UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

UN FRAMEWORK PARA EL MANEJO DE CONSISTENCIA EN DISEÑOS UML 2.0

JOCELYN PAOLA SIMMONDS WAGEMANN

2005

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

UN FRAMEWORK PARA EL MANEJO DE CONSISTENCIA EN DISEÑOS UML 2.0

JOCELYN PAOLA SIMMONDS WAGEMANN

COMISIÓN EXAMINADORA		**CALIFICACIONES**		
		NOTA (n^o)	(Letras)	FIRMA
PROFESOR GUÍA: M. CECILIA BASTARRICA	:	.....	.....	.....
PROFESOR CO-GUÍA: LUIS GUERRERO	:	.....	.....	.....
PROFESOR INTEGRANTE: OSCAR HERRERA	:	.....	.....	.....
NOTA FINAL EXAMEN DE TÍTULO	:	.....	.....	.....

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO DE CHILE
ENERO, 2005

Resumen

El presente trabajo tiene como objetivo principal el desarrollo de un framework flexible que permita tratar el problema de consistencia entre diagramas definidos en UML 2.0. ¿Por qué esta procupación específica por UML 2.0? La razón es simple: UML es el lenguaje de facto para el modelamiento orientado a objetos desde el año 1997 y UML 2.0 es su versión actual.

UML 2.0 es el estándar dentro de la industria, pero esto no significa que sea la solución definitiva. El estándar no define una clara relación entre la semántica de los distintos diagramas, ni ofrece políticas de versionamiento en el caso de la evolución de un modelo. Los creadores de UML justifican estas decisiones aduciendo a que no toda inconsistencia es accidental. Por ejemplo, cuando se hace un diseño en forma top-down, se inicia el proceso de diseño con un modelo incompleto, por lo tanto inconsistente.

Esta posición tiene una consecuencia importante: dado que UML no obliga a mantener la consistencia entre los distintos tipos de diagramas, las herramientas que cumplen con el estándar tampoco están obligadas a mantenerlo. Debido al tamaño y complejidad de los sistemas desarrollados se hace indispensable el uso de herramientas CASE para apoyar el proceso de desarrollo. Por lo tanto, el usuario de UML tiene pocas opciones - debe procuparse de la consistencia de sus diseños en forma manual.

Algunas inconsistencias han sido estudiadas en profundidad, con técnicas y herramientas diferentes para cada tipo de inconsistencia. Estas razones motivaron la implementacion de una herramienta usable y extendible que aplique una política uniforme con respecto a la detección de problemas, donde el usuario no esta pendiente del formalismo usado.

El resultado de esta memoria es una aplicación denominada “Model Consistency Checker” (MCC) y el formalismo que utiliza es “Description Logics” (DL), dado que experimentos anteriores probaron que este formalismo servía de base para la detección y resolución de un conjunto amplio de problemas de consistencia. Racer, un motor de inferencia para DL que es completo y eficiente, brinda los servicios de razonamientos usados por MCC.

La interfaz gráfica del sistema está integrada como plug-in a Poseidon, una herramienta CASE para UML. En consecuencia, la aplicación generada necesita de Poseidon y Racer para funcionar adecuadamente y como ambas son portables (Linux, Mac OS X y Windows) y MCC esta implementada en Java, se obtiene una herramienta portable.

Dada la importancia de sistemas basados en componentes, se experimentó con la extensión de la herramienta para incluir análisis de consistencia de diagramas de componentes con respecto a los diagramas ya considerados. Se definió la inconsistencia *Servicio no disponible*, que ocurre cuando la interfaz ofrecida por una componente incluye algún servicio que no está implementado como una operación pública por alguna de las clases interiores de la componente. Esta extensión fue un éxito.

Agradecimientos

Quisiera agradecer a todas las personas que participaron en esta memoria, en forma directa o indirecta. Mi familia, mi pololo, mis amigos, los academicos y funcionarios del DCC. Por las palabras de aliento, los consejos, las fiestas y las memorias...

Nota: un agradecimiento especial para la persona que me introdujo a B & G...

Índice general

Capítulo 1

Introducción

El desarrollo de software industrial es un esfuerzo colaborativo de un equipo multidisciplinario donde participan analistas, diseñadores, implementadores y testeadores por un lado y clientes y usuarios finales por el otro lado. Dada la complejidad y el tamaño de este tipo de proyectos, surge la necesidad de un canal estandarizado de comunicación entre los diferentes actores involucrados, que le de al equipo la capacidad de poder hablar un "idioma común". Así, los diferentes roles involucrados en el desarrollo de aplicaciones pueden expresar y compartir ideas en forma fluida y no ambigua.

Dada la necesidad de un lenguaje de especificación, varias metodologías surgieron, algunas propuestas por empresas de desarrollo, otras por laboratorios académicos. Gran parte de estas metodologías usaron diagramas como una forma de representar diseños, por la baja curva de aprendizaje asociada. Entre estas metodologías, existían varios puntos comunes con respecto a la información representada. Además, varias tecnologías eran complementarias. En particular, la unión de las metodologías de Booch (Booch)[4], OMT-2 (Rumbaugh)[38] y OOSE (Jacobson)[23] llevo a la creación de UML (Unified Modelling Language), en 1996.

UML es un lenguaje de modelamiento visual que provee al equipo de desarrollo de software de una familia de diagramas que permiten especificar la estructura y el comportamiento del sistema a construir. Cada tipo de diagrama representa un aspecto distinto de un sistema, y los diagramas combinados dan una visión global del software. Este lenguaje fue adoptado por el Object Management Group como el estándar de-facto para el modelamiento orientado a objetos en 1997 (UML 1.1). UML está definido por una sintaxis abstracta, un conjunto de reglas de buen uso (especificadas en OCL - Object Constraint Language) y semántica

expresada en lenguaje natural.

La especificación anterior de UML es la version 1.5 [33], y fue publicada en Marzo del 2003. Con respecto a la especificación anterior (UML 1.4), los cambios se centraron en la inclusión de la semántica de acciones ejecutables y procedimientos. Inconsistencias detectadas en estas especificaciones por la comunidad no fueron arregladas, sino que se inicio la elaboración de la especificación 2.0 de UML, que finalmente se aprobó en Septiembre del 2004. Las especificaciones de UML 2.0 incluyen la Superstructure y Infrastructure [37, 29], las cuales definen la sintaxis y semántica del lenguaje, OCL[31] y XMI[34] (XML Metadata Interchange).

1.1. Motivación

Con esta nueva versión de UML (2.0), se espera dar solución a varios problemas detectados en las versiones anteriores. Uno de los problemas más importantes detectado es la imposibilidad de especificar la relación entre distintas versiones de modelos, diagramas y elementos de diagramas. En el figura 1.1 se pueden apreciar las tres posibles relaciones entre modelos. Dos modelos están relacionados horizontalmente cuando pertenecen a una misma versión de un diseño, por ejemplo, la relación entre el diagrama de clase v1 y el diagrama de estado v1. Se dice que dos modelos están relacionados verticalmente cuando un modelo es un refinamiento de otro, por ejemplo, la relación entre el diagrama de clase v1 y la primera versión del código fuente. Finalmente, un modelo es la evolución de otro si pertenece a una versión posterior de un mismo diseño, por ejemplo, el diagrama de clase v1 y el diagrama de clase v2.

No existe mecanismo alguno en la especificación anterior de UML (1.5) que relacione elementos de modelos relacionados en forma horizontal, vertical o evolutiva. Al no estar especificada esta relación, se pierde mucha información implícita que puede ser usada para revisar la consistencia lógica entre diagramas de un modelo y entre modelos. Según las especificaciones parciales disponibles de UML 2.0 [36, 35, 30, 32] aún no se ha incluido en la nueva especificación un mecanismo que cumpla con esta función. Una solución a este problema ha sido planteada en [41, 43] con la declaración de un profile para la evolución de modelos y la detección de inconsistencias usando Description Logics (DL)[3].

DL es un fragmento decidible de lógica de primer orden. Se define una terminología que permite describir al dominio que es modelado, usando un lenguaje de representación. Esto

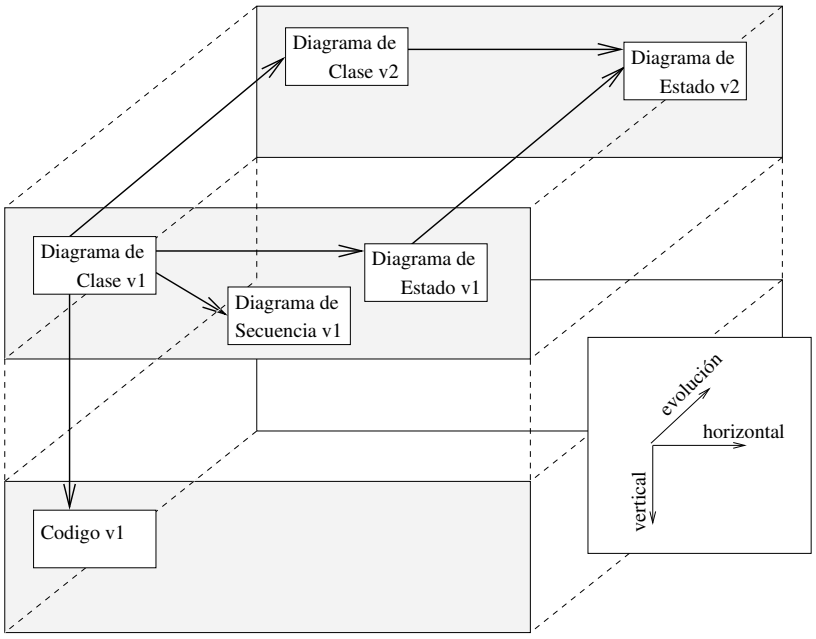

Figura 1.1: Ejemplo de las relaciones que existen entre modelos UML

se hace especificando los conceptos existentes en el dominio y las relaciones que los unen. El conjunto de conceptos y relaciones que definen un dominio se conoce como el *Tbox*. Estos conceptos y relaciones[1] se pueden instanciar, imitando objetos reales del sistema a modelar. El conjunto de aserciones de instanciación se conoce como la *Abox*. Se puede entonces razonar con respecto a la consistencia de estos individuos.

