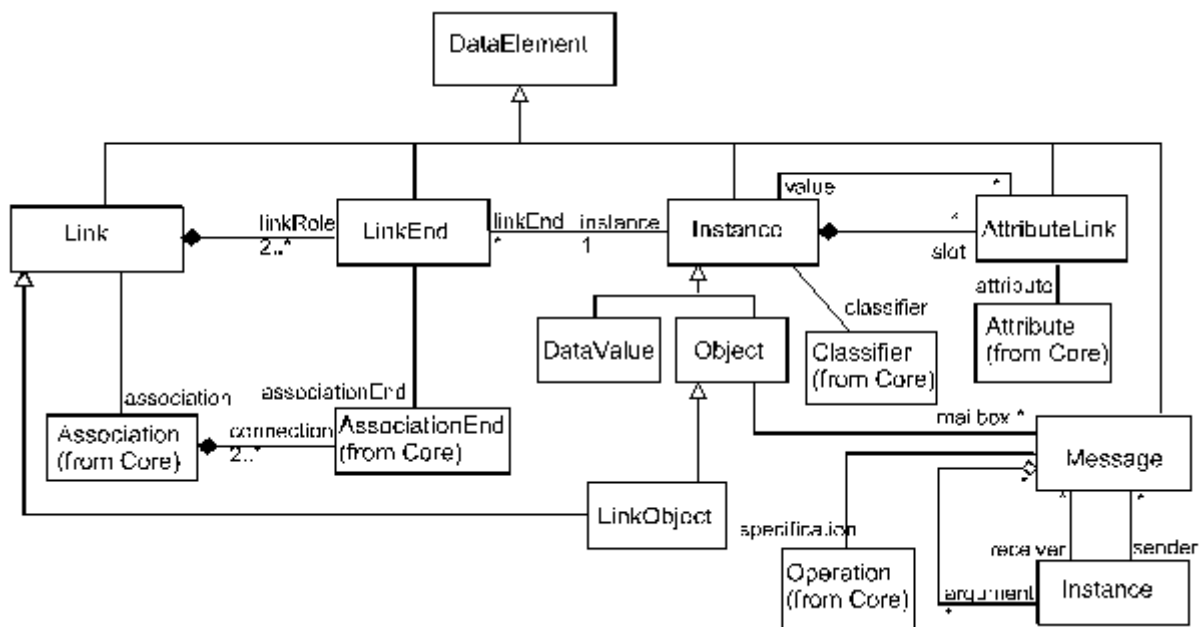


Figura 3.10: DataElements

#### 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 : valor del AttributeLink.

attribute : el atributo representado por el AttributeLink.

##### Instance

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 posee un conjunto de conexiones (links) a otras instancias. El conjunto de slots mas el conjunto de conexiones determinan el estado de la instancia. Instance es una metaclassa abstracta.

### **Asociaciones**

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

### **Link**

Un link es una conexión entre instancias. Cada link es una instancia de una asociación del nivel del modelo, es decir que un link conecta instancias de las clases asociadas o sus subclases. 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, si son navegables.

### **Asociaciones**

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

### **LinkEnd**

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 : instancia conectada al LinkEnd.  
associationEnd : AssociationEnd que declara al LinkEnd.

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

### **Object**

Es una instancia originada por una clase. Los valores de los slots de un objeto pueden cambiar a lo largo de su vida. La clase de un objeto también puede cambiar, esto significa que las propiedades definidas por la clase modificada son dinámicamente agregadas (o eliminadas) al objeto.

Durante su vida un objeto está definido por: su identidad; su clase; su estado interno (valores de sus atributos y conexiones); y su mailbox privado.

### **Asociaciones**

mailBox : 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, [Pons 00]).

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. 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  $(\forall o:\text{Object})$ , 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.

$$\forall m: \text{mailBox}(o) \text{ receiver}(m)=o$$

**[2] No Dangling behavior:**

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

$$\forall m: \text{mailBox}(o) \text{ specificaton}(m) \in \text{operations}(\text{classifier}(o))$$

**[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:

$$[update(o,a,v)] \text{ value}(o,a)=v, \text{ expresando la modificación del valor de un atributo del objeto } o.$$

**[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):

$$\text{enabled}(o,m) \rightarrow m \in \text{mailBox}(o)$$

$$\langle o,m \rangle \text{true} \rightarrow m = \text{first}(\text{mailBox}(o))$$

**[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 de [Pons 00].

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

**[6] Efectos:**

Cuando una transición se dispara, la secuencia de acciones denotada por *effect(t)* debe ser ejecutada (sólo se consideran acciones asincrónicas). 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:

$$[o,m] \{ \text{currentStates}(o) = \text{currentStates}(o) - \{ \text{source}(t) \} \cup \{ \text{target}(t) \} \mid t \in \text{firing} \}$$

$$\wedge \forall t: \text{firing } \forall a: \text{effect}(t) \text{ sent}(a)$$

$$\wedge \text{mailBox}(o) = \text{mailBox}(o) - \text{first}(\text{mailBox}(o))$$

donde *firing* es el conjunto de transiciones abilitadas corrientemente y está definido de la siguiente forma:

$$\text{firing} = \text{firingTransitions}(\text{behavior}(\text{classifier}(o)),m)$$

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:

$$\text{sent}(\text{create}(c)) = \text{Enabled}(\text{newObject}(c))$$

$$\text{sent}(\text{destroy}(\text{exp})) = \text{Enabled}(\text{destroy}(\text{eval}(\text{exp})))$$

$$\text{sent}(\text{update}(\text{exp1}, a, \text{exp2})) = \text{Enabled}(\text{update}(\text{eval}(\text{exp1}), a, \text{eval}(\text{exp2})))$$

$$\text{sent}(\text{call}(\text{op}, \text{args})) = \langle \text{op}, \text{eval}(\text{self}), \text{eval}(\text{head}(\text{args})), \text{eval}(\text{tail}(\text{args})) \rangle \in \text{mailBox}(\text{eval}(\text{head}(\text{args})))$$

Por ejemplo, ejecutar un evento de invocación involucra evaluar las expresiones de objetos para obtener al receptor del mensaje y 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 recibido será eventualmente procesado.

$$\forall m:\text{Message} \forall o:\text{Object} (m \in \text{mailBox}(o) \rightarrow \diamond \langle o.m \rangle \text{true})$$

**[8] Destrucción de objetos.**

Como consecuencia de la acción de destrucción el objeto deja de existir.

$$[\text{destroy}(o)] \neg \text{Exists}(o)$$

**Especificación en Lógica Dinámica**

En esta sección se describe la especificación formal de los elementos en el nivel de los datos. Esta especificación consta de una signatura  $\Sigma_{\text{SYS}} = (S_{\text{SYS}}, \leq, F_{\text{SYS}}, P_{\text{SYS}}, A_{\text{SYS}})$  y una fórmula  $\gamma_{\text{SYS}}$  sobre  $\Sigma_{\text{SYS}}$ . 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  $\gamma_{\text{SYS}}$  es la conjunción de dos conjuntos disjuntos de fórmulas,  $\gamma_S$  y  $\gamma_D$  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.

Specification of AttributeLink	
Sorts	AttributeLink
Taxonomy	AttributeLink $\leq$ DataElement
Updatable functions	
	value: AttributeLink $\rightarrow$ Instance attribute: AttributeLink $\rightarrow$ Attribute
Updatable predicates	
Actions	
Axioms	$\forall a:$ 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 $\leq$ Element
Nonupdatable functions	
Updatable predicates	
Actions	
Axioms	
Static axioms	
Dynamic axioms	
End specification of DataElement	

Specification of Instance	
Sorts	Instance
Taxonomy	Instance $\leq$ DataElement
Updatable functions	
	slots: Instance $\rightarrow$ Set of AttributeLink linkEnds: Instance $\rightarrow$ Set of LinkEnd classifier: Instance $\rightarrow$ Classifier <i>additional functions:</i> value: Instance x Name $\rightarrow$ Set of Instance allLinks: Instance $\rightarrow$ Set of Link allOppositeLinkEnds: Instance $\rightarrow$ Set of LinkEnd parts: Instance $\rightarrow$ Set of Instance allParts: Instance $\rightarrow$ Set of Instance
Updatable predicates	
Actions	
Axioms	$\forall i$ : Instance
Static axioms	
	<p><i>axioms for additional functions</i></p> <p>[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.  <math>value(i,n) = \{ value(l) \mid l \in slots(i) \wedge name(attribute(l))=n \} \cup \{ instance(l) \mid l \in allOppositeLinkEnds(i) \wedge name(associationEnd(l))=n \}</math></p> <p>[2] allLinks returns a set containing all links in which the instance participates.  <math>allLinks(i) = map\ link\ linkEnds(i)</math></p> <p>[3] allOppositeLinkEnds returns a set containing all opposite LinkEnds of the instance.  <math>allOppositeLinkEnds(i) = \{ e \in (\cup_{l \in links(i)} linkRoles(l)) \mid instance(e) \neq i \}</math></p> <p>[4] parts returns a set containing the parts of a composite instance.  <math>parts(i) = \{ p:Instance \mid \exists k:Link \exists l1: linkRoles(k) \exists l2: linkRoles(k) (instance(l1)=i \wedge instance(l2)=p \wedge aggregation(associationEnd(l1))=\#composite) \}</math></p> <p>[5] allParts returns all nested parts of a composite instance.  <math>allParts(i) = parts(i) \cup ( \cup_{p \in parts(i)} allParts(p) )</math></p> <p><i>well-formedness axioms:</i></p> <p>[1] the AttributeLinks matches the declarations in the Classifier.  <math>\forall l: AttributeLink\ l \in slots(i) \leftrightarrow attribute(l) \in allAttributes(classifier(i))</math></p> <p>[2] the links matches the declarations in the Classifier.  <math>\forall l: Link\ l \in allLinks(i) \rightarrow association(l) \in allAssociations(classifier(i))</math></p> <p>[3] An Instance may not belong by composition to more than one composite Instance.  <math>\exists e1,e2: oppositeLinkEnds(i) aggregation(associationEnd(e1))=\#composite \wedge aggregation(associationEnd(e2))=\#composite \rightarrow e1=e2</math></p> <p>[4] Satisfaction of Constraints. Constraints always evaluate true.  <math>\forall c: allConstraints(classifier(i)) (eval(c)[self:=i] = true)</math></p> <p>[5] symmetry  <math>l \in linkEnds(i) \leftrightarrow instance(l)=i</math></p>
End specification of Instance	

Specification of Link	
Sorts	Link
Taxonomy	Link $\leq$ DataElement
Updatable functions	
	association: Link $\rightarrow$ Association linkRoles: Link $\rightarrow$ Seq of LinkEnd <i>additional functions</i> connectedElements: Link $\rightarrow$ Seq of Instance
Updatable predicates	
Actions	
Axioms	$\forall l$ : Link
Static axioms	

<i>axioms for additional functions</i>	
[1]connectedElements(l) returns a sequence containing all the instances connected by the link. connectedElements(l) = map instance linkRoles(l)	
<i>well-formedness axioms</i>	
[1] The set of LinkEnds must match the set of AssociationEnds of the Association. $\forall e: AssociationEnd \ e \in \text{map associationEnds linkRoles}(l) \leftrightarrow e \in \text{allConnections}(\text{association}(l))$	
[2] There are not two Links of the same Association which connects the same set of Instances in the same way. $\forall l1,l2: Link \ \text{association}(l1)=\text{association}(l2) \wedge \text{connectedElements}(l1)=\text{connectedElements}(l2) \rightarrow l1=l2$	
Dynamic axioms	

End specification of Link

### Specification of LinkEnd

Sorts	Object
Taxonomy	LinkEnd $\leq$ DataElement
Nonupdatable functions	
Updatable functions	
	instance: LinkEnd $\rightarrow$ Instance associationEnd: LinkEnd $\rightarrow$ AssociationEnd
Updatable predicates	
Actions	
Axioms	$\forall l: LinkEnd$
Static axioms	
	[1] The type of the Instance must match the type of the AssociationEnd $\text{type}(\text{associationEnd}(l)) = \text{classifier}(\text{instance}(l))$
Dynamic axioms	

End specification of LinkEnd

### Specification of Message

Sorts	Message
Taxonomy	Message $\leq$ DataElement
Nonupdatable functions	
	$\langle \rangle$ : Operation x Instance x Instance x Seq of Instance $\rightarrow$ Message specification: Message $\rightarrow$ Operation. sender: Message $\rightarrow$ Instance receiver: Message $\rightarrow$ Instance arguments: Message $\rightarrow$ Seq of Instance
Updatable functions	
NonUpdatable predicates	
	IsLocal: Message
Actions	
Axioms	$\forall m: Message$
Static axioms	
	[1] specification $\langle op,s,r,p \rangle = op$ [2] sender $\langle op,s,r,p \rangle = s$ [3] receiver $\langle op,s,r,p \rangle = r$ [4] arguments $\langle op,s,r,p \rangle = p$ <i>well-formedness axioms</i> [1] the receiver understands the message. specification(m) $\in$ 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,

	$match(<>, <>) = true$ $match(a-args, p-pars) = classifier(a) = type(p) \wedge match(args, pars)$
Dynamic axioms	
End specification of Message	

<b>Specification of Object</b>	
Sorts	Object
Taxonomy	$Object \leq Instance$
Nonupdatable functions	
Updatable functions	$mailBox: Object \rightarrow Seq\ of\ Message$ $currentStates: Object \rightarrow Set\ of\ State$
Updatable predicates	
Actions	$newObject: Class\ x\ Object \rightarrow Creation$ $update: Object\ x\ Name\ x\ Instance \rightarrow Modification$ $-.: Object\ x\ Message \rightarrow Modification$ $destroy: Object \rightarrow Cancellation$
Axioms	$\forall o: Object,$
Static axioms	
	<i>well-formedness axioms</i> [1] $\forall m: mailBox(o)\ receiver(m) = o$ [2] $\forall m: mailBox(o)\ specificaton(m) \in operations(classifier(o))$
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 $\langle o.m \rangle true \rightarrow m = first(mailBox(o))$ $enabled(o.m) \rightarrow m \in mailBox(o)$ [5] call actions: reception of a message m $\langle o.m \rangle true \rightarrow \exists t: Transition\ t \in firingTransitions(behavior(classifier(o)), m)$ [6] effect of call actions. $[o.m](currentStates(o) = currentStates(o) - \{source(t) \mid t \in firing\} \cup \{target(t) \mid t \in firing\})$ $\wedge \forall t \in firing\ \forall a: effect(t)\ sent(a) \wedge mailBox(o) = mailBox(o) - first(mailBox(o))$ where, $firing = firingTransitions(behavior(classifier(o)), m)$ [7] fairness conditions $\forall m: Message\ \forall o: Object\ (m \in mailBox(o) \rightarrow \diamond \langle o.m \rangle true)$ [8] destruction of objects $[destroy(o)] \neg exists(o)$ [9] Behavioral correctness: the behavior of operations (defined by the state machine) satisfies the pre and post conditions of the corresponding specifications. $\forall \langle op, s, r, p \rangle: Message\ classifier(r) = owner(op) \rightarrow$ $(eval(precondition(op)[self/r, parameters/p]) = true$ $\rightarrow [r. \langle op, s, r, p \rangle] eval(postcondition(op)[self/r, parameters/p]) = true)$
End specification of Object	

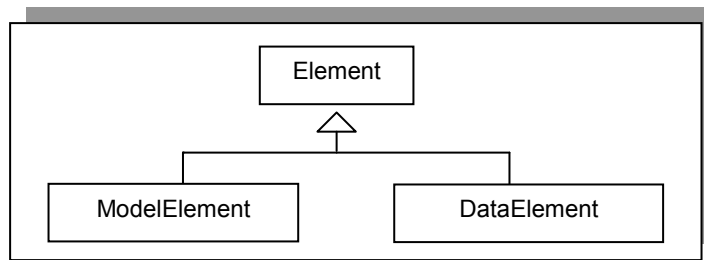
### 3.4. Integración de ambos niveles: La M&D-theory

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:

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

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

- La signatura  $\Sigma_{M\&D}$  incluye a la signatura  $\Sigma_{UML}$ . Esto significa 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 metaclasses o metasorts.
- La signatura  $\Sigma_{M\&D}$  incluye a la signatura  $\Sigma_{SYS}$ . Esto significa 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  $\leq$ . 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 figura 3.11 ilustra esta jerarquía de sorts:



**Figura 3.11: Jerarquía de Sorts**

- Hay un símbolo de predicado exists:  $Element \rightarrow Boolean$ . La extensión de un sort (conjunto de todas las 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:

$$(\forall i: Instance) \neg exists(i).$$

- El predicado enabled:  $Action \rightarrow Boolean$  define el conjunto de acciones 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 (descrita al inicio de este capítulo). Las signaturas de estos símbolos son:

$$newObject: Classifier \times Object \rightarrow Creation ; newLink: Association \times Link \rightarrow Creation.$$

- 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 conexión (referenciada por k) correspondiente a la asociación denotada por el término a. Las funciones polimórficas:

$instances: Classifier \rightarrow Set\ of\ Instance ; instances: Association \rightarrow Set\ of\ Link$   
representan los conjuntos de instancias (o links) creados a partir de un Classifier (o Association). Están definidos de la siguiente forma:

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

$$\forall a: Association \forall i: Link (i \in instances(a) \leftrightarrow association(i)=a)$$

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:  $\phi_{M\&D} = \phi_{UML} \wedge \gamma_{SYS} \wedge \phi_{JOINT}$ . En principio,  $\phi_{UML}$  es la fórmula sobre  $\Sigma_{UML}$  que define las características de los elementos de modelado y se obtiene mediante la conjunción de todos los axiomas del nivel de los modelos. Luego,  $\gamma_{SYS}$  es la fórmula construida sobre  $\Sigma_{sys}$  que describe las características de los elementos modelados y se obtiene mediante la conjunción de todos los axiomas del nivel de los datos). Finalmente,  $\phi_{JOINT}$  es una fórmula construida utilizando el

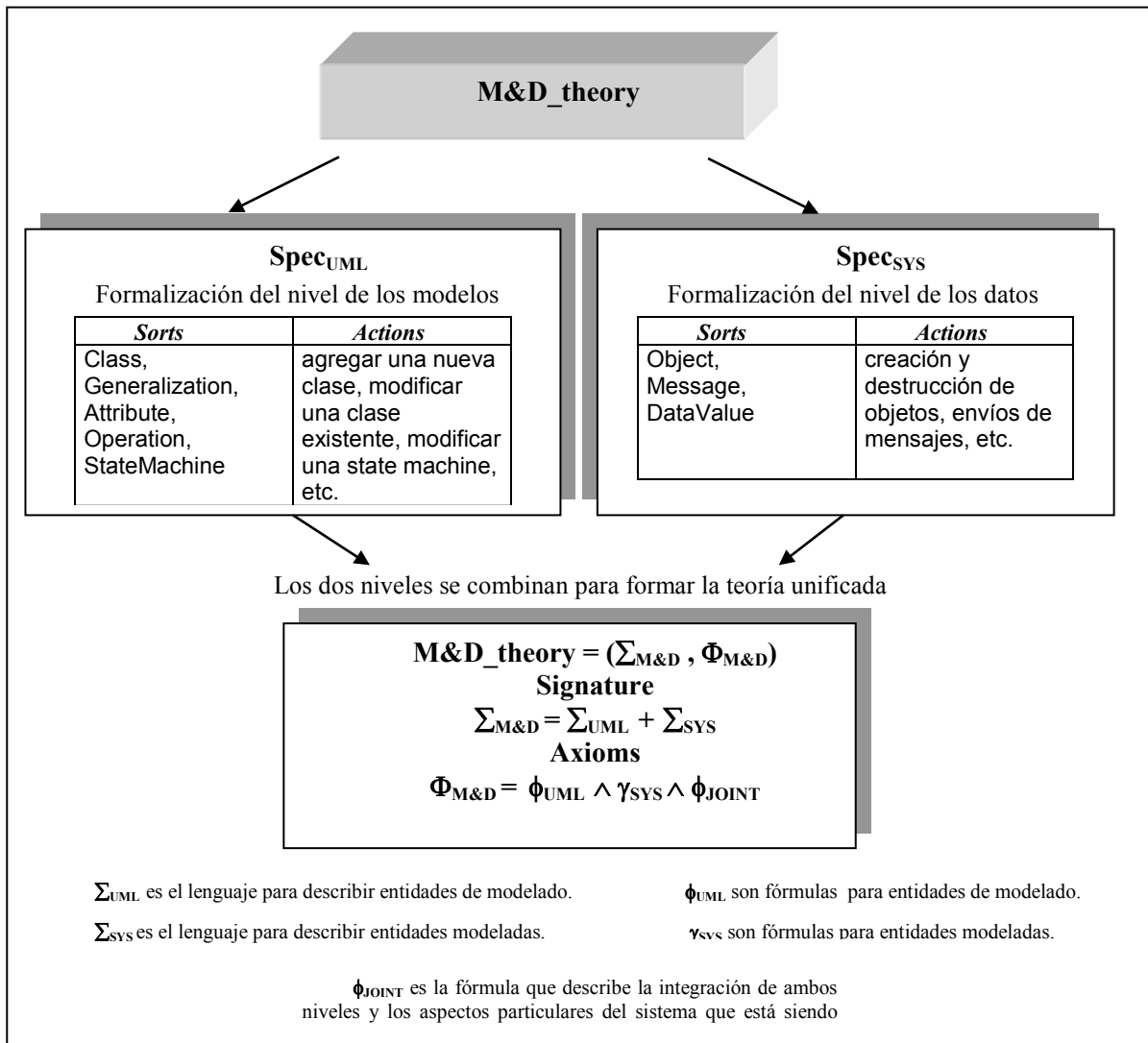


lenguaje integrado *M&D*, y por lo tanto, puede expresar propiedades de los modelos, propiedades de los datos y propiedades relacionando ambos niveles. La fórmula  $\phi_{\text{JOINT}}$  se obtiene uniendo dos grupos de axiomas:

$\phi_{\text{GENERAL}}$  : describe características generales a todos los sistemas. Un ejemplo de este tipo de fórmulas son los axiomas de buena formación.

$\phi_{\text{SPECIFIC}}$  : describe características específicas de cada sistema. Esta fórmula es la conjunción de: axiomas de instanciación  $\phi_{\text{INST}}$  y axiomas de completación  $\phi_{\text{COMP}}$ .

La figura 3.12 ilustra la estructura de la *M&D\_theory*.



**Figura 3.12: M&D-Theory**

### 3.5. 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 se describen el dominio semántico y las correspondientes reglas de interpretación para las construcciones de UML.

### 3.5.1. El dominio semántico

El dominio semántico donde las construcciones de UML serán interpretadas está formado por sistemas de transición de la forma  $U=(S^U, w_0, m_U)$ . Un sistema de transición es un conjunto de mundos posibles con una relación de transición entre ellos, tal como fue descrito en el capítulo 3 de [Pons 00]. 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  $\Sigma$ . Sea  $U=(A, m_U)$  una  $\Sigma_N$ -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:

$$U=(S^U, w_0, m_U)$$

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  $f: s_1, \dots, s_n \rightarrow s$  entonces  $w(f): U_{s_1}, \dots, U_{s_n} \rightarrow U_s$ .
  - si  $p \in P_N$  entonces  $w(p) = p^U$  (la interpretación fija dada por el álgebra  $U$ ).
  - si  $p \in P_U$  y  $p: s_1, \dots, s_n$  entonces  $w(p): U_{s_1}, \dots, U_{s_n}$
  - 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  $\alpha$  una relación binaria llamada la relación entrada/salida de  $\alpha$ :
 
$$m_U(\alpha) \subseteq S^U \times S^U$$

Notar que el dominio de las interpretaciones es un álgebra heterogénea, es decir una  $\Sigma_N$ -álgebra, cuyos elementos incluyen tanto datos (por ejemplo objetos) como meta-datos (por ejemplo clases). La figura 3.13 ilustra una estructura de mundos posibles, donde se observa la dicotomía datos vs. meta-datos (sobre fondo gris y fondo blanco respectivamente).

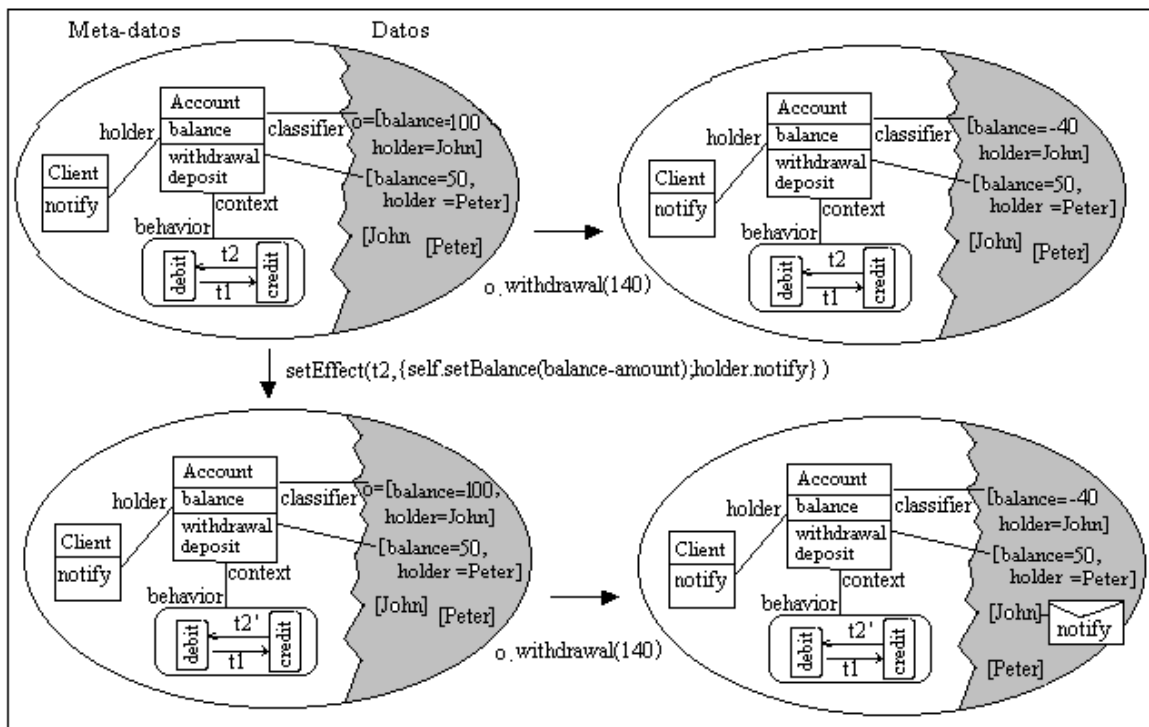


Figura 3.13: Evolución en ambas direcciones.

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 3.13 muestra un ejemplo de evolución en ambas direcciones. Es importante notar que como consecuencia de la evolución de la especificación (es decir la modificación del efecto de la transición  $t_2$ , al agregarle una nueva acción: enviar el mensaje notify al holder) el comportamiento del objeto  $o$  ha sufrido una modificación colateral.

### 3.6. Función de interpretación semántica

En las secciones anteriores se definió el dominio semántico donde se interpretará al lenguaje de modelado UML. El siguiente paso 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 denominada **Sem**.

**Sem**: ConstruccionesUML  $\rightarrow$  DominioSemántico

Es importante destacar que la interpretación semántica de UML definida 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**  $\xrightarrow{\text{translation}}$  **M&D-theory**  $\xrightarrow{\text{semantics}}$  **Dominio-semántico**

De hecho, 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 de [Pons 00]. 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  $\leq r$  maximal del conjunto de modelos minimales de spec). Es decir,  $\text{semantics}(\text{spec}) = \{M \mid M \equiv \text{min-max}(\text{spec})\}$

Para obtener la interpretación semántica de UML sólo resta definir la función *translation*. Esta función 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, es decir una M&D-theory cuyos únicos axiomas son  $\phi_{\text{UML}}$ ,  $\phi_{\text{SYS}}$  y  $\phi_{\text{GENERAL}}$  (no contiene axiomas específicos  $\phi_{\text{SPECIFIC}}$ ). Las reglas van enriqueciendo progresivamente esta teoría, agregando dos clases de axiomas específicos:

- de instanciación  $\phi_{\text{INST}}$ : describen los elementos de modelado usados en el modelo.
- de completación  $\phi_{\text{COMP}}$ : 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.

Se describirá la función *translation* mediante un ejemplo. La figura 3.14 muestra el modelo UML de un sistema bancario mientras que la figura 3.15 contiene la teoría asociada al mismo.

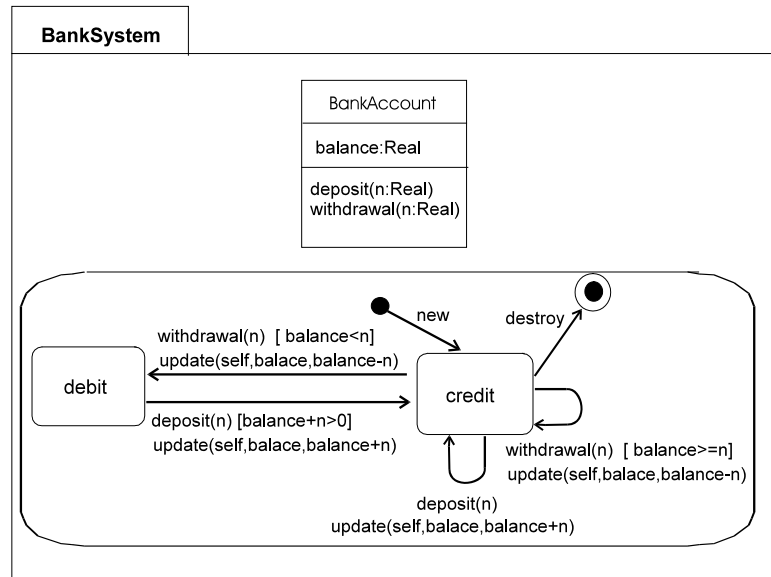


Figura 3.14: Modelo UML de un Sistema Bancario

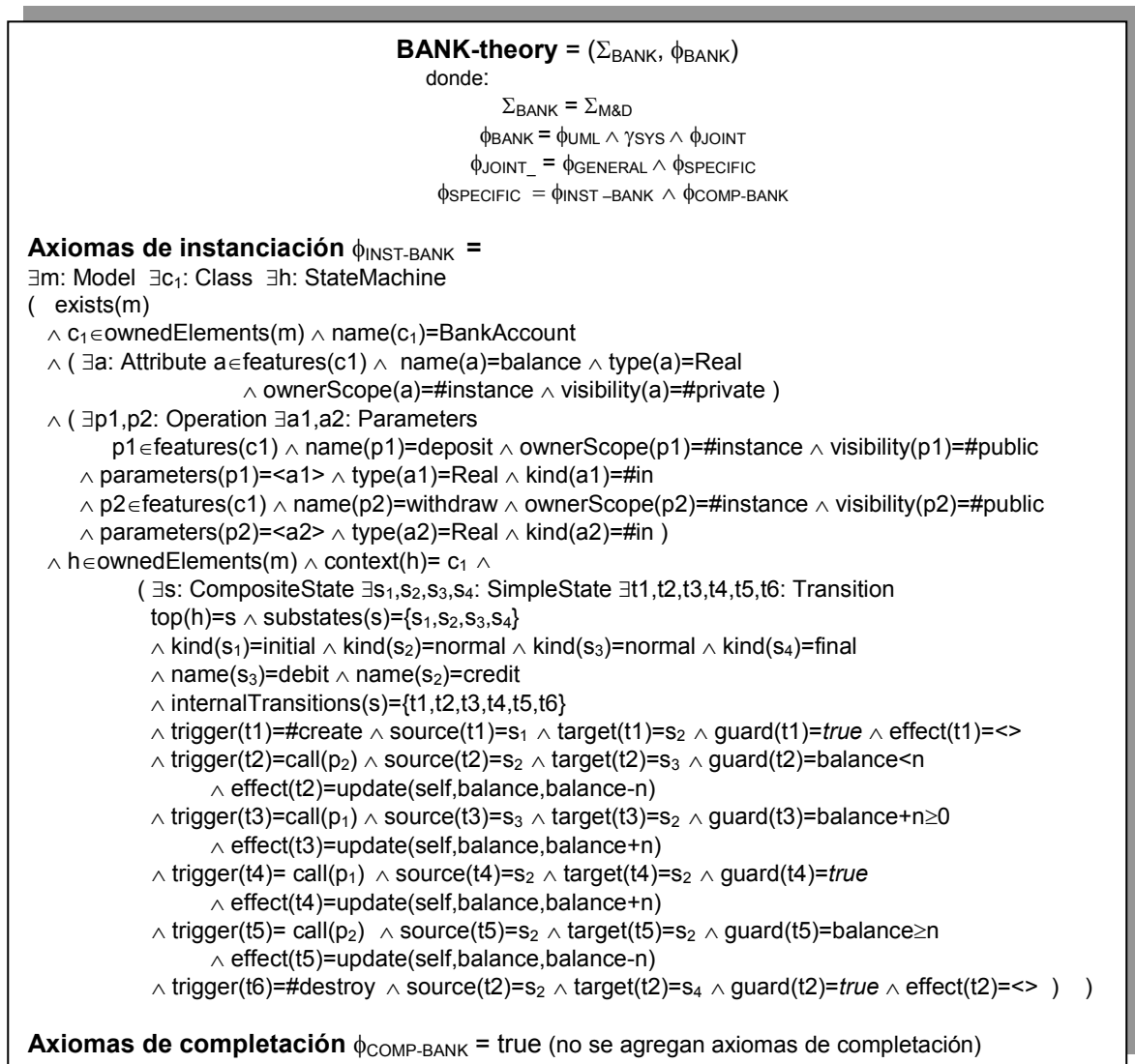


Figura 3.15: Teoría asociada al modelo UML del Sistema Bancario

Por claridad en los ejemplos se utiliza la notación nombreDeAtributo en lugar del término `value(self,nombreDeAtributo)`. También se utiliza la notación `obj.nombreDeAtributo=nuevoValor`, en lugar del término `update(obj,nombreDeAtributo,nuevoValor)`.

### 3.7. Conclusiones

**Ventajas de la integración:** 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.

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	<p>spec no debe contener dos atributos con el mismo nombre dentro de la misma clase:</p> $\forall c: \text{Classifier} \forall a1, a2: \text{Attribute}$ $a1 \in \text{attributes}(c) \wedge a2 \in \text{attributes}(c)$ $\wedge \text{name}(a1) = \text{name}(a2) \rightarrow a1 = a2$	<p>Los valores de los atributos de los objetos de sys deben corresponderse con las declaraciones en las respectivas clases:</p> $\forall a: \text{AttributeLink} \forall i: \text{Instance}$ $a \in \text{slots}(i) \rightarrow$ $\text{attribute}(a) \in \text{allAttributes}(\text{classifier}(i))$ $\wedge \text{isA}(\text{classifier}(\text{value}(a)), \text{type}(\text{attribute}(a)))$
Aspectos Dinámicos	<p>Refinamiento del modelo spec por el agregado de una nueva clase:</p> $\text{addAttribute} : \text{Class} \times \text{Attribute} \rightarrow \text{Act}$ $\forall c: \text{Classifier} \forall a: \text{Attribute}$ $[\text{addAttribute}(c, a)] a \in \text{attributes}(c)$	<p>Los objetos de sys reaccionan al recibir mensajes:</p> $\_ \_ : \text{Object} \times \text{Message} \rightarrow \text{Act}$ $\forall o: \text{Object} \forall m: \text{Message}$ $\langle o, m \rangle \text{true} \rightarrow \exists t: \text{Transition}$ $t \in \text{firingTransitions}(\text{behavior}(\text{classifier}(o), o, m))$

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.



# 4. Rational Rose

## 4.1. Introducción

Rational Rose es una herramienta CASE que soporta el modelado visual del software orientado a objetos. Fue desarrollada por la empresa Rational Software Corporation [Rational], la cual agrupa a los creadores de UML. Rational Rose permite crear, analizar, diseñar, visualizar, modificar y manipular los componentes de un modelo. Provee soporte para dos elementos esenciales en la ingeniería de software moderna: desarrollo basado en componentes y desarrollo iterativo controlado. Dichos conceptos son independientes, pero su utilización combinada es natural y beneficiosa. A esto se le suma la posibilidad de trabajo en grupos, la generación de código y la ingeniería inversa.

La arquitectura de diagramas y modelos de Rational Rose facilita el uso de notaciones para el modelado orientado a objetos ampliamente conocidas, tales como UML, COM (Component Object Modeling), OMT, y Booch. Aunque actualmente UML no es soportado completamente por esta herramienta, gran parte de los elementos de modelado pueden ser representados, pero no con todas las propiedades definidas por el lenguaje.

Rational Rose provee la notación necesaria para especificar y documentar la arquitectura de un sistema de software. La arquitectura lógica es capturada en diagramas de clases que representan las abstracciones fundamentales del sistema en desarrollo. La arquitectura de los componentes es capturada en diagramas de componentes, los cuales se focalizan en la organización real de los módulos de software dentro del ambiente de desarrollo. La arquitectura de despliegue es capturada en diagramas de despliegue, los cuales presentan la configuración del sistema en tiempo de ejecución como nodos de procesamiento.

Con esta herramienta es posible representar gráficamente el comportamiento del sistema a través diagramas de casos de uso. Como alternativa a los casos de uso provee diagramas de colaboración. Los diagramas de estados proporcionan técnicas de análisis adicionales para las clases con un comportamiento dinámico significativo.

Rational Rose/C++ Demo Version 4.0.3 es la versión utilizada para este trabajo. Es una versión reducida (con limitaciones de uso) de la herramienta Rational Rose 4.0, con el objetivo de ser difundida gratuitamente en las universidades.