En [41, 43] se usó el metamodelo de UML para la definición del dominio, modelando las metaclases como conceptos y las asociaciones entre estas metaclases como roles. Los modelos UML creados por usuarios se modelaron como instancias de los conceptos. De esta forma, se pudo hacer pruebas para determinar si era posible detectar problemas de consistencia entre diagramas. Experimentos llevados a cabo usando DL para detectar inconsistencias en ejemplos traducidos manualmente fueron exitosos, lo que validó la técnica. La dificultad de aprender DL y la falta de una herramienta que traduzca los modelos UML en DL implicaba que no fuera usable como técnica de validación a gran escala.

[1]Conocidas como roles en la terminologia del área

1.2. Objetivos

El objetivo general de esta memoria es la implementación de un framework gráfico que permita a usuarios como analistas y diseñadores de sistemas detectar y solucionar inconsistencias entre los distintos diagramas que pertenecen a un modelo. Las inconsistencias a detectar y solucionar son las mencionadas en [41], las cuales tendrán que ser reformuladas para incorporar los cambios introducidos en la nueva especificación de UML. Dado que DL es un formalismo con una alta curva de aprendizaje, la idea es ocultarlo del usuario final.

A continuación se presenta una lista de objetivos más específicos:

- Estudiar el mapeo XMI - DL, para poder realizar traducciones en forma automática.
- Diseñar el framework de manera flexible, para que la incorporación de cambios en la especificación UML se pueda hacer sin tener que reescribir la herramienta. También debe soportar la incorporación de nuevas revisiones de consistencia.
- En [41, 43] se estudió la existencia de 18 problemas de consistencia entre diagramas de clase, estado y secuencia. Como objetivo se tiene la implementación de la detección de 5 de estos problemas, como prueba de concepto.

1.3. Metodología

Para un adecuado término del trabajo es necesario desarrollar los siguientes pasos:

- Análisis de requerimientos del framework.
- Estudio de soluciones para problemas similares mediante una revisión bibliográfica.
- Generar una propuesta de solución.
- Definir el ambiente de trabajo y las herramientas necesarias.
- Generar un diseño para dicha solución.
- Implementación.
- Ajustes y documentación.

1.4. Antecedentes

Distintas metodologías han sido propuestas en la literatura para la revisión de consistencia de modelos en UML. Una metodología general para el manejo de problemas de consistencia asociados a la herencia entre máquinas de estado protocolares ha sido propuesta por Engels et al. [10]. Los requerimientos de consistencia se describen usando los Procesos Secuenciales Comunicados (CSP) como el modelo matemático. Entonces, los usuarios de esta técnica deben estar familiarizados con la teoría de CSP. Ehrig y Tsiolakis [9] han investigado la consistencia entre diagramas de clase y diagramas de secuencia. Usan grafos para representar a los diagramas de clase y describen a los diagramas de secuencia utilizando gramáticas. Nuevamente, el usuario debe estar familiarizado con un formalismo más, el cual sólo sirve para tratar a una familia de problemas de consistencia.

El verificar si una secuencia de acciones especificada en un diagrama de colaboración puede ser realizada dado un conjunto de máquinas de estado es un problema que ha sido investigado por Schfer et al. [39]. Este grupo de colaboradores ha desarrollado la herramienta HUGO, un prototipo que revisa la correspondencia entre máquinas de estado y diagramas de colaboración. Modelos PROMELA [17] son utilizados para representar las máquinas de estado y los diagramas de colaboración se formalizan a través de autómatas de Buchic. Finalmente, se usa el verificador de modelos SPIN para comparar el modelo con el autómata.

Este mismo problema ha sido analizado por Litvak et al. [24]. Este grupo utiliza algoritmos en vez de verificadores externos de modelos. Concretamente, han implementado la herramienta BVUML, la cual recibe los diagramas de estado y secuencia como archivos XMI producidos por ArgoUML. Utilizar HUGO implica de que el usuario debe traducir sus modelos a distintos formalismos, lo cual es un proceso engorroso y propenso a errores si la máquina de estado a chequear es compleja y el usuario no esta completamente familiarizado con el formalismo. La herramienta BVUML evita este problema de traducción al aprovechar la interfaz gráfica ofrecida por ArgoUML.

Rational Rose [21] es una herramienta CASE para UML bastante popular, que ofrece funcionalidades para llevar a cabo revisiones de consistencia de modelos. Estas revisiones se aplican sobre el modelo completo, aplicándose todas las pruebas disponibles - el usuario no puede especificar cuales serán los chequeos a aplicar. Si el modelo es complejo, el proceso puede ser bastante largo. La salida de la revisión, que se presenta directamente al usuario, es críptica y debe ser accedida desde el log de errores. Una interfaz fue desarrollada para

este revisor interno - el Rose Model Checker [27]. Esta herramienta toma la salida producida por el revisor interno de Rational y trata de generar resultados más comprensibles. Este da soporte para problemas de enlaces no existentes entre elementos, por ejemplo, clases y asociaciones no existentes y genera también estadísticas sobre el modelo, pero no provee revisiones de consistencia más complejas, como comportamiento no compatible o servicios no disponibles.

Uno de los problemas más comunes en el desarrollo de frameworks evolutivos es cómo hacer evolucionar la arquitectura del sistema sin impactar a las aplicaciones ya existentes. Dada la variedad de las metodologías usadas para modelar arquitecturas de sistemas, existen pocas herramientas de apoyo a este proceso. Por ejemplo, en [8] se presenta un profile para UML (UML-F) que se usa para formalizar el proceso de mantención y evolución de frameworks, pero todavía no existen herramientas de apoyo para su uso.

Usar lenguajes de descripción de arquitectura (ADL) formales para hacer la descripción de una arquitectura es algo que se reserva usualmente para sistemas críticos, dado que la aplicación de ADLs a un problema es un proceso largo y complejo. Además, la mayoría del trabajo hecho en ADLs es académico. Existen además, varios ADL, así que también existe el problema de comunicar y comparar especificaciones en distintos ADL. UML 2.0 provee varios elementos especializados para la especificación de arquitecturas, y aún cuando no es el ADL más expresivo, es bastante poderoso si se considera de que las especificaciones escritas en UML se integran con los diseños detallados en UML, y además son fáciles de entender al ser diagramático. Existe también bastante experiencia de uso de UML en la industria.

1.5. Organización de la memoria

Los capítulos 2 y 3 hacen una descripción general de UML y Description Logics, respectivamente. En el capitulo 4, se define la solución a implementar. El análisis y diseño de la solución propuesta se ve en el capítulo 5. En el capítulo 6 se especifican los detalles más importantes de la implementación. No se busca dar un detalle de todo lo implementado; si eso es lo que se desea, se puede ver el código fuente [42]. El capítulo 7 muestra a través de un ejemplo, los resultados obtenidos en el presente trabajo. Finalmente en el capítulo 8 se encuentra una breve discusión sobre los alcances, los futuros trabajos y se describen las conclusiones de esta memoria.

Capítulo 2

UML

Al desarrollar grandes proyectos de ingeniería de software, es altamente conveniente contar con un proceso formal de modelamiento, en el cual se estudien las posibles soluciones, especificándose finalmente la solución escogida en la forma de un modelo. Este modelo es una herramienta clave para la comunicación de un equipo de desarrollo, y además, permite analizar el diseño propuesto. Este proceso es necesario sobre todo cuando los sistemas a construir son de gran tamaño y/o complejidad. Adicionalmente, contar con una notación estándar que apoye este proceso de diseño promueve la comunicación y el éxito del proceso como un todo. UML (Unified Modeling Language) es una posible notación.

2.1. UML 1.5

UML[33] es un lenguaje de modelamiento visual que provee al equipo de desarrollo de software con una familia de diagramas que permiten especificar la estructura y el comportamiento del sistema a construir. Como lenguaje de modelamiento visual, define varios conceptos de modelamiento, la semántica de estos, la notación visual correspondiente y una guía de uso. UML como lenguaje ha sido ampliamente aceptado en el desarrollo de sistemas que usan orientación a objetos y componentes como paradigmas.

En un proyecto, existe la necesidad de especificar los requerimientos y el diseño. Para esto, UML ofrece tres categorías de modelos: de requerimientos, estructurales y dinámicos (especificación del comportamiento del sistema). Una especificación completa de un sistema deberá incluir diagramas de las tres categorías, generando una visión global del sistema.

La categoría de modelos de requerimientos está representada por los modelos de Casos de Uso. Estos diagramas muestran en forma compacta la funcionalidad ofrecida por el sistema, y los agentes externos con los cuales interactúa.

Con respecto a la estructura de un sistema, se puede especificar la arquitectura lógica y la física. La arquitectura lógica del sistema queda detallada en diagramas de Clases, donde se especifican las clases del sistema y el conjunto estático de las relaciones entre éstas. También es posible modelar la estructura lógica usando objetos en vez de clases - un diagrama de objetos estáticos corresponde a una instancia del diagrama de clases, mostrando una vista detallada del sistema en un instante específico.

La arquitectura física se detalla usando diagramas de componentes y deployment[1]. Un componente es un módulo del sistema que puede ser instalado en forma independiente y cuya implementación puede ser reemplazada sin afectar al resto del sistema. Encapsula su implementación y define un conjunto de interfaces. En los diagramas de componentes, se muestra la organización de los componentes y las dependencias entre ellos. En los diagramas de deployment, se especifican configuraciones de componentes y nodos (ubicaciones) donde serán instalados los componentes.

Finalmente, con respecto a la especificación del comportamiento del sistema, se puede modelar el comportamiento de las clases de un sistema en forma individual - máquinas de estado, o en conjunto - diagramas de interacción. Cada máquina de estado está relacionada con una clase del diagrama de clases. Estas máquinas son autómatas finitos no determinísticos, es decir, desde un estado es posible tomar varios caminos simultáneamente, siendo escogido uno al azar. Los diagramas de actividad son complementarios a estas máquinas, permitiendo la especificación de una operación de una clase.

Para la especificación de la interacción entre objetos en el transcurso del tiempo, existen dos tipos de diagramas: colaboración y secuencia. Semánticamente, estos diagramas son equivalentes, sólo muestran la información de una forma distinta. En el diagrama de colaboración, se muestra las interacciones entre los objetos organizadas en torno a la estructura del diagrama de clases. El diagrama de secuencia se centra en los períodos de vida de los diferentes objetos. En ambos diagramas se muestran los mensajes que reciben y envían los distintos objetos.

[1]Deployment se refiere al proceso de distribuir e instalar una aplicación

2.2. Diagramas

A continuación se detallará más en profundidad cada uno de los dagramas incluidos en UML. La notación de los elementos no ha cambiado en las distintas versiones, para mantener compatibilidad entre modelos en distintas versiones. Se pueden encontrar más detalles con respecto a estos diagramas y su uso en [5, 12].

2.2.1. Diagrama de Casos de Uso

Los diagramas de casos de uso muestran la funcionalidad esperada del sistema, como casos de uso individuales. Estos se representan con un óvalo por cada caso de uso, identificándolo por un nombre que evoque la secuencia de acciones asociadas al caso de uso. Además, se definen las entidades que interactuarán con el sistema - llamadas actores. Estos actores pueden ser otros sistemas, humanos, etc. Cada actor se representa con un ícono de una persona, con el nombre del actor por debajo.

Se muestran las relaciones entre los casos de uso y los actores como un grafo. Las relaciones que se pueden incluir en un diagrama de casos de uso son las asociaciones entre actores y casos de usos, generalizaciones entre actores, generalizaciones entre casos de uso y extensiones e inclusiones entre casos de uso. Una asociación entre un actor y un caso de uso significa la interacción del actor con esta parte del sistema. Este es el único tipo de relación que existe entre actores y el sistema.

La generalización es una relación de categorización entre elementos más generales (los ancestros) y elementos más específicos (los descendientes). A medida que se atraviesa la jerarquía de actores o casos de uso generadas por relaciones de generalización, siempre se agrega información a los actores o casos de uso.

Una relación de extensión define que el comportamiento de un caso de uso puede ser extendido utilizando el comportamiento ya definido en otro caso de uso. Por ejemplo, una bencinera puede extender sus servicios *limpiando* las ventanas del auto después de echar bencina, tal como se muestra en la figura 2.1. El orden en que se incluye comportamiento extendido depende del caso de uso que extienda su comportamiento. En el mismo ejemplo, la bencinera deberá decidir si lavar los vidrios antes o después de echar bencina.

Una relación de inclusión define que un caso de uso incorpora el comportamiento de otros casos de uso como propio. En el ejemplo de la bencinera, que además de bencina,

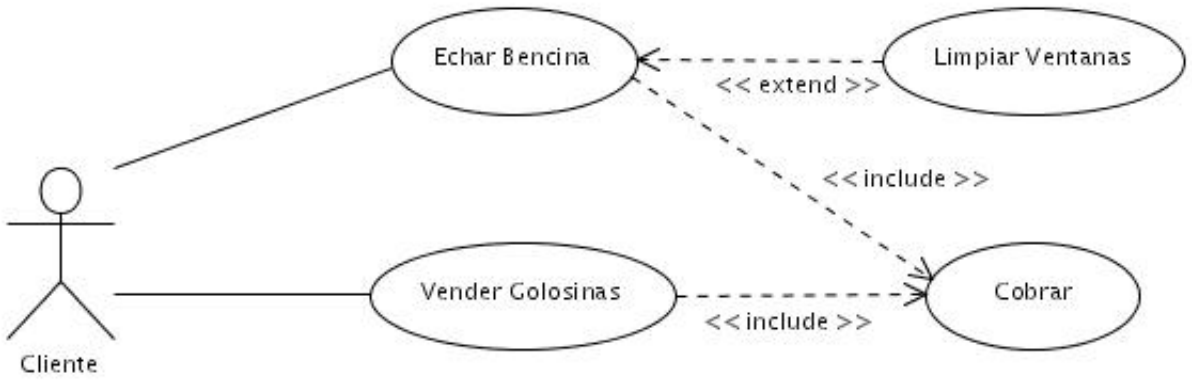

Figura 2.1: Ejemplo de un diagrama de casos de uso

vende golosinas, los casos de uso "Echar bencina" y "Vender golosina" ambos incluyen el comportamiento de un tercer caso de uso, "Cobrar", en la cual se mapea la acción de cobrar al cliente por sus compras.

2.2.2. Diagrama de Clase

Un diagrama de clase es un grafo, donde los nodos son elementos del tipo clasificador y los arcos son las distintas relaciones estáticas que existen entre estos elementos. El nombre de "diagrama de clase" es convención, porque este diagrama puede contener clases, interfaces, y paquetes. En este diagrama se especifica los elementos del sistema, que serán referenciados por el resto de los diagramas, lo cual lo convierte en el diagrama central de un modelo. Se puede ver un ejemplo de este tipo de diagrama en la figura 2.2.

Clases e interfaces son representados de la misma forma, en la forma de una caja rectangular. Dado que las clases son más frecuentes que las interfaces, se diferencia las interfaces etiquetándolas como interfaces. Una clase representa un concepto del sistema que está siendo modelado. Una clase define la estructura (atributos) y comportamiento (métodos) de los objetos que la instancian. Se permite la definición de clases anidadas.

Las relaciones que existen entre clases son: asociación, agregación, composición y generalización. La relación más básica entre clases es asociación, representada por una línea. El caso más común es una asociación binaria, pero se permite la modulación de relaciones entre N clases. Cada extremo de una asociación debería llevar un nombre y una multiplicidad.

Para relaciones más complejas, se permite modelar propiedades de la relación a través de una clase de asociación.

Agregación y composición son casos especiales de asociación. Agregación indica de que la clase involucrada en una relación de agregación forma parte de la otra clase involucrada en la relación. La relación de composición es una variedad más fuerte de agregación - la instancia involucrada sólo puede participar en una relación de composición a la vez, y además, el objeto compuesto tiene la responsabilidad sobre la creación y destrucción de sus partes.

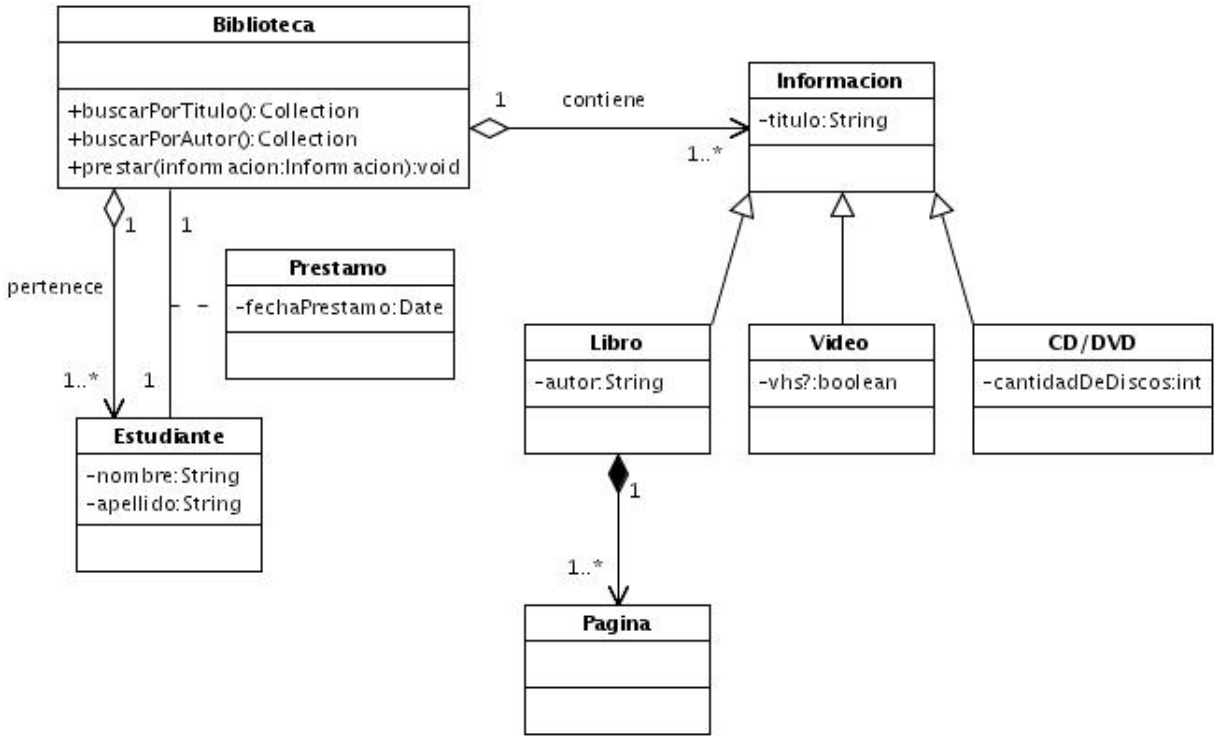

Figura 2.2: Ejemplo de un diagrama de clases

La generalización es una relación de categorización entre elementos más generales (los ancestros) y elementos más específicos (los descendientes). A medida que se atraviesa la jerarquía de clases generadas por relaciones de generalización, siempre se agrega información a las clases, heredándose atributos, métodos y asociaciones. Una instancia de una clase siempre debe poder reemplazar a una instancia de cualquiera de sus ancestros.