## 4.2. Vistas de un modelo en Rational Rose

El método para el análisis y diseño OO en *Rose* recomienda el uso de vistas estáticas y dinámicas de un modelo lógico y físico del sistema en desarrollo. La utilización de la notación de la herramienta permite crear y refinar estas vistas dentro de un modelo completo del dominio del problema y el sistema de software. El modelo completo contiene clases, paquetes lógicos, objetos, operaciones, paquetes de componentes, módulos, procesadores, dispositivos y las relaciones entre ellos. Cada uno de estos componentes del modelo posee propiedades que lo caracterizan e identifican. Rose proporciona íconos gráficos para representar cada componente y relación del modelo.

Un modelo contiene además diagramas y especificaciones, los cuales proveen un medio de visualización y manipulación de los componentes del modelo y sus propiedades. Puesto que los diagramas se utilizan para ilustrar las múltiples vistas de un modelo, cada ítem que representa un componente del modelo puede estar presente en varios, uno ó ninguno de dichos diagramas.

Un proyecto de software en Rational Rose está organizado en cuatro vistas:

- Vista de casos de uso
- Vista lógica
- Vista de componentes
- Vista de despliegue

Cada vista contiene, por defecto, un diagrama principal, pero puede ser extendida con diagramas y elementos adicionales.

Los tipos de diagramas que soporta Rational Rose 4.0 son:

- Diagrama de casos de uso
- Diagrama de clases
- Diagrama de estados
- Diagrama de colaboración
- Diagrama de secuencia
- Diagrama de componentes
- Diagrama de despliegue

## 4.3. Ambiente de la aplicación

En las siguientes secciones serán explicados los conceptos necesarios para la especificación en Rational Rose de elementos de modelado que están en el alcance de este trabajo.

### 4.3.1. Interfaz gráfica

La interfaz gráfica de trabajo que Rational Rose presenta al usuario está compuesta por seis ventanas, cuyas denominaciones son: Application, Browser, Documentation, Diagram, Toolbox y Specification. A través de estas ventanas, el usuario será capaz de mostrar, crear, modificar, manipular y documentar los elementos en un modelo. La figura 4.1 muestra la ventana principal de la herramienta.

#### Application Window

Es la ventana principal de la herramienta. Contiene una barra de título, una barra de menús, una barra de herramientas, y un área de trabajo que puede contener las ventanas Toolbox, Browser, documentation, diagram y specification.

#### Browser

El Browser es una alternativa a los menús y barras de herramientas, para visualizar, navegar y manipular los items del modelo fácilmente. El Browser permite:

Vista jerárquica de varios items en el modelo.

Utilizar **drag&drop** para modificar características del modelo.

Actualización automática de los items del modelo para reflejar cambios en el Browser.

#### Diagram Window

Permite crear y modificar las vistas gráficas del modelo. Cada ícono en un diagrama representa un elemento en el modelo. Distintas propiedades de un elemento pueden mostrarse con diferentes íconos en distintas vistas del modelo. Una vez que un ícono se visualiza en el diagrama puede ser seleccionado, desplazado, redimensionado, eliminado del diagrama o eliminado del modelo.

Los diagramas son contenidos por los componentes del modelo que representan: un paquete lógico contiene un diagrama de clases, creado por defecto, llamado "Main", y los diagramas de clases y de interacciones creados por el usuario; un paquete de componentes contiene diagramas de componentes; una clase contiene su diagrama de estados; un modelo contiene su diagrama de despliegue.



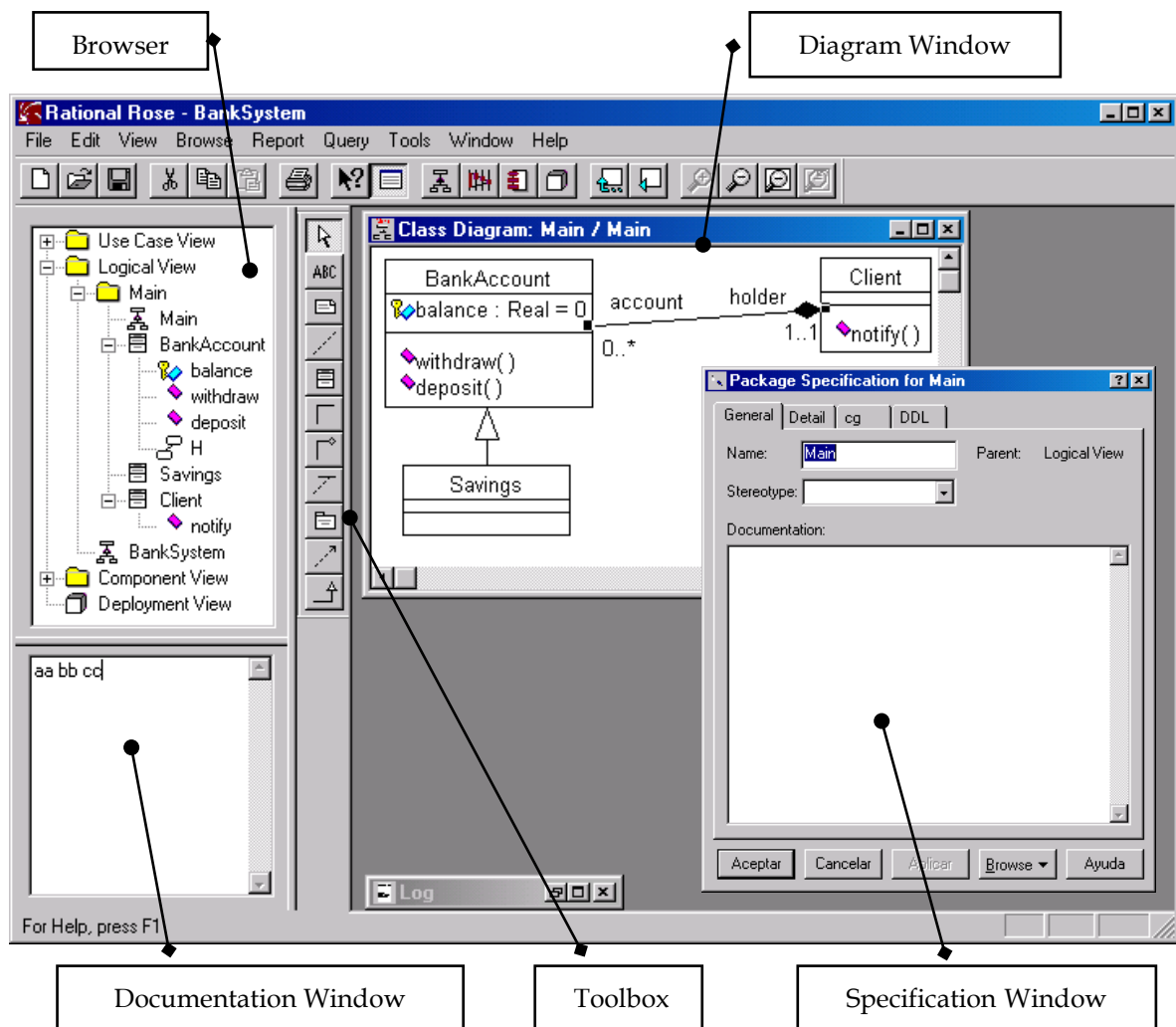


Figura 4.1: Interfaz gráfica principal

Un nuevo diagrama se crea desde el menú **Browse>xxx diagram**, donde **xxx** es el tipo de diagrama.

Una alternativa es, desde el Browser, seleccionar el componente del modelo que contendrá al diagrama, y hacer click en la opción **new>xxx diagram** del menú contextual.

### **Toolbox**

Consiste de un grupo de botones apropiados para el tipo de diagrama activo. Por ejemplo, si la ventana activa es un diagrama de clases, el Toolbox presenta botones para crear clases, asociaciones, etc. Para crear un elemento del modelo desde el Toolbox se debe hacer click sobre el botón adecuado y luego hacer click en el diagrama, donde quedará ubicado el nuevo ícono representando al nuevo elemento.

### **Documentation Window**

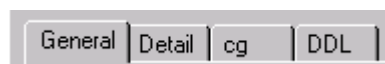
Se utiliza para describir textualmente a los elementos del modelo. La descripción puede incluir información sobre el rol, las restricciones, propósito y comportamiento esencial de cada elemento. Para visualizar la documentación de un elemento, basta con seleccionar el ítem correspondiente.

### **Specification Window**

Permite mostrar y modificar las propiedades y relaciones de un elemento del modelo. La información en una especificación es presentada textualmente y parte de ella puede ser mostrada en el ícono que representa al elemento en un diagrama. Se accede a la especificación de cada elemento a través del menú contextual de su ítem en el Browser **-Specification-**, ó desde el menú contextual de su ícono en un diagrama **-Specification>...**.

En la Specification Window las propiedades de un elemento se dividen en grupos relacionados, a los cuales se accede por medio de tabs. Hay cuatro tabs presentes en la especificación de la gran mayoría de los elementos de modelado: General Tab, Detail Tab, cg Tab y DDL Tab.

El primer Tab, presente en todas las especificaciones, es titulado *General* y contiene información sobre el nombre, la documentación y el dueño de cada elemento en el modelo. Detail Tab contiene información sobre propiedades más específicas de cada elemento del modelo. Los dos últimos, cg Tab y DDL Tab, contienen propiedades útiles para la generación de código y no se tratan en este trabajo. En general, la parte superior de una Specification Window se presenta con, por lo menos, este grupo de tabs:



## **4.4. Especificación de un modelo en Rational Rose**

En esta sección se explica como especificar un modelo en esta herramienta. Se tratan solamente los elementos de modelado que están dentro del alcance de esta tesis y los tipos de diagramas donde pueden ser representados.

Con la creación de un nuevo modelo en la herramienta se inicializan las cuatro vistas del modelo con sus respectivos diagramas principales. En el caso de la vista lógica, contenida en el paquete con nombre “Logical View”, el diagrama de clases “Main” es el que contendrá los íconos de los paquetes de clases que representan la estructura lógica del dominio del problema y el sistema a modelar.

En primer lugar se da un vistazo al tipo de diagrama que puede mostrar los elementos. Por cada elemento se explican las distintas formas de crearlo y especificar propiedades que están dentro del alcance de la M&D-Theory. Como ejemplo de cada elemento, se toma un elemento del modelo UML del sistema bancario visto en el capítulo 3 y se muestran figuras de su representación en el Browser (en el caso de ser posible), las partes importantes de cada Tab de especificación y su representación en los diagramas.

## 4.4.1. Diagrama de clases

Un diagrama de clases contiene íconos que representan clases (*classes*), paquetes lógicos (*packages*) y sus relaciones (*associations*, *generalizations*). Se pueden crear uno o más diagramas para representar las clases contenidas en cada paquete del modelo. Cada diagrama será contenido por el paquete con las clases que representa. La figura 4.2 ilustra los diagramas de clases necesarios para representar el modelo del sistema bancario.

Hay tres maneras de crear un diagrama de clases:

Desde el menú principal **Browse>Class Diagram...**

Desde el botón correspondiente (**Browse>Class Diagram**) en el Toolbar.

Desde el Browser, con el menú contextual del paquete al que representará, **New>Class Diagram**.

### 4.4.1.1. Toolbox del diagrama de clases

Cuando un diagrama de clases está activo, un Toolbox específico es mostrado para asistir al usuario en la construcción del diagrama. En la figura 4.3 se detallan sus posibilidades.

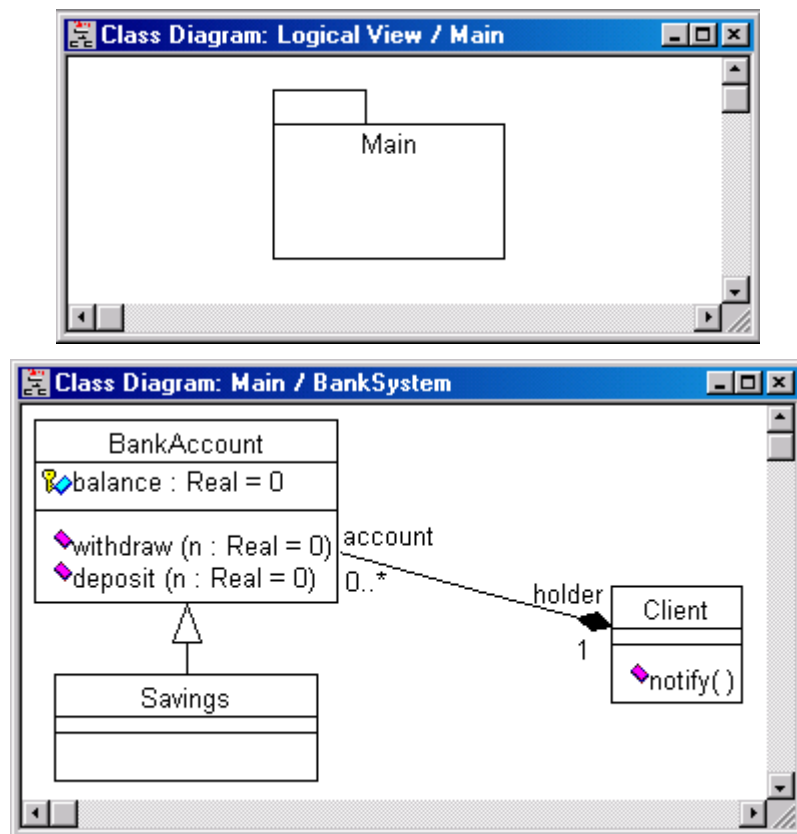
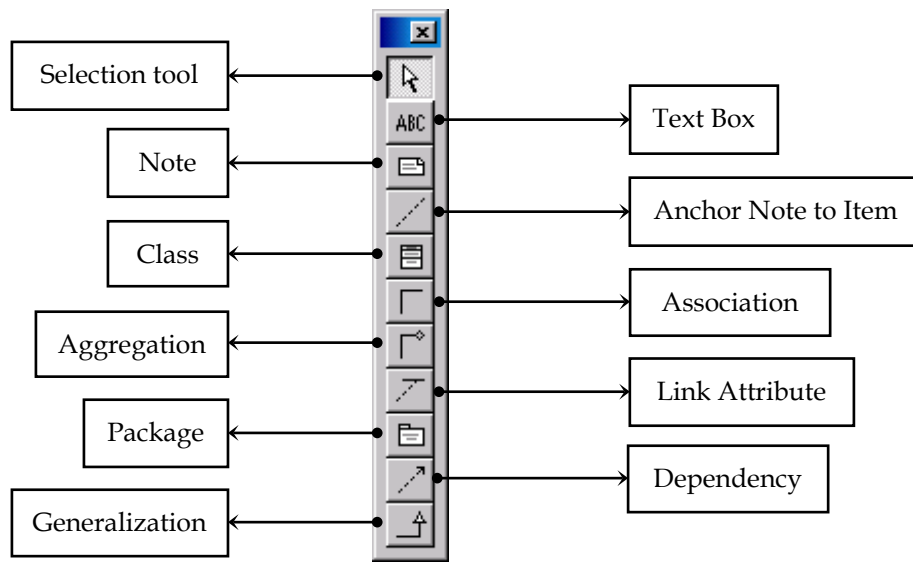


Figura 4.2: Diagramas de clases



## 4.4.1.2. Creación y especificación de elementos de modelado

### Model

Es el package con nombre “Logical View” presente en cada especificación de un modelo en Rational Rose. La figura 4.4 muestra el ítem que representa al modelo del sistema bancario en el Browser y su especificación en el General Tab. Las maneras de crear un elemento *Model* son las siguientes:

Seleccionando **File>New** del menú principal.

Seleccionando el botón  (**Create New Model**) del Toolbar.

### Especificación

*name*: El *Package* “Logical View” no permite el cambio de su nombre. Se toma como nombre del elemento *Model*, el nombre del archivo (sin la extensión) en el cual es salvado el modelo que se está especificando.

*package*: en este caso es nulo y no se puede modificar.

*ownedElements*: se pueden visualizar, agregar y eliminar, a través del Browser (clases y subsistemas) ó

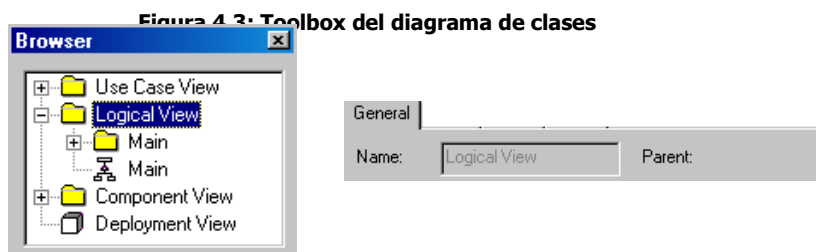


Figura 4.4: Elemento Model

de cada uno de los diagramas que contiene el *Package* “Logical View”.

### Subsystem

Un elemento *Subsystem* es cualquiera de los paquetes pertenecientes al ítem “Logical View” del Browser, sin importar el nivel de anidamiento. La figura 4.5 muestra, en el modelo del sistema bancario, su ítem en el Browser, su especificación en General tab, y su ícono en un diagrama de clases. Hay tres maneras de crear un elemento *Subsystem*:

Seleccionando el *Package* al que pertenecerá, y **new>Package** de su menú contextual.

Seleccionando **Tools>Create>Package** del menú principal.

Seleccionando el botón **Package** del Toolbox y haciendo click en un diagrama del *Package* al que pertenecerá.

### Especificación

*name*: se puede especificar con texto en su ítem en el Browser, en su ícono el diagrama o en el campo Name del General Tab.

*package*: el General Tab muestra en un campo estático el *Package* que contiene al subsistema. La modificación de esta propiedad se realiza mediante **drag&drop** del ítem del subsistema entre items de paquetes en el Browser.

*ownedElements*: se pueden visualizar, agregar y eliminar, a través del Browser (clases y subsistemas) ó de cada uno de los diagramas que contiene el subsistema.

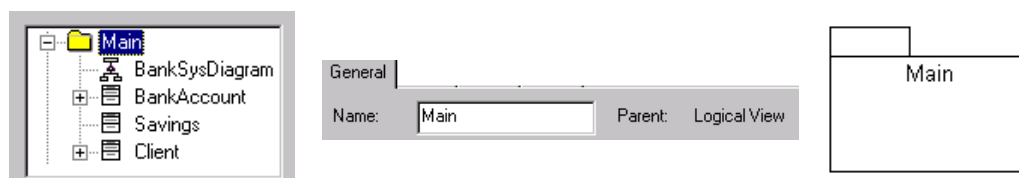


Figura 4.5: Elemento Subsystem

## Class

Hay tres formas de crear un elemento *Class*:

Seleccionando el *Package* que lo contendrá, y **new Class** de su menú contextual.

Seleccionando **Tools>Create>Class** del menú principal.

Seleccionando el botón **Class** del Toolbox y haciendo click en un diagrama del *Package* al que pertenecerá.

La figura 4.6 muestra la clase “BankAccount” en Rational Rose, su ítem en el Browser, su especificación en los tabs y su ícono en un diagrama de clases.

## Especificación

*package*: el General Tab muestra en un campo estático el *Package* que contiene a la clase. La modificación de esta propiedad se debe hacer desde el Browser.

*name*: se especifica con texto en su ítem en el Browser, en su ícono del diagrama o en el campo Name del General Tab.

*isAbstract*: para especificar que la clase es abstracta debe seleccionarse el check box Abstract. Por defecto, la clase no es abstracta.

*isActive*: se especifica seleccionando la opción Active en la propiedad Concurrency del Detail Tab.

*features*: los *Attributes* y *Operations* pueden ser vistas, agregadas o eliminadas desde el Attributes Tab y Operations Tab respectivamente.

*associationEnds*: los *AssociationEnds* pueden ser vistas o eliminadas desde el Relations Tab.

Nota: se asume que el valor de la propiedad Type es *Class*. La especificación de las demás propiedades es ignorada.

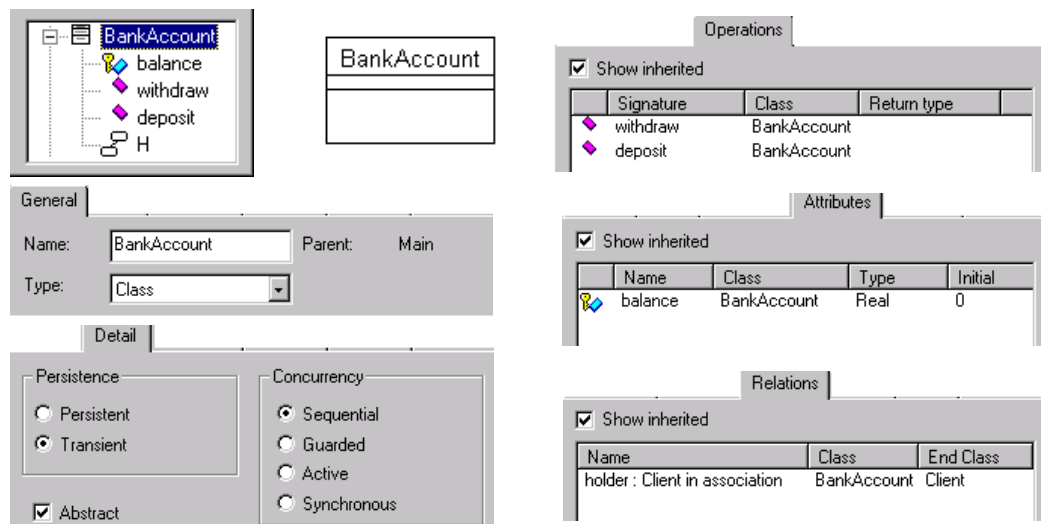


Figura 4.6: Elemento Class

## Attribute

Hay tres maneras de crear un elemento *Attribute*:

Seleccionando el ítem *Class* al que pertenecerá, y **new>Attribute** de su menú contextual.

Seleccionando **Insert** en el Attributes Tab de la especificación de la clase a la que pertenecerá.

Seleccionando **Insert New Attribute** del menú contextual de algún ícono que represente a la clase que lo contendrá.

La figura 4.7 muestra como ejemplo al atributo “balance” de la clase “BankAccount”

## Especificación

*owner*: el General Tab muestra en un campo estático la clase que contiene al atributo. La modificación de esta propiedad se realiza mediante **drag&drop** entre items de clases en el Browser.

*package*: esta propiedad es determinada por la propiedad Package de la clase a la que pertenece el atributo.

*name*: Se especifica con texto en su ítem en el Browser, o en el campo Name del General Tab.

*type*: se especifica en el campo Type del General Tab ingresando el nombre de una clase ó un tipo de dato.

*initialValue*: se especifica en el campo Initial Value del General Tab ingresando un valor. Tanto esta propiedad como Type, también se pueden modificar desde el ítem en el Browser o desde un ícono que represente a su clase y muestre el atributo.

*visibility*: Se asigna seleccionando una de las opciones del campo Export Control en el General Tab. Las opciones contempladas son: Public, Protected o Private, que corresponden a #public, #protected y #private. La opción seleccionada por defecto es Private. Si se selecciona Implementation se toma #private como valor

*ownerScope*: Seleccionando el check box Static en el Detail Tab, se indica que su valor es #class. Si no se selecciona, su valor es #instance.

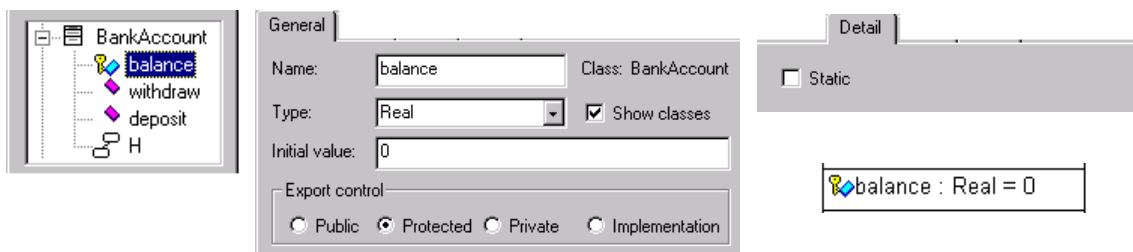


Figura 4.7: Elemento Attribute

## Operation

Hay tres maneras de crear un elemento *Operation*:

Seleccionando el ítem *Class* al que pertenecerá, y **new>Operation** de su menú contextual.

Seleccionando **Insert** en el Operations Tab de la especificación de la clase a la que pertenecerá.

Seleccionando **Insert New Operation** del menú contextual de algún ícono que represente a la clase que lo contendrá.

La figura 4.8 muestra las operaciones de la clase “BankAccount” en Rational Rose, y la especificación de la operación “deposit”.

## Especificación

*owner*: el General Tab muestra en un campo estático el nombre de la clase a la que pertenece la operación. La modificación de esta propiedad se realiza mediante **drag&drop** del ítem de la operación, entre items de clases en el Browser.

*name*: se escribe sobre el ítem, o en la propiedad Name de General Tab.

*package*: esta propiedad es determinada por la propiedad package de la clase a la que pertenece la operación, y que se muestra en el campo estático Class del General Tab.

*visibility*: Se asigna seleccionando una de las opciones del campo Export Control en el General Tab. Las opciones contempladas son: Public (#public), Protected (#protected) o Private (#private). La opción seleccionada por defecto es Public.

*concurrency*: Se asigna seleccionando una de las opciones del campo Concurrency en el Detail Tab. Las opciones contempladas son: Sequential (#sequential), Guarded (#concurrent) o Synchronous (#concurrent). La opción seleccionada por defecto es Sequential.

*parameters*: los parámetros pueden ser visualizados, agregados o eliminados desde el campo Arguments en el Detail Tab.

*precondition*: se especifica en el campo Pre conditions del Pre Conditions Tab, ingresando las invariantes que son asumidas por la operación. Si el campo es vacío, la invariante es *true*.

*postcondition*: se describe la expresión en el campo Post conditions del Post Conditions Tab. Si el campo es vacío, la expresión es *true*.

*body*: la expresión que describe la implementación de la operación se ingresa en el campo Semantics, en el Semantics Tab.

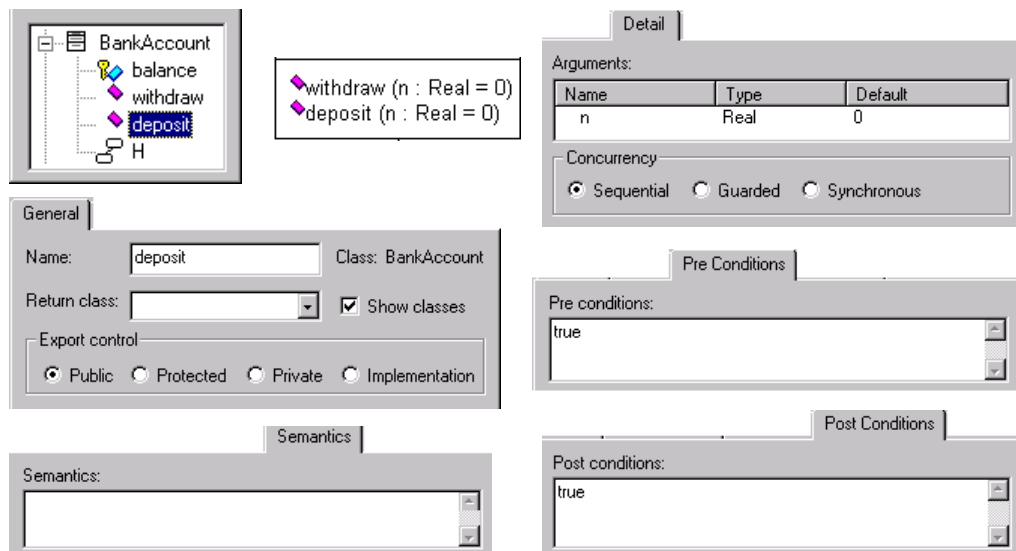


Figura 4.8: Elemento Operation

## Generalization

Existen dos procedimientos con los cuales se obtiene una generalización:

Seleccionando el botón **Generalization** del Toolbox del diagrama donde se encuentran las clases a relacionar. Hacer click en el ícono de la subclase y arrastrar la línea hasta el ícono de la superclase.

Seleccionando **Tools>Create>Generalization** del menú principal y siguiendo el procedimiento anterior sobre el diagrama.

La figura 4.9 ilustra la relación de generalización entre las clases “BankAccount” y “Savings”.

## Especificación

*name*: se especifica en la propiedad Name de General Tab.

*package*: esta propiedad es determinada por la propiedad package de la subclase de la generalización, que es mostrada en el campo estático Class del General Tab.

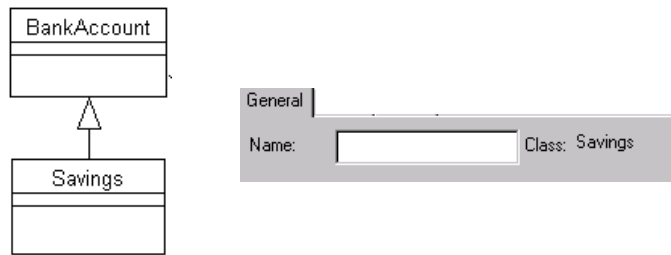
*subtype*: es posible modificarlo en el diagrama seleccionando el final de la línea (ícono) que representa a la generalización y que es más cercano al ícono de la subclase. Luego se arrastra la línea hasta el ícono de la nueva subclase.

*supertype*: es posible modificarlo en el diagrama seleccionando el final de la línea (ícono) que representa a la generalización y que es más cercano al ícono de la superclase. Luego se arrastra la línea hasta el ícono de la nueva superclase.

## Association

Existen dos procedimientos con los cuales se obtiene una asociación:

Seleccionando el botón **Association** del Toolbox del diagrama donde se encuentran las clases a relacionar. Hacer click en uno de los íconos de las clases a relacionar y arrastrar la línea hasta el ícono de la otra clase.



**Figura 4.9: Elemento Generalization**

Seleccionando **Tools>Create>Association** del menú principal y siguiendo el procedimiento anterior sobre el diagrama.

La figura 4.10 ilustra la relación de asociación entre las clases “BankAccount” y “Client”.

### Especificación

*name*: se especifica en la propiedad Name de General Tab.

*package*: esta propiedad es mostrada en el campo estático Parent del General Tab. Su valor corresponde al paquete que contiene el diagrama donde es representada la relación.

*connections*: son dos *AssociationEnds* cuyos nombres y clases destino se especifican en los campos Role A(B) y Class(A)B, respectivamente. Las asociaciones en Rational Rose son binarias, por lo tanto, no se permiten más de dos *AssociationEnds*. Se puede modificar este conjunto, arrastrando uno de los finales de la línea que representa a la asociación hasta otra clase.

### AssociationEnd

Un elemento *AssociationEnd* es creado implícitamente cuando una asociación es creada, ó reemplazada una de sus clases destino. La creación de asociaciones bidireccionales se explicó anteriormente.

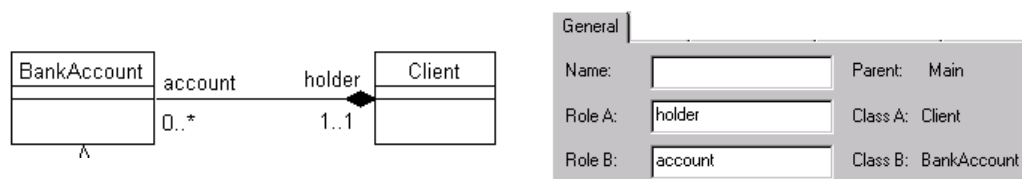
También es posible que se generen implícitamente cuando es creada una agregación, una asociación unidireccional o reemplazada una de sus clases destino. Las formas de crear una agregación o una asociación unidireccional son las siguientes:

Seleccionando el botón **Association** del Toolbox del diagrama donde se encuentran las clases a relacionar. Hacer click en el ícono de clase “parte” y arrastrar la línea hasta el ícono de la clase “agregado”.

Seleccionando **Tools>Create>Aggregate Association** del menú principal y siguiendo el procedimiento anterior sobre el diagrama.

Seleccionando **Tools>Create>Unidirectional Association** del menú principal. Luego, en el diagrama, se selecciona el ícono de la clase no accesible desde la asociación, y arrastrando la línea hasta la clase que podrá ser navegada a través de la asociación.

La figura 4.11 ilustra el *AssociationEnd* “holder” de la asociación entre las clases “BankAccount” y “Client”.



**Figura 4.10: Elemento Association**



## Especificación

*name*: se especifica en el campo Role del Role A(B) General Tab.

*type*: el Role A(B) General Tab muestra en un campo estático llamado Class con el nombre de la clase que conecta.

*association*: es la asociación a la que pertenece el *AssociationEnd*.

*package*: es determinado por la propiedad package de la asociación a la que pertenece.

*isNavigable*: se indica que el associationEnd es navegable marcando el check box Navigable en el Role A(B) Detail Tab.

*targetScope*: se indica que su valor es #classifier, marcando el check box Static en el Role A(B) Detail Tab. Por defecto el check box no está marcado y el valor es #instance.

*multiplicity*: se define en el campo Cardinality del Rol A(B) Detail Tab. Ya existen algunos valores predeterminados para seleccionar. La sintaxis para describirla es la siguiente:

Valor	Descripción
1	Una instancia
n	Número ilimitado
0..n	Cero o más
1..n	Una o más
0..1	Cero o Una
<literal>	Número exacto
<literal>..n	Número exacto o más
<literal>..<literal>	Rango especificado
<literal>..<literal>,<literal>	Rango especificado o número exacto
<literal>..<literal>, <literal>..<literal>	Múltiples rangos especificados

*aggregation*: se especifica a través de las propiedades Aggregate y Containment del Role Detail Tab. El valor es #none si el check box Navigable no es marcado. El valor es #shared si Aggregate es cierto, y By Referenced o Unspecified son seleccionados para Containment. El valor es #composite si las propiedades Aggregate y By value son seleccionadas.

*qualifier*: los atributos se visualizan, se agregan y se eliminan en el list box llamado Key/Qualifiers, en el Role Detail Tab. Por cada atributo se define un nombre y un tipo.

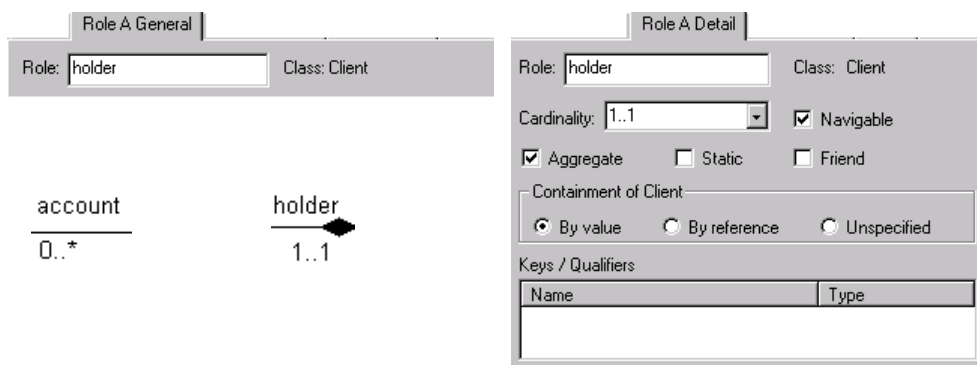


Figura 4.11: Elemento AssociationEnd

## Parameter

Un elemento *Parameter* se crea por medio del list box del Detail Tab en la especificación de una operación. Desde el menú contextual del listbox se selecciona **Insert** y un nuevo parámetro es agregado

a la lista existente. Luego puede ser modificado el orden entre ellos. La figura 4.12 muestra la especificación del parámetro “n” de la operación “withdraw”.

### Especificación

La especificación de las propiedades de un elemento *Parameter* se realiza en el campo Arguments donde se creó.

*name*: se selecciona el nombre actual (columna Name), luego se realiza un click sobre la selección y se escribe el nuevo nombre.

*type*: se selecciona el tipo actual (columna Type), luego se realiza un click sobre la selección y se escribe el nuevo tipo. Se dispone de un menú contextual con los nombres de las clases existentes en el modelo.

*defaultValue*: el cuerpo de la expresión se escribe en la columna Default siguiendo el mismo método que name y type.

*package*: Lo determina la propiedad package de la operación de la cual forma parte.

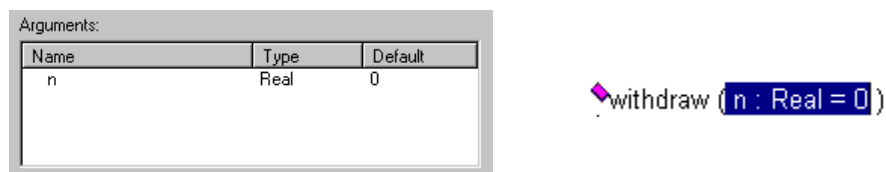


Figura 4.12: Elemento Parameter

### AssociationClass

La creación un elemento *AssociationClass* se lleva a cabo con los siguientes pasos:

Crear una asociación entre dos clases.

Crear una clase.

Seleccionar el botón **Link Attribute** del Toolbox del diagrama donde se encuentran la clase y la asociación a relacionar. O seleccionar **Tools>Create>Link Attribute** del menú principal.

Hacer click en el ícono (línea) de la asociación y arrastrar la línea punteada hasta el ícono de la clase.

Una alternativa para la creación es seleccionar el nombre de una clase del menú contextual del campo Link class del Detail Tab en la especificación de la asociación.

### Especificación

La especificación de una *AssociationClass* (figura 4.13) se realiza desde las especificaciones de la clase y la asociación que la componen.

*name*: lo determina la propiedad Name de la clase.