Las interfaces especifican conjuntos de métodos que son visibles con respecto a una clase o componente. Esta especificación no incluye la implementación de estos métodos. En UML

1.5, estas interfaces no tienen atributos o estados, sólo métodos. Las clases que se relacionan con una interfaz pueden: ofrecer una implementación de la interfaz, o requerir una implementación de la interfaz. Las interfaces pueden también estar relacionadas a través de relaciones de generalización.

Finalmente, los paquetes agrupan clases e interfaces y las relaciones entre estos, en conjuntos lógicos. El paquete define el espacio de nombres de los elementos que pertenecen a él. Los paquetes pueden estar subdivididos en más paquetes, para estructurar mejor los elementos que pertenecen al paquete. Se definen relaciones de dependencia entre paquetes, definiendo que la funcionalidad de un paquete depende de la funcionalidad ofrecida por otro paquete.

2.2.3. Diagrama de Objetos

Diagramas de objetos es un caso especial de diagramas de clases. Muestra, en la forma de un grafo de objetos, una configuración del sistema en un momento determinado del tiempo. Estos objetos muestran valores reales de datos. Como las clases en los diagramas de clases, los objetos se muestran como cajas rectangulares, anteponiéndose un ":" al nombre de la clase que instancia el objeto. Se puede identificar los objetos en forma individual, colocando un nombre antes del ":". En un diagrama de objetos puede haber más de una instancia de la misma clase.

El uso de este tipo de diagramas es bastante limitado, usándose principalmente para mostrar ejemplos de estructuras de datos complejas. Se usan también para mostrar configuraciones especiales de sistemas, como por ejemplo, la configuración inicial de un sistema. Un ejemplo de este tipo de diagrama se muestra en la figura 2.3.

2.2.4. Diagrama de Componentes

Un diagrama de componentes muestra las dependencias entre componentes del sistema, como también clasificadores que los especifican (clases, interfaces, etc.) y los artefactos que implementan los componentes (código fuente, clases compiladas, ejecutables, etc.). Este diagrama muestra un grafo de componentes conectadas por relaciones. Entre componentes se puede tener relaciones de dependencia y composición.

Las clases e interfaces que forman parte de un componente están relacionadas con su com-

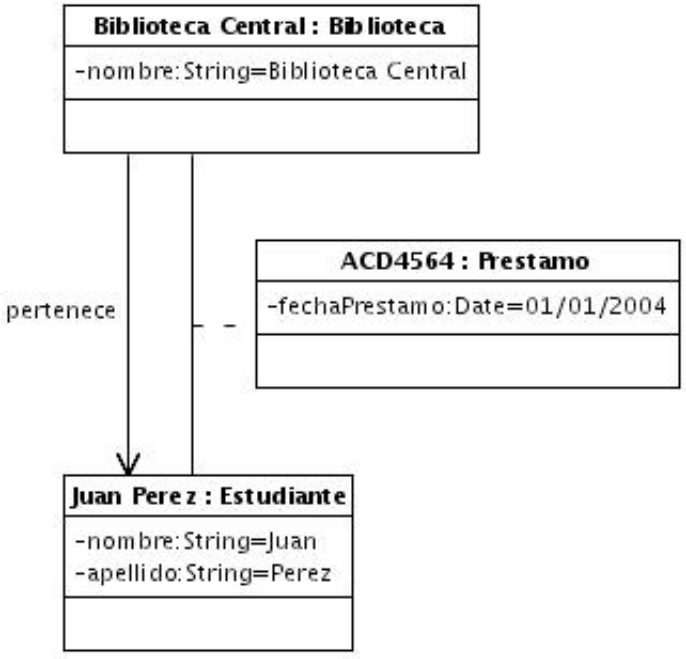

Figura 2.3: Ejemplo de un diagrama de objetos

ponente contenedor a través de relaciones de "residencia". De la misma forma, artefactos que implementan componentes están conectados a éstos a través de relaciones de "implementación". Un uso típico de este diagrama es mostrar dependencias de bibliotecas estáticas entre programas.

Los componentes no tienen sus propias características, como atributos u operaciones, pero pueden actuar como contenedores para clasificadores que sí tienen características propias. Es común asociar interfaces a los componentes, para identificar los servicios ofrecidos por los elementos que residen en el componente. En estos casos, las relaciones de dependencia entre los componentes se dibujan entre las interfaces correspondientes. Un ejemplo de este tipo de diagrama se muestra en la figura 2.4.

2.2.5. Diagrama de Deployment

Los diagramas de deployment se usan para especificar vistas estáticas de configuraciones dinámicas de nodos y de los componentes que se ejecutan en esos nodos. Los nodos son el hardware específico donde se instalará la aplicación y los componentes son partes de la

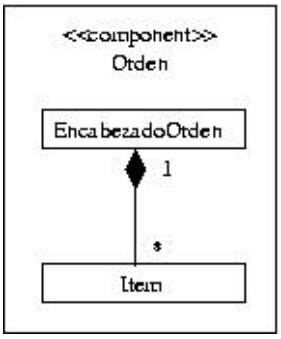

Figura 2.4: Ejemplo de un diagrama de componentes

aplicación. Los arcos entre los nodos son las vías de comunicación del sistema. Este tipo de diagrama es esencial cuando un software va a ser instalado en forma distribuida en varias máquinas. También se usan pasa explorar la arquitectura de sistemas embebidos, para experimentar cómo interactúan el hardware y software.

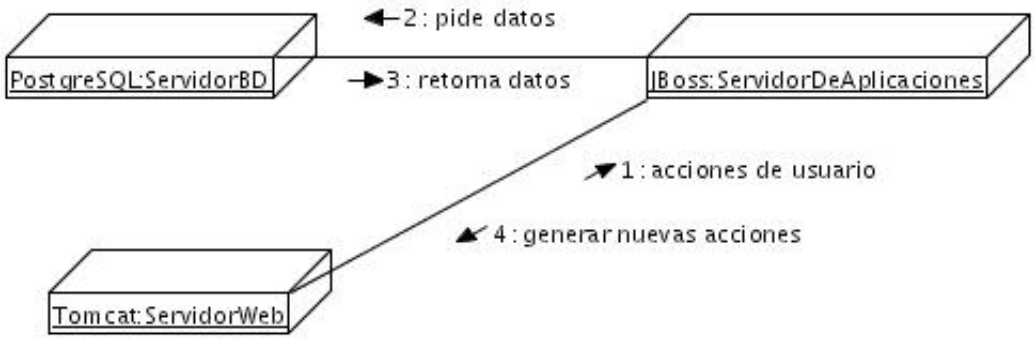

Figura 2.5: Ejemplo de un diagrama de deployment

2.2.6. Diagrama de Estado

Los diagramas de estado especifican el comportamiento de las entidades capaces de desarrollar comportamiento dinámico, en la forma de autómatas finitos no determinísticos. La idea es modelar los cambios de estado de una instancia al recibir eventos. Estos estados representan las diferentes etapas de la vida de una instancia de la clase especificada. El uso más común es para describir el comportamiento de las instancias de una clase. También se puede ocupar para especificar el comportamiento de casos de uso, actores, etc.

La relación entre dos estados se conoce como una transición. Se dice que una transición es "activada" cuando se cambia de un estado a otro a través de una transición. Se escoge una transición si es que estando en el estado originario de la transición, se produce el evento que activa la transición. Si un evento activa a más de una transición, se elige una de las transiciones en forma arbitraria. Las transiciones también pueden tener condiciones booleanas asociadas, llamadas guardias, las cuales se evalúan cuando ya se ha activado la transición. También se les puede asociar acciones, las cuales se ejecutarán sólo si la transición es gatillada y las condiciones asociadas son verdaderas. Una transición se representa con una flecha entre dos estados. El evento, condición y acción asociadas se escriben junto a la transición, como *evento [condición] / acción*. Un ejemplo de este tipo de diagrama se muestra en la figura 2.6.

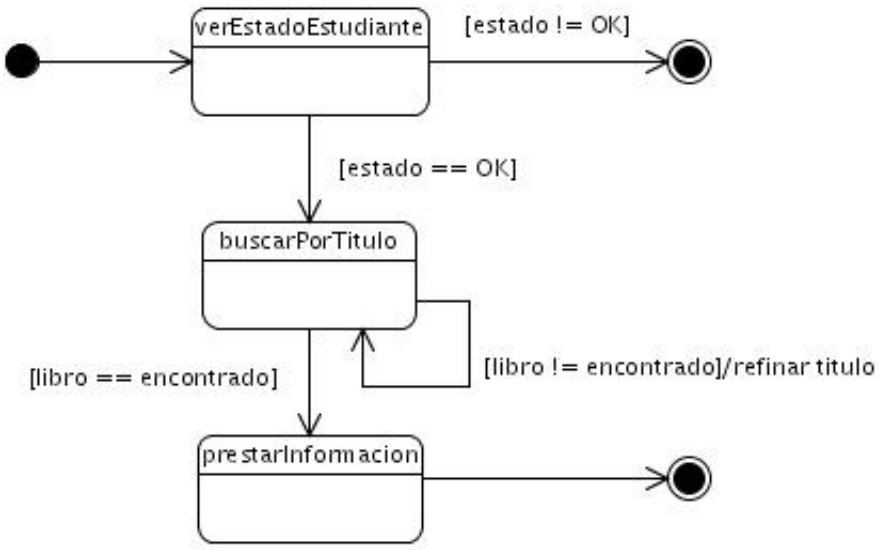

Figura 2.6: Ejemplo de un diagrama de estado, para la clase Biblioteca

Cada diagrama de estado puede tener un sólo estado inicial, donde comienza cualquier secuencia de acciones que puede llevar a cabo una instancia de una clase. En contraste, un diagrama de estado puede tener tantos estados terminales como quiera. Los estados iniciales se representan con círculos rellenos negros y los finales con círculos rellenos blancos con un círculo relleno negro más pequeño dentro. Los estados propios de una clase son estados, donde se puede llevar a cabo acciones o esperar la producción de algún evento. Estos se representan con cajas con los bordes redondeados.

Los estados pueden ser simples o compuestos. Un estado compuesto se puede descomponer

en dos ó más subestados. Cualquier estado sólo se puede refinar de esta forma. Además, cualquier subestado de un estado compuesto puede ser un estado compuesto. Un estado compuesto puede tener su propio estado inicial y sus estados finales.

2.2.7. Diagrama de Actividad

Los diagramas de actividades se utilizan para modelar el orden de una serie de actividades en la forma de un grafo. También se puede utilizar este diagrama para modelar el flujo de datos y objetos entre actividades. Los nodos son actividades concretas como llamadas a operaciones y actividades compuestas, las cuales son conjuntos de actividades que representan acciones compuestas como métodos. Al igual que las máquinas de estado, estos diagramas tienen un estado inicial y varios estados finales. La notación para este diagrama es la misma usada en las máquinas de estado.

A diferencia de las máquinas de estado, donde una transición entre estados se activa al ocurrir un evento externo, una actividad se activa cuando termina la actividad previa y los objetos o datos necesarios como entrada están listos. Este diagrama está orientado a modelar las actividades que ocurren, y no a especificar qué objeto lleva a cabo qué tarea.

Este diagrama soporta el modelamiento de concurrencia dinámica, es decir, invocaciones paralelas de una operación. Se utiliza estados de sincronización para la comunicación entre procesos concurrentes. Para indicar el flujo de datos, se puede incluir objetos en este diagrama. La actividad que se conecta a un objeto debe producir como salida un objeto de la clase correspondiente al objeto al cual se conecta. De la misma forma, una actividad que recibe un objeto debe poder recibir un objeto de la clase correspondiente. Un ejemplo de este tipo de diagrama se muestra en la figura 2.7.

2.2.8. Diagrama de Colaboración

Un diagrama de colaboración muestra un grafo de las interacciones entre instancias, en forma no estructurada. Las interacciones se muestran como flechas entre los objetos, etiquetadas por los mensajes que son enviados por los objetos. Los mensajes corresponden a las operaciones de las clases correspondientes. Estos mensajes tienen adjunto un número, que indica el orden del mensaje en la secuencia de comunicación entre los objetos. El sentido de la flecha indica el sentido de la interacción. Cuando se anidan llamados, la numeración de

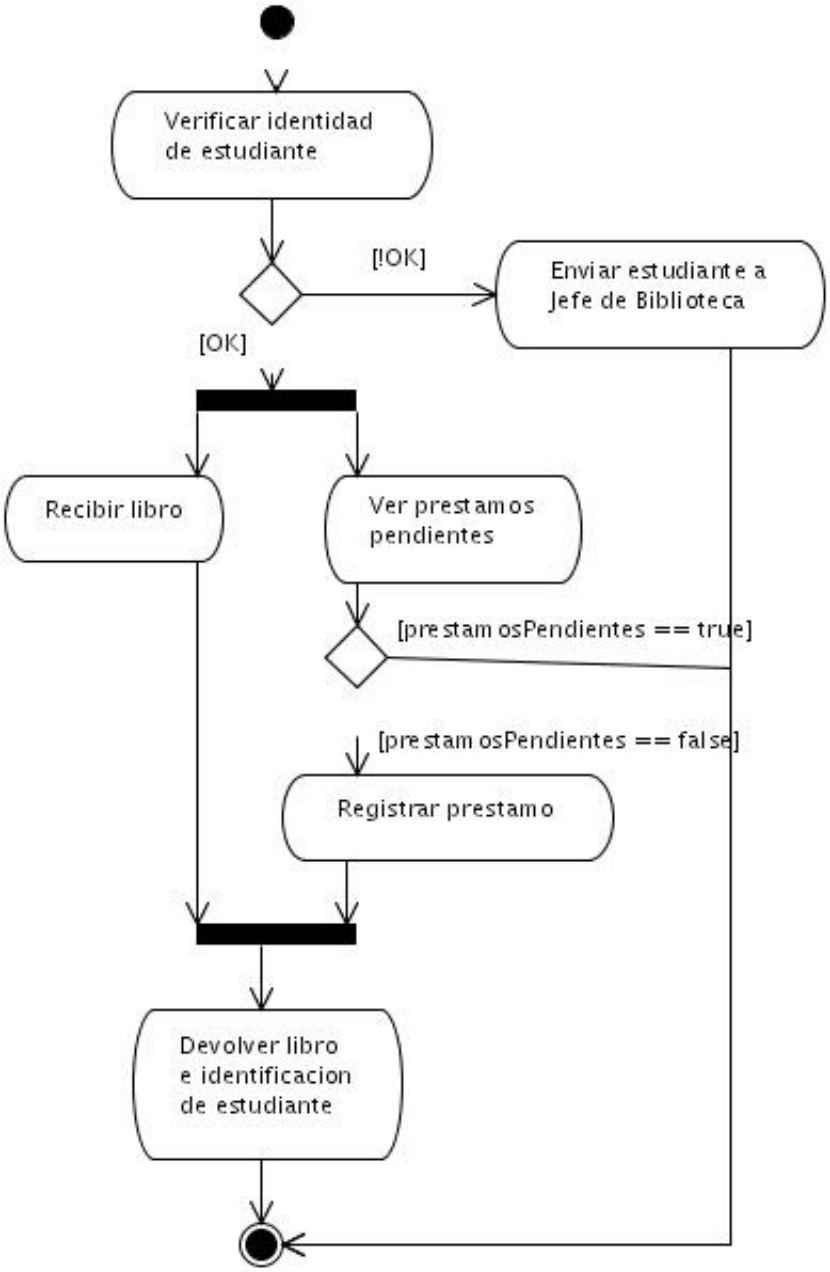

Figura 2.7: Ejemplo de un diagrama de actividad

los mensajes también se anida. Un ejemplo de este tipo de diagrama se muestra en la figura 2.8.

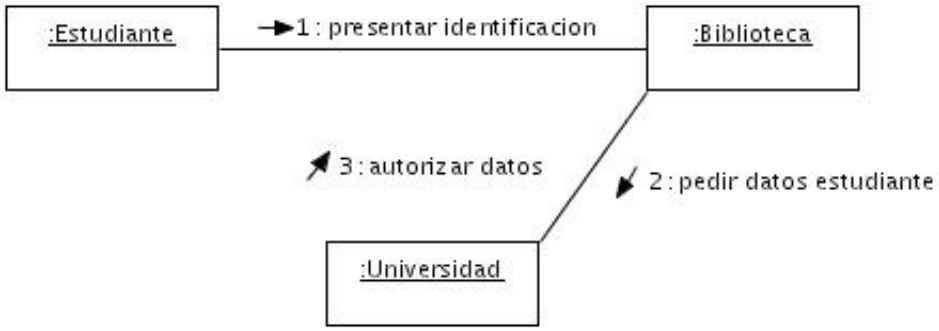

Figura 2.8: Ejemplo de un diagrama de colaboración

2.2.9. Diagrama de Secuencia

Este diagrama muestra la misma información que el diagrama de colaboración, pero en forma estructurada, utilizando para esto dos dimensiones: tiempo e instancias. El tiempo se representa en el eje vertical, en forma creciente hacia abajo. En el eje horizontal, se colocan los diferentes objetos que interactúan en el diagrama. No tiene importancia el orden horizontal de las instancias. Cada instancia tiene su propio eje de tiempo, en la cual está activo el objeto respectivo. Estos ejes son paralelos entre sí. Las interacciones entre instancias se muestran como flechas horizontales entre los ejes de tiempo de estas. También pueden ser diagonales hacia abajo. Se usan diferentes estilos de líneas y flechas dependiendo del tipo de mensaje que se está enviando, como por ejemplo, la creación, destrucción de una instancia, comunicación sincrónica, retornos, etc. Un ejemplo de este tipo de diagrama se muestra en la figura 2.9.

2.3. Cambios introducidos por UML 2.0

La especificación anterior de UML es la version 1.5 [33], data de Marzo del 2003. Con respecto a la especificación anterior (UML 1.4), los cambios se centraron en la inclusión de la semántica de acciones ejecutables y procedimientos. Inconsistencias detectadas en estas especificaciones por la comunidad no fueron arregladas, sino que se comenzó trabajo en la elaboración de la especificación 2.0 de UML, que finalmente se aprobó en Septiembre del

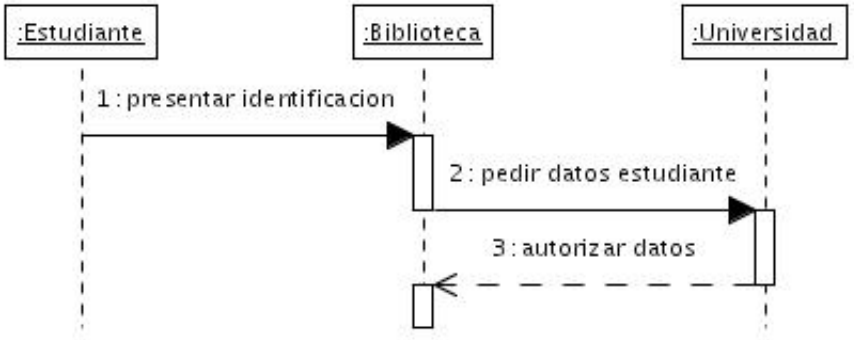

Figura 2.9: Ejemplo de un diagrama de secuencia

2004. Las especificaciones de UML 2.0 incluyen la Superstructure y Infrastructure [37, 29], las cuales definen la sintaxis y semántica del lenguaje, OCL[31] y XMI[34] (XML Metadata Interchange).