*package*: lo determina la propiedad Parent de la asociación.

### Data Type

La creación de un elemento *Data Type* (figura 4.14 a) se realiza a partir de la especificación de las siguientes propiedades en otros elementos:

Propiedad Type de un elemento *Attribute*.

Propiedad Return type de un elemento *Operation*.

Propiedad Type de un elemento *Parameter*.

Propiedad Type dentro de la propiedad Keys/Qualifiers de un elemento *AssociationEnd*.

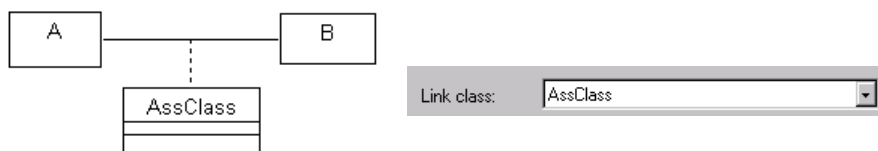


Figura 4.13: Elemento AssociationClass

## Constraint

Solamente es posible crear elementos *Constraints* (figura 4.14 b) para *Associations* y *AssociationEnds*. La especificación se realiza de la siguiente manera:

Un elemento *Constraint* para una *Association* se especifica en el campo Constraints, del Detail Tab de su especificación.

Un elemento *Constraint* para una *AssociationEnd* se especifica en el campo Constraints, del Role A(B) Detail Tab de la especificación de una asociación.



Figura 4.14 a: Elementos DataType

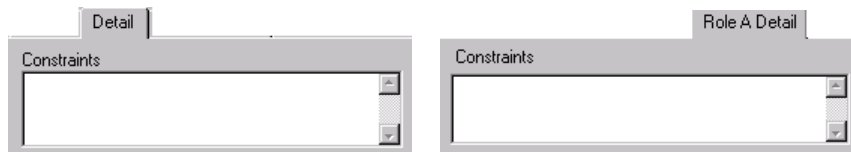


Figura 4.14 b: Elementos Constraint

## 4.4.2. Diagrama de estados

Un diagrama de estados contiene íconos que representan una máquina de estados. Una máquina de estados está compuesta de estados (*states*) y sus relaciones (*transitions*). Se puede crear un diagrama por cada clase del modelo para representar el comportamiento de sus instancias. Cada diagrama pertenecerá a la clase a la cual le describe la máquina de estados.

Un diagrama de estados se crea seleccionando el ítem de una clase, que no posea máquina de estados, y seleccionando **State Diagram** de su menú contextual. La figura 4.15 muestra el diagrama correspondiente a la máquina de estados de la clase “BankAccount” del sistema bancario.

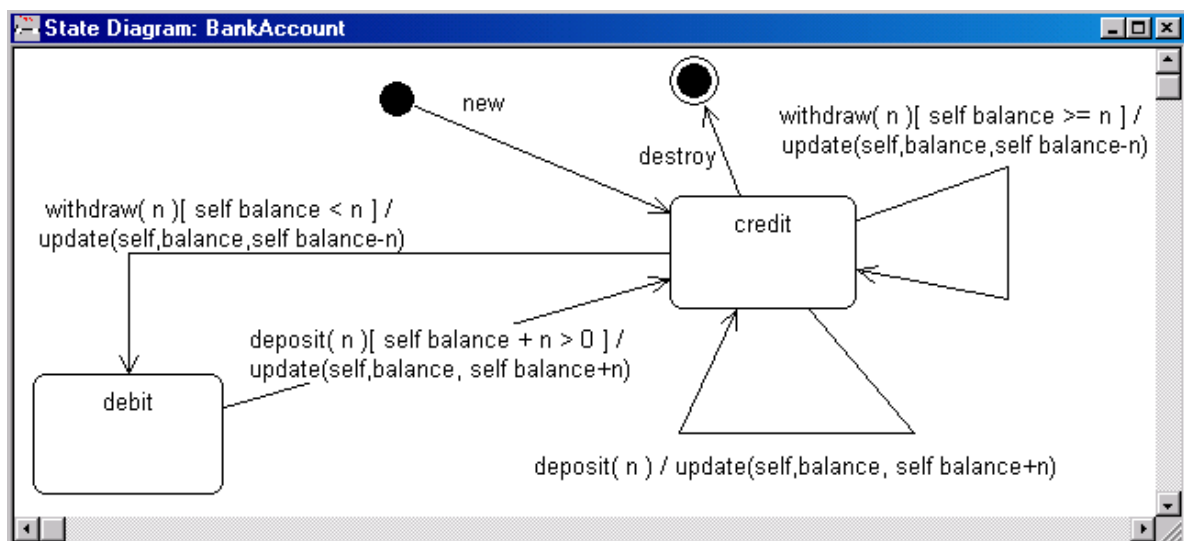


Figura 4.15: Diagrama de Estados

### 4.4.2.1. Toolbox del diagrama de estados

Cuando la ventana de un diagrama de estados está activa, un Toolbox ad hoc (figura 4.16) es mostrado para asistir al usuario en la construcción del diagrama.

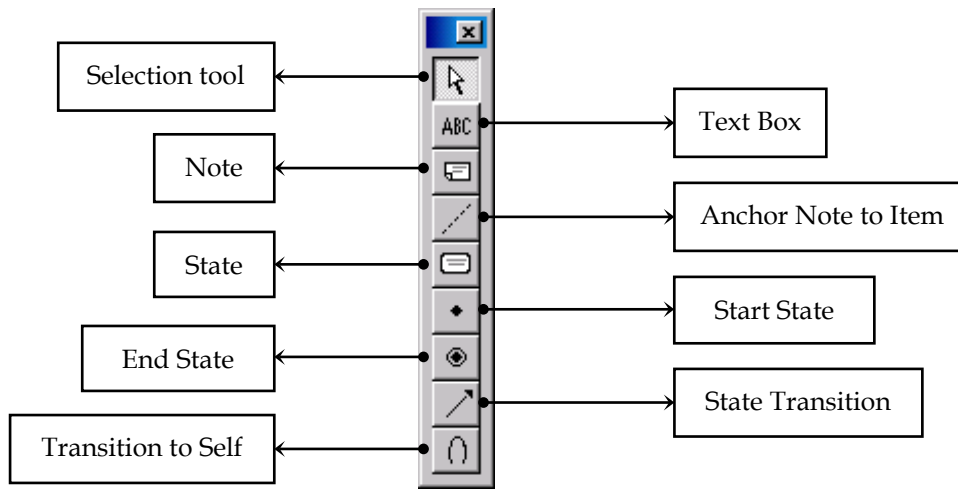


Figura 4.16: Toolbox del diagrama de Estados

### 4.4.2.2. Creación y especificación de elementos de modelado

#### StateMachine

La creación de un elemento *StateMachine* resulta de la creación de un diagrama de estados para una clase (ver figura 4.17).

#### Especificación

*name*: se especifica en el Browser, seleccionando el ítem del diagrama de estados, haciendo click sobre el nombre actual y luego escribiendo el nuevo nombre.

*package*: lo determina la clase.

*context*: es la clase que contiene el diagrama de estados. Esta propiedad es inmodificable.

*top*: no se debe crear explícitamente. Suponer que el diagrama de estados es el interior del un estado compuesto que contiene a todos los demás estados de la máquina y se llama "topState".

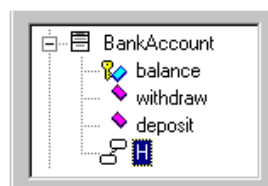


Figura 4.17: Elemento StateMachine

#### SimpleState

Un elemento *SimpleState* se crea, con el diagrama de estados como ventana activa, de las siguientes maneras:

Seleccionar el botón **State** del Toolbox o **Tools>Create>State** del menú principal y luego hacer click en el diagrama de estados.

Seleccionar el botón **Start State** del Toolbox o **Tools>Create>Start State** del menú principal y luego hacer click en el diagrama de estados.

Seleccionar el botón **End State** del Toolbox o **Tools>Create>End State** del menú principal y luego hacer click en el diagrama de estados.

La figura 4.18 muestra los tres tipos de estados y parte de la especificación del estado “debit” de la máquina de estados de la clase “BankAccount”.

### Especificación

*name*: se especifica en la propiedad Name de General Tab, o sobre el ícono del diagrama.

*package*: lo determina el diagrama de estados donde se halla representado el ícono del *SimpleState*.

*parent*: es el estado compuesto que lo contiene en el diagrama.

*incoming*: es el conjunto de transiciones cuyas flechas de representación en el diagrama apuntan al ícono del *SimpleState*.

*outgoing*: es el conjunto de transiciones cuyas flechas de representación en el diagrama salen desde el ícono del *SimpleState*.

*kind*: su valor depende de cómo fue creado el *SimpleState*. Los posibles valores son:

#normal : si fue creado por la primera opción y no se representó ningún estado en el interior de su ícono en el diagrama.

#initial : si fue creado por la segunda opción.

#final : si fue creado por la segunda opción.

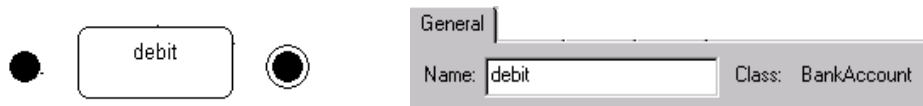


Figura 4.18: Elementos SimpleState

### CompositeState

Un elemento *CompositeState* se crea con los siguientes pasos:

Seleccionar el botón **State** del Toolbox o **Tools>Create>State** del menú principal.

Hacer click en el diagrama de estados.

Representar por lo menos un estado en el interior del ícono del estado creado.

### Especificación

*name*: Se puede describir en el General Tab o dentro de su ícono en el diagrama. El nombre “topState” del estado *top* de la máquina es inmodificable, porque no se visualiza.

*package*: lo determina el diagrama de estados donde se halla representado el ícono del *CompositeState*.

*parent*: es el estado compuesto que lo contiene en el diagrama. Los estados que se encuentran en el nivel más alto, en realidad, tendrán como parent al *top* de la máquina de estados. El *parent* de “topState” será *nullElement*.

*incoming*: es el conjunto de transiciones cuyas flechas de representación en el diagrama apuntan al perímetro del ícono del *CompositeState* o a uno de sus subestados.

*outgoing*: es el conjunto de transiciones cuyas flechas de representación en el diagrama salen desde el perímetro del ícono del *CompositeState* o sus subestados.

*substates*: es el conjunto de estados cuyos íconos están contenidos dentro del ícono del *CompositeState* en el diagrama de estados.

*internalTransitions*: es el conjunto de transiciones cuyas flechas están contenidas en el ícono del *CompositeState*, o sea relacionan sus subestados.

*isTop*: es falso en todos los casos, excepto en el *CompositeState* creado implícitamente con la máquina de estados.

## Transition

Las formas de crear un elemento *Transition* son las siguientes:

Seleccionar el botón **State Transition** del Toolbox, o **Tools>Create>Transition** del menú principal, siendo el diagrama de estados la ventana activa. Hacer click en el ícono del estado origen y arrastrar la línea hasta el ícono del estado destino.

Seleccionar el botón **Transition to self** del Toolbox, **Tools>Create>Loop** del menú principal, siendo el diagrama de estados la ventana activa. Hacer click en el ícono del estado que será fuente y destino a la vez. Es una transición hacia el mismo estado.

La figura 4.19 muestra la representación gráfica y especificación de la transición desde el estado “debit” hacia estado “credit”.

## Especificación

*source*: es el estado desde cuyo ícono parte la flecha que representa a la transición. Su nombre se puede ver en el campo *Transition between substates>From* del *Detail Tab*. Se modifica, solamente, desde el diagrama.

*target*: es el estado cuyo ícono es apuntado por la flecha que representa a la transición. Su nombre se puede ver en el campo *Transition between substates>To* del *Detail Tab*. Se modifica, solamente, desde el diagrama.

*trigger*: un *Event* puede ser especificado en los campos *Event* y *Arguments* del *Detail Tab*.

*guard*: una *Guard* puede ser especificada en el campo *Condition* del *Detail Tab*.

*effect*: una *Action* puede ser especificada en el campo *Action* del *Detail Tab*.

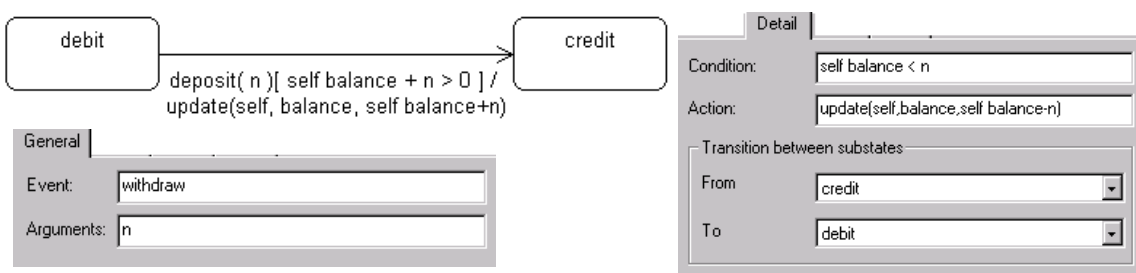


Figura 4.19: Elemento Transition

## Event

La creación de un elemento *Event* se realiza especificando las propiedades *Event* y *Arguments* del *General Tab* de una transición. Distintas subclases de *Event* pueden ser generadas, dependiendo del valor de dichas propiedades. Las alternativas se detallan en la siguiente tabla:

M&D-theory	Rational Rose	
Subclase de Event	Propiedad Event	Propiedad Arguments
CreationEvent	create	
CreationEvent	new	
DestructionEvent	destroy	
TimeEvent	timeOut	
CallEvent	<operationName>	<parametersList>

<operationName>: el nombre de una operación correspondiente a la clase o superclase que contiene a la máquina de estados.

<parametersList>: lista de nombres de los parámetros de la operación, en el ámbito de la máquina de estados. Van separados mediante comas “,”. La cantidad de nombres debe ser igual a la cantidad de parámetros especificados en la operación. Cada nombre no debe, necesariamente, coincidir con el nombre del *Parameter* en el modelo.

## Especificación

Se realiza sobre las mismas propiedades que especifican el *Event*. Se modifican en el General Tab de la transición, o en la etiqueta de la transición en el diagrama. Ver figura 4.20.

*name*: es el contenido del campo de la propiedad Event. En el caso de *CallEvents*: “<operationName>(<parametersList>)”.

*operation*: (*CallEvent* solamente) es la operación en el modelo cuyo nombre y cantidad de parámetros coincide con la propiedad Event y la cantidad de nombres en la propiedad Arguments, respectivamente.

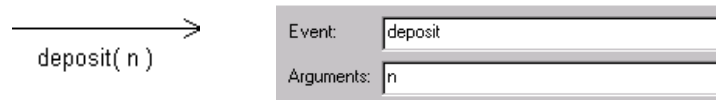


Figura 4.20: Elemento Event

## Guard

Un elemento *Guard* se crea especificando las propiedad Condition en el Detail Tab de una transición. Ver figura 4.21.

## Especificación

*expression*: se describe con un cuerpo de bloque Smalltalk, teniendo en cuenta que sus argumentos serán *self* (la instancia de la clase que recibió el evento) y los parámetros especificados en la propiedad Arguments del General Tab de la transición. La instancia puede recibir mensajes tales como un nombre de uno de sus atributos o un nombre de una *AssociationEnd* opuesta. Al evaluarlo en un ambiente Smalltalk con los argumentos adecuados, debe devolver un valor booleano.



Figura 4.21: Elemento Guard

## ActionSpec

La creación de un elemento *ActionSpec* se realiza especificando las propiedad Action en el Detail Tab de una transición (figura 4.22). Distintas subclases de *ActionSpec* pueden ser generadas de acuerdo al valor de dicha propiedad. Las alternativas se detallan en la siguiente tabla:

M&D-theory	Rational Rose
Subclase de ActionSpec	Propiedad Action
CreationActionSpec	create(<className>)
DestroyActionSpec	destroy(<objExp>)
CallActionSpec	call(<operationName>,<objExpList>)
LocalInvocationSpec	update(<objExp>,<attributeName>,<objExp>)

<className>: nombre de una clase del modelo.

<attributeName>: nombre de uno de los atributos de la clase, o superclase, a la que pertenece la máquina de estados. También puede ser el nombre de una *AssociationEnd* opuesta.

<objExp>: se describe con un cuerpo de bloque Smalltalk, teniendo en cuenta que sus argumentos serán *self* (la instancia de la clase que recibió el evento) y los parámetros especificados en la propiedad Arguments del General Tab de la transición. La instancia puede recibir mensajes tales como un nombre de argumento ó un nombre de *AssociationEnd* opuesta. Al evaluarlo en un ambiente Smalltalk con los argumentos adecuados, debe devolver un objeto.

<objExpList>: secuencia de <objExp> separadas mediante comas “,”.

## Especificación

### *CreationActionSpec*

*classifier*: el nombre de la clase, que creará una instancia, es el único argumento especificado para la palabra clave “create”.

### *DestroyActionSpec*

*expression*: único argumento para la palabra clave “destroy”. Describe el cuerpo de una *ObjectExpression* que al evaluarla deberá devolver la instancia que dejará de existir.

### *CallActionSpec*

*operation*: primer argumento para la palabra clave “call”. Es el nombre de la operación que se quiere invocar.

*arguments*: son los argumentos descriptos luego del nombre de operación. Describen cuerpos de *ObjectExpression* que al evaluarlos devolverán las instancias que necesita la operación para ejecutarse.

### *LocalInvocationSpec*

*receiver*: primer argumento para la palabra clave “update”. Describe el cuerpo de una *ObjectExpression* que al evaluarla deberá devolver la instancia que recibirá la invocación local.

*attributeName*: segundo argumento para la palabra clave “update”. Es el nombre de atributo del objeto que se modificará mediante la invocación local.

*newValue*: tercer argumento para la palabra clave “update”. Describe el cuerpo de una *ObjectExpression* que al evaluarla deberá devolver el nuevo valor para el atributo descrito en *AttributeName*, del objeto descrito en *Receiver*.

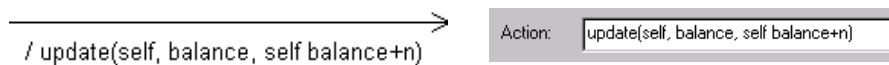


Figura 4.22: Elemento ActionSpec

## 4.5. Representación textual de un modelo

La información representada en un modelo especificado con la aplicación CASE Rational Rose, se almacena en archivos con extensión .mdl o .ptl. Estos tipos de archivos utilizan el formato ASCII para representar la información gráfica y textual contenida en los modelos, y se los denomina *petals* (pétalos). Debido al formato ASCII utilizado, es posible transferir modelos y componentes de modelos con independencia de plataformas.

Mediante la opción **File>Save** o **File>Save as...** del menú principal, se puede almacenar el modelo actual en la aplicación en un archivo con extensión .mdl. Si se selecciona uno o varios elementos del modelo y se elige **File>Export** en el menú principal, se pueden almacenar determinados items, en un archivo con extensión .ptl.

Este tipo de archivos también contiene información sobre la posible generación de código en algún lenguaje. Gran parte de la información especificada no es de interés para este trabajo. En la siguiente sección se describirá en forma incompleta la gramática del contenido de un archivo .mdl. Se destacará la estructura principal del texto del archivo y la especificación de los elementos de modelado de relevancia descriptos anteriormente.



## 4.5.1. Gramática de un archivo .mdl

### 4.5.1.1. Estructura principal

La estructura del archivo .mdl (<PetalFile>) se divide, en principio, en dos partes principales:

```
<PetalFile>
  <PetalVersion>
  <Design>
```

#### PetalVersion

Contiene datos sobre la versión de la herramienta con la que fue generado el archivo. Propiedades adicionales, varían en distintas versiones.

```
<PetalVersion>
(object Petal
  version          40)
```

#### Design

Contiene toda la información del modelo y sus vistas. Esto comprende la especificación de los elementos de modelado y la representación textual de sus correspondientes íconos gráficos.

```
<Design>
(object Design "Logical View"
  is_unit          <Boolean>
  is_loaded        <Boolean>
  file_name        <FilePath>          {Path completo del archivo .mdl}
  quid             <Identity>
  defaults         (...{Datos referentes a la visión de la aplicación: fuentes, ubicación, etc.}...)
  root_usecase_package (...{Datos referentes a la vista de casos de uso: modelos, diagramas, etc.}...)
  root_category    <Class_Category>    {con nombre obligatorio "Logical View"}
  root_subsystem   (...{Datos relacionados con la vista de componentes}...)
  process_structure (...{Datos sobre la vista de despliegue}...)
  properties       (...{especificación de propiedades útiles para la generación de código}...)
```

### 4.5.1.2. Elementos destacados

#### ClassCategory

Representa un ítem *Package* del Browser de Rational Rose. Contiene la representación textual de características propias, diagramas de clases, clases, asociaciones y otros paquetes.

```
<ClassCategory>
(object Class_Category <String>          {nombre}
  quid             <Identity>          {identificación única para el modelo}
  exportControl    <ExportEnum>
  global           <Boolean>
  subsystem        <String>
  logical_models   ( list unit_reference_list <Class|Association|Class_Category>* )
  logical_presentations (...{Propiedades gráficas de los diagramas de clases}... )
```

#### Class

Su representación contiene propiedades específicas a una clase, y eventualmente puede contener representaciones de atributos, operaciones, referencias a superclases y un diagrama de estados.

```
<Class>
(object Class <String>          {nombre}
  quid             <Identity>          {identificación única para el modelo}
  abstract         <Boolean>
  concurrency      <ConcurrencyType>
  operations       ( list Operations <Operation>* )
  class_attributes ( list class_attribute_list <ClassAttribute>* )
  superclasses     ( list inheritance_relationship_list <InheritanceRelationship>* )
  .....
  statemachine     <StateMachine>
```

statediagram (...{Datos sobre la representación gráfica del diagrama de estados}. . . ) . . . )

## Operation

Describe las propiedades del ítem *Operation* y puede contener la especificación de varios parámetros.

### <Operation>

```
(object Operation <String>
  quid <Identity>
  parameters ( list Parameters <Parameter>* )
  result <String>
  concurrency <ExportEnum>
  pre_condition <SemanticInfo>
  post_condition <SemanticInfo>
  semantics <SemanticInfo>      ...)
```

## ClassAttribute

Representación del elemento *Attribute*. También utilizada para *Qualifiers*.

### <ClassAttribute>

```
(object ClassAttribute <String>
  quid <Identity>
  type <String>
  initv <String>
  exportControl <ExportEnum>
  static <Boolean>      ....)
```

## InheritanceRelationship

Contiene la información necesaria para representar una *Generalization* en el modelo.

### <InheritanceRelationship>

```
(object Inheritance_Relationship
  quid <Identity>
  label <String>
  supplier <String>      {nombre de superclase}
  quidu <Identity>      {de la superclase}      ...)
```

## Association

Especifica propiedades de la relación *Association* y contiene la especificación de sus dos roles.

### <Association>

```
(object Association <String>
  quid <Identity>
  roles (list role_list <Role>*)
  Constraints <Text>
  AssociationClass <String>      {si es AssociationClass}      ...)
```

## Role

Especificación de las propiedades de un *AssociationEnd*. Incluye la representación de una lista de *Attributes*, que actúan como *Qualifiers*.

### <Role>

```
(object Role <String>
  quid <Identity>
  label <String>      {nombre}
  supplier <String>      {nombre de clase que conecta}
  quidu <Identity>      {de la clase que conecta}
  client_cardinality <Cardinality>
  s_navigable <Boolean>
  is_aggregate <Boolean>
  Containment <ContainmentEnum>
  is_principal <Boolean>
  friend <Boolean>
  static <Boolean>
  keys ( list class_attribute_list <ClassAttribute>* )      ....)
```

## StateMachine

Describe información referente a la máquina de estados representada en un *StateDiagram*. Contiene una lista con las especificaciones de los estados que la conforman.

### <StateMachine>

(object State\_Machine  
states (list States <State>\* )

## State

Describe propiedades de *State*, *StartState* o *EndState*. Contiene las especificaciones de las transiciones que parten desde el estado.

### <State>

(object State <String>  
quid <Identity>  
transitions (list transition\_list <StateTransition>\*)  
type <StateType>  
statemachine <StateMachine>  
actions (...{Acciones internas al estado}. . . ) . . . )

## StateTransition

Especificación de las propiedades de una *Transition*, del evento que la dispara, su condición y la acción que ejecuta.

### <StateTransition>

(object State\_Transition  
quid <Identity>  
supplier <String> {nombre del estado destino}  
quidu <Identity> {del estado destino}  
Event <Event>  
condition <String>  
action <Action>  
sendEvent (...{Datos sobre un evento que envía la transición}... ) ....)

## Event y Action

### <Event>

(object Event <String>  
parameters <String>)

### <Action>

(object action <String>  
quid <Identity>)

## Tipos básicos

<ExportEnum> ::= "Public" | "Private" | "Protected" | "Implementation"

<ConcurrencyType> ::= "Guarded" | "Active" | "Synchronous"

<StateType> ::= "StartState" | "Normal" | "EndState"

<ContainmentEnum> ::= "By Reference" | "By Value"

<Cardinality> ::= (value cardinality <String>) {Ver tabla de Multiplicidad}

<SemanticInfo> ::= (object Semantic\_Info  
PDL <Text>)

<Boolean> ::= TRUE | FALSE

<Identity> ::= " {Número hexadecimal} "

<Text> ::= <String> | ( | <String> )<sup>+</sup>

<String> ::= " {Secuencia de caracteres ASCII} "

<FilePath> ::= " {Ubicación en un sistema de archivos} "



# 5. Herramienta. Manual de Uso

## 5.1. Arranque de la aplicación

Hay dos opciones para ejecutar la herramienta desde el ambiente Visual Works [VisualWorks 98]:

- Escribir *ModelCheckerInterface open* en una ventana Workspace, luego seleccionar el texto y elegir **do it** del menú contextual.
- Abrir el Resource Finder desde el Visual Launcher **-Browse>Resources-**, seleccionar la clase ModelCheckerInterface, el recurso windowSpec, y luego el botón **Start**.

## 5.2. Ventana principal de la aplicación

La interfaz gráfica principal en la cual un usuario interactúa con la especificación formal en Lógica Dinámica de un modelo orientado a objetos, se puede separar en ocho componentes importantes: Model Level, Elements Repository, Data Level, Taxonomy, Signature, Axioms, Label y Display. La barra de título de la ventana contiene el nombre “Model Checker” y el nombre del archivo (.dlm) en el que fue salvado el modelo formal visualizado actualmente en la aplicación. Ver figura 5.1.

### Model Level

Muestra una estructura jerárquica de íconos que representan entidades de modelado pertenecientes al modelo formal especificado (nivel del modelo). Cada elemento del modelo es representado por un ícono que representa a su Sort, y su atributo *name*. Cada ítem puede ser seleccionado, o expandido, para consultar sus características ó ser modificado. Pueden formar parte de este árbol elementos tales como clases, asociaciones, paquetes, atributos, operaciones, máquinas de estados etc.

### Data Level

Muestra una estructura jerárquica de íconos que representan entidades modeladas pertenecientes al modelo formal especificado (nivel de los datos). Elementos que pueden aparecer en este árbol son: objetos, links, mensajes, etc.

### Elements Repository

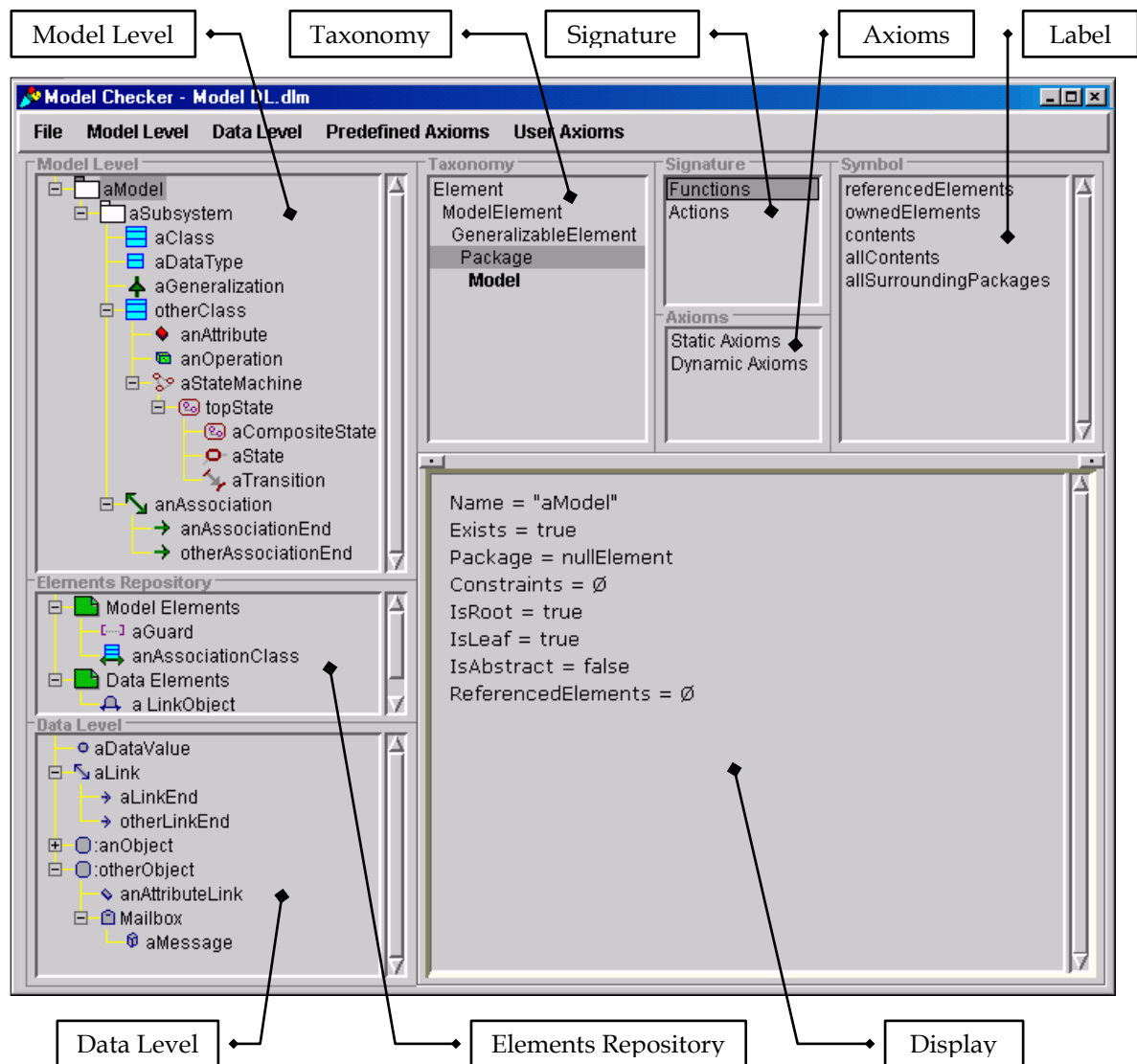
Se utiliza como depósito de elementos del nivel del modelo y elementos del nivel de los datos, que no pertenecen al modelo formal en su estado actual. Las entidades creadas por un usuario se ubican, en principio, en este árbol. Dichas entidades pueden ser incorporadas, a futuro, al modelo formal mediante acciones de evolución. Asimismo, elementos eliminados del modelo pasan a formar parte de este depósito.

### Taxonomy

Muestra una lista de nombres de Sorts organizada jerárquicamente conforme a la relación  $\odot$  de la M&D-Theory. Varía de acuerdo al ítem seleccionado en cualquiera de los niveles del modelo formal. El nombre del Sort al cual pertenece es resaltado, y acompañado por los nombres de los Sorts ancestros (superSorts). Seleccionando cada Sort puede accederse a la signatura relacionada, y a los axiomas definidos por la teoría donde sus instancias tienen primordial participación.

**Signature**

**Figura 5.1: Ventana principal**



Presenta la signatura relacionada con el Sort seleccionado. Es una lista conformada por las opciones "Functions", "Predicates" y/o "Actions". Seleccionando cada una de ellas, se accede a las funciones, los predicados y las acciones cuyas declaraciones contienen en primer lugar al Sort seleccionado.

### Axioms

Tipos de propiedad donde las instancias del Sort seleccionado tienen una participación preponderante. La lista de opciones puede estar conformada por: "Static Axioms" y/o "Dynamic Axioms".

### Label

Muestra una lista de símbolos de funciones, predicados o acciones, y listas de títulos descriptivos para los axiomas, correspondientes a la opción seleccionada en Signature o en Axioms. El rótulo de la lista cambia entre "Symbol" y "Title". La opción seleccionada en esta lista puede ser evaluada en el actual estado del modelo.

### Display

En esta región se pueden observar los atributos más destacados de cada uno de los elementos del modelo. También se visualizan las declaraciones y comentarios de cada opción seleccionada en Label, y los resultados de los chequeos sobre el modelo.

## 5.2.1. Barra de menús

Desde la barra principal de menús de la aplicación se accede a cinco menús cuyas opciones se detallan a continuación:

### 5.2.1.1. Menú File

**Parse Rose Model...** : despliega una ventana de diálogo con el árbol de directorios, para que el usuario seleccione un archivo con extensión .mdl (**model**). Este tipo de archivos es generado por Rational Rose y contiene la representación textual de un modelo especificado con la notación UML.

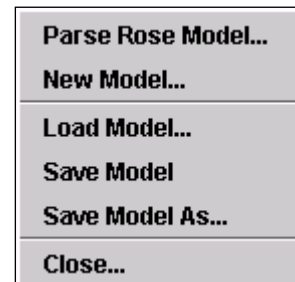
**New Model...** : crea un nuevo modelo formal en la aplicación.

**Load Model...** : despliega una ventana de diálogo con el árbol de directorios, para que el usuario seleccione un archivo con extensión .dlm (**dynamic logic model**). Este tipo de archivos contiene un modelo especificado en Lógica Dinámica para ser utilizado en esta aplicación.

**Save** : guarda el modelo actual en la aplicación, en un archivo con extensión .dlm y el nombre pre asignado.

**Save As...** : despliega una ventana de diálogo con el árbol de directorio, para que el usuario seleccione la ubicación y el nombre (con extensión .dlm) del archivo que contendrá el modelo actual en la aplicación.

**Close**: cierra la aplicación.



### 5.2.1.2. Menú Model Level

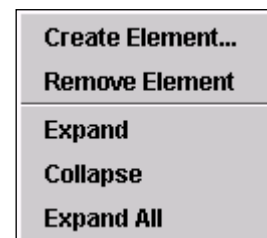
**Create Element...** : Permite crear un elemento en el nivel del modelo, es decir una instancia del metamodelo. Ventana de diálogo mediante, el usuario elige el Sort y el nombre del elemento, y este se ubica en la carpeta “Model Elements” de Elements Repository.

**Remove Element** : Elimina el elemento seleccionado en la carpeta “Model Elements” de Elements Repository.

**Expand** : Expande el nodo seleccionado en Model Level. El número de hijos desplegados depende de los atributos de cada elemento.

**Collapse** : Contrae los subárboles del nodo seleccionado en Model Level

**Expand All** : Expande el árbol que representa el modelo hasta el nivel más bajo posible.



### 5.2.1.3. Menu Data Level

**Create Element...** : Permite crear un elemento apropiado para el nivel de los datos, para representar una instancia del modelo. Ventana de diálogo mediante, el usuario elige el Sort y el dataElement se ubica en la carpeta “Data Elements” de Elements Repository con un nombre predeterminado.

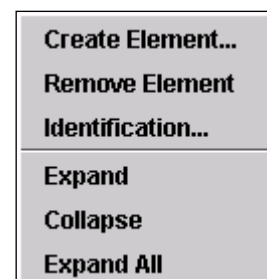
**Remove Element** : Elimina el elemento seleccionado en la carpeta “Data Elements” de Elements Repository.

**Identification...** : El usuario puede darle un nombre representativo a cada elemento presente en el nivel de los datos. Permite distinguir visualmente a instancias con distinta identidad.

**Expand** : Expande el nodo seleccionado en Data Level.

**Collapse** : Contrae el nodo seleccionado en Data Level.

**Expand All** : Expande todas los nodos que representan instancias en el nivel de los datos.





#### 5.2.1.4. Menú Predefined Axioms

**Browse Static Axioms...** : Abre una ventana que posibilita al usuario consultar y evaluar sobre el modelo actual cada una de las fórmulas estáticas definidas en la M&D-Theory. Las fórmulas están clasificadas de acuerdo al Sort que cumpla un rol más preponderante en la fórmula.

**Browse Static Axioms**

**Browse Dynamic Axioms**

**Browse Dynamic Axioms...** : Abre una ventana que posibilita al usuario consultar y evaluar sobre el modelo actual cada una de las fórmulas dinámicas definidas en la M&D-Theory.

#### 5.2.1.5. Menú User Axioms

**Browse** : Despliega una ventana donde el usuario puede navegar grupos de axiomas estáticos y axiomas dinámicos creados adicionalmente. Los axiomas pueden ser evaluados sobre el modelo presente en la aplicación, o eliminados. Los axiomas estáticos pueden ser modificados.

**Edit Static Axiom** : Abre un editor de fórmulas estáticas en Lógica Dinámica que permite al usuario crear sus propios axiomas y documentarlos.

**Edit Dynamic Axiom** : Abre un editor de fórmulas dinámicas en Lógica Dinámica que permite al usuario crear sus propios axiomas y documentarlos. Se puede optar entre la edición de precondiciones ó postcondiciones sobre acciones primitivas de evolución.