2.3.1. Interfaces

El desarrollo de aplicaciones basadas en componentes [26] ha tenido un auge en los últimos años. Esta técnica se enfoca en construir sistemas grandes integrando componentes ya desarrolladas. El uso de componentes ya desarrolladas significa que los costos de desarrollo deberían bajar y los tiempos de desarrollo (que ahora son de "armado") también deberían bajar. La idea central de este movimiento es la suposición de que hay partes de grandes sistemas que se repiten suficientes veces como para escribirlos una vez. Estas aplicaciones deberían ser más flexibles y mantenibles. Modificar una funcionalidad significa integrar un componente nuevo en reemplazo del viejo. Un cliente debería poder especificar las funcionalidades que él considera necesarias y se arma una configuración de la aplicación a su medida.

Las versiones anteriores de UML no proveían conceptos muy desarrollados de interfaces y arquitecturas como para el modelamiento de este tipo de sistemas. Se decidió agregar soporte para desarrollo por componentes, agregándolos como conceptos de primera clase a UML 2.0. Con este fin, se introdujo el concepto explícito de puerto (port). Un puerto es usado para exportar servicios específicos de un clasificador. Se cambió la notación de las interfaces, para especificar en forma no ambigua quién es el que ofrece los servicios (ball) y quién los requiere (socket) (ver la figura 2.10). Las interfaces ahora pueden también tener atributos virtuales. Para especificar mejor las interfaces, se permite especificar las secuencias de invocación de las

operaciones de la interfaz usando máquinas de estado protocolares. Una máquina de estado (ver Sección 2.2.6) es protocolar si no tiene acciones de entrada o salida asociadas a sus estados.

Figura 2.10: Ejemplo de la nueva notación de interfaces

2.3.2. Semántica de relaciones

Con respecto a las especificaciones anteriores, había confusión con respecto a varias relaciones que aparecían en la sintaxis abstracta del metamodelo. Había elementos que a veces se mostraban como atributos y otras veces como asociaciones. Esto se ha solucionado en la especificación 2.0 al utilizar sólo asociaciones.

Las dudas que existían con respecto a las especificaciones anteriores con respecto a las relaciones de dependencia de interfaces se han solucionado en esta especificación, incorporando una nueva notación más explícita.

También se ha clarificado la relación de generalización. En las especificaciones anteriores, estaba claro que cuando un par de clases estaban relacionadas por herencia, la subclase heredaba los atributos y operaciones del padre. Lo que no estaba claro era si las asociaciones del padre también se heredaban. Dado lo que se deducía del texto, se podía suponer que se heredaban. Además, no había una sintaxis para indicar que una asociación era heredada de otra. En esta especificación, se ha incluido la herencia de asociaciones explícita para solucionar este problema.

Siguiendo en el tema de las generalizaciones, las especificaciones anteriores no especifican qué relación existe entre máquinas de estado de clases relacionadas por herencia. La nueva especificación incluye gran parte del trabajo de Engles[10] acerca de la herencia entre máquinas de estado, diferenciando entre la sustitución y la extensión de funcionalidades.

2.3.3. Clases compuestas

Los patrones estructurales [13] de diseño describen cómo clases y objetos pueden ser combinados para formar estructuras más grandes. Dos ejemplos de este tipo de patrón son el Composite y Decorator[13]. En el patrón Composite, se tienen objetos que están compuestos por otros, que a la vez también pueden ser compuestos. El patrón Decorator se refiere a cuando se le agrega un wrapper (envoltorio) a una clase, agregando nuevas funcionalidades a través del wrapper (los métodos de la clase original aún son invocados).

Con las especificaciones anteriores, era obligatorio modelar este tipo de situación usando herencia. Un concepto similar existía en la especificación anterior, pero sólo aplicable a objetos. Ahora se ha extendido a clases. En esta especificación, se ha introducido el modelo de clase estructurado. Esto significa que una clase puede estar compuesta de subpartes, con jerarquías de contención, definidas en forma estática.

2.3.4. Secuencia vs Colaboración

En las especificaciones anteriores, diagramas de secuencia y de colaboración han estado especificados juntos, dado que semánticamente contienen la misma información - la descripción de los participantes y la descripción de los patrones de comunicación entre los participantes. Estos conjuntos de instancias se unificaron en un conjunto de instancias para ambos diagramas. Además, se modeló un patrón de comunicación más general para tomar en cuenta los casos especiales de ambos diagramas - por ejemplo, diagramas de colaboración permiten anidación por sintaxis, pero los de secuencia no, donde se hace anidación con condiciones booleanas en los mensajes.

El modelo resultante era engorroso de usar, con muchos elementos superfluos. Estos elementos son almacenados en una forma estandarizada, utilizando el estándar XMI (XML Metadata Interchange), también establecido por la OMG. Varias herramientas [2, 14] no usaban estas construcciones para generar los XMI correspondientes, rompiendo con el estándar de la OMG. La solución en este estándar fue rehacer esta parte del metamodelo, separando los diagramas de secuencia de los de colaboración.

Aprovechando el remodelamiento de los diagramas de secuencia, se le agregó bastante poder de expresión. Ya no se trabaja con diagramas de secuencia, sino que con fragmentos de diagramas que pueden ser compuestos usando ciertos operadores para generar diagramas

más complejos. Esto hace que los diagramas sean más compactos y más fácil de mantener. También se puede descomponer los diagrama por eje de tiempo de un objeto y por mensaje.

Operadores para manejar fragmentos de secuencias:

- sd: fragmento de diagrama con nombre
- ref: referencia a un fragmento de diagrama
- loop: repetir un fragmento
- alt: elegir entre dos fragmentos
- par: regiones concurrentes
- seq: orden parcial
- strict: orden estricto
- opt: opcional
- neg: negación

2.3.5. Actividad vs Estado

En las especificaciones anteriores, diagramas de estado y de actividad han estado especificados juntos, dado que se consideraba que los diagramas de actividad eran un caso particular de las máquinas de estado. Al hacer esto, se impusieron varias condiciones poco naturales a los diagramas de actividad, restrigiendo su expresividad y posible uso. Esto era necesario porque la forma de activación de transiciones en los dos diagramas es completamente distinta. Las transiciones de las máquinas de estado se gatillan con la ocurencia de eventos externos, mientras que las transiciones de los diagramas de actividad se gatillan con la finalización de actividades anteriores.

En la especificacion 2.0 de UML, el metamodelo para actividades fue separado del de máquinas de estado y rediseñado por completo. Aún cuando se mantenga la misma sintaxis, la semántica ha sido remodelada por completo. Más específicamente, una actividad ya no es una subclase de máquina de estado. Los diagramas de actividad ahora están basados en las redes de Petri en vez de autómatas finitos no-determinísticos. El metamodelo define varios

niveles de expresividad. Uno de estos niveles incluye el control de flujo y la invocación de procedimientos. Un nivel más expresivo permite la especificación de flujo de datos.

2.4. OCL

Al ser un lenguaje de especificación visual, los diagramas de UML no siempre son lo suficientemente expresivos como para especificar todos los aspectos que son relevantes en una especificación. Un ejemplo de esto es la especificación de restricciones adicionales con respecto a los objetos que pertenecen a un modelo. A la falta de un lenguaje de especificación formal de restricciones, estas se describen usando lenguaje natural. El problema con el lenguaje natural es que es muy fácil caer en ambigüedades de especificación. Existen lenguajes tradicionales de especificación, pero estos tienen una alta curva de aprendizaje y la necesidad de una formación matemática por parte del usuario, lo que lo hace difícil de usar al usuario promedio.

OCL[31] - Object Constraint Language - fue desarrollado para suplir la necesidad de un lenguaje formal que fuera fácil de leer y escribir, en el contexto de UML. Este lenguaje es un lenguaje puro de especificación, por lo que las expresiones escritas en él no tienen efectos secundarios, es decir, no modifican el estado del modelo consultado. El estado de un sistema nunca va a cambiar como efecto de evaluar una expresión escrita en OCL. En cambio, los usuarios de OCL escriben expresiones que especifican como el sistema cambia de estado.

Es importante notar de que OCL no es un lenguaje de programación, no hay control de flujo. OCL es un lenguaje tipeado. Los tipos en OCL pertenecen a una de dos categorías: el conjunto de tipos predefinidos por la especificación de OCL, como enteros, booleanos, etc. y el conjunto de tipos definidos por el usuario, lo que corresponde a las clases definidas en un modelo. Para que una expresión en OCL este bien formada, debe cumplir con las reglas de tipado del lenguaje. Lo bueno de usar OCL es que da la flexibilidad de independencia de lenguajes de programación.

La siguiente es una lista de los usos más frecuentes de OCL:

- para especificar invariantes con respecto a la definición de clases en el diagrama de clases
- para especificar condiciones previas y posteriores con respecto a operaciones y métodos
- para especificar guardias

- para especificar los receptores de mensajes y acciones
- para especificar restricciones con respecto a operaciones
- para especificar reglas de deducción para atributos de modelos

Capítulo 3

Description Logics

Las ideas presentadas en esta memoria intentan tratar con el problema de consistencia entre diagramas definidos en modelos UML. El uso de un formalismo para representar los diagramas y modelos permitirá la investigación y manipulación de la semántica de los modelos elaborados por el usuario. Los sistemas de representación del conocimiento (SRC) permiten a través de definiciones de alto nivel, la representación sistematizada de dominios, mientras que situaciones específicas se traducen como los individuos que representan a los conceptos definidos. De esta forma, las consecuencias implícitas asociadas al conocimiento explícitamente representado pueden ser descubiertas a través del uso de este tipo de sistemas.

3.1. Conceptos Básicos

Los sistemas de representación terminológica permiten la definición de una terminología que describe al dominio que se quiere representar, usando el lenguaje definido por el sistema para este fin. Una vez establecida la representación del dominio, como conceptos y las relaciones entre estos, es posible razonar acerca de los individuos que pueblan al dominio modelado. Description Logics es un fragmento decidible y completo de la lógica de primer orden. Existen varias implementaciones de este tipo de sistema, cada uno con su propia variante de DL, lo que significa que estos sistemas varían en su poder expresivo. La decidibilidad y complejidad de estos formalismos han sido examinados en gran detalle en la literatura [20, 19].

3.1.1. Conceptos y Roles

Constructor	**Sintaxis**	**Semántica**
Concepto atómico	C	$C^{\mathcal{I}}$
Concepto universal	$\top$	$\Delta^{\mathcal{I}}$
Negación atómica ($\mathcal{C}$)	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
Unión	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
Intersección ($\mathcal{U}$)	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
Subsunción de conceptos	$C_1 \sqsubseteq C_2$	$C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$
Cuantificador universal	$\forall R.C$	$\{d_1 \mid \forall d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \rightarrow d_2 \in C^{\mathcal{I}})\}$
Cuantificador existencial ($\mathcal{E}$)	$\exists R.C$	$\{d_1 \mid \exists d_2 \in \Delta^{\mathcal{I}}.(R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}})\}$
Restricciones numéricas no cualificadas ($\mathcal{N}$)	$(\geq n\ R)$ $(\leq n\ R)$	$\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \geq n\}$ $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \leq n\}$
Restricciones numéricas cualificadas ($\mathcal{Q}$)	$(\geq n\ R.C)$ $(\leq n\ R.C)$	$\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}}\}\rvert \geq n\}$ $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2) \wedge d_2 \in C^{\mathcal{I}}\}\rvert \leq n\}$
Restricciones numéricas funcionales ($\mathcal{F}$)	$(\geq 2\ R)$ $(\leq 1\ R)$	$\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \geq 2\}$ $\{d_1 \mid \lvert\{d_2 \mid R^{\mathcal{I}}(d_1, d_2)\}\rvert \leq 1\}$
one-of ($\mathcal{O}$)	$\{a_1, \ldots, a_n\}$	$\{d \mid d = a_i^{\mathcal{I}} \text{ for some } a_i\}$
Actores (role fillers) ($\mathcal{B}$)	$\exists R.\{a\}$	$\{d \mid R^{\mathcal{I}}(d, a^{\mathcal{I}})\}$
Rol	R	$R^{\mathcal{I}}$
Unión de roles ($\mathcal{R}$)	$R_1 \sqcap R_2$	$R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$
Subsunción de roles ($\mathcal{H}$)	$R_1 \sqsubseteq R_2$	$R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$
Roles inversos ($\mathcal{I}$)	R^{-1}	$\{(d_1, d_2) \mid R^{\mathcal{I}}(d_2, d_1)\}$

Cuadro 3.1: Operadores básicos usados en DL

Cada SRC basado en DL tiene un lenguaje de especificación de conceptos; este lenguaje es una combinación de los operadores básicos de DL. Además, estos sistemas ofrecen funcionalidades para llevar a cabo el razonamiento automatizado. Cada variante de la familia de DL se caracteriza por los constructores de conceptos y roles que combina. La tabla 3.1[1] muestra un resumen de los constructores de DL más comunes, además de la semántica de cada uno. Estos constructores de conceptos y roles se usan para construir expresiones más complejas, con las cuales se describe la terminología que representa al dominio.

A las variantes de DL más conocidas se les asigna nombres. El lenguaje $\mathcal{FL}^-$ [6] es la variación de DL que incluye la cuantificación universal, la unión de conceptos y la cuantificación existencial de la forma $\exists R.\top$. $\mathcal{FL}^-$ es una formalización de los "marcos" de Minsky [28]. La unión de conceptos está implícita en la estructura de los marcos, dada la necesidad de comprobar si un conjunto de condiciones se cumple en forma simultánea. Usando la cuantificación de roles es posible caracterizar las instancias individuales al definir los atributos de los marcos: usando la cuantificación existencial, se puede establecer que existe al menos un valor válido para el atributo, en cambio, el cuantificador universal establece que los valores posibles del atributo cumplen con ciertas condiciones.

La lógica $\mathcal{AL}$[40] extiende $\mathcal{FL}^-$ al agregar la negación de los conceptos atómicos. La costumbre es colocar como prefijo el nombre de la variante original, colocando a continuación los nombres de los operadores con que se extiende la variante. Por ejemplo, la variante $\mathcal{ALC}$ es $\mathcal{AL}$ más la negación de conceptos.

3.1.2. Bases de conocimiento e inferencia

El lenguaje de especificación de conceptos permite la definición de la terminología que será usada en la creación de bases de conocimiento, donde después se podrán llevar a cabo inferencias sobre el dominio representado. La terminología usada en la creación de una base de conocimiento es especificada como un conjunto finito de expresiones creadas a partir de los operadores permitidos por la variante de DL en particular. Las expresiones que definen conceptos se conocen con el nombre de axiomas terminológicos. Este conjunto de definiciones se conoce como el *Tbox*. La base de conocimiento, la cual contiene los individuos que instancian los conceptos definidos en la *Tbox* se llama *Abox*. La *Abox* contiene todo el conocimiento referente al dominio, en la forma de un conjunto finito de expresiones que relacionan los conceptos y roles a individuos. Estas expresiones se conocen como aserciones.

Se dice que un SRC es completo si puede garantizar que todas las inferencias válidas serán descubiertas. Las implementaciones de algunos sistemas prefieren limitar la expresividad del sistema para poder garantizar resultados manejables y completos. Tener un lenguaje de definición de conceptos menos expresivo limita el poder del sistema, y el número de situaciones donde se puede aplicar este tipo de sistema es reducido. Algunos sistemas incluyen operadores que son reconocidos como inmanejables o hasta no decidibles, dado que permiten mayor expresividad a la hora de describir una terminología, pero los clasificadores de conceptos que

ofrecen estos sistemas son incompletos por estas mismas razones [25]. En estos casos, los usuarios deben estar conscientes de que podrían existir inferencias que el sistema no puede encontrar.

3.1.3. Servicios de razonamiento

Los servicios de razonamiento se usan para razonan acerca de las propiedades en general, y la consistencia de las bases de conocimiento en particular. Existen servicios que se aplican al nivel de la *Tbox* y otros que se aplican al nivel de la *Abox*. Los servicios estándares provistos por sistemas de representación del conocimiento son: la subsunción, el chequeo de instancias, el chequeo de relaciones, el chequeo de consistencia de conceptos y de bases de conocimientos. Estos servicios se pueden ejecutar suponiendo la semántica de mundo abierto o mundo cerrado. La semántica de mundo cerrado significa que el sistema supone que los individuos y las relaciones existentes entre estos es el único conocimiento existente, es decir, el conocimiento actual está completo. La semántica de mundo abierto significa que el conocimiento actual no se considera completo.

Cuando se construye el *Tbox* para una base, se usa la subsunción para determinar si el conocimiento expresado es consistente y/o redundante. Para hacer esto, se construye una jerarquía de conceptos, basado en la relación "es-un", es decir, buscando cuales son los conceptos que incluyen la definición del concepto mas general. Los algoritmos de clasificación, los cuales revisan la consistencia de los conjuntos de axiomas terminológicos, dependen directamente de la subsuncion. Por ejemplo, un concepto no esta satisfacible[1] si es subsunido por el concepto vacío $\perp$.

Con respecto a la *Abox*, se ofrece el chequeo de instancias. Esto se refiere a la determinación de cuales son los conceptos que son instanciados por un individuo. Un individuo es inconsistente con respecto a una *Tbox* si instancia al concepto vacío $\perp$. De una forma similar se puede averiguar cuáles son los individuos que instancian un determinado concepto. Determinar si dos individuos están relacionados a través de un rol se llama chequeo de consistencia. Estos dos servicios son cruciales para determinar qué información se conoce acerca de los individuos en la base de conocimiento. Chequeo de consistencia de conceptos y de bases de conocimientos se usa para determinar si la información especificada en la base de conocimientos es coherente.

[1]Un concepto no es satisfacible si no puede ser instanciado, es decir, es inconsistente con el resto del conocimiento

La investigación dentro de la área de la lógica descriptiva se ha enfocado principalmente en entender la relación que existe entre la combinación de operadores que define una variante de DL y la complejidad de los servicios de razonamiento que se ofrece. El estudio práctico del comportamiento de las diferentes variedades de DL ha permitido obtener un buen entendimiento de las propiedades de los constructores del lenguaje y cómo interactúan.

3.2. Uso de DL para formalizar UML

UML 2.0 se especifica mediante un metamodelo. Este metamodelo define en forma extensa cuáles son los elementos que existen, dónde se pueden usar y da un conjunto de reglas de buen uso para cada elemento. El cumplir con este estándar no requiere que una herramienta CASE implemente control de consistencia entre modelos, diagramas y elementos. Esto significa que errores pueden ser introducidos en los modelos por descuidos, por ejemplo, errores de ortografía, cambio de palabras por sinónimos, etc. Pero este tipo de inconsistencia temporal en los modelos no se introduce solamente en forma accidental - a veces, es necesario introducir inconsistencias en los modelos, dado que no existe un modelo completo en los pasos intermedios del proceso de diseño.

Es usual comenzar la etapa de diseño con un modelo más general, el cual es sucesivamente refinado durante el proceso completo de desarrollo del sistema. Por ejemplo, pueden existir especificaciones incompletas de clases, o links que referencian operaciones no existentes, dado que no todas las responsabilidades han sido definidas todavía. Estas inconsistencias son tolerables durante la fase de diseño, pero los modelos finales que serán la guía para los implementadores deben ser consistentes. Además, en la medida en que los modelos crezcan y adquieran mayor complejidad, es necesario pensar en una forma de llevar a cabo revisiones automáticas de consistencia, para poder determinar si un modelo propuesto es válido o no.