**Save As...** : Posibilita guardar el grupo de axiomas editados en un archivo con extensión .axm (axiom).

**Load...** : despliega una ventana de diálogo con el árbol de directorios, para que el usuario seleccione un archivo con extensión .axm. Los axiomas cargados se suman a los presentes.

**Reset** : Se eliminan todos los *User Axioms* presentes en la aplicación.

**Browse**

**Edit Static Axiom**

**Edit Dynamic Axiom**

**Save As...**

**Load...**

**Reset...**

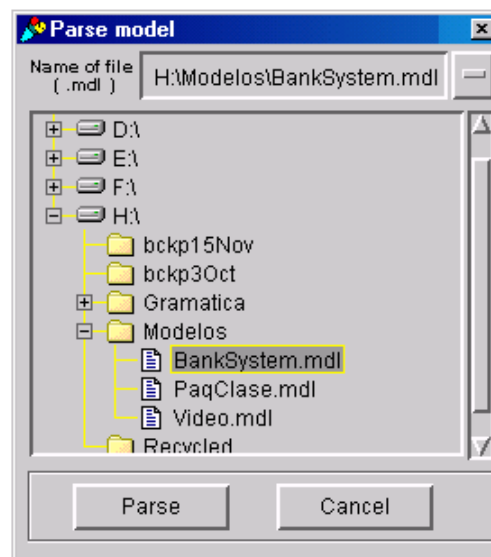


Figura 5.2: Elección del modelo UML

## 5.3. Traducción de un modelo UML

Mediante la opción **File>Parse Rose Model**, se debe seleccionar un archivo que contenga la representación textual de una especificación gráfica en notación UML, generada por la herramienta CASE Rational Rose 4.0 (versión Students). Como ejemplo se parsea el modelo en Rational Rose del sistema bancario visto en los capítulos anteriores, y salvado con el nombre `BankSystem.mdl`. Ver figura 5.2.

Una vez traducido el modelo, se pueden visualizar cada uno de los términos LD instanciados en el componente Model Level de la interfaz gráfica. De acuerdo con el axioma de inicialización de la teoría, en el estado inicial no existen elementos modelados, tales como objetos, mensajes, etc. Por esta razón Data Level no contiene elementos luego de la traducción. También se eliminan elementos residuales en Elements Repository. La apariencia de la herramienta luego de la traducción del modelo Rose es ilustrada en la figura 5.3.

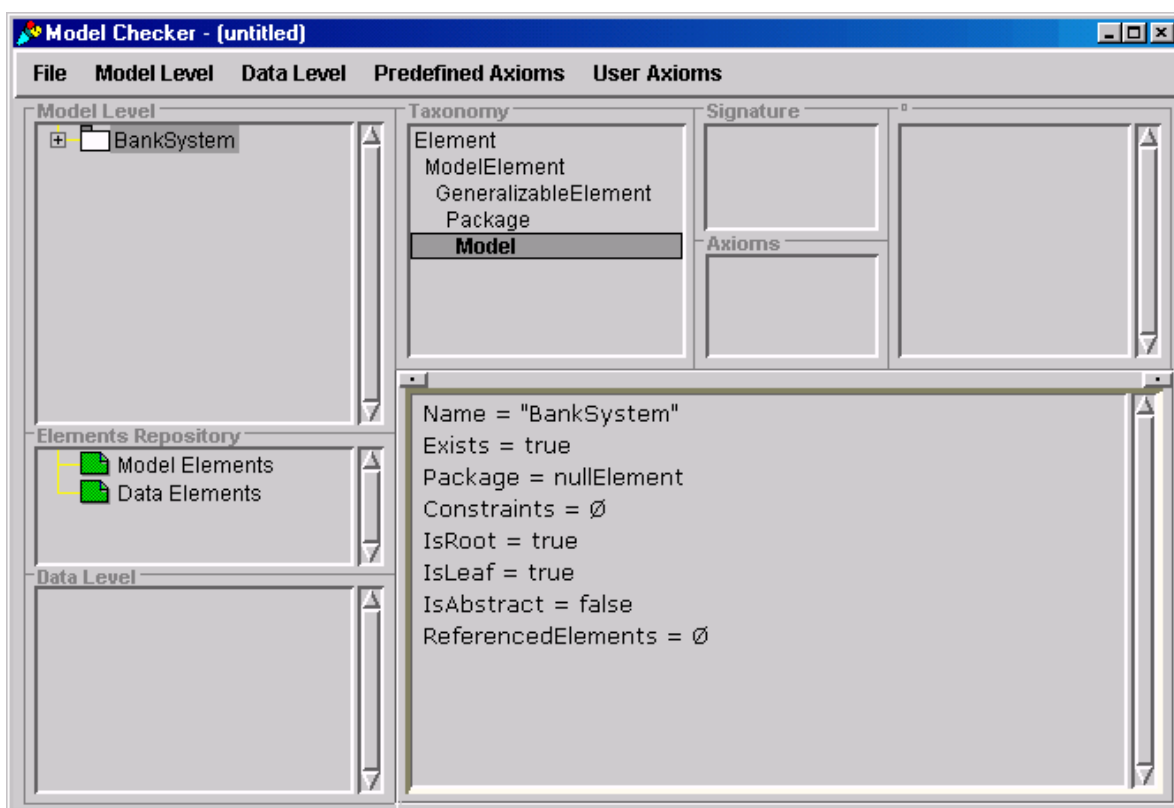


Figura 5.3: Modelo Traducido

## 5.4. Navegando los elementos del modelo formal

Todos los elementos LD que componen el modelo formal están representados en los distintos niveles del árbol. Cada elemento es representado con un ícono que representa a su Sort, y el atributo *name* a la derecha del ícono. La selección de un elemento del modelo, ícono mediante, influye en el resto de los componentes gráficos de la interfaz. Por ejemplo si se selecciona el elemento raíz del modelo parseado, en Taxonomy se muestran el Sort de dicho elemento (*Model*) y sus supersorts (*Package*, *GeneralizableElement*, *ModelElement* y *Element*). En Display se detallan un grupo de atributos y asociaciones que caracterizan al elemento. En Signature y Axioms se presentan los tipos de axiomas y signatura disponible para el Sort seleccionado.

El acceso a los distintos elementos del modelo y el sistema modelado, a través de sus árboles, se realiza expandiendo el nodo que los representa. Hay tres maneras: desde el menú contextual **expand**, haciendo click en el símbolo “+” junto al ícono del elemento, o **Model(Data) Level>Expand**. Mediante la expansión se puede acceder a determinados elementos relacionados. Un elemento no se representa dos veces en el árbol, se accede por un único camino.

Los elementos visualizados luego de una expansión dependen del tipo del elemento expandido y su estado. A continuación se presentan algunos ejemplos sobre el modelo “BankSystem” y se ilustran en la figura 5.4.

- ♦ **A:** la expansión de un elemento *Package* **p** muestra **ownedElements(p)**.
- ♦ **B:** la expansión de un elemento *Class* **c** muestra **features(c) ∪ {behavior(c)}**.
- ♦ **C:** la expansión de un elemento *Operation* **o** muestra **parameters(o)**.
- ♦ **D:** la expansión de un elemento *Association* **a** muestra **connections(a)**.
- ♦ **E:** la expansión de un elemento *Transition* **t** muestra **{guard(t)} ∪ {trigger(t)} ∪ {effect(t)}**.
- ♦ **F:** la expansión de un elemento *CompositeState* **s** muestra **substates(s) ∪ internalTransitions(s)**.

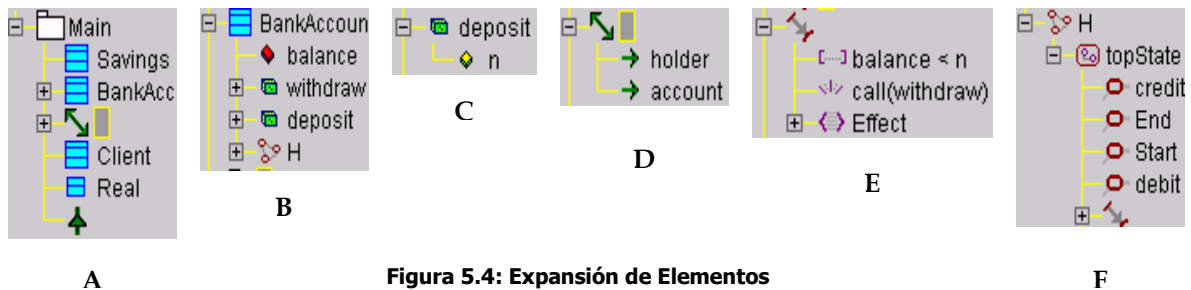


Figura 5.4: Expansión de Elementos

## 5.5. Consultando los elementos del modelo formal

Muchas de las características del modelo no se presentan con sólo navegar por los distintos elementos visualizados en Model Level y Data Level. Para obtener más información se pueden evaluar las funciones y los predicados de la teoría.

En la ventana de la figura 5.5 se observa que con sólo seleccionar y expandir el nodo que representa a la clase “BankAccount” se obtiene la siguiente información:

- En Model Level se presentan como subnodos su atributo “balance”, sus operaciones “withdraw” y “deposit”, y su máquina de estados “H”.
- En Display se muestra la interpretación de funciones y predicados destacados de la clase como *name*, *exists*, *package*, *isRoot*, *isLeaf*, *isAbstract*, *isActive* y *behavior*. Las funciones *features* y *constraints*, se muestran cuando su interpretación es el conjunto vacío.
- La jerarquía Taxonomy, muestra su Sort *Class* seleccionado por defecto y los sorts que lo generalizan: *Classifier*, *GeneralizableElement*, *ModelElement* y *Element*.
- La lista Signature indica que para el Sort *Class* hay funciones y predicados para evaluar, y acciones para ejecutar.
- La lista Axioms indica que para el Sort *Class* hay axiomas estáticos y dinámicos definidos en la teoría para evaluar. En el caso de la evaluación de los axiomas estáticos participan todas las instancias del Sort *Class*, y no solamente la clase “BankAccount”.

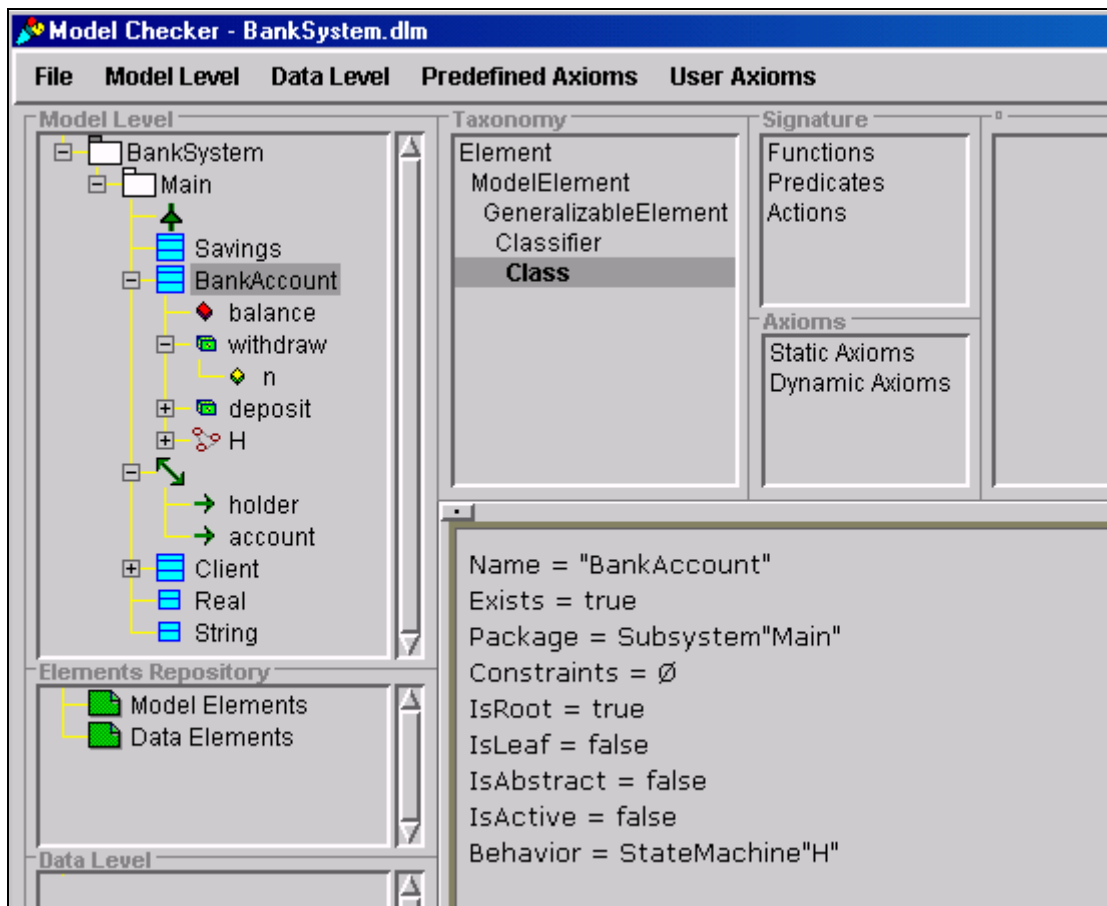


Figura 5.5: Selección y expansión de la clase "BankAccount"

### 5.5.1. Evaluación de una función

Los pasos a seguir para evaluar una función sobre el modelo formal son los siguientes:

- Seleccionar el elemento del modelo o del sistema modelado que será primer argumento de la función.
- Seleccionar Functions en Signature. Si no figura Functions en la lista, se debe a que el Sort no es primer argumento de ninguna declaración de función en la teoría.
- Seleccionar su Sort o algún supersort en Taxonomy.
- Recorrer las distintas funciones disponibles para el Sort del elemento en Label y observar su declaración y comentario en Display.
- Se pueden consultar más funciones disponibles seleccionando los distintos superSorts en Taxonomy.
- Una vez elegida la función se selecciona **evaluate** del menú contextual de Label.
- Elegir argumentos a través de los diálogos en caso de ser necesario.
- El resultado de la interpretación de la función se muestra en Display.

#### Ejemplo

Las figuras 5.6 muestran los pasos seguidos por un usuario para evaluar la función *allOperations* (del Sort *Classifier*) a la clase "Savings" del modelo "BankSystem". La figura 5.6.a retrata el momento en que se ordena la evaluación y la figura 5.6.b el resultado de la interpretación.

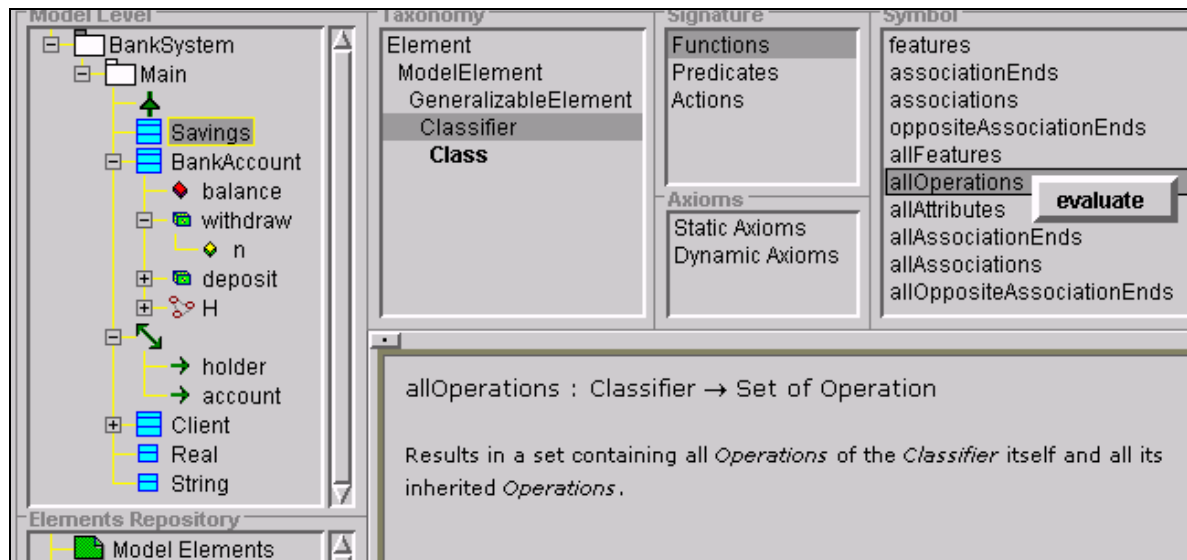


Figura 5.6.a: Evaluación de la función *allOperations*

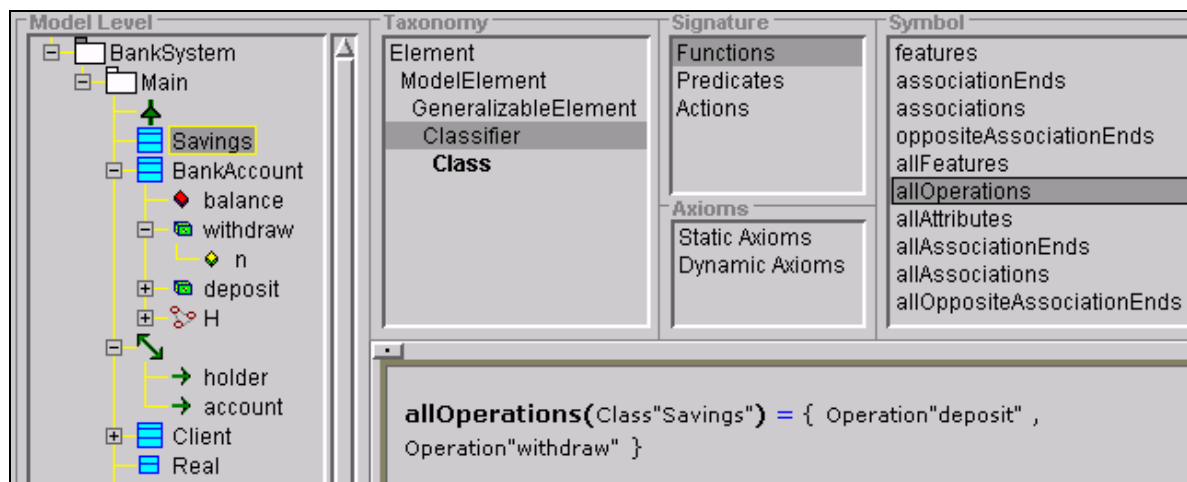


Figura 5.6.b: Resultado de la evaluación de la función *allOperations*

## 5.5.2. Evaluación de un predicado

Los pasos a seguir para evaluar un predicado sobre el modelo formal son los siguientes:

- Seleccionar el elemento del modelo o del sistema modelado que será primer argumento del predicado.
- Seleccionar Predicates en Signature. Si no figura Predicates en la lista, se debe a que el Sort no es primer argumento de ninguna declaración de predicado en la teoría.
- Consultar los predicados disponibles a través de la lista Label.
- En el cuadro Display se puede observar el tipo y un comentario sobre el predicado.
- Para acceder a predicados adicionales para evaluar sobre el elemento, se recurre a los Supersorts en Taxonomy.
- Una vez elegido el predicado, se selecciona **evaluate** del menú contextual.
- El primer argumento del predicado es el elemento seleccionado en alguno de los niveles y eventualmente pueden desplegarse ventanas de diálogo para elegir otros argumentos.
- El resultado de la evaluación del predicado se observa en Display.

## Ejemplo

Las figuras 5.7 muestran los pasos seguidos para evaluar el predicado *partOf* (Sort Classifier) con las clases "Savings" y "Client" como argumentos. La figura 5.7.a retrata el momento en que se ordena la evaluación, a partir de la selección de "Savings", y la elección del segundo argumento (una instancia del Sort Classifier). El resultado de la evaluación del predicado se muestra en la figura 5.7.b.

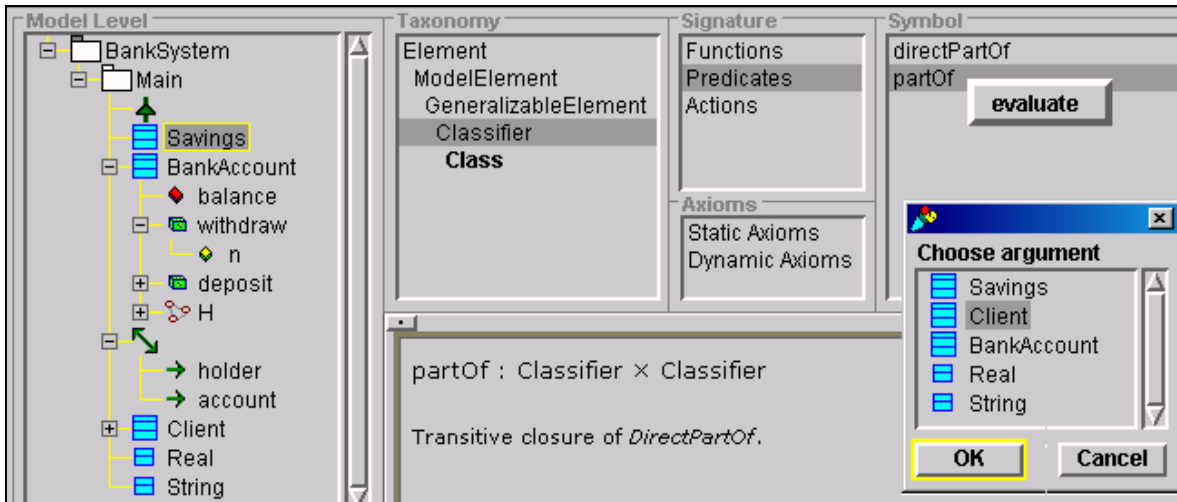


Figura 5.7.a: Evaluación del predicado *partOf*

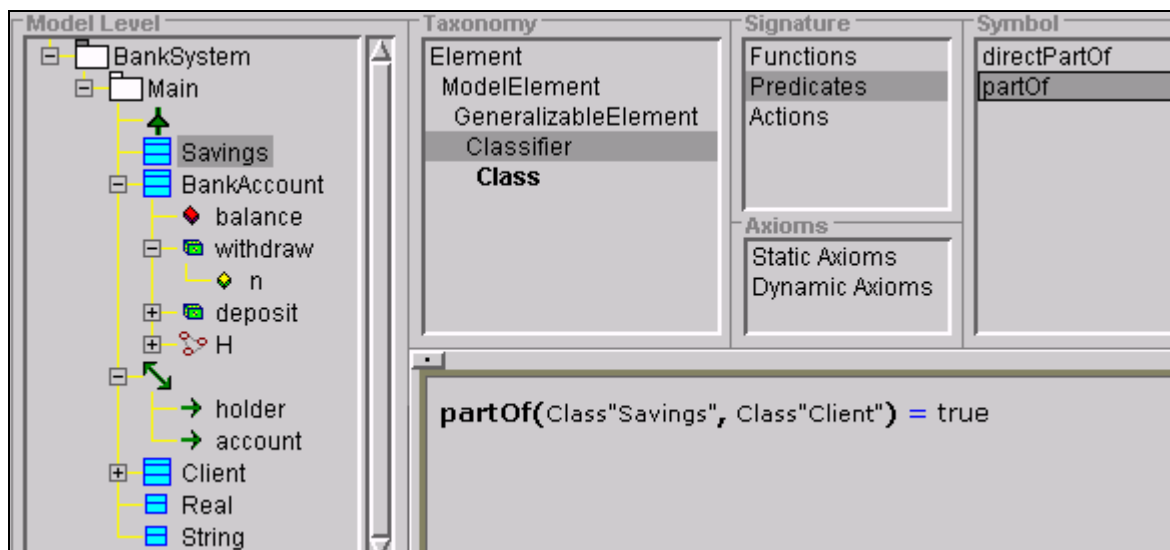


Figura 5.7.b: Resultado de la evaluación del predicado

## 5.6. Evolución del modelo formal

El modelo formal puede evolucionar a través de la ejecución de acciones. Cada acción ejecutada produce la modificación de uno o más elementos en los dos niveles. Las modificaciones pueden comprobarse en la herramienta, por modificación de la estructura de alguno (o los dos) de los árboles, a través de la información presentada en Display, o evaluando funciones y predicados sobre los elementos afectados. Pequeñas diferencias se suscitan en la ejecución de las distintas clases de acciones, por esta razón se explican separadamente con ejemplos adecuados.

Las acciones primitivas se pueden clasificar como:

**Acciones de Modificación (Modification Actions):** modifican un elemento en el modelo o en el sistema modelado.

**Acciones de Creación (Creation Actions):** agregan un nuevo elemento en el modelo o en el sistema modelado.

**Acciones de Cancelación (Cancellation Actions):** eliminan un elemento existente en el modelo, o en el sistema modelado.

## 5.6.1. Ejecución de una acción de modificación

Se deben llevar a cabo los siguientes pasos:

- Seleccionar un elemento del modelo o del sistema modelado sobre el que se desea ejecutar la acción, es decir modificar alguna de sus características.
- Seleccionar Actions en Signature. Si no figura Actions en la lista, se debe a que el Sort no es primer argumento de ninguna declaración de acción en la teoría.
- Consultar las acciones disponibles a través de la lista Label (Symbol), donde se muestra el símbolo de cada acción, y a través de Display donde se muestra la declaración de la acción y un comentario sobre qué hace.
- Para acceder a otras acciones que se puedan ejecutar sobre el elemento, se recurre a los Supersorts en Taxonomy.
- Una vez elegida la acción, se selecciona **evaluate** del menú contextual en Label.
- El primer argumento de la acción es el elemento seleccionado en alguno de los niveles y eventualmente pueden desplegarse ventanas de diálogo para elegir otros argumentos. Los diálogos varían de acuerdo al tipo de argumento necesario. Si el tipo del argumento es un Sort de la teoría, se presentan los elementos disponibles de dicho Sort. Si el tipo es un *UMLDataType* se presentan distintas opciones, por ejemplo un InputField para el Sort Name o Multiplicity, lista de opciones para *EnumerationLiterals* o *Booleans*, etc.
- El resultado de la ejecución es la modificación del elemento receptor de la acción y se refleja en alguna de las características mostradas en Display o a través de la evaluación de funciones y predicados.

### Ejemplo

Las figuras 5.8 muestran la ejecución de la acción *setAggregation* (del Sort *AssociationEnd*) sobre el elemento "holder", eligiendo el valor literal #shared como argumento. La figura 5.8.a muestra el estado del final de asociación antes de la ejecución de la acción. La figura 5.8.b retrata el momento en que se ordena la ejecución, y la elección del argumento (un literal de tipo *AggregationKind*). 5.8.c presenta el resultado de la modificación, o sea, el nuevo estado del final de asociación.

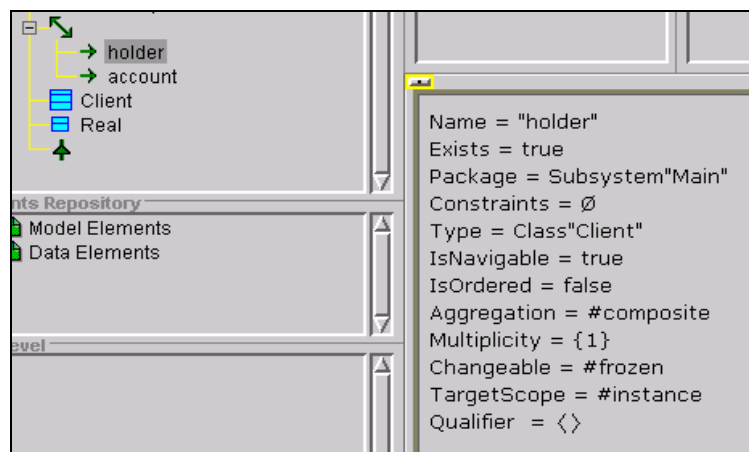


Figura 5.8.a: AssociationEnd "holder"

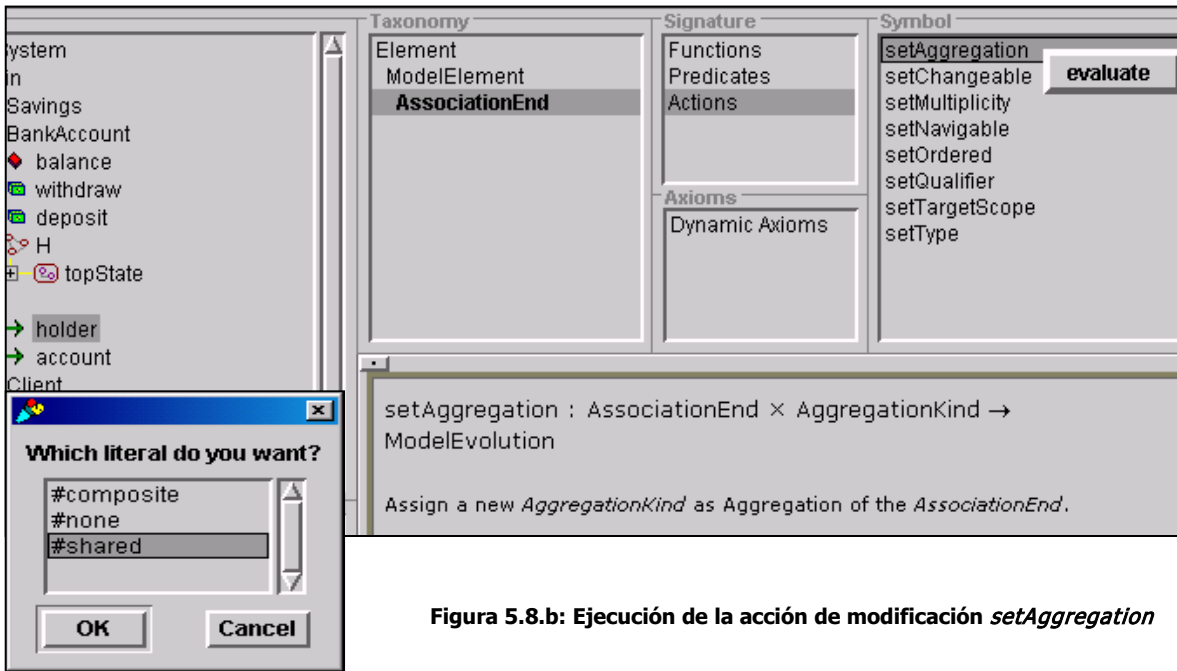


Figura 5.8.b: Ejecución de la acción de modificación *setAggregation*

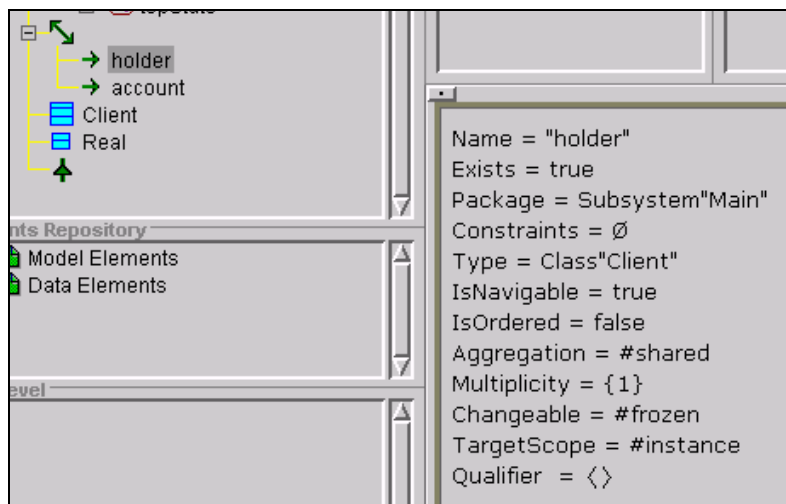


Figura 5.8.c: Resultado de la acción de modificación

## 5.6.2. Ejecución de una acción de creación

Para ejecutar este tipo de acciones, antes que nada, se debe disponer de un elemento del mismo Sort que la entidad a crear en Elements Repository. Caso contrario, se debe seleccionar **Model Level>Create Element** o **Data Level>Create Element** del menú principal, luego elegir el Sort y el nombre del nuevo elemento, que se ubicará automáticamente en Elements Repository, en la carpeta Model Elements o Data Elements, según corresponda.

Preparar el nuevo elemento para que cumpla con las precondiciones necesarias para ser incorporado al modelo y mantenga la buena formación de sus elementos. Por ejemplo: un atributo debe tener un tipo antes de ser agregado a una clase, una generalización debe tener subtipo y supertipo, etc. Las precondiciones y postcondiciones para ejecutar la acción se pueden verificar con axiomas dinámicos y se explica más adelante.

Los pasos a seguir son:

- Seleccionar el elemento del modelo que será primer argumento de la acción de creación.
- Elegir la acción siguiendo los mismos pasos de la sección anterior.



- Evaluar la acción eligiendo como segundo argumento al nuevo elemento.
- El resultado de la ejecución es la incorporación del elemento creado por la acción al nivel del modelo o al nivel de los datos. Este hecho se visualiza en el árbol representado en Model Level, o en Data Level. También es posible que se refleje en las características, del elemento receptor de la acción, mostradas en Display o a través de la evaluación de funciones.

### Ejemplo

Se agrega un subsistema “R” como subpaquete del modelo “BankSystem” mediante la acción *addSubpackage* (del Sort *Package*). Las figuras 5.9 ilustran el ejemplo. 5.9.a muestra la preparación del elemento a crear: elección del Sort *Subsystem*, atributo *name* de la entidad de modelado y su posterior ubicación en Elements Repository. En la figura 5.9.b se ordena ejecutar la acción y se elige como argumento, el subsistema “R” (Sort *Package*). Por último, 5.9.c muestra el cambio de estado del modelo producido por la transición *addSubpackage*. Se puede apreciar que el nodo que representa al subsistema “R” es un nuevo hijo del nodo raíz que representa al modelo “BankSystem”.

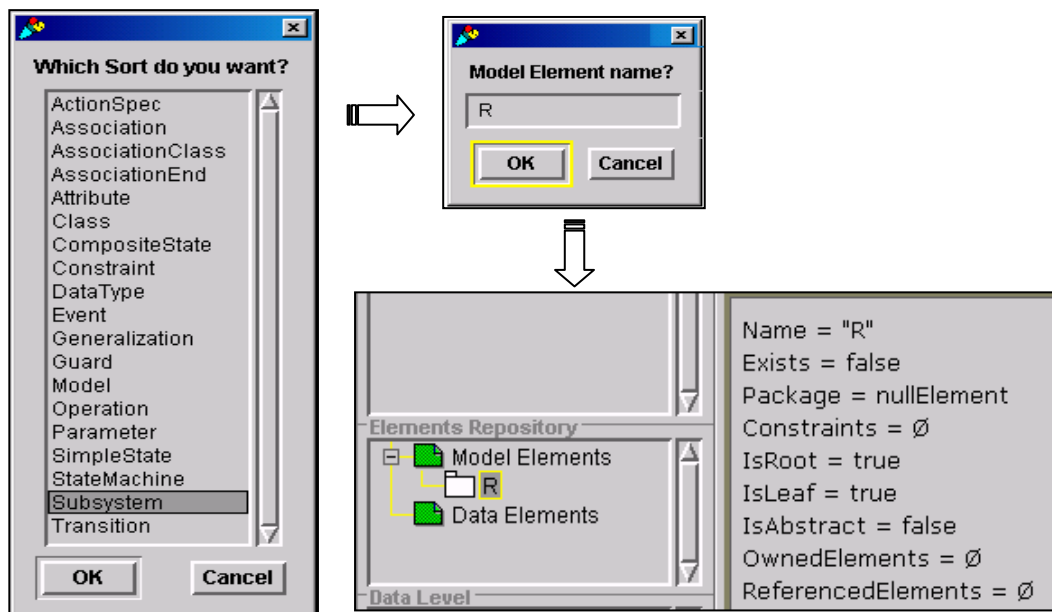


Figura 5.9.a: Preparación del subsistema "R"

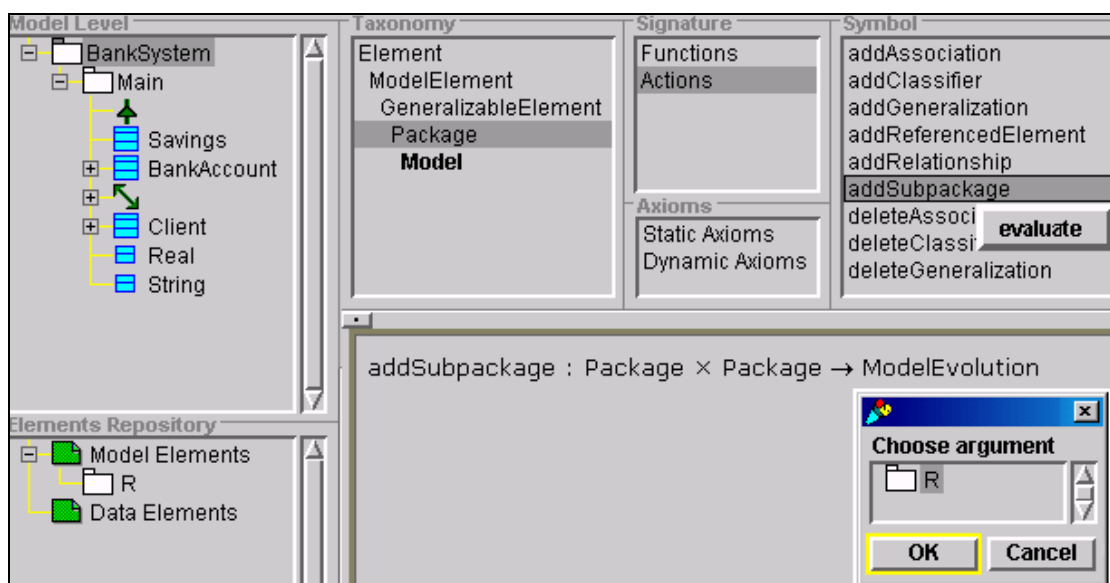


Figura 5.9.b: Ejecución de la acción de creación *addSubpackage*

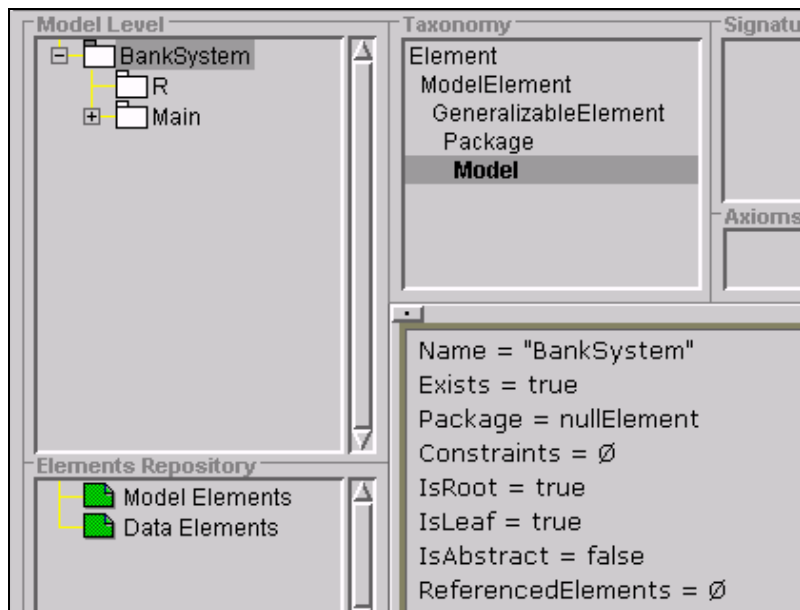


Figura 5.9.c: Resultado de la acción de creación

### 5.6.3. Ejecución de una acción de cancelación

- Seleccionar un elemento del modelo o del sistema modelado, el cual será primer argumento de la acción a ejecutar. Hay dos posibilidades: el elemento seleccionado contiene al elemento a eliminar del modelo, o el elemento seleccionado será el eliminado.
- Preparar el elemento a eliminar para que cumpla con las condiciones necesarias para ser eliminado del modelo y mantener su buena formación.
- Elegir la acción siguiendo los pasos de la sección de acciones de modificación.
- Ejecutar la acción con **evaluate** desde el menú contextual. En caso de ser necesario, se elige como argumento el elemento a eliminar.
- El resultado de la ejecución es la eliminación del elemento del nivel del modelo o del nivel de los datos. Este hecho se visualiza en el árbol representado en Model Level, o en Data Level, y el elemento eliminado se incorpora automáticamente a Elements Repository. También es posible que se refleje en las características de algunos de los elementos que estaban relacionados con el elemento eliminado, mostradas en Display o a través de la evaluación de funciones.

#### Ejemplo

A través de sucesivas transiciones del modelo formal provocadas por las acciones **addClassifier**(Model“Main”,DataType“String”), **addFeature**(Class“Client”,Attribute“firstName”) y **addFeature**(Class“Client”, Attribute“lastName”), el estado actual del modelo contiene una clase “Client” con dos atributos “firstName” y “lastName” de tipo “String”. En las figuras 5.10 se muestra la aplicación la acción de cancelación *deleteFeature* (del Sort *Classifier*) para eliminar el atributo “firstName”, de la clase “Client”, y a su vez del modelo. La figura 5.10.a corresponde al momento en que se da la orden de ejecución de la acción y se elige el atributo “firstName” como argumento de la misma. 5.10.b refleja el cambio de estado del modelo en el árbol de Model Level. Concluida la ejecución de la acción el atributo “firstName” se ubica automáticamente en la carpeta Model Elements de Elements Repository.

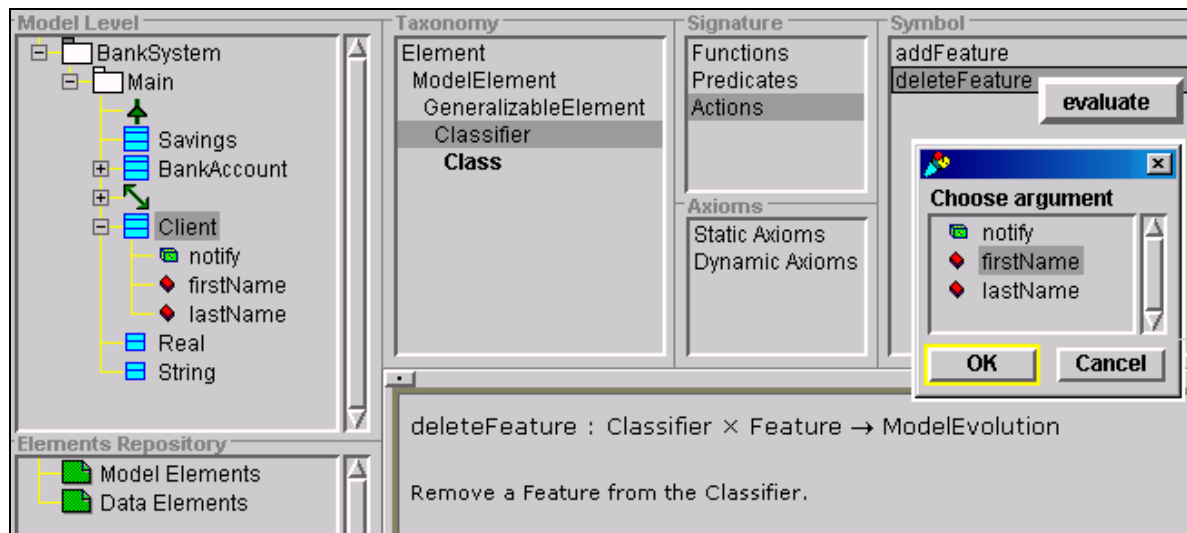


Figura 5.10.a: Ejecución de la acción de cancelación *deleteFeature*

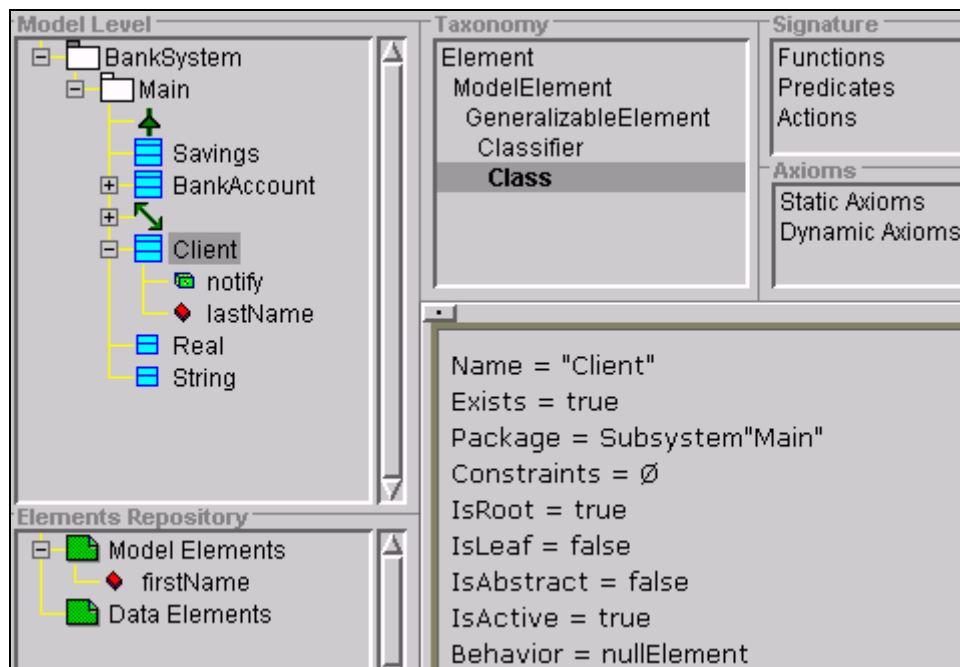


Figura 5.10.b: Resultado de la acción de cancelación

## 5.7. Evaluación de propiedades sobre el modelo

El usuario cuenta con la posibilidad de evaluar propiedades estáticas y dinámicas sobre los modelos. Para dicha tarea la herramienta está provista con bibliotecas de axiomas definidos en la teoría, que se denominan axiomas predefinidos (*predefined axioms*).

Los axiomas estáticos son organizados en Sorts, pero evalúan el modelo completo, o sea que no necesitan argumentos específicos. El axioma no pertenece al Sort, pero describe alguna propiedad que deben cumplir sus elementos (instancias del Sort) en un modelo, o un sistema modelado.

Los axiomas dinámicos evalúan un modelo antes y después de una transición de sus posibles estados, por causa de una acción primitiva. Se puede verificar si una acción es aplicable en el modelo, evaluando precondiciones. Por otro lado, las postcondiciones simulan la ejecución de una acción para verificar el efecto sobre sus argumentos y la propagación sobre otros elementos del modelo o sistema modelado.

## 5.7.1. Evaluación de un axioma estático

### Opción 1

- Seleccionar un elemento cuyo Sort, o alguno de sus ancestros, participe de la propiedad estática que se desea evaluar sobre el modelo formal.
- Seleccionar Static Axioms en la lista Axioms.
- Recorrer los axiomas clasificados en el Sort seleccionando los distintos campos de la lista Label (que contiene un título distintivo de cada axioma). En Display se observa la fórmula y un comentario del axioma seleccionado.
- La propiedad se evalúa sobre el modelo seleccionando **evaluate** del menú contextual de Label.
- El resultado de la evaluación, *true* o *false*, se muestra en Display.

### Ejemplo

Se evalúa un axioma relacionado con el Sort *Package*, que expresa que ningún paquete del modelo contiene clases o tipos de datos *-Classifiers-* con el mismo nombre. La figura 5.11.a muestra la selección y evaluación del axioma estático “Classifiers’ name”. Como puede observarse, el elemento seleccionado es el subsistema “Main”, pero el resultado sería el mismo seleccionando cualquier entidad del modelo desde donde se pueda acceder al Sort *Package*. La figura 5.11.a muestra el resultado de la evaluación del axioma.

The screenshot shows the following details:

- Model Level:** A tree view with 'BankSystem' as the root, containing 'Main', 'Savings', 'BankAccount', 'Client', 'Real', and 'String'. 'Main' is selected.
- Taxonomy:** A list of elements: 'ModelElement', 'GeneralizableElement', 'Package', and 'Subsystem'. 'Subsystem' is selected.
- Signature:** A list of categories: 'Functions', 'Actions', and 'Axioms'. Under 'Axioms', 'Static Axioms' and 'Dynamic Axioms' are listed. 'Static Axioms' is selected.
- Title:** A list of labels: 'Classifiers' name.', 'Packages' name.', 'Associations' name.', 'Generalizations' supertype', 'Attributes' type', 'AssociationEnds' type', 'StateMachines' context', 'Classifiers' behavior', and 'Lyfe dependency'. 'Classifiers' name.' is selected, and an 'evaluate' button is visible next to it.
- Display:** Shows the text "In a Package the Classifier names are unique." and the logical formula:
 
$$\forall p:\text{packages}(M) \ \forall c1,c2:\text{classifiers}(M) \ c1 \in \text{contents}(p) \wedge c2 \in \text{contents}(p) \wedge \text{name}(c1)=\text{name}(c2) \rightarrow c1=c2$$

Figura 5.11.a: Evaluación del axioma estático "Classifiers' name"

The screenshot shows the following details:

- Model Level:** Same as Figure 5.11.a, with 'Main' selected.
- Taxonomy:** Same as Figure 5.11.a, with 'Subsystem' selected.
- Signature:** Same as Figure 5.11.a, with 'Static Axioms' selected.
- Title:** Same as Figure 5.11.a, with 'Classifiers' name.' selected.
- Display:** Shows the word **true** in blue text, indicating the successful evaluation of the axiom.

Figura 5.11.b: Resultado de la evaluación del axioma estático

## Opción 2

- Abrir el Browser de axiomas estáticos desde el menú principal de la herramienta, eligiendo **Predefined Axioms>Browse Static Axioms**.
- Seleccionar el Sort adecuado, en la lista de todos los Sorts de la teoría que tienen axiomas estáticos asociados.
- Buscar el axioma requerido, a través de Title. En Formula&Comment se muestra la información restante de cada axioma.
- El axioma seleccionado se evalúa con el botón **evaluate**.
- El resultado aparece en Formula&Comment.
- Con el botón **Update** se vuelven a mostrar los datos del axioma seleccionado.
- Con el botón **Close** se cierra el Browser.

## Ejemplo

Como ejemplo se evalúa un axioma relacionado con el Sort *GeneralizableElement*, que expresa que en el modelo no se permite la herencia circular entre instancias de dicho Sort. La figura 5.12 muestra la selección del Sort “GeneralizableElement” y del axioma estático “Circular inheritance”, y el resultado de la evaluación sobre el modelo formal instanciado en la aplicación. El resultado se muestra en la misma figura por simplicidad, pero en la realidad la herramienta reemplaza el texto de Formula&comment por *true*.

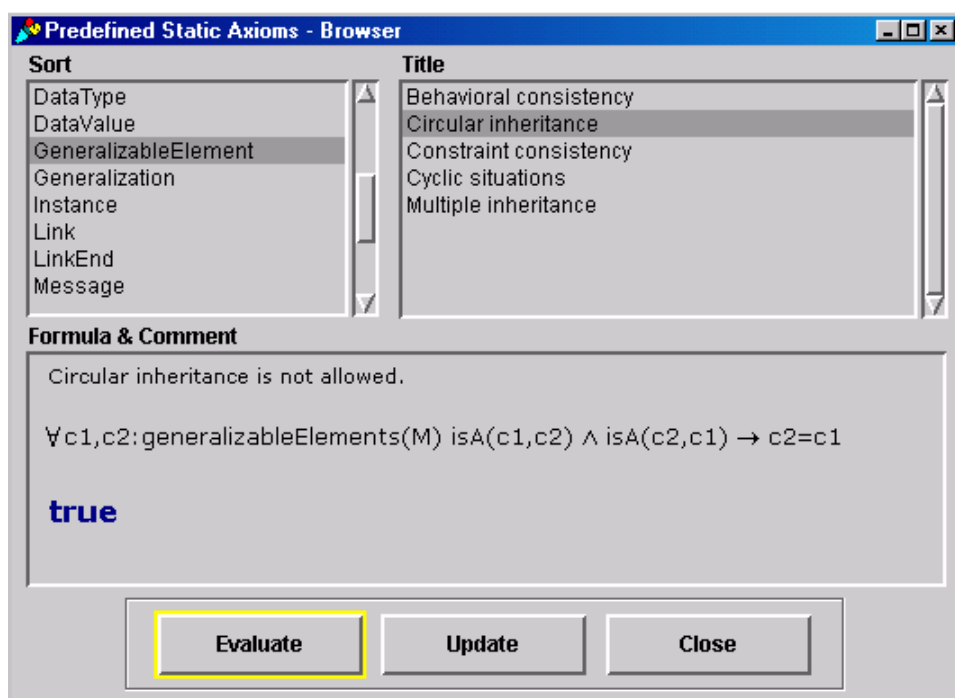


Figura 5.12: Evaluación del axioma estático "Circular inheritance"

## 5.7.2. Evaluación de un axioma estático

### Opción 1

- Seleccionar un elemento del modelo o del sistema modelado, que será el source de la acción involucrada en el axioma dinámico.
- Seleccionar Dynamic Axioms en Axioms. Si no figura Dynamic Axioms en la lista, se debe a que el Sort no es source de ninguna declaración de acción en la teoría, o las acciones son triviales y no necesitan especificarse con preconditions y postconditions.

- Consultar los axiomas disponibles a través de la lista Label, donde, como título por defecto, se muestra el símbolo de cada acción entre los símbolos “<” y “>” para las precondiciones y entre “[“ y “]” para las postcondiciones. En Display se visualiza a las fórmulas disgregadas en partes significativas que merezcan ser comentadas, con la intención de lograr mayor legibilidad.
- Para acceder a otros axiomas que se puedan evaluar sobre el elemento seleccionado, se recurre a los Supersorts en Taxonomy.
- Con un axioma dinámico escogido, se selecciona **evaluate** del menú contextual en Label.
- El primer argumento de la acción involucrada en el axioma es el elemento seleccionado en alguno de los niveles y eventualmente pueden desplegarse ventanas de diálogo para elegir otros argumentos necesarios para simular la acción.
- El resultado (*true* o *false*) se muestra en Display, y no se produce ningún cambio en los distintos niveles (modelo y datos). El modelo permanece en el estado anterior a la ejecución del axioma dinámico.

### Ejemplo 1

Como primer ejemplo se evalúa el axioma dinámico que expresa las precondiciones necesarias para agregar un atributo a una clase o a un tipo de dato (instancias del Sort *Classifier*). En este caso se desea saber si están dadas las condiciones para agregar el atributo “lastName” de tipo “String” a la clase “Client”, y la validez del modelo formal se conserve. La figura 5.13.a muestra la selección del axioma “<addFeature(c,f1)>” correspondiente al Sort *Classifier*, la orden de evaluación y la selección del argumento. El resultado se muestra en la misma figura por simplicidad. Es evidente que la clase ya posee un atributo con el nombre “lastName”, en consecuencia el resultado es *false*. Si se agregase con una acción, el modelo perdería su buena formación. El resultado se muestra en la misma figura por simplicidad.

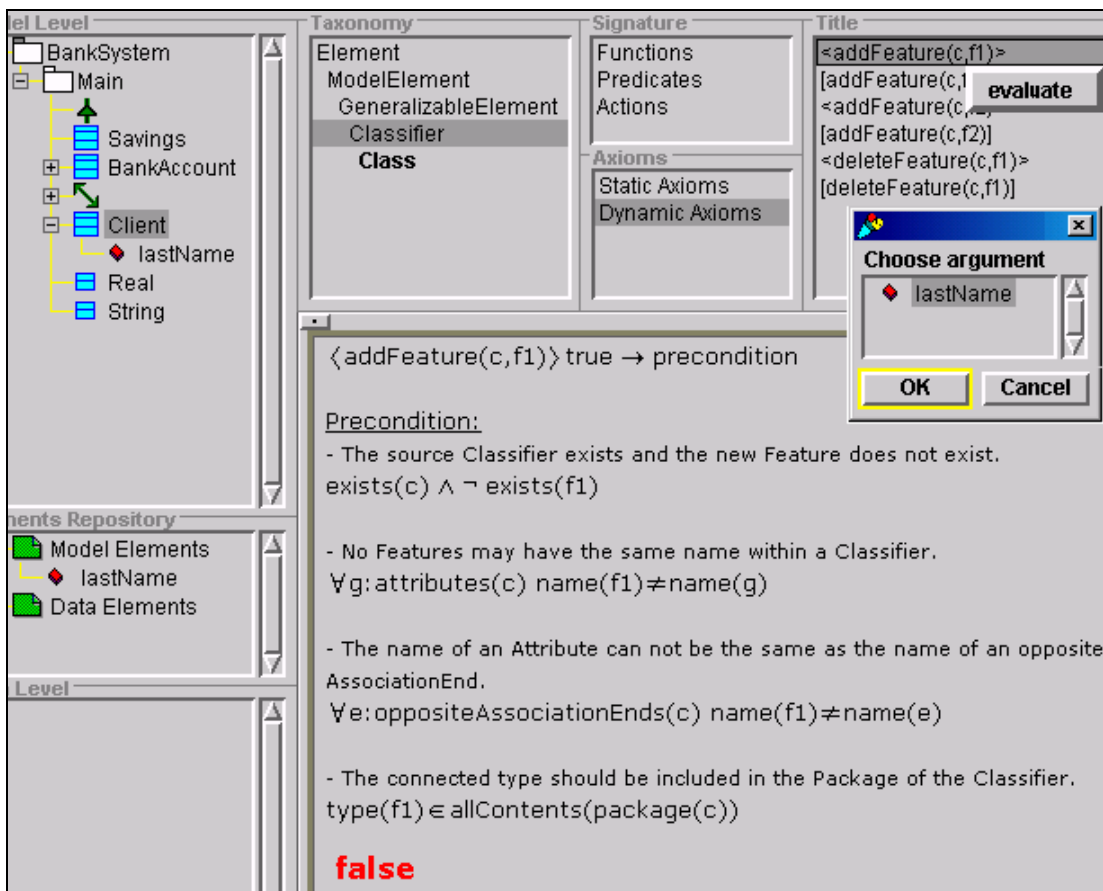


Figura 5.13.a: Evaluación de la precondición <addFeature(c,f1)>

## Ejemplo 2

Previo a llevar a cabo el segundo ejemplo se modificó el nombre del atributo mediante la acción `setName(Class“Client”, Attribute“firstName”)` accedida desde el Sort *ModelElement*. En la figura 5.13.b se muestra la evaluación de la precondition “`<addFeature(c,f1)>`” con la misma clase y el atributo renombrado como argumentos. Ahora, el resultado de la evaluación es *true* y significa que las condiciones de aplicabilidad de la acción se cumplen.

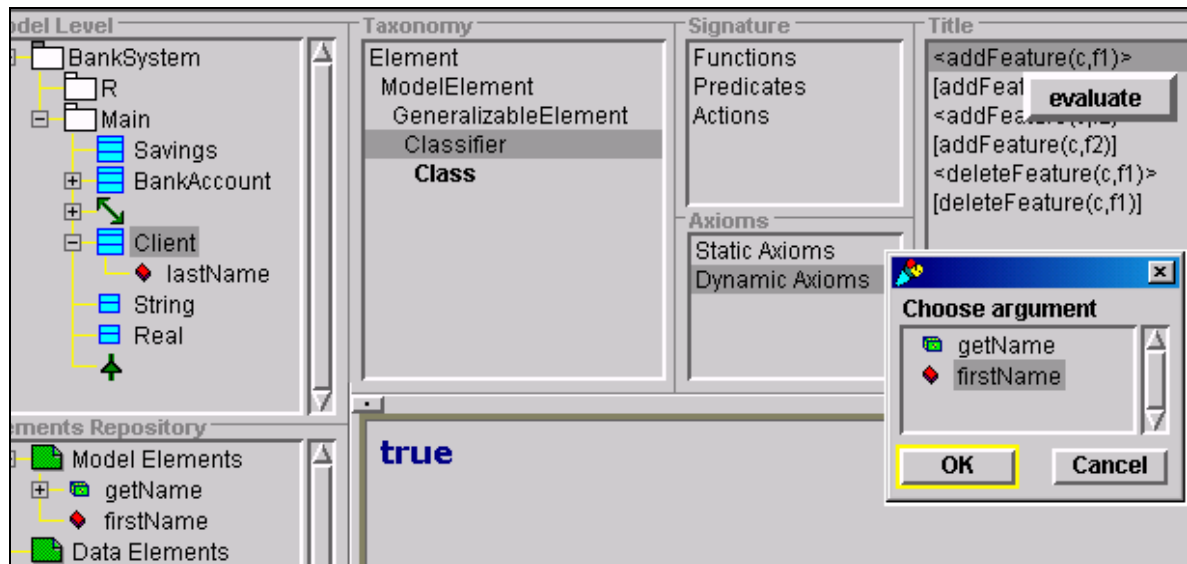


Figura 5.13.b: Segunda evaluación de la precondition `<addFeature(c,f1)>`

## Opción 2

- Abrir el Browser de axiomas dinámicos desde el menú principal, seleccionando **Predefined Axioms>Browse Dynamic Axioms**.
- Seleccionar el Sort adecuado, o un ancestro, del elemento que será el source de la acción especificada por el axioma dinámico.
- Buscar el axioma requerido en la lista Title. En Formula&Comment se muestra la información restante de cada axioma (ídem Display).
- Con el botón **Evaluate** se evalúa el axioma elegido.
- El resultado (*true* o *false*) aparece en Formula&Comment.
- Con el botón **Update** se vuelven a mostrar los datos del axioma seleccionado.
- Con el botón **Close** se cierra el Browser.

## Ejemplo

Se simula la incorporación de la operación “getName” a la clase “Client”, y se evalúan los efectos sobre el modelo formal. La operación contiene un parámetro “m” de tipo “String”. La acción simulada es `addFeature(Class“Client”, Operation“firstName”)`. En la figura 5.14 se observa la ventana del Browser con la selección del Sort *Classifier* y del axioma dinámico “[addFeature(c,f2)]”, y se resumen la elección del primer -la clase- y el segundo -la operación- argumento, y el resultado de la evaluación.

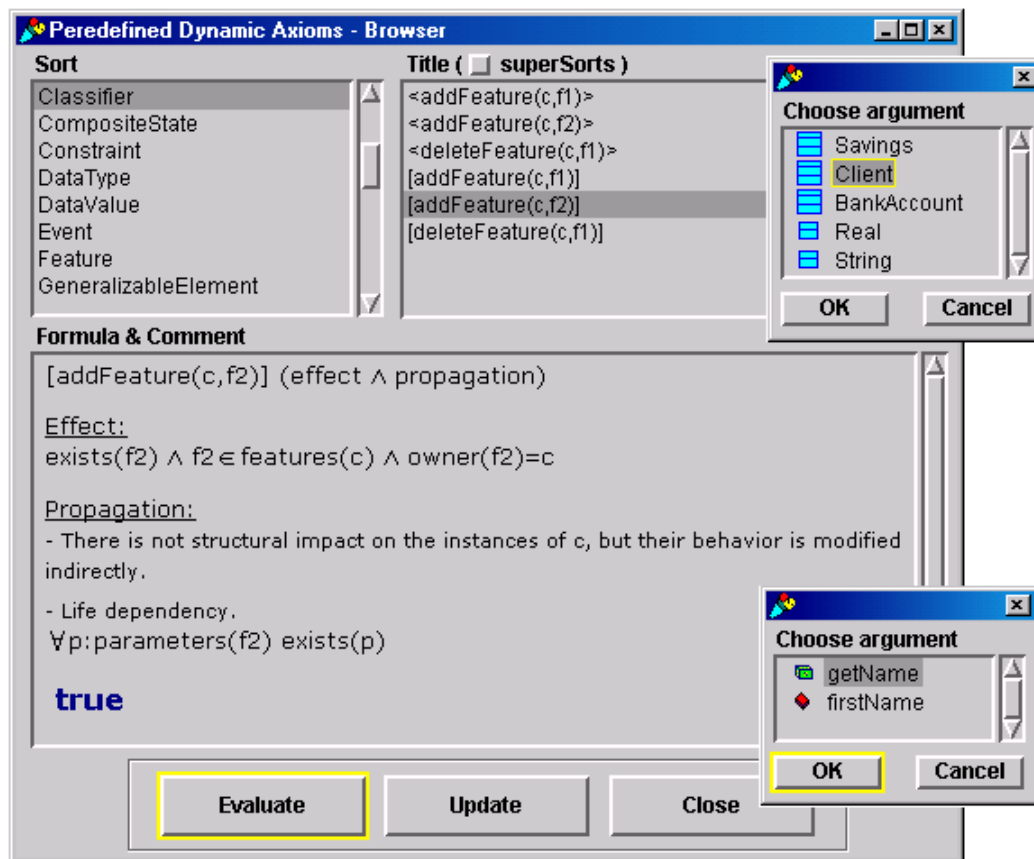


Figura 5.14: Evaluación de la postcondición  $[addFeature(c, f2)]$

## 5.8. Edición de axiomas

Un usuario puede crear sus propias bibliotecas de propiedades estáticas y dinámicas, para evaluarlas posteriormente sobre instancias de determinados modelos. Conjuntos de propiedades relacionadas pueden ser salvados en un archivo, y luego recuperados de acuerdo a las necesidades o características del modelo presente en la aplicación.

### 5.8.1. Editor de axiomas estáticos

Al editor de axiomas estáticos se accede desde el menú principal, eligiendo **User Axioms>Edit Static Axiom**. La ventana de trabajo desplegada se muestra en la figura 5.15. Sus principales componentes se detallan a continuación.

**Title** : en este campo se escribe un título descriptivo para el axioma.

**Comment** : se describe en lenguaje natural la propiedad expresada en la fórmula.

**Variable M** : es una variable especial cuyo valor por defecto es el modelo formal instanciado en la herramienta, en el momento en que se evalúa la propiedad. Su valor se modifica solamente al instanciarse un nuevo modelo.



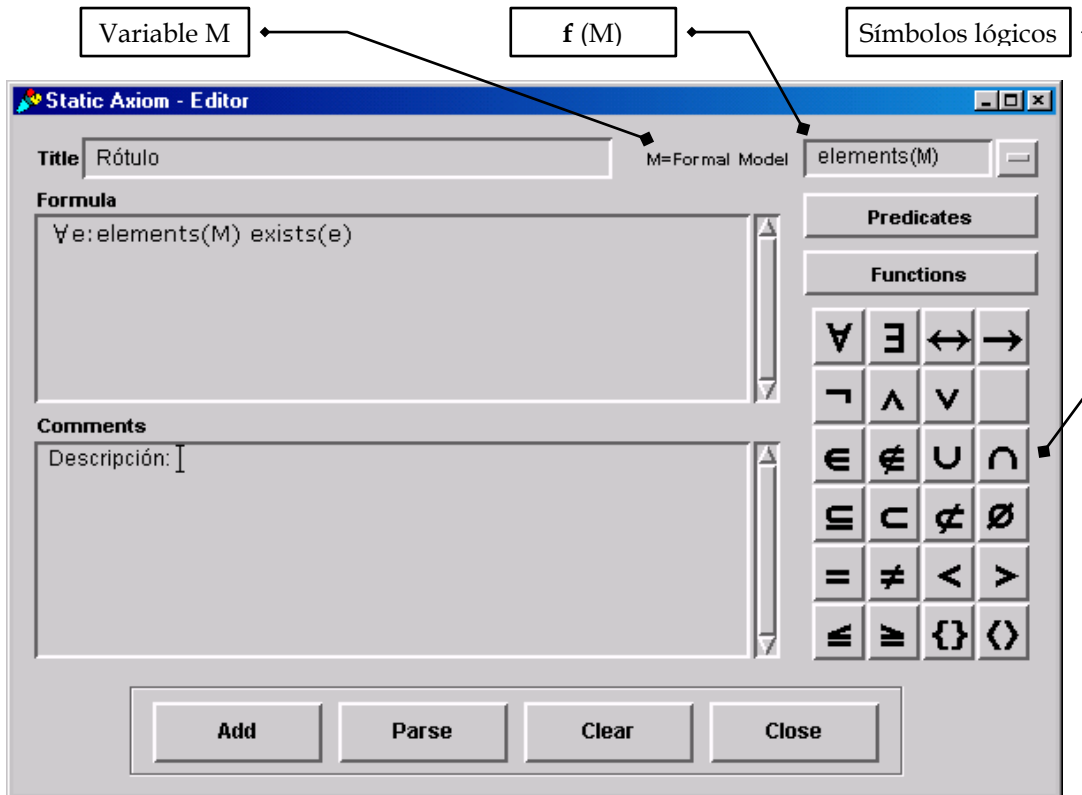
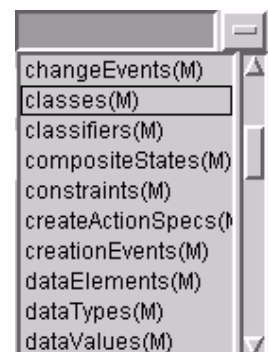


Figura 5.15: Editor de axiomas estáticos

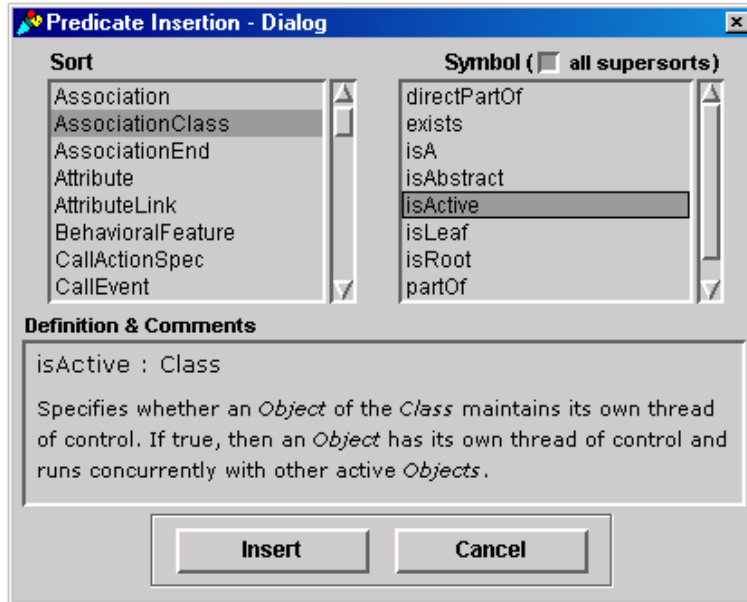
**Formula** : se escribe el axioma en términos de lógica de 1er orden. La fórmula debe estar cuantificada con elementos del modelo. El orden de precedencia, de mayor a menor, de los símbolos de una fórmula se especifica en la siguiente tabla:

1	Términos (funciones, variables y constantes)
2	Predicados
3	$\neg$
4	$\wedge, \vee$
5	$\leftrightarrow, \rightarrow$
6	$\forall, \exists$

**f(M)** : son funciones cuya interpretación resulta en el conjunto de todas los elementos de un Sort determinado, presentes en el modelo formal contenido en la variable M. Son necesarias para cuantificar las instancias de un Sort de la teoría. Por ejemplo, si se quiere especificar una propiedad sobre las clases, entonces *classes(M)* representa el conjunto finito de clases presentes en el estado del modelo al evaluarse la propiedad.





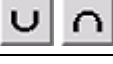




**Predicates** : facilita la escritura de predicados en la fórmula. El botón abre un diálogo que permite consultar todos los predicados de la teoría. Pueden clasificarse por Sort, o por Sort y todos sus ancestros. Es posible ver su declaración de tipos y un comentario. Resumiendo, por cada Sort seleccionado se muestran los predicados aplicables a sus instancias. Mediante **Insert** el símbolo del predicado es insertado en el campo Formula del editor. La siguiente figura ilustra esta ventana de diálogo:



**Functions** : facilita la escritura de funciones en la fórmula. El botón **Functions** abre un diálogo que permite consultar todas las funciones de la teoría. Su utilización es similar a **Predicates**.

**Símbolos lógicos** : grupo de botones que permiten insertar en el campo Formula algunos de los símbolos lógicos necesarios para editar una expresión. El símbolo es insertado en la última posición del cursor en dicho campo. En la siguiente tabla se detalla la función de cada uno de los botones:

	Insertan los símbolos de cuantificador universal y existencial
	Insertan los símbolos conectivos proposicionales bicondicional, implicación, negación, conjunción y disyunción
	Insertan las constantes nullElement y conjunto vacío.
	Insertan símbolos de predicado de pertenencia e inclusión en conjuntos
	Inserta los símbolos de función de unión e intersección de conjuntos
	Insertan símbolos de predicado de comparación
	Insertan los símbolos necesarios para definir conjuntos y secuencias por extensión

**Add** : agrega el axioma estático editado a la biblioteca *User Axioms* de la aplicación.

**Parse** : verifica la sintaxis de la expresión en Formula e indica si es una fórmula bien formada (f.b.f.). Es conveniente realizarlo antes de ejecutar **Add**.

**Clear** : elimina los datos ingresados por el usuario.

**Close** : finaliza la tarea del editor axiomas.

### Ejemplo

Si el lenguaje orientado a objetos a utilizar en la etapa de implementación no soporta herencia múltiple, el usuario puede crear un axioma estático que exprese que en el modelo sólo se permite herencia simple. El aspecto del editor durante la edición del nuevo axioma lo ilustra la figura 5.16.

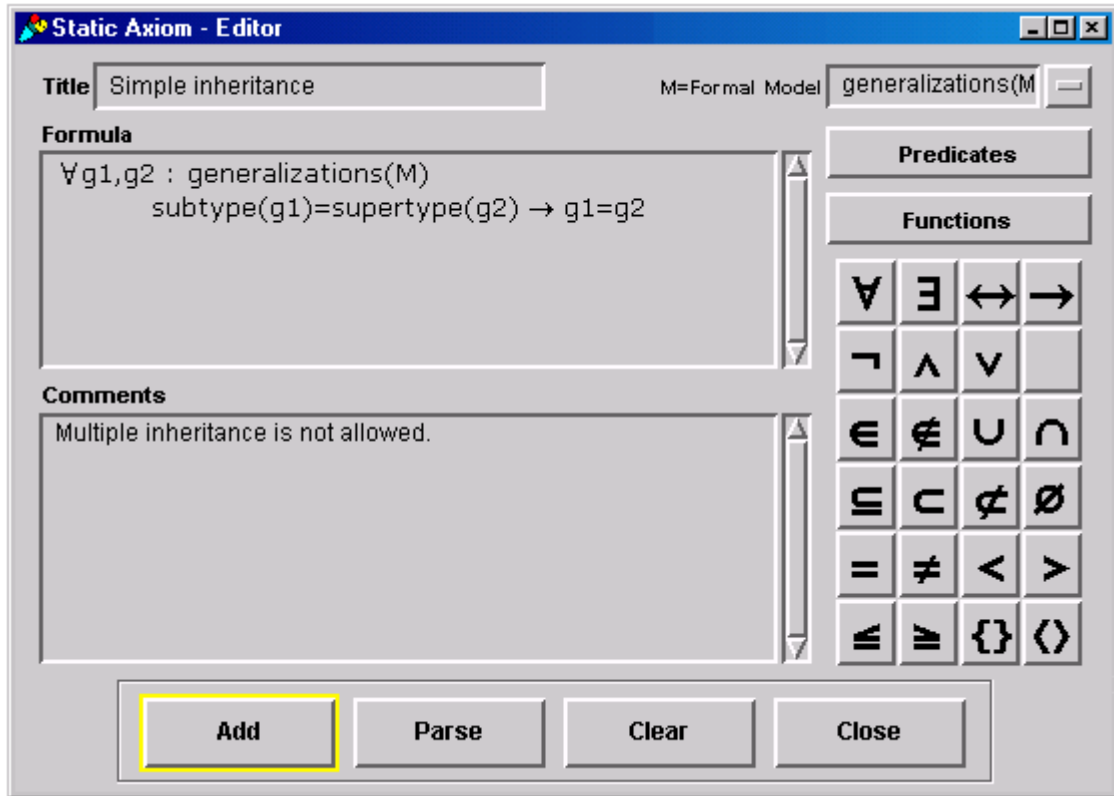


Figura 5.16: Edición de un axioma estático

## 5.8.2. Editor de axiomas dinámicos

### 5.8.2.1. Editor de precondiciones

Al editor de precondiciones se puede acceder desde el menú principal, al elegir **User Axioms>Edit Dynamic Axiom>Edit Precondition**. La figura 5.17 presenta el aspecto de la ventana del editor.

**Action** : El botón **Action** abre un diálogo que permite consultar todos las acciones de la teoría. Pueden clasificarse por Sort, o por jerarquías de Sort. Es posible ver su declaración de tipos y un comentario. Por cada Sort seleccionado se muestran los acciones ejecutables sobre sus instancias. Mediante **Insert**, el símbolo de la acción es insertado en el campo Action.

**Add Precondition** : agrega la expresión de Formula y el comentario, como una condición más del axioma dinámico.

**Parse** : verifica la sintaxis de la expresión en Formula, que describe a una precondición y no a todo el axioma.

**Add** : agrega el axioma dinámico a la biblioteca *User Axioms* de la aplicación.

Los campos Title, Formula, Comment, f(M), y los botones **Predicates**, **Functions**, **Clear**, **Close** y los iconos de símbolos lógicos cumplen la misma función que en el editor de axiomas estáticos.

### Edición de una precondición

- Con el botón **Action** se elige la acción para la cual se especificarán las precondiciones de ejecución.
- Ingresar las variables para la acción en el campo Action.
- El Title del axioma se describe automáticamente con <nombreDeAcción(argumentos.)>, aunque se puede modificar posteriormente.
- Se ingresan iterativamente las condiciones, con sus respectivos comentarios, presionando el botón **Add Precondition** . Conviene verificarlas individualmente con **Parse**  a medida que se incorporan al axioma dinámico.
- Agregar el axioma, mediante **Add**  a la biblioteca *User Axioms* de la aplicación.

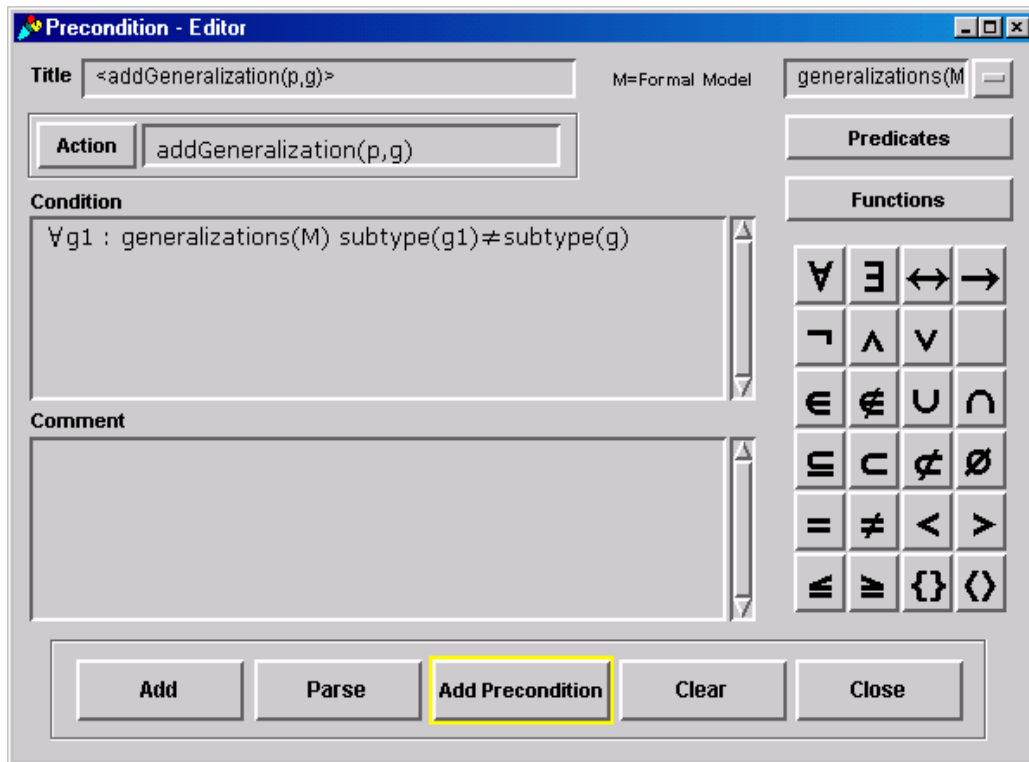


Figura 5.17: Editor de precondiciones

### 5.8.2.2. Editor de postcondiciones

Al editor de postcondiciones se puede acceder desde el menú principal, al elegir **User Axioms>Edit Dynamic Axiom>Edit Postcondition**. La figura 5.18 presenta el aspecto de la ventana del editor.

**Add Effect:** agrega la expresión descrita en Formula, como un efecto de la *postcondition*.

**Add Propagation:** agrega la expresión descrita en Formula, como una propagación de la ejecución de la acción.

### Edición de una postcondición

- Con el botón **Action** se elige la acción para la cual se especificarán el efecto y la propagación de ejecución de la acción.
- Ingresar las variables para la acción en el campo Action.
- El Title del axioma se describe automáticamente con [nombreDeAcción(argumentos)], aunque se puede modificar posteriormente.

- Se ingresan iterativamente las formulas que describen los efectos (de aplicar la acción), con sus respectivos comentarios, presionando **Add Effect**. Conviene verificarlos individualmente con **Parse** a medida que se incorporan al axioma dinámico.
- Se ingresan iterativamente las fórmulas que describen la propagación (de aplicar la acción) con sus respectivos comentarios, a través de **Add Propagation**. Conviene verificarlos individualmente con **Parse**, a medida que se incorporan al axioma.
- Agregar el axioma, mediante **Add**, a la biblioteca *User Axioms* de la aplicación.

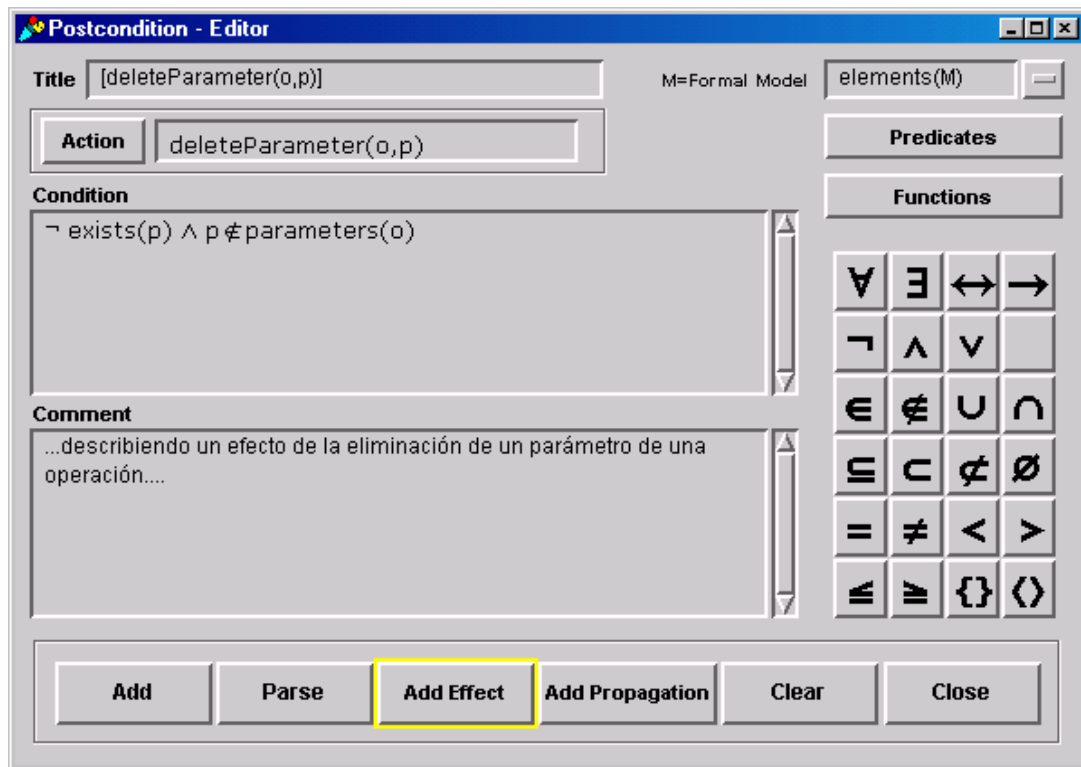


Figura 5.18: Editor de postcondiciones

### 5.8.3. Utilizando los axiomas editados

#### 5.8.3.1. Consulta y evaluación de axiomas

Se accede al Browser de axiomas editados por el usuario eligiendo **User Axioms>Browse**. Marcando el tipo de axioma -Static Axioms o Dynamic Axioms-, se puede consultar cada uno de ellos seleccionándolos sobre la lista Title. En Formula&Comment se muestran la fórmula y sus comentarios.

Con un axioma seleccionado, se pueden llevar a cabo las siguientes acciones:

**Evaluate** : evaluar la propiedad sobre el modelo formal presente en la herramienta.

**Modify** : modificar el axioma (sólo disponible para los axiomas estáticos).

**Remove**: eliminar el axioma de la biblioteca de propiedades actual.

#### Ejemplo

A través del Browser se consulta y evalúa el axioma estático “Simple inheritance”, ingresado en el ejemplo anterior a la biblioteca de axiomas del usuario. En la figura 5.19 se muestra el aspecto del Browser en el momento de la evaluación. El resultado, por simplicidad, se observa en la misma ventana.

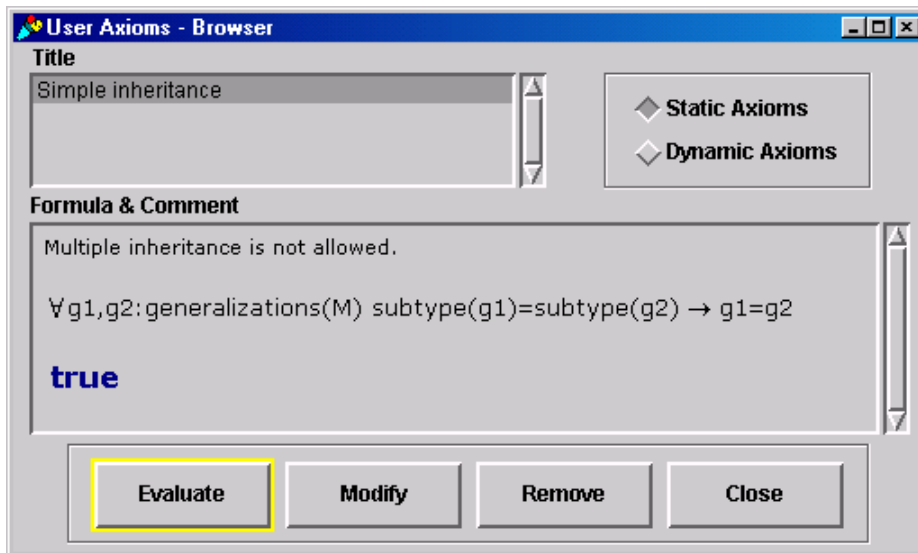


Figura 5.19: Evaluación de un axioma de usuario

### 5.8.3.2. Manejo de bibliotecas de axiomas

#### Guardando bibliotecas de axiomas

Seleccionando **User Axioms>Save As...** se despliega una ventana de diálogo para que el usuario elija nombre y ubicación de un archivo con extensión `.axm`. Dicho archivo guardará todos los axiomas estáticos y dinámicos contenidos en la biblioteca de axiomas del usuario de la aplicación.

#### Cargando bibliotecas de axiomas

Seleccionando **User Axioms>Load..** se despliega una ventana de diálogo para que el usuario elija nombre y ubicación de un archivo con extensión `.axm`. Los axiomas contenidos en el archivo son agregados a la biblioteca de axiomas del usuario de la aplicación.

#### Eliminando la biblioteca de axiomas

Si se selecciona **User Axioms>Reset...** se eliminan todos los axiomas contenidos actualmente en la biblioteca de propiedades creadas por usuarios.

## 5.9. Persistencia de los estados de un modelo formal

#### Salvando el estado de un modelo

Se puede guardar el estado actual del modelo presente en la aplicación. Seleccionando **File>Save Model** se guarda en el archivo cuyo nombre aparece en la barra de título de la aplicación. Seleccionando **File>Save Model As...** se abre una ventana de diálogo para elegir nombre y ubicación del archivo con extensión `.dlm`.

#### Recuperando el estado de un modelo

Se recupera un modelo formal en un estado determinado seleccionando **File>Load Model...**, y eligiendo el archivo con extensión `.dlm` requerido.

## 5.10. Evolución en el nivel de los datos

A partir del modelo del sistema bancario utilizado como ejemplo en el resto del capítulo, y en un estado donde aún el axioma de inicialización de la teoría se cumple (es decir no existen entidades modeladas) se expone la creación de algunas instancias.

### 5.10.1. Creación de un objeto -Object-

Se describe la creación de una instancia del Sort *Object*, utilizando la clase “Savings” del modelo del sistema bancario:

- Seleccionar **Data Level>Create Element>Object** para generar una entidad modelada “an Object”, perteneciente al Sort *Object*. El nuevo elemento se ubica automáticamente en la carpeta Data Elements de Elements Repository. Ver figura 5.20.a.
- Seleccionar el source de la acción de creación, la clase “Savings”.
- Seleccionar la acción “newObject” del Sort *Class*.
- Ejecutar la acción y elegir el objeto “an Object” como argumento. Ver figura 5.20.b.

El efecto de la acción, es la incorporación del objeto al nivel de los datos con sus slots inicializados. En el nivel del modelo, la clase “Savings” hereda el atributo “balance” de la clase “BankAccount”. Por esta razón el objeto creado contiene un slot “balance” con valor inicial cero. El estado actual del objeto está dado por el estado simple “credit” que pertenece a la máquina de estados “H”, que define el comportamiento de las instancias de la clase “Bank Account”. El *mailbox* privado del objeto no tiene mensajes.

En el árbol de Data Level, el nodo que representa al nuevo objeto posee un subnodo correspondiente a su único slot, cuyo Sort es *AttributeLink*. Algunos atributos se pueden visualizar en Display, el resto mediante evaluación de funciones y predicados. Ver figura 5.20.c.

**Identidad de las entidades modeladas:** Las entidades modeladas no poseen el atributo *name* como en el caso de las entidades de modelado. Para diferenciarlas a nivel identidad, por ejemplo dos objetos de la clase “Savings” con el mismo valor en su *slot*, se les puede asignar una identificación mediante **Data Level>Identification** del menú principal. Más Adelante se cambiará la identificación del objeto recientemente creado “aSavings” por “S1”.

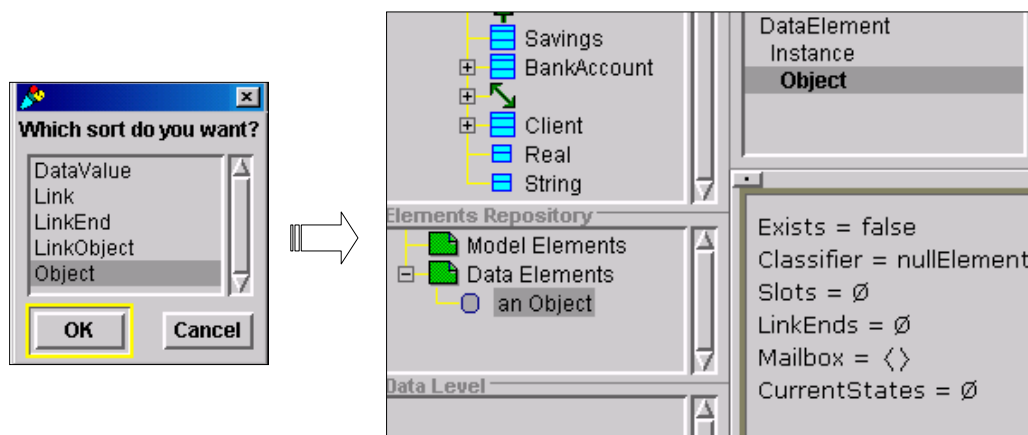


Figura 5.20.a: Preparación del objeto

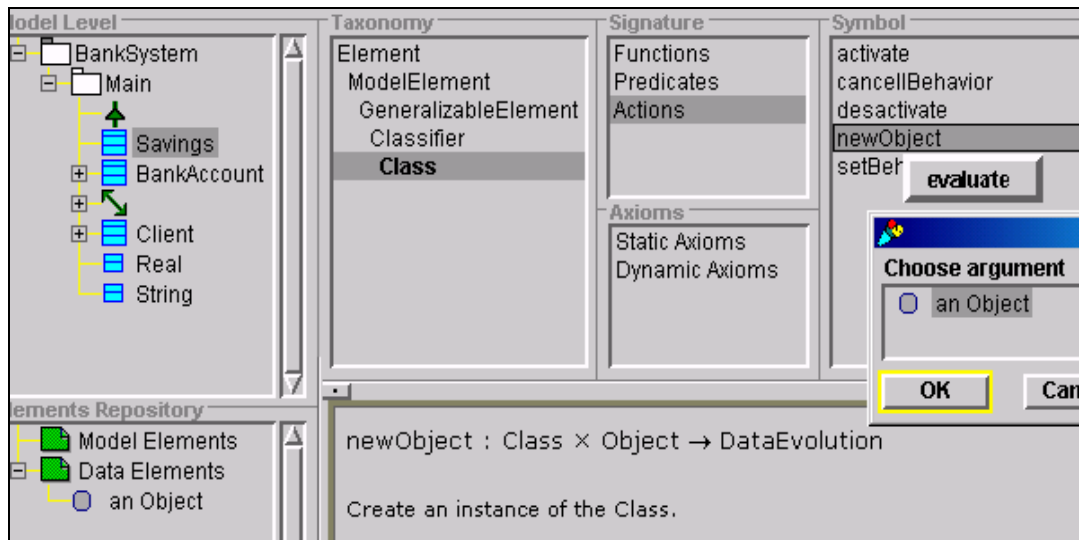


Figura 5.20.b: Ejecución de la acción de creación *newObject*

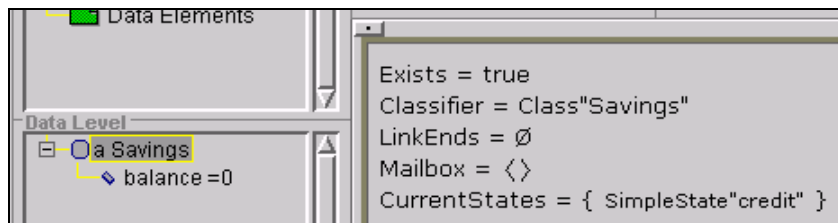


Figura 5.20.c: Efecto de la acción de creación *newObject*

## 5.10.2. Creación de un valor -DataValue-

En este caso se detalla la creación de un valor instancia del tipo de datos “Real”:

- Seleccionando **Data Level>Create Element>DataValue**, se incorpora una entidad modelada con identificación “a DataValue”, perteneciente al Sort *DataValue*. El ítem se ubica en la carpeta Data Elements de Elements Repository.
- Seleccionar como source de la Creation Action, el tipo de dato “Real”.
- Seleccionar la acción “newDataValue” del Sort *DataValue*.
- Ejecutar la acción, e ingresar los argumentos: el objeto “a DataValue” y el número real 50. El segundo argumento debe ser un literal evaluable por el compilador Smalltalk, por ejemplo enteros, fracciones, fechas, strings, symbols, etc. Figura 5.21.a.

El efecto de la acción, es la incorporación del valor 50 de tipo Real al nivel de los datos. Su representación en Data Level y en Display se presenta en la figura 5.21.b.

Siguiendo los métodos vistos de creación de objetos y valores, se agregan dos datos adicionales. El nivel de los datos evoluciona con la instanciación de la clase “Client” y del tipo “Real”. Las instancias creadas son el objeto con identificación “John” y el valor 100.



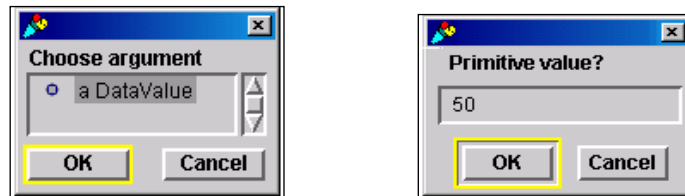
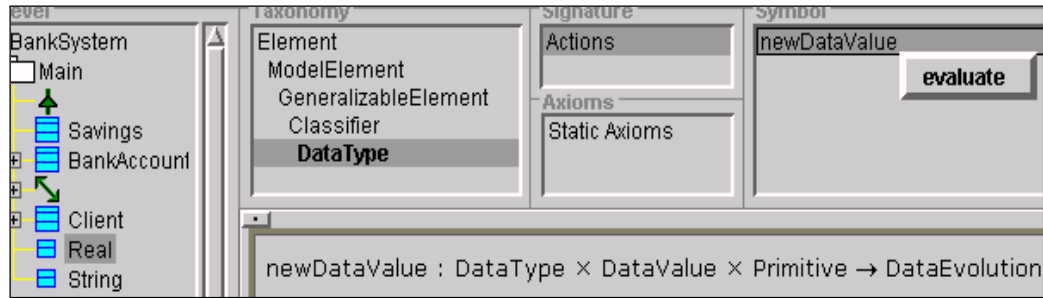


Figura 5.21.a: Ejecución de la acción de creación *newDataValue*

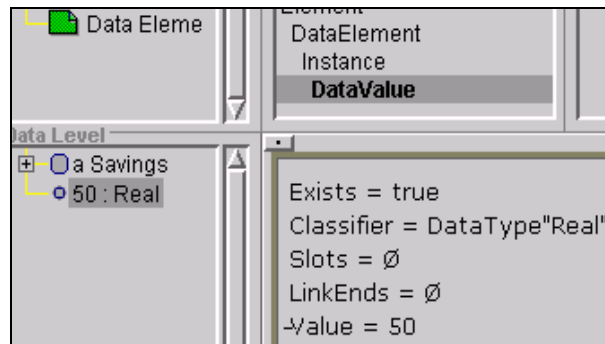


Figura 5.21.b: Creación del valor

### 5.10.3. Creación de una conexión -Link-

En los siguientes pasos se describe la creación de una instancia de la asociación entre las clases “BankAccount” y “Client”. Se creará un *link* entre el objeto “S1” de la clase “Savings” y “John” de la clase “Client”.

- En primer lugar se deben crear los dos *linkEnds* que conectan a cada instancia con el link. La creación de un elemento del Sort *LinkEnd* se realiza mediante **Data Level>Create Element>LinkEnd** del menú principal. Luego se conecta cada *LinkEnd* con la instancia y *AssociationEnd* correspondientes mediante las modification actions *setInstance* y *setAssociationEnd*, respectivamente. Todas estas acciones se realizan en Elements Repository.
- Seleccionando **Data Level>Create Element>Link**, se incorpora una entidad modelada “a Link”, perteneciente al Sort *Link*. Se ubica automáticamente en la carpeta Data Elements de Elements Repository.
- Mediante la acción *addLinkRole* se agregan los dos *LinkEnds* preparados en el anterior paso.
- Seleccionar la asociación “ ” en Model Level, y evaluar la acción *newLink* del Sort *Association*. Elegir “a Link” como argumento.

El efecto de la acción, es la incorporación del *Link* y sus *LinkEnds* al nivel de los datos, conectando las instancias con identificación “John:Client” y “S1”. Al *Link* se lo identifica como “JohnAccount”. La aplicación de la acción y su efecto se ilustra en la figura 5.22.

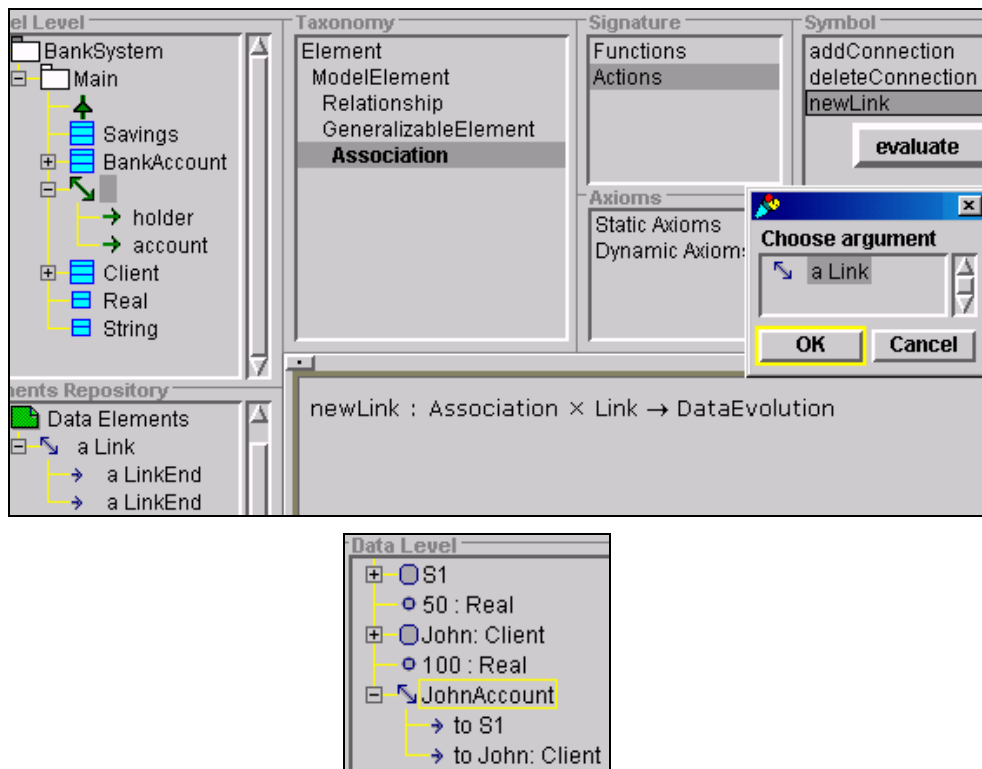


Figura 5.22: Creación del link

#### 5.10.4. Creación de un mensaje -Message-

Los siguientes pasos describen la creación de un mensaje de depósito enviado por el cliente “John” a la cuenta bancaria “S1”. La instancia del *Sort Message* se agregará a la *mailbox* del objeto receptor del mensaje.

- Seleccionar el ítem de la operación “deposit” en Model Level y la acción “< >” perteneciente al *Sort Operation*. Observar la signatura de la operación.
- Ejecutar la acción (figura 5.23.a) eligiendo los siguientes argumentos:

El objeto que envía el mensaje: “John”.

El objeto receptor del mensaje: “S1”.

Secuencia de parámetros que necesita el mensaje: 50.

El efecto de la acción, es la creación del mensaje “deposit(50)” y su ingreso al *mailbox* del objeto “S1” de la clase "Savings". La evolución del nivel de los datos se puede observar en el árbol de Data Level. Ver figura 5.23.b.

El objeto “S1” recibe efectivamente el mensaje “deposit(50)”, mediante la acción de modificación *-.-* accesible desde el *Sort Object*. En este caso, el resultado de la ejecución lo determina la máquina de estados que describe el comportamiento de las instancias de la clase “Savings”. El resultado de la recepción del mensaje es: el valor del *slot* “balance” es 50, el estado actual del objeto sigue siendo “credit” y el mensaje es eliminado del *mailbox*. Ver figura 5.23.c.

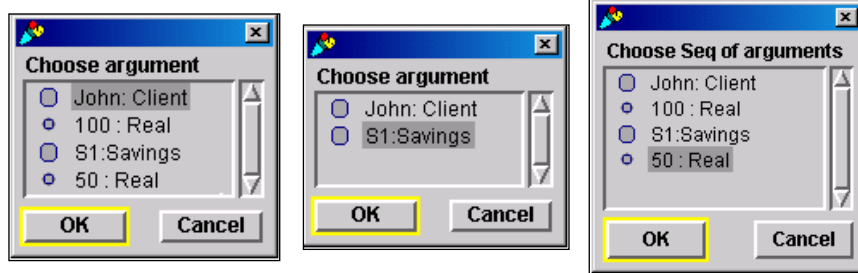
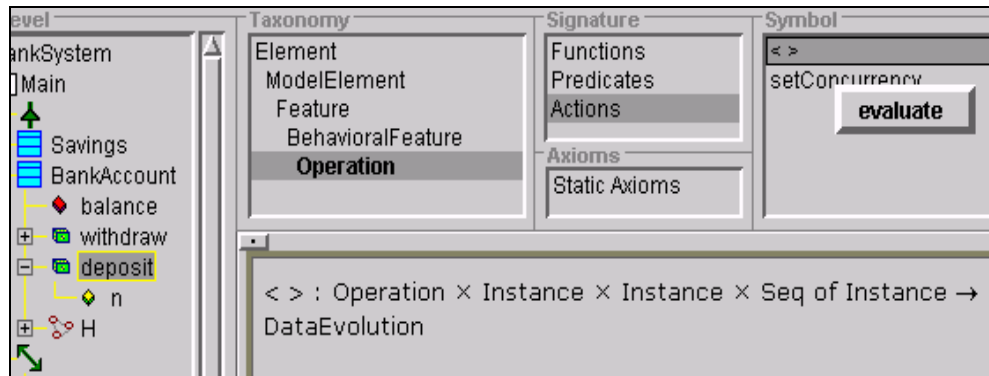


Figura 5.23.a: Ejecución de la acción de creación del mensaje

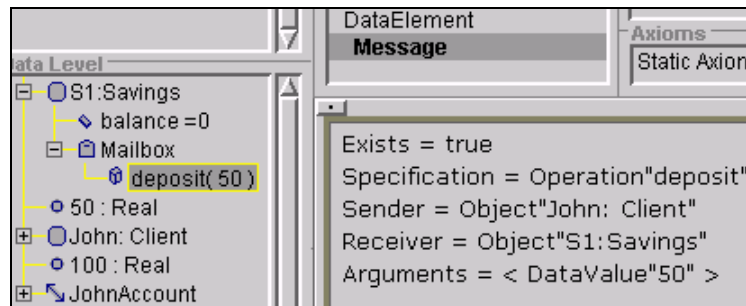


Figura 5.23.b: Efecto de la acción de creación <>

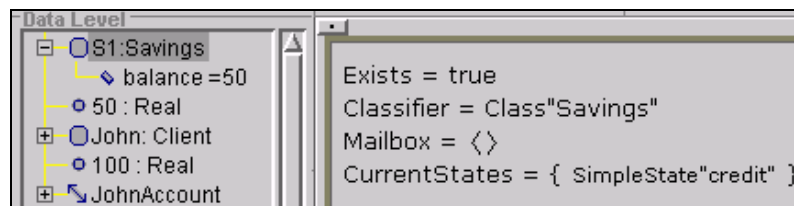


Figura 5.23.c: Efecto de la acción de modificación --

## 5.11. Evolución en ambos niveles

Con un ejemplo similar al de la figura 3.13, se describe cómo la evolución de la especificación del modelo afecta el comportamiento del sistema modelado, en el modelo formal especificado en la herramienta. Al estado del modelo en el ejemplo anterior, se le realizaron algunas modificaciones:

- destroy(DataValue“100:Real”)**- cancelación del valor 100.
- newData Value(DataType“Real”,80)**- creación del valor 80 de tipo “Real”.
- <>(Operation“withdraw”,Object“John”,Object“S1”,DataValue“80”)**- creación del mensaje “withdraw(80)” enviado por el cliente “John” y contenido en el *mailbox* de la cuenta “S1”.

La visualización del estado de los dos niveles (datos y meta-datos) en la herramienta se presenta en la figura 5.24, destacando la transición “t2” de la máquina de estados de “BankAccount”. A este estado del modelo se lo denomina  $W_i$ .

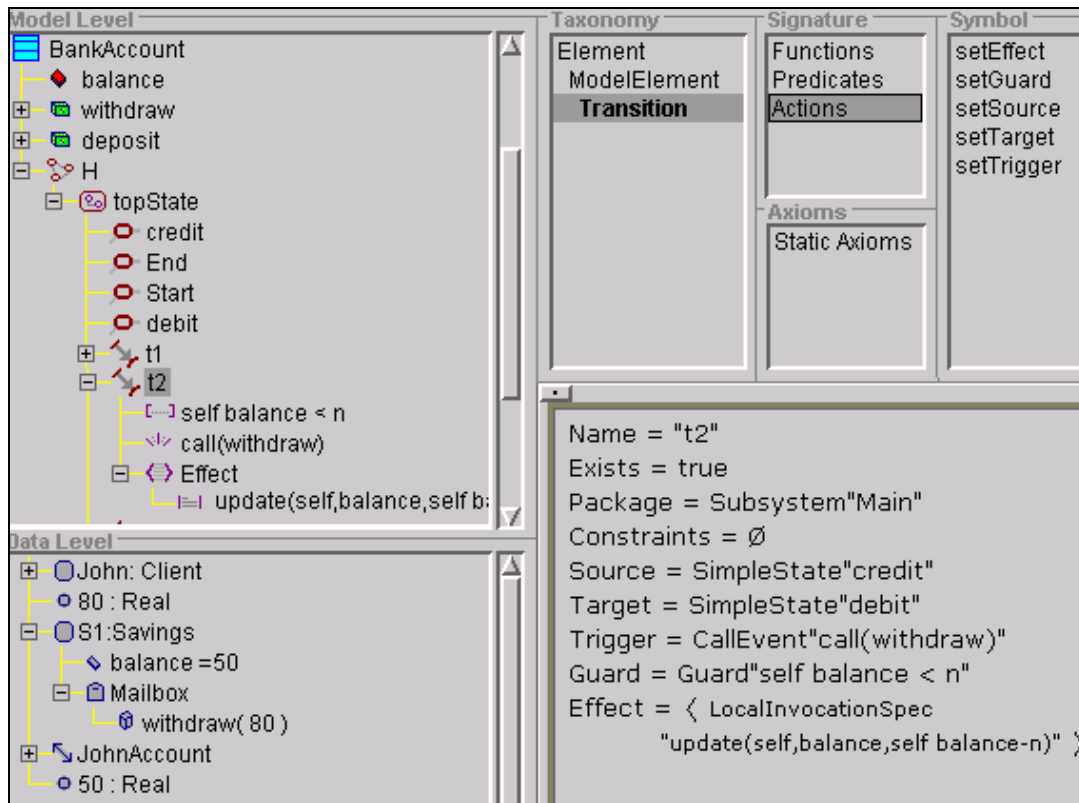


Figura 5.24: Estado  $W_i$  del modelo formal

Con la ejecución del mensaje pendiente en el *mailbox* de la cuenta “S1”, el balance se convierte en negativo y el estado del objeto pasa a ser de débito. La acción ejecutada sobre el modelo formal es “-.-”, ubicada en el Sort *Object* :

`-.-( Object“S1:Savings” , Message“withdraw(80)” )`

La acción produce evolución en el nivel de los datos. Como efecto colateral se crea el valor -30, que es el nuevo valor del *slot* “balance” de la cuenta. El estado actual del modelo formal en la aplicación se visualiza en la figura 5.25 y se lo denomina  $W_j$ .

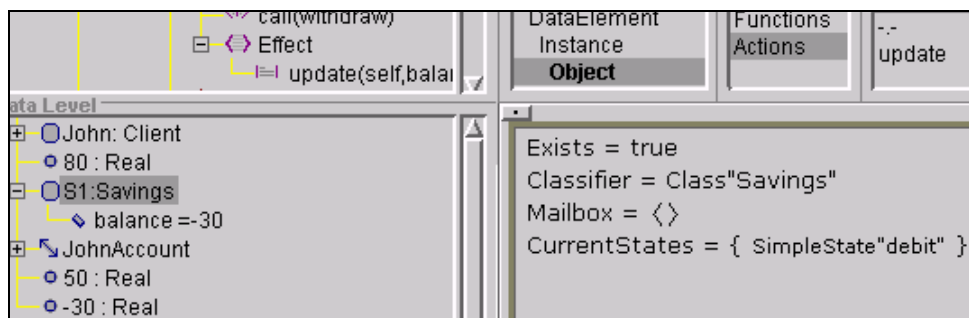


Figura 5.25: Estado  $W_j$  del modelo formal

Ahora se retorna al estado de partida,  $W_i$ , y se modifica el efecto de la transición “t2” agregando el envío de un mensaje de notificación al cliente dueño de la cuenta, mediante la acción “setEffect”, accedida desde el Sort *Transition*:

`setEffect( Transition“t2” , < LocalInvocationSpec“update(self, balance, self balance - n)” ,  
CallActionSpec“call(notify, self holder)” > )`

Con la ejecución de esta acción el modelo formal evoluciona en el nivel del modelo. El nuevo estado, denominado  $W_k$ , se observa en la figura 5.26.

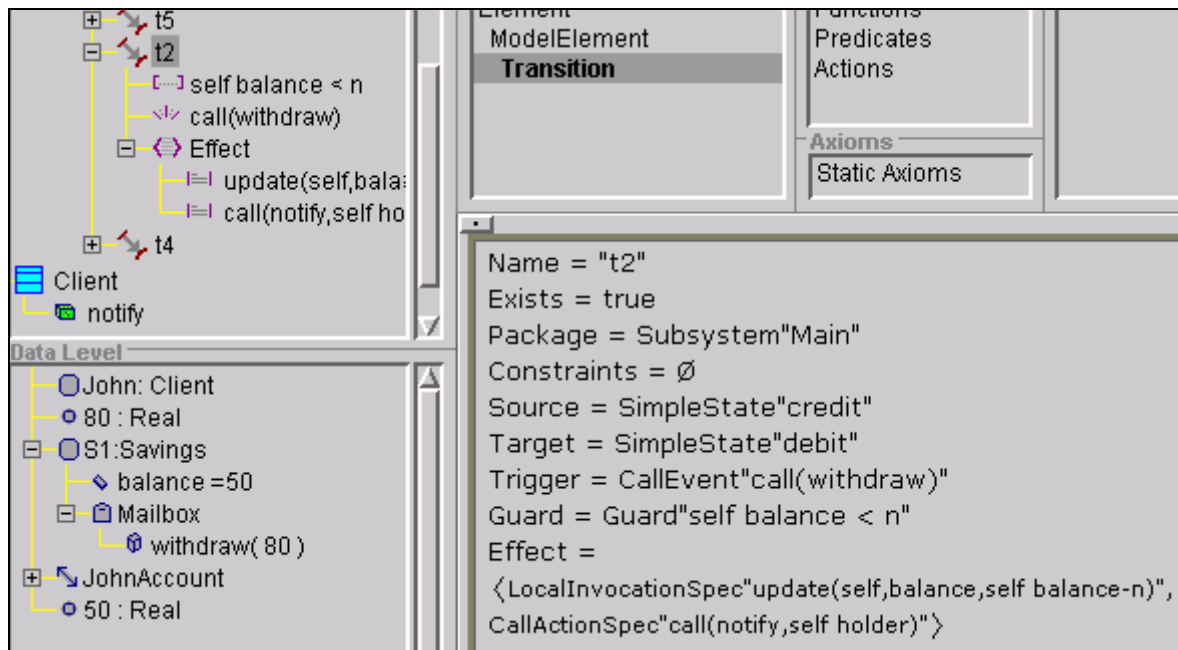


Figura 5.26: Estado  $W_k$  del modelo formal

Ahora se realiza el mismo paso que en la primera ocasión sobre  $W_i$ . Con la modificación del efecto la transición "t2" consumada se ejecuta el mensaje pendiente en el *mailbox* de la cuenta "S1":

-.-( Object"S1:Savings" , Message"withdraw(80)" )

La acción produce las mismas modificaciones (transición de  $W_i$  a  $W_j$ ) que con la anterior especificación, pero además, la cuenta "S1" le envía el mensaje "notify()" al cliente "John". En consecuencia, el comportamiento de los objetos del nivel de datos es afectado por la evolución del nivel del modelo. La figura 5.27 muestra el estado final  $W_f$  del modelo formal reflejado por la herramienta.

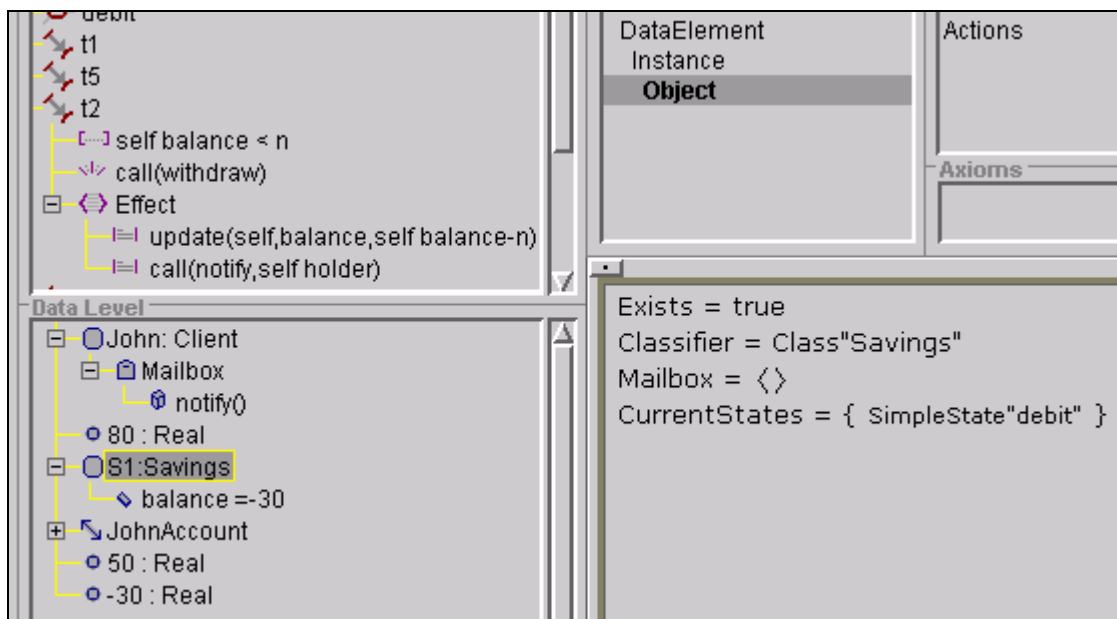


Figura 5.27: Estado  $W_f$  del modelo formal

# 6. Herramienta: Diseño

## 6.1. Organización

La documentación del diseño de la aplicación se presenta con el modelo de clases organizado en paquetes lógicos. Cada paquete contiene un grupo de clases (submodelo) relacionadas que cumplen algún objetivo importante en el diseño. Por cada paquete se presentan diagramas de clases -con clases, subpaquetes, sus relaciones, atributos y operaciones más interesantes- y la documentación de las clases más importantes –atributos, asociaciones, responsabilidades, colaboraciones-. Cuando la finalidad de un paquete involucra demasiadas clases, es descompuesto en subpaquetes para mayor claridad.

## 6.2. Paquete Principal

El diagrama de clases de la figura 6.1 ilustra los paquetes de mayor nivel del modelo y sus dependencias.

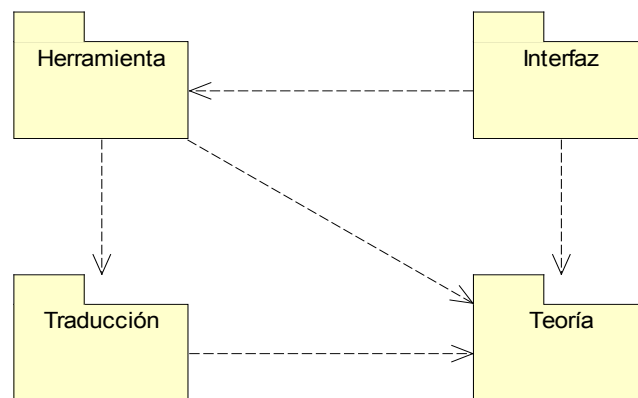


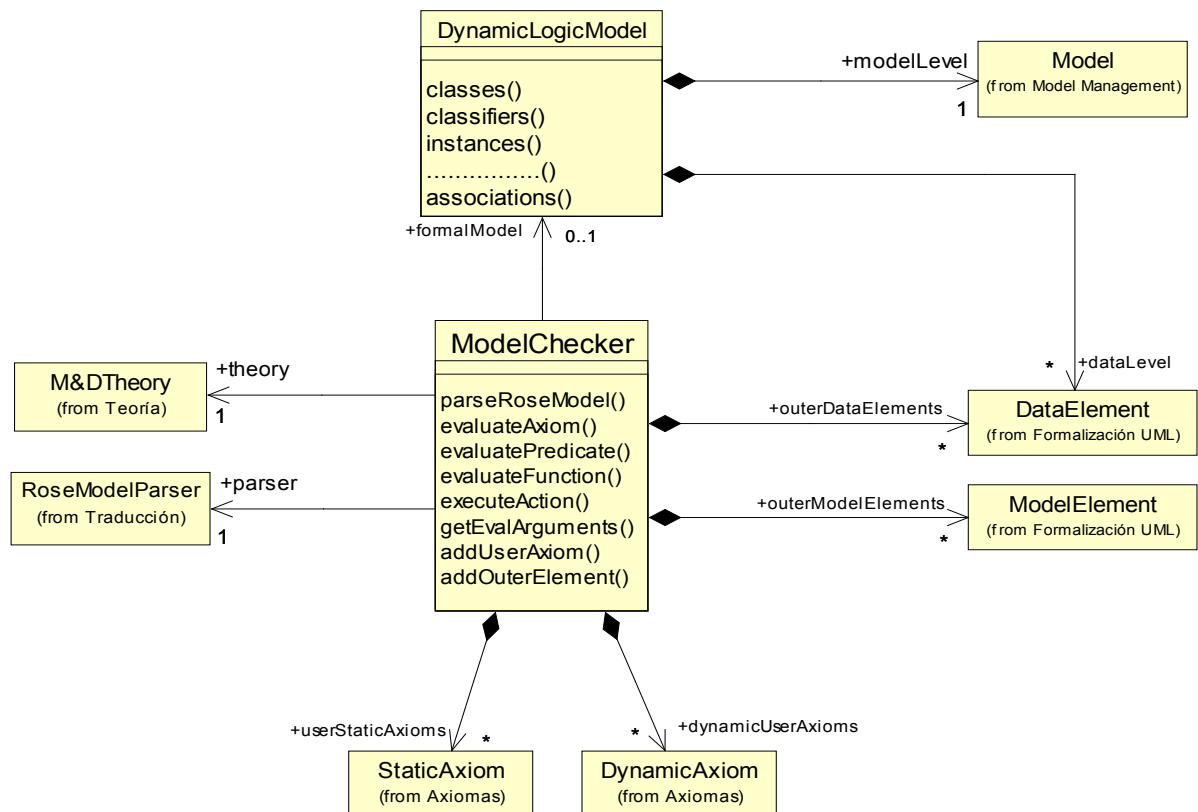
Figura 6.1: Modelo – Paquetes principales

### 6.2.1. Paquete Herramienta

El diagrama de clases principal del modelo Herramienta se presenta en la figura 6.2.

#### ModelChecker

Provee un marco para la interpretación semántica de un modelo especificado en UML, en un modelo cuyos elementos pertenecen a un dominio semántico formalmente definido por la M&D-Theory. Posibilita la interpretación de funciones y predicados sobre los elementos del modelo formal en sus distintos estados, así como la transición entre estados del modelo formal mediante la ejecución de acciones de evolución, y la evaluación de formulas estáticas y dinámicas. Permite administrar elementos que dejan de existir en determinado estado y crear otros para ser incorporados a un estado futuro, tanto en el nivel de los datos como en el nivel de los modelos. Admite la incorporación de axiomas, dinámicos o estáticos, específicos para determinados modelos.



**Figura 6.2: Paquete Herramienta – Diagrama de clases**

### Asociaciones

theory : teoría que provee el formalismo para especificar modelos.

parser : traductor de un modelo UML, en un modelo formal.

formalModel : modelo traducido a la M&D theory.

outerModelElements : entidades de modelado que no existen el estado actual del modelo formal.

outerDataElements : entidades modeladas que no existen el estado actual del modelo formal.

userStaticAxioms : axiomas estáticos incorporados a la teoría.

userDynamicAxioms : axiomas dinámicos incorporados a la teoría.

### Responsabilidades

Traducir modelos UML a modelos formales.

Evaluar (interpretar y ejecutar) funciones, predicados y acciones sobre el modelo formal.

Verificar axiomas estáticos y dinámicos.

Proveer argumentos para la evaluación de funciones, predicados, acciones y axiomas dinámicos.

Crear, Mantener y eliminar de elementos formales externos al mundo actual del modelo formal.

Agregar, eliminar y evaluar axiomas específicos de un modelo formal.

### Colaboraciones

RoseModelParser: traducción del modelo UML a Lógica Dinámica.

DynamicLogicModel : información sobre elementos que contiene.

MDTheory : proporciona la taxonomía de Sorts, signatura y axiomas de la teoría.

Sort : creación de un elemento.

Axiom: evaluación de un axioma.

TermDeclaration: evaluación de funciones, predicados y acciones.

ArgumentsManager : obtención de argumentos para la evaluación de terms y axioms.

### DynamicLogicModel

Es la especificación formal de un modelo orientado a objetos en términos de la M&D-theory. Mantiene una dicotomía entre elementos de modelado y elementos del sistema modelado. Cada estado de una instancia de esta clase corresponde a un estado del modelo.

**Asociaciones:**

modelLevel : es el elemento de modelado Model que contiene a todos los elementos de modelado. Es la raíz de la jerarquía de paquetes en la que se organiza el nivel del modelo.

dataLevel : conjunto de entidades modeladas que conforman el nivel de los datos.

**Responsabilidades**

Retornar todas las instancias de un Sort dado.

Agregar y eliminar elementos de ambos niveles.

**Colaboraciones**

Requiere que el Model, elemento raíz del nivel del modelo, le devuelva las instancias de un Sort dado.

## 6.2.2. Paquete Teoría

Este paquete contiene las clases y relaciones que modelan la teoría dinámica de primer orden M&D-theory. A la vez este paquete se subdivide en cinco paquetes mas para favorecer la descripción de los distintos componentes de la teoría. La figura 6.3 muestra los subpaquetes del paquete Teoría y sus dependencias. La figura 6.4 muestra el diagrama de clases principal de este paquete.

### M&DTheory

Representa a la teoría dinámica de primer orden M&D-theory. Está conformada por una signatura (taxonomía de Sorts, funciones, predicado y acciones) y un conjunto de axiomas.

**Asociaciones**

sorts : conjunto de Sorts organizados taxonómicamente por la relación  $\leq$ .

functions : conjunto de declaraciones de funciones pertenecientes a la signatura de la teoría.

predicates : conjunto de declaraciones de predicados pertenecientes a la signatura de la teoría.

actions : conjunto de declaraciones de acciones pertenecientes a la signatura de la teoría.

staticAxioms : conjunto de axiomas estáticos definidos por la teoría.

dynamicAxioms : conjunto de axiomas dinámicos definidos por la teoría.

**Responsabilidades**

Crear el conjunto de Sorts ordenados parcialmente por la relación  $\leq$ .

Creación y asignación de Sorts a las declaraciones de funciones, predicados y acciones.

Distinguir conjunto de Sorts cuyas instancias son entidades de modelado, del conjunto de Sorts de entidades modeladas.

Creación de axiomas estáticos y dinámicos.

Determinar la taxonomía de un Sort dado.

Devolver conjuntos de funciones, predicados, acciones donde un Sort dado es primer argumento de su declaración.

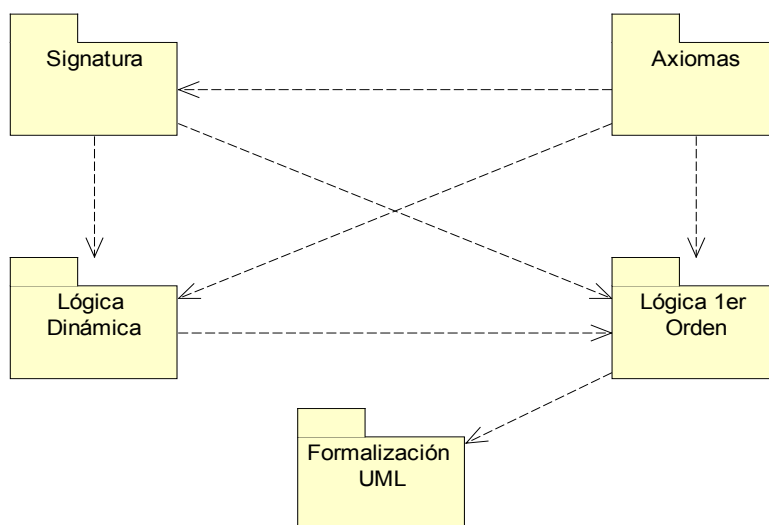


Figura 6.3: Paquete Teoría – Subpaquetes



### Colaboraciones

Sort: proporciona el conjunto de Sorts que lo generalizan.

TermDeclaration: el Sort de cada uno de sus argumentos.

### Sort

Representa un tipo para un elemento del dominio semántico la teoría. Cualquier entidad de modelado o entidad modelada en un modelo formal, pertenece a un Sort.

### Atributos

name : símbolo que distingue al Sort en la signatura de la teoría.

### Asociaciones

supersorts : conjunto de Sorts que lo generalizan, nivel inmediato superior, en la relación  $\leq$ .

implementation : clase que implementa el comportamiento de sus instancias.

### Responsabilidades

Crear un elemento (o término) para el modelo formal (o dominio de la teoría).

Retornar el conjunto de todos los Sorts que lo generalizan.

Obtener los argumentos posibles de su tipo para una evaluación.

### Colaboraciones

Pedir a la clase que implementa instancias de su tipo, la creación de un elemento.

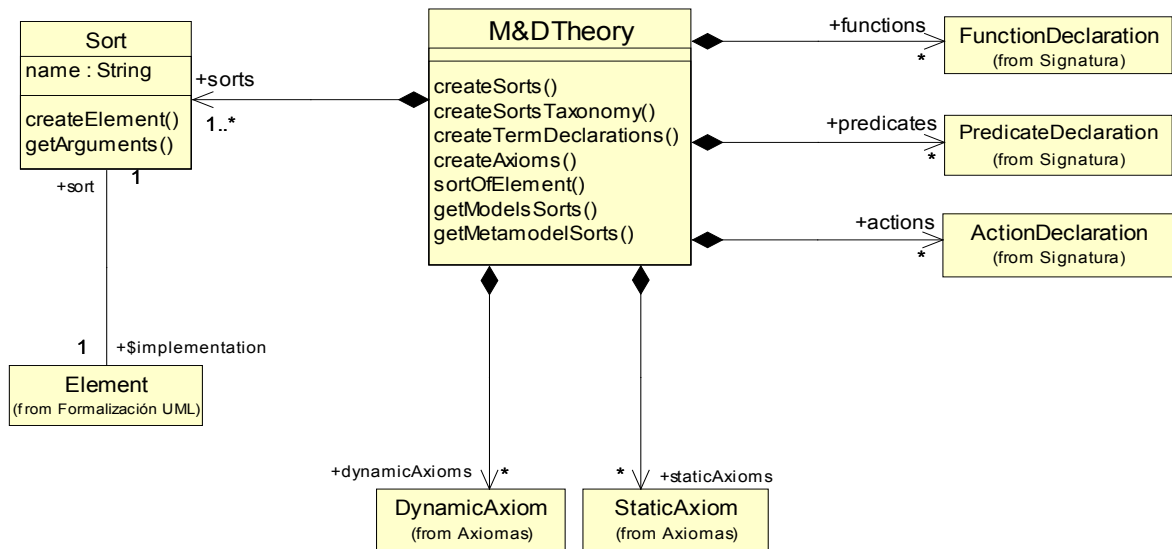


Figura 6.4: Paquete Herramienta – Diagrama de clases

## 6.2.2.1. Paquete Signatura

El diagrama de clases de este subpaquete de Teoría se ilustra en la figura 6.5.

### TermDeclaration

Sus subclasses representan declaraciones de funciones, predicados y acciones, en la signatura de la teoría.

### Atributos

symbol : símbolo del término.

comment : texto que contiene una breve descripción informal de la interpretación semántica de la declaración.

### Asociaciones

argumentsSort : secuencia de Sorts que participan en la declaración.

### Responsabilidades

Evaluar funciones (terms) con instancias de los Sort que forman parte de la declaración.

### Colaboraciones

Pedir a cada Sort los argumentos posibles para la evaluación del term.

### FunctionDeclaration

Representa una declaración de función en la signatura de la teoría.

### Asociaciones

resultSort : Sort del elemento que retorna la evaluación de la función.

function : interpretación semántica de la declaración.

### PredicateDeclaration

Representa una declaración de un predicado en la signatura de la teoría.

### Asociaciones

predicate : interpretación semántica de la declaración.

### ActionDeclaration

Representa una declaración de una acción en la signatura de la teoría.

### Asociaciones

action : interpretación semántica de la declaración.

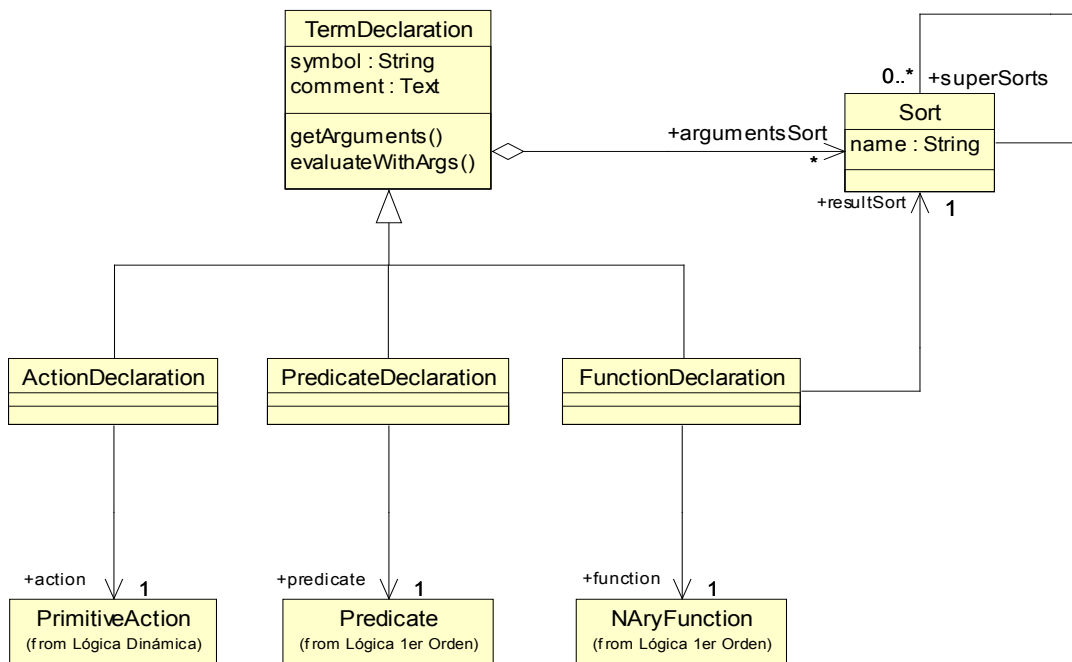


Figura 6.5: Paquete Teoría:Signatura– Diagrama de clases

## 6.2.2.2. Paquete Axiomas

Este paquete agrupa las clases que modelan la posibilidad de evaluar axiomas sobre un modelo. El diagrama de clases de este subpaquete de Teoría se presenta en la figura 6.6.

### Axiom

Representa un axioma de la teoría.

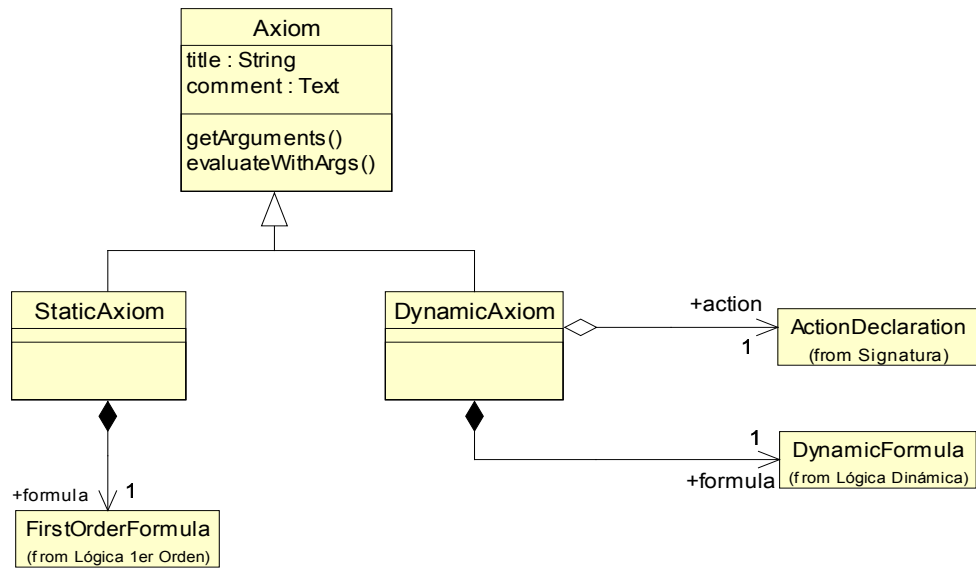
### Atributos

title : Título descriptivo del axioma.

comment : Descripción informal de la invariante o la semántica de acción representada por el axioma.

### Responsabilidades

Obtener los argumentos posibles para evaluar la fórmula.



**Figura 6.6: Paquete Teoría:Axiomas– Diagrama de clases**

### StaticAxiom

Representa un axioma estático de la teoría. Es decir, especifica una invariante, propiedad estática o una regla de buena formación de los modelos.

#### Asociaciones

formula : formula de primer orden cuantificada, para evaluarse sobre el modelo formal.

#### Responsabilidades

Evaluar una formula estática.

Obtener el modelo formal, argumento para evaluar la fórmula estática.

#### Colaboraciones

Quantifier: se evalúa sobre un modelo formal.

ModelChecker : provee el modelo formal.

### DynamicAxiom

Representa un axioma dinámico de la teoría. Es decir, define la semántica de las acciones de evolución de los modelos.

#### Asociaciones

formula : formula dinámica, para evaluar sobre el modelo formal.

action : la declaración de acción involucrada en la formula dinámica.

#### Responsabilidades

Evaluar una formula dinámica sobre el modelo simulando su evolución.

Obtener los argumentos posibles para evaluar la fórmula dinámica.

#### Colaboraciones

DynamicFormula: se evalúa sobre un modelo formal.

### 6.2.2.3. Paquete Lógica Dinámica

Este paquete agrupa las clases que modelan la lógica dinámica de primer orden. El diagrama de clases de este subpaquete de Teoría se presenta en la figura 6.7. La clasificación de acciones de evolución primitivas consideradas para este trabajo se presenta en la jerarquía de clases ilustrada por la figura 6.8.

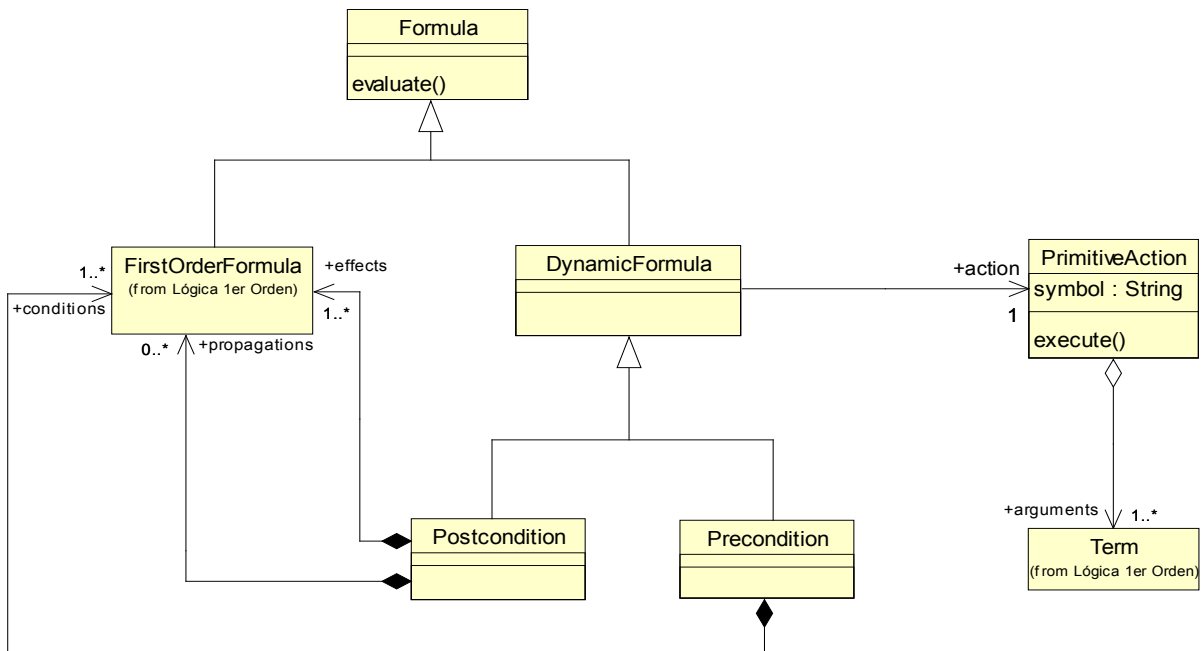


Figura 6.7: Paquete Teoría:Lógica Dinámica– Diagrama de clases

### DynamicFormula

Representa una fórmula modal de la Lógica Dinámica order-sorter de primer orden. En este contexto las fórmulas modales definirán la semántica de las acciones, es decir la evolución de los modelos.

#### Asociaciones

action : acción primitiva de evolución especificada por la fórmula.

#### Colaboraciones

PrimitiveAction : requiere que la acción se ejecute con determinados argumentos.

### Precondition

Especifica las condiciones de aplicabilidad de la acción en el modelo formal.

#### Asociaciones

conditions : conjunto de formulas que expresan las precondiciones para ejecutar la acción.

#### Responsabilidades

Evaluar la validez de la conjunción de todas las condiciones sobre un modelo formal.

### Postcondition

Describe los efectos y las propagaciones en el modelo de la ejecución de la acción en el modelo formal.

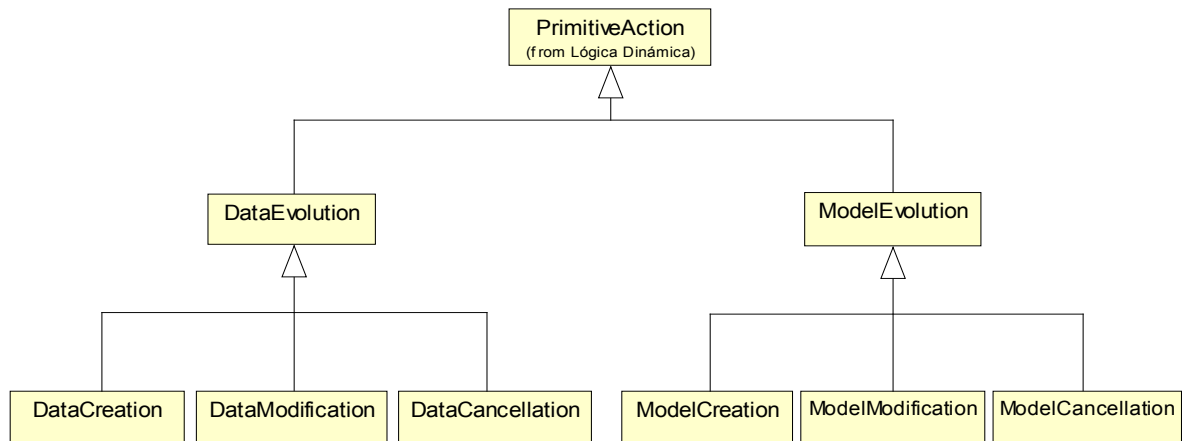
#### Asociaciones

effects : conjunto de formulas que expresan los efectos directos de aplicar la acción, es decir las modificaciones que sufre el source de la acción.

propagations : conjunto de formulas que describen el efecto indirecto de aplicar la acción y que son necesarios para mantener la consistencia del modelo formal.

#### Responsabilidades

Simular la aplicación de la acción sobre el modelo formal, y evaluar la validez de la conjunción de todos los efectos y propagaciones.



**Figura 6.8: Paquete Teoría:Lógica Dinámica– Jerarquía de acciones primitivas**

### **PrimitiveAction**

Abstrae la modificación atómica sobre un modelo. Puede representar evolución del nivel del modelo o evolución de los datos.

#### *Asociaciones*

arguments : secuencia de términos sobre los que se ejecuta la acción.

#### *Responsabilidades*

Se ejecuta sobre el valor de su primer argumento, denominado source, y toma el valor del resto de los argumentos como parámetros de la acción.

#### *Colaboraciones*

Term : requiere el valor de cada uno de los términos para ejecutarse.

### **ModelEvolution**

Acción primitiva que modifica el modelo.

### **DataEvolution**

Acción primitiva que modifica el sistema modelado.

### **ModelCreation**

Acción primitiva que agrega un nuevo elemento al modelo.

### **ModelModification**

Acción primitiva que modifica un elemento del modelo.

### **ModelCancellation**

Acción primitiva que elimina un elemento existente del modelo.

### **DataCreation**

Acción primitiva que agrega un nuevo elemento al sistema modelado.

### **DataModification**

Acción primitiva que modifica un elemento del sistema modelado.

### **DataCancellation**

Acción primitiva que elimina un elemento existente del sistema modelado.

## **6.2.2.4. Paquete Lógica de primer orden**

Este paquete agrupa las clases que modelan las fórmulas y términos necesarios para representar expresiones en lógica de primer orden. La jerarquía de clases que modelan las fórmulas están representadas en el diagrama de clases de la figura 6.9. La jerarquía de clases que modelan a los términos está representadas en el diagrama de clases de la figura 6.10.

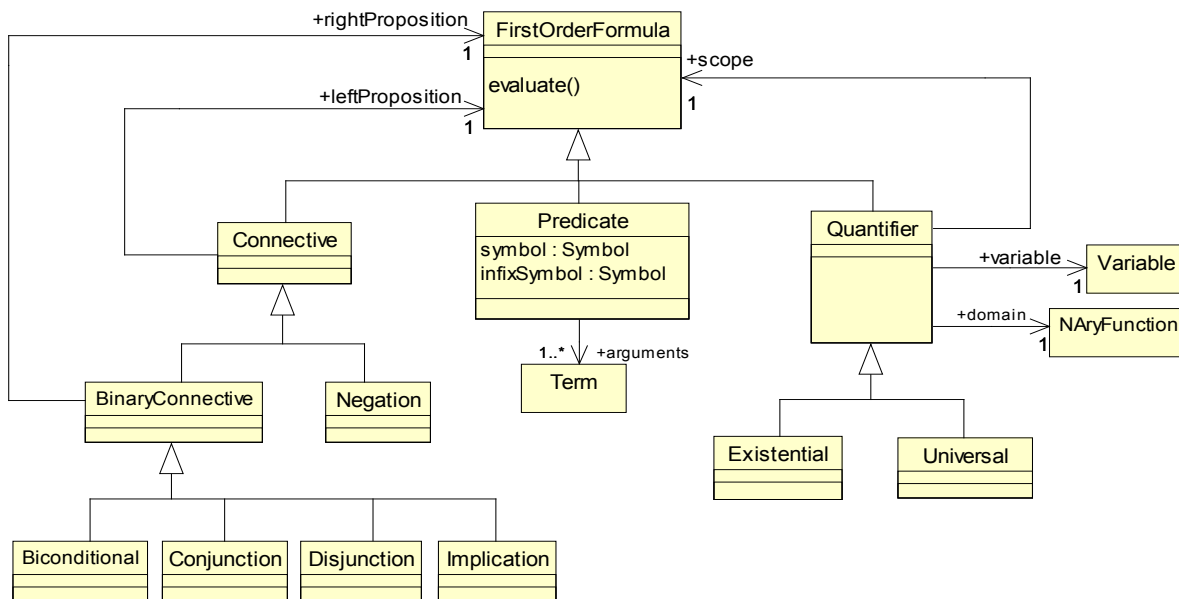


Figura 6.9: Paquete Teoría:Lógica 1er orden– Jerarquía de fórmulas

## FirstOrderFormula

Representa una fórmula de la Lógica de Predicados de primer orden.

### Responsabilidades

Previa valuación de las variables libres que contiene, la fórmula de primer orden puede evaluarse.

## Connective

La jerarquía de Connective representa construcciones de la Lógica proposicional, tales como las construidas a partir de los operadores proposicionales o conectivos:  $\wedge$  Conjunción,  $\vee$  Disyunción,  $\neg$  Negación,  $\rightarrow$  Implicación,  $\leftrightarrow$  Bicondicional.

### Asociaciones

leftProposition: fórmula de primer orden, proposición necesaria para construir inductivamente la fórmula proposicional. En el caso de Negation es la única. En el caso de Implication y Biconditional se la denomina antecedente.

rightProposition: fórmula de primer orden, proposición necesaria para construir inductivamente las fórmulas proposicionales con conectivos binarios. En el caso de Implication y Biconditional se la denomina consecuente.

### Colaboraciones

FirstOrderFormula: requiere que se evalúen sus subfórmulas.

## Predicate

Representa un predicado n-ario, a evaluarse sobre elementos de modelado.

### Atributos

symbol: en este contexto, el símbolo coincide con el nombre de un mensaje de predicado implementado en algunas de las clases necesarias para la formalización del metamodelo UML.

infixSymbol : símbolo para la representación del símbolo de predicado en forma infija.

### Asociaciones

arguments : secuencia de términos, acorde a la aridad del predicado.

### Colaboraciones

Term: necesita el valor de cada uno de sus argumentos(términos) para evaluarse.

## Quantifier

Representa una cuantificación en la Lógica de predicados de primer orden. La cuantificación puede ser universal (cuantificador  $\forall$ ) o existencial (cuantificador  $\exists$ ).

**Asociaciones**

variable: es la variable ligada al cuantificador.

domain : función que determina la valuación para la variable ligada al cuantificador.

scope : fórmula de primer orden, alcance de la variable ligada al cuantificador.

**Colaboraciones**

NAryFunction : requiere la evaluación de la función para obtener el conjunto de elementos que serán la valuación de la variable ligada en la evaluación de la fórmula.

FirstOrderFormula : evaluación de la subfórmula Scope para cada sustitución de la variable ligada al cuantificador.

**Term**

Representa un término de la lógica de predicados de primer orden.

**Responsabilidades**

Devuelve un valor.

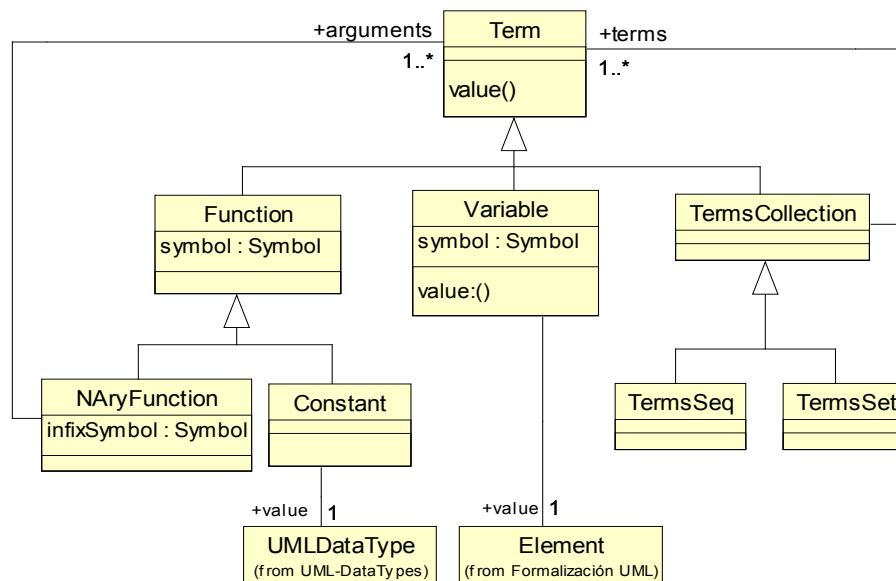


Figura 6.10: Paquete Teoría:Lógica 1er orden–Jerarquía de términos

**Variable**

En el contexto de esta aplicación una variable es un contenedor, rotulado con un símbolo, de elementos del modelo formal. Su valor es precisamente el elemento que contiene, y puede cambiar a lo largo de la vida de la variable.

**Constant**

Función de aridad cero. Su valor no se modifica y puede ser alguna instancia de UML-Data Types, o NullElement.

**NaryFunction**

Representa una función de aridad superior a cero. Al igual que los predicados, su symbol coincide con algún mensaje de función implementado en algunas de las metaclasses necesarias para la implementación del metamodelo UML. Su valor depende de los valores de sus términos, argumentos de la función.

**TermsCollection**

Representa una secuencia o conjunto de términos. Su valor está dado por la secuencia o conjunto de valores de los términos que contiene. En una fórmula aparecen como conjuntos o secuencias definidos por extensión, en este caso con términos.

### 6.2.2.5. Paquete Formalización de UML

Este paquete contiene el modelo de clases necesarias para la formalización de la porción del metamodelo de UML cuyas instancias son utilizadas como dominio semántico de la M&D-Theory. La primera subdivisión de paquetes se realiza en base a los dos niveles de la arquitectura de las notaciones modelado que integra la formalización: el nivel del modelo y el nivel de los datos. Luego, en el paquete correspondiente al nivel de los modelos se sigue intencionalmente la estructura de paquetes del metamodelo de UML.

Los estereotipos de los atributos y operaciones de cada clase indican si representan funciones (<<F>>), predicados (<<P>>) o acciones (<<A>>) en la Lógica Dinámica. En el caso de los associations siempre representan funciones y no se especifica un estereotipo. Se debe tener en cuenta que el lenguaje utilizado para la implementación del metamodelo no soporta herencia múltiple, por lo tanto se simula con herencia simple de una de las superclases y asociación con la restante. La figura 6.11 ilustra los dos subpaquetes principales y las clases que representan a los sorts distinguidos de la teoría.

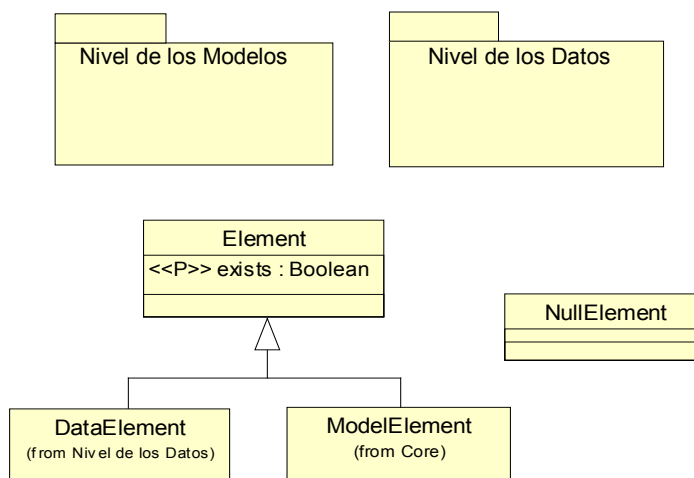


Figura 6.11: Paquete Teoría:Lógica 1er orden-Formalización UML

### Paquete Nivel de los Modelos: Foundation: UML-Data Types

Estas clases son necesarias para la implementación del metamodelo en Smalltalk. Otras clases necesarias, como String, Integer, Float, Boolean y Time, son tomadas directamente de la librería de clases básicas del lenguaje. Las subclases de UMLDataType implementadas se presentan en el diagrama de la figura 6.12.a. La jerarquía de clases para representar enumeraciones se ilustra en la figura 6.12.b. Cada subclase puede crear tantas instancias como cantidad de literales permita su rango. Ver el patrón de diseño Singleton [Alpert et 98; Gamma et al 94].

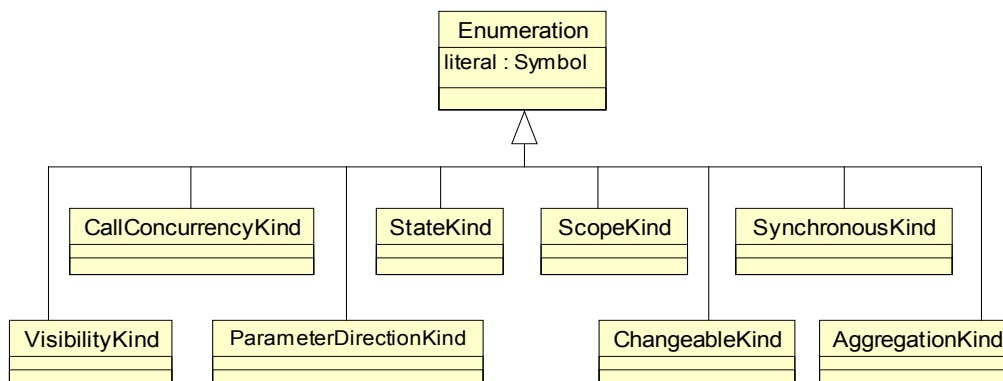


Figura 6.12.b: Paquete UML-Data Types- Jerarquía de Enumeraciones



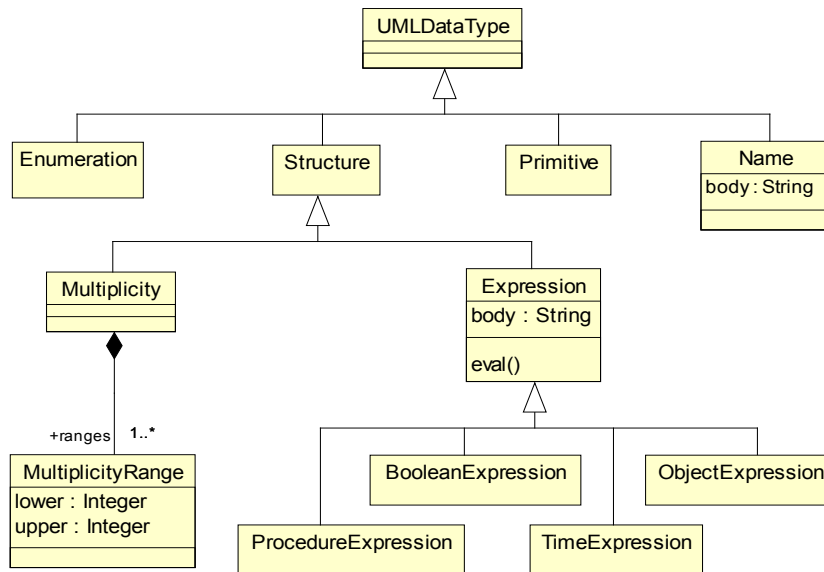


Figura 6.12.a: Paquete Formalización UML: Foundation: UML-Data Types

### Paquete Nivel de los Modelos: Foundation:Core

Como en la especificación de UML, las clases que modelan esta porción del metamodelo se dividen en dos diagramas para mayor claridad. Ver figuras 6.13 a y b.

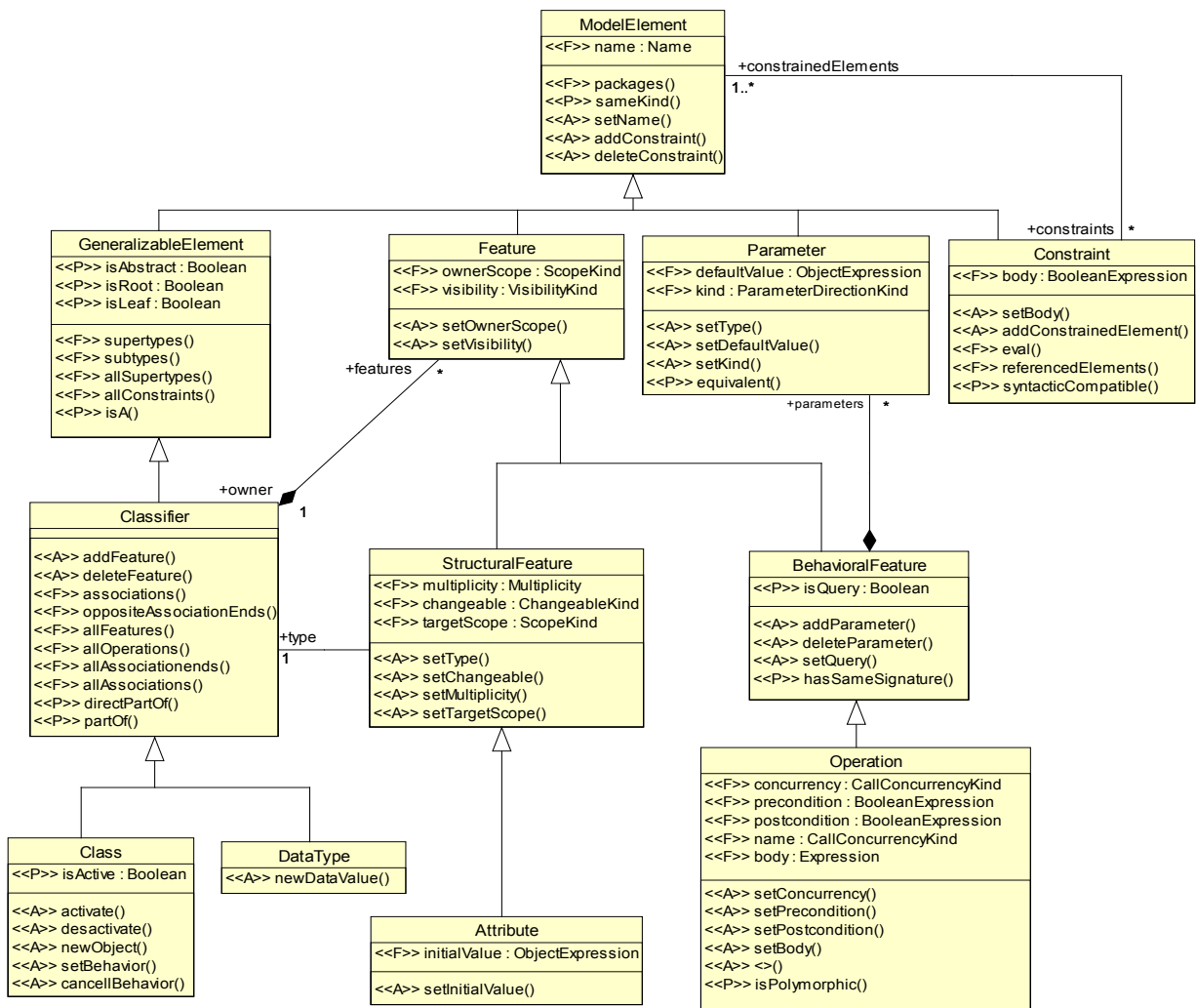
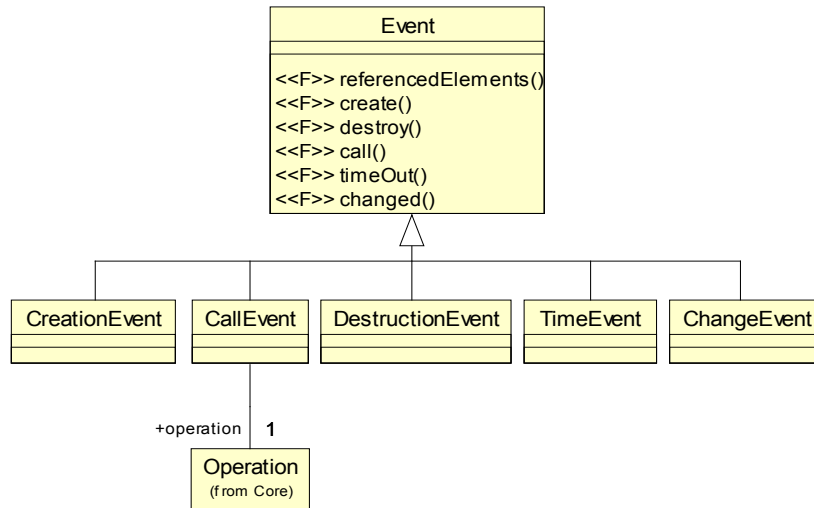
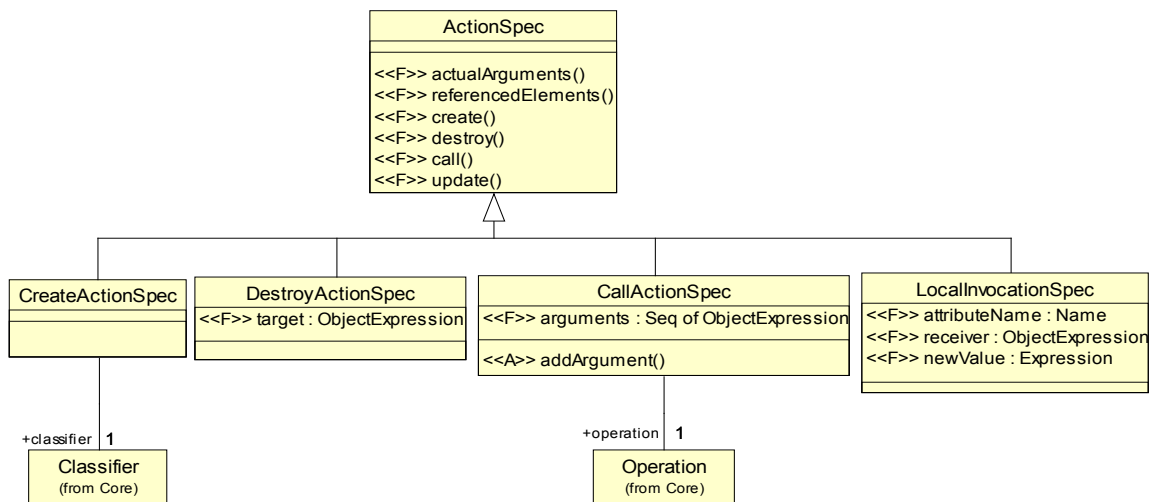


Figura 6.13.a: Paquete Foundation:Core- Structural Backbone





**Figura 6.14.b Paquete Behavioral Elements: State Machines -Jerarquía Events**



**Figura 6.14.c: Paquete Behavioral Elements: State Machines – Jerarquía ActionSpecs**

### Paquete Nivel de los Modelos:Model Management

En la figura 6.15 se ilustra el diagrama de clases que modelan el paquete Model Management del metamodelo de UML.

### Paquete Nivel de los Datos

La figura 6.16 ilustra el diagrama de clases que modelan el nivel de los datos de la formalización del metamodelo de UML.

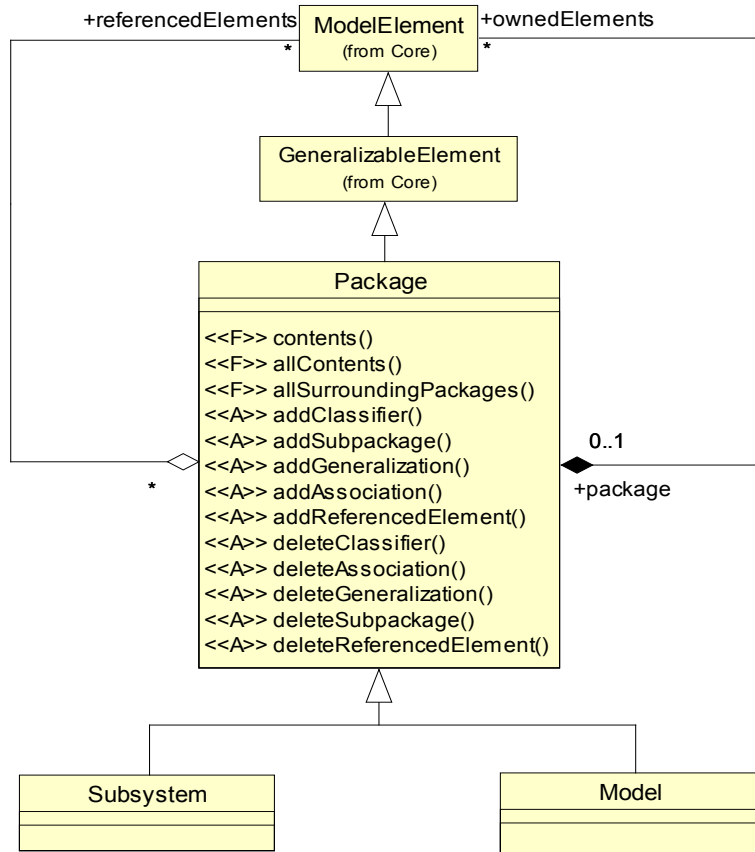


Figura 6.15: Paquete Model Management

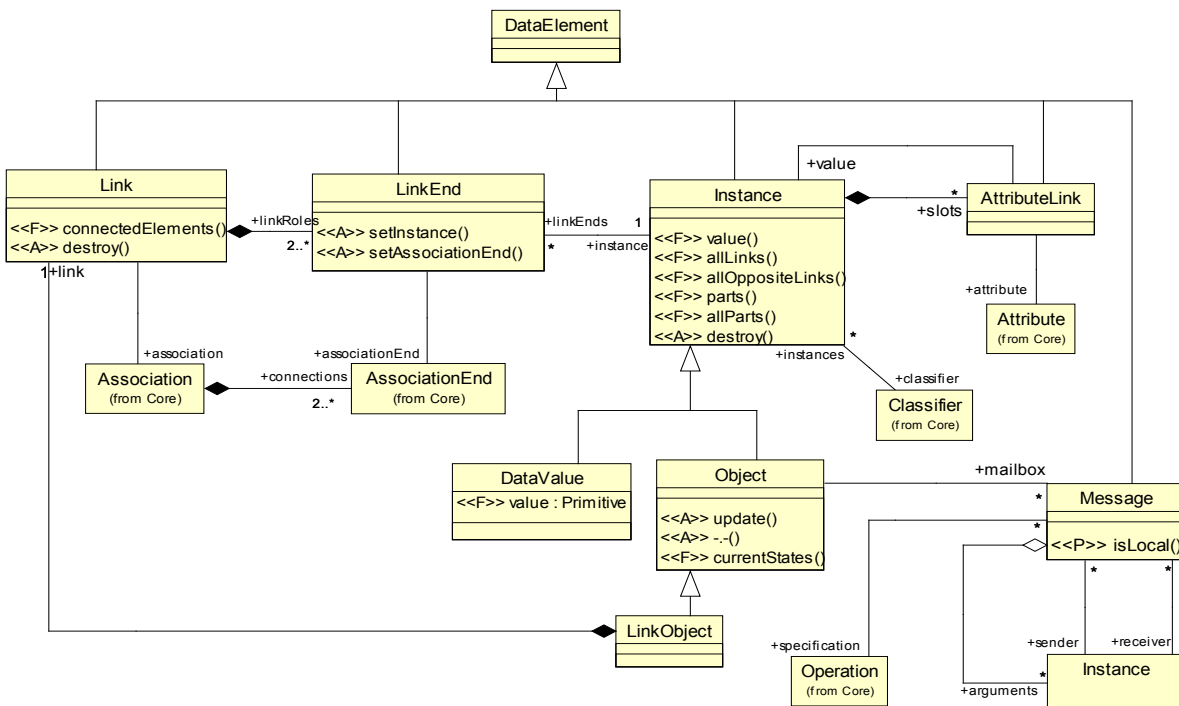


Figura 6.16: Paquete Nivel de los Datos

## 6.2.3. Paquete Traducción

La traducción de un modelo UML especificado en Rational Rose a su correspondiente M&D-theory, se lleva a cabo en dos pasos:

- 1 - Generar un modelo Rose en el ambiente Smalltalk.
- 2 - Interpretar el modelo Rose para obtener la teoría M&D.

### Primer Paso

La representación textual del modelo Rose, se parsea a través de una instancia de la clase RoseModelParser en colaboración con instancias de RoseModelScanner y RoseModelBuilder. Las clases RoseModelParser y RoseModelScanner fueron generadas por T-Gen [Graver 92], una herramienta para la generación automática de parsers de strings a objetos, mediante la especificación de la gramática de un archivo .mdl detallada en el capítulo 4.

La clase RoseModelBuilder actúa en conjunto con las anteriores, con el objetivo de crear objetos que representen a cada elemento del modelo Rose que sea de interés para este trabajo. Se obtiene una instancia de RoseModel, compuesta por instancias de la jerarquía RoseElement. Un RoseModel conserva la estructura del modelo generado por Rational Rose, aunque fueron descartados todos sus atributos de especificación gráfica, generación de código, etc.

### Segundo Paso

El RoseModel generado es traducido por una instancia de DynamicLogicModelMaker, la cual interpreta cada uno de los elementos de modelado en UML (RoseElements) y los convierte a elementos formales de la Lógica Dinámica. El modelo formal resultante es una instancia de DynamicLogicModel, con la cual se trabajará en la aplicación.

El grupo de clases necesarias para llevar a cabo estos pasos se representa en el diagrama de clases de la figura 6.17.

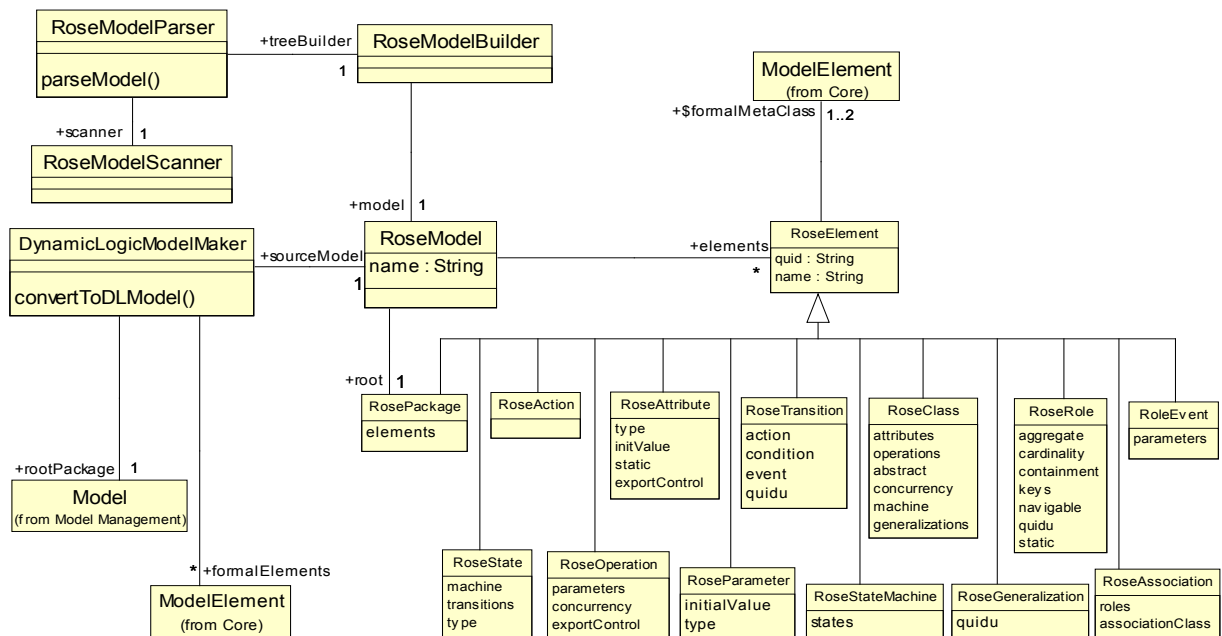


Figura 6.17: Paquete Traducción

## 6.2.4. Paquete Interfaz

Este paquete contiene las clases necesarias que proveen a la herramienta de una interfaz gráfica, para que el usuario interactúe con el modelo formal, edite fórmulas, traduzca modelos UML, etc. De acuerdo a las principales funciones de las clases que lo componen, el paquete es descompuesto en dos subpaquetes: Interfaz principal y Edición de Axiomas. Los respectivos diagramas son ilustrados por las figuras 6.18 y 6.19 respectivamente.

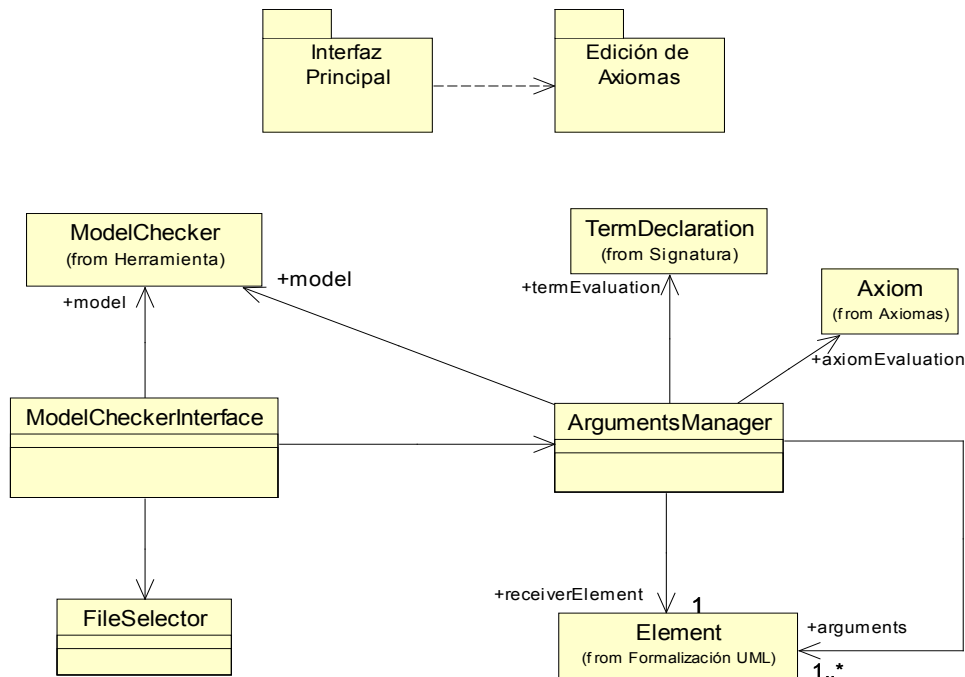


Figura 6.18: Paquete Interfaz principal

### ModelCheckerInterface

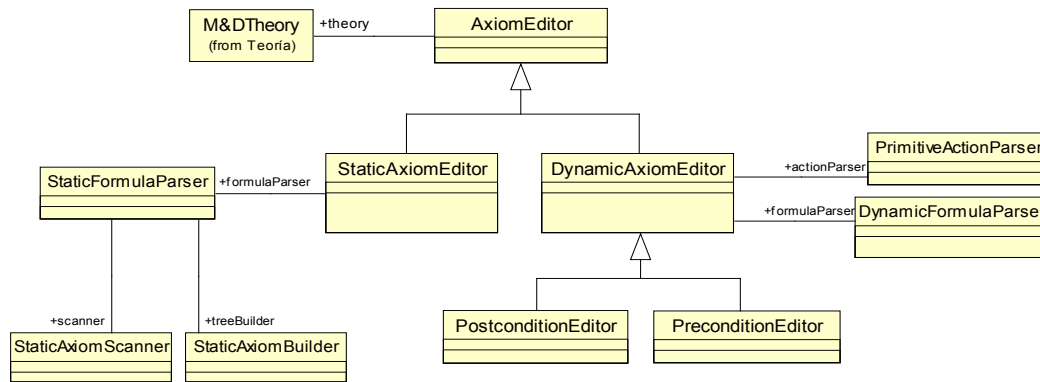
Provee la interfaz gráfica principal de la herramienta con la que los usuarios interactúan. A través del mecanismo de dependencias MVC [Howard 95; Horan 96] muestra la evolución del modelo formal. Es la encargada de abrir diálogos para seleccionar archivos, Browsers de axiomas, editores de axiomas y diálogos para elegir argumentos. Las vistas de los elementos del modelo en forma de árbol de íconos se realizó con el widget TreeView creado por el framework Aragon [Aragon].

### ArgumentsManager

Ante la evaluación de predicados, acciones, funciones, o axiomas sobre el modelo, ofrece al usuario diálogos para la elección de los argumentos disponibles en el modelo. Cada diálogo se adecua al sort de los elementos, o si el argumento es individual o una secuencia de elementos.

### FileSelector

Provee un diálogo para la navegación y selección de archivos de tipo .mdl, .dlm, y .axm. La vista del sistema de directorios se realizó con el widget TreeView de Aragon.



**Figura 6.19: Paquete Edición de axiomas**

### AxiomEditor

Esta jerarquía de clases permite al usuario contar con la interfaz gráfica adecuada para editar sus propios axiomas estáticos y dinámicos. Presenta botones para incluir símbolos lógicos, o seleccionar funciones, predicados y acciones disponibles en la teoría. Las clases StaticFormulaParser, DynamicFormulaParser y PrimitiveActionParser son las responsables de parsear fórmulas estáticas, fórmulas dinámicas y acciones primitivas respectivamente. Dichas clases y StaticFormulaScanner, DynamicFormulaScanner y PrimitiveActionScanner fueron generadas por T-Gen. A la par, son necesarias las clases StaticFormulaBuilder, DynamicFormulaBuilder y PrimitiveActionBuilder para crear las instancias de la jerarquía Formula, es decir fórmulas que puedan evaluarse en la teoría. En el diagrama se omiten algunas clases para simplicidad.

# Conclusiones

La gran aceptación que tuvo la unificación de los lenguajes gráficos de modelado orientado a objetos por parte de los ingenieros de software, impulsó activas discusiones acerca de la precisión semántica de UML. La necesidad de integrarlo con técnicas formales de análisis y verificación puede satisfacerse ocultando formalismos matemáticos detrás de la notación gráfica. Hasta el momento, la propuesta más exitosa para lograr la integración consiste en definir formalmente la semántica de UML, es decir, fijar reglas para asociar estructuras sintácticas del lenguaje de modelado con elementos en un dominio semántico formalmente definido. El modelo conceptual basado en lógica dinámica resumido en el capítulo 3 de este trabajo, representa formalmente la información expresada en determinados modelos UML.

La herramienta desarrollada implementa un método de transformación automático, de acuerdo al conjunto de reglas definidas en la M&D-theory, para crear un modelo formal a partir de los modelos expresados en UML. Luego, permite trabajar con el modelo formal en la aplicación de mecanismos de chequeo, evaluación de fórmulas, evolución en los distintos niveles del modelo, etc. Se espera que la herramienta evolucione a medida que se completa la formalización de UML en la propuesta mencionada, para contar con la posibilidad de evaluar cualquier instancia del metamodelo de UML. También queda pendiente el proceso inverso, el cual consiste en que la herramienta transforme el modelo formal en un modelo UML, apto para seguir especificándolo en Rational Rose. De esta manera, se lograría que la herramienta se integre al proceso CASE llevado a cabo por Rational Rose, aportando semántica formal al en el análisis y el diseño orientado a objetos.





# Referencias Bibliográficas

- [Alpert et 98] S. Alpert, K. Brown and B. Woolf. The Design Patterns Smalltalk Companion. Addison-Wesley, January 1998.
- [Aragon] Fraunhofer Institute IITB. [www.aragon.iitb.fhg.de](http://www.aragon.iitb.fhg.de) . December 1998.
- [Argo] Argo/UML Proyect, [www.argouml.tigris.org](http://www.argouml.tigris.org). OCL Compiler, [www.drdsden-ocl.sourceforge.net](http://www.drdsden-ocl.sourceforge.net) Technische Universitat Dresden. OCL Parser, [www-4.ibm.com/software/ad/standards/ocl-download.html](http://www-4.ibm.com/software/ad/standards/ocl-download.html). OCL specification in the UML.
- [Boehm 88] B.Boehm, A espiral model of software development and enhancement, IEEE Computer, 21(5), May 1988.
- [Boehm and Papaccio 88] B.Boehm and P.papaccio, Understanding and controlling software costs, IEEE Transactions on Software Engineering, 14(10), October 1988.
- [Booch 94] G.Booch, Object Oriented Analysis and Design with Applications, Second Edition, Addison-Wesley Publishing Company, Inc, 1994.
- [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.
- [Budd 91] Timothy Budd, An introduction to object-oriented programming, Addison-Wesley Publishing Company, 1991.
- [Coad and Yourdon 91] P.Coad and E.Yourdon, "Object Oriented Analysis", Yourdon Press, Englewood Cliffs,NJ, 1991.
- [DeMarco79] Tom DeMarco, Structured Analysis and System Specification. Englewood Cliffs, NJ:Prentice Hall, 1979.
- [Evans et al 98] Evans,A., France,R., Lano,K. y Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Beyond the notation, Muller and Bezivin editors, Lecture Notes in Computer Science 1618, Springer-Verlag 1998.
- [Evans et al 99] Evans,A., France,R., Lano,K. y Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, H.Kilov editor, Kluwer Academic Publishers 1999.
- [France et al 97] R.France, J.Brueel and M.Larrondo-Petrie. An integrated object-oriented and formal modeling environment, Journal of Object Oriented Programming (JOOP), 1997.
- [Gamma et al 94] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- [Gilb 88] T. Gilb, Principles of Software Engineering Management, Addison Wesley, Reading, 1988.
- [Gomaa and Scott 81] H.Gomaa and D.Scott, Prototyping as a tool in the specification of user requirements, proceedings 5<sup>th</sup> International Conference on Software Engineering, San Diego, March 1981.
- [Graver 92] Justin O.Graver. T-Gen User's Guide. University of Florida, 1992.
- [Horan 96] Bernard Horan, Laura Hill and Mario Wolezco. Advanced VisualWorks Course. [www.cs.uiuc.edu](http://www.cs.uiuc.edu).
- [Howard 95] Tim Howard. The Smalltalk Developer's Guide to Visual Works. SIGS Books, 1995.
- [Jacobson et al 99] I.Jacobson, M.Christerson, P.Jonsson and G.Overgaard, Object Oriented Software Engineering, Addison Wesley, 4<sup>th</sup> edition, 1992.

- [Jacobson et al 99] Ivar Jacobson, Grady Booch and James Rumbaugh, The Unified Software Development Process. Addison Wesley 1999.
- [Jones 90] C B Jones, Systematic software construction using VDM. Prentice Hall, 1990.
- [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.
- [Kim and Carrington 99] Kim, S. y Carrington,D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML’99 Conference, Lecture Notes in Computer Science 1723 (1999).
- [Lano and Bicaregui 98] Formalizing the UML in Structured Temporal Theories, Kevin Lano, Jean Bicaregui, Second ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische Universitat Munchen.
- [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].
- [Pons 00] Tesis de doctorado de la Facultad de Ciencias Exactas de la UNLP. “Una teoría dinámica como fundamento formal del proceso de desarrollo de software basado en modelos”. Febrero de 2000.
- [Rational] Rational Software Corporation. [www.rational.com](http://www.rational.com).
- [Rumbaugh et al. 91] J.Rumbaugh, M.Blahá, W.Premarlani, “Object Oriented Modeling and Design”, Prentice Hall, 1991.
- [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.
- [Spivey 92] M.Spivey. The Z notation: a reference manual. Prentice Hall, Englewood Cliffs, NJ, Second edition, 1992.
- [UML 97] The Unified Modeling Language (UML) Specification – Version 1.1, September 1997. Joint Submission to the Object Management Group (OMG).
- [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 99] UML 1.3. The Unified Modeling Language (UML) Specification – Version 1.3. Object Management Group. [www.omg.org](http://www.omg.org).
- [VisualWorks 98] VisualWorks Non-Commercial 3.0. ObjectShare Inc. May 8, 1998. [www.objectshare.com](http://www.objectshare.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, Buenos Aires, 1998.
- [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.
- [Wong 84] C.Wong, A successful software development, IEEE Transactions on Software Engineering, 10(6), November 1984.