Entonces, como UML en sí no tiene una semántica formal, no se pueden hacer revisiones automáticas de consistencia. Para abordar este problema, es necesario introducir un formalismo que permita manipular los modelos y los diagramas, basándose en la semántica de estos y no su representación visual. Sistemas de representación del conocimiento que usen DL como lenguaje de especificación terminológica pueden ser usados para formalizar UML. El metamodelo es la guía usada para definir el dominio y los modelos de usuarios se traducen como individuos. La ventaja de este enfoque es que existen varias implementaciones de este tipo de sistema[15, 18, 22], los cuales pueden ser integrados con herramientas CASE para

UML para formalizar UML.

3.2.1. Aplicando DL a UML

La figura 3.1 muestra la relación que existe entre el metamodelo de UML y los modelos creados por usuarios. Esta figura sólo muestra una pequeña parte del metamodelo, que en este caso incluye algunas de las metaclases y meta-asociaciones necesarias para la especificación de diagramas de clase, secuencia y componentes. El metamodelo[37, 29] es especificado en su totalidad usando diagramas de clase. Cada elemento creado en un modelo de usuario instancia la metaclase correspondiente. Por ejemplo, en el diagrama de clase, `CarroDeCompras` es una clase, así que instancia la metaclase `Class`. La asociación entre las clases `CarroDeCompras` y `Cliente` es una instancia de la metaclase `Association`. La relación de generalización entre las clases `Cliente` y `ClientePreferente` instancia la metaclase `Generalization`.

Las relaciones de `<<instantiate>>` (instanciación) también se aplican a los elementos usados en los otros diagramas ofrecidos por UML. Por ejemplo, en el diagrama de componentes, la componente `Orden` es instancia de la metaclase `Component`. Las clases interiores de la componente, las encargadas de implementar los servicios ofrecidos por la componente, se rigen por el mismo subconjunto del metamodelo que los diagramas de clase. En el diagrama de secuencia, los objetos instancian la metaclase `Object`. Las propiedades que tiene cada elemento usado en un modelo de usuario está definido por la metaclase correspondiente. Dado el tamaño y complejidad del metamodelo, este está subdividido en paquetes, definiéndose en forma concreta cuáles paquetes son usados por cuáles diagramas. Más detalles sobre el metamodelo se pueden obtener en la especificación de UML [37].

En la figura 3.2 se muestra cómo se traduce una parte del metamodelo mostrado en 3.1 a DL, para ser usado como parte de la definición del dominio (*Tbox*) necesario para poder razonar acerca de modelos y diagramas en UML. Por ejemplo, existe la metaclase `ModelElement`, de la cual heredan todas las metaclases, y tiene un atributo `name`. Entonces, se define el concepto `ModelElement` que tiene el atributo `name`. Por definición, un `Model` es un `ModelElement`, dado que la metaclase `Model` hereda de `ModelElement`, por lo que la definición del concepto `Model` hereda la definición del atributo `name` al especificar que un `Model` es un `ModelElement`. Además, existe una asociación (`owned-element`) entre las metaclases `Model` y `ModelElement`, dado que un modelo contiene elementos. Esto se traduce como el rol `owned-element` entre los dos conceptos, especificada de la siguiente forma:

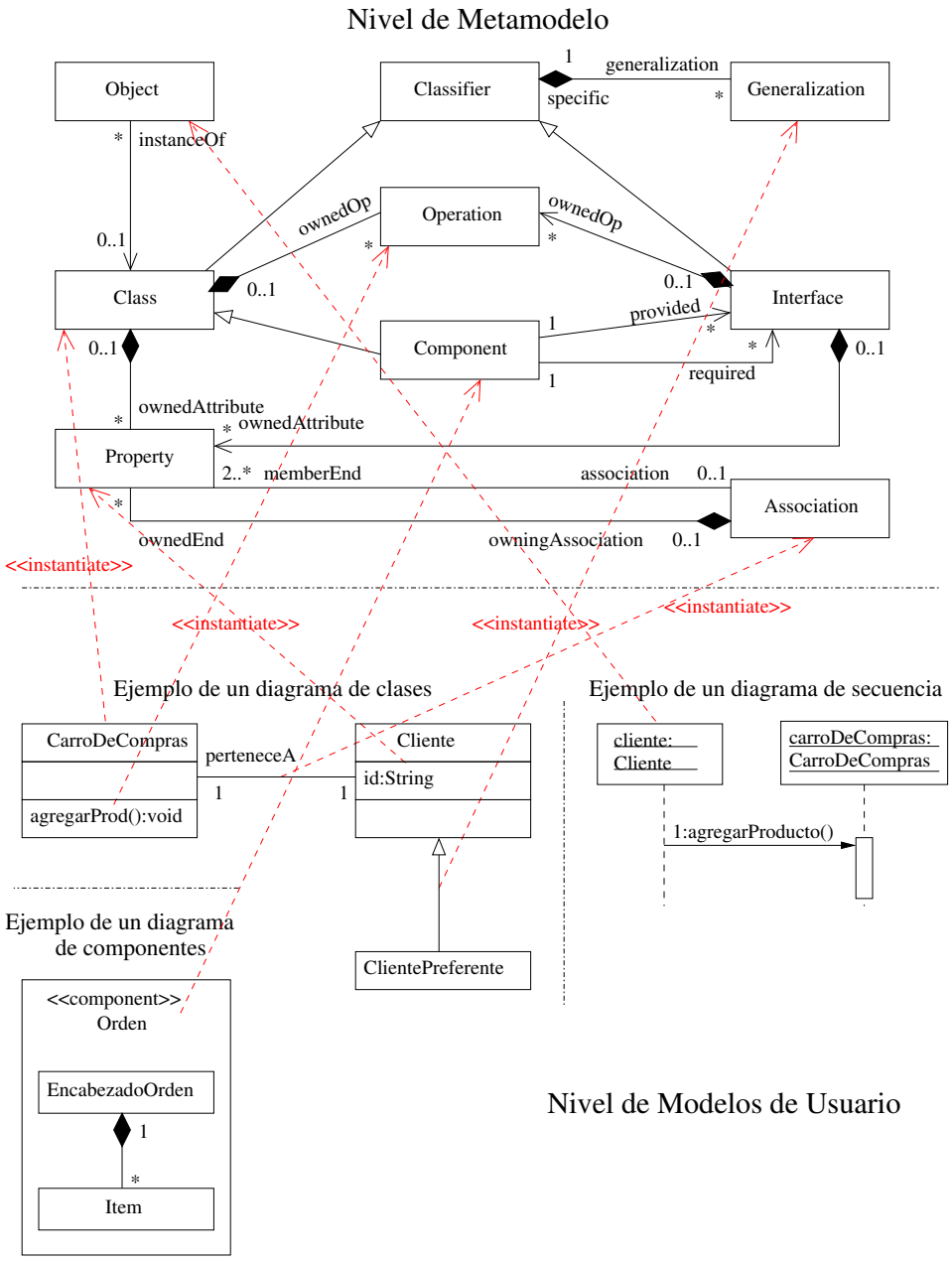

Figura 3.1: La relación entre el metamodelo y modelos de usuario

`(all owned-element ModelElement)`. Esto significa que toda instancia que se relacione con una instancia del concepto `Model` a través de este rol debe ser instancia del concepto `ModelElement`.

De la misma forma, la metaclase `Class` hereda de `ModelElement`, definiendo además dos nuevos atributos: `isAbstract` y `isLeaf`. En la definición del concepto `Object`, restricciones cualificadas de rol se usan para forzar la multiplicidad establecida en el metamodelo para la asociación entre `Class` y `Object` (de que un objeto es instancia de exactamente una clase). Esto se traduce como `(exactly 1 instance-of)`.

```
; Definicion del concepto ModelElement
(implies ModelElement (a name))

; Definicion del concepto Model
(implies Model
     (and ModelElement (all owned-element ModelElement)))

; Definicion del concepto Class
(implies Class
     (and ModelElement
     (a isAbstract)
     (a isLeaf)))

; Definicion del concepto Object
(implies Object
     (and ModelElement
     (exactly 1 instance-of)
     (all instance-of Class)))
```

Figura 3.2: *Tbox* parcial correspondiente al metamodelo de UML

En la figura 3.3 se muestra la traducción de una parte del diagrama de clase de ejemplo que se muestra en la figura 1, para que pase a formar parte del *Abox*. Todos los diagramas pertenecen a un modelo llamado `modelo1`, así que se crea un individuo que instancia el concepto `Model` y se asigna "modelo1" como valor del atributo `name`. El segundo conjunto de definiciones corresponde a la traducción de la clase `Cliente`. Lo primero que se hace nuevamente es crear un individuo que instancia al concepto `Class`. A continuación, se asigna el valor "Cliente" al atributo `name`. Después, se relaciona la clase con el modelo al que pertenece usando la relación `owned-element`. Finalmente, se llenan los valores de los atributos definidos para el concepto `Class` - en este caso, de que la clase `Cliente` no es abstracta ni es una hoja (es decir, tiene subclases).

El último conjunto de definiciones corresponde a la traducción del objeto `unCarroDeCompras`. El atributo `name` del individuo es inicializado con el valor "unCarroDeCompras".

```
; Individuo que representa al modelo1
(instance inst-modelo1 model)
(constrained inst-modelo1 name-of-modelo1 name)
(constraints (string= name-of-modelo1 "modelo1"))

; Individuo que representa la clase Cliente
(instance inst-Cliente class);
(constrained inst-Cliente name-of-Cliente name)
(constraints (string= name-of-Cliente Çliente"))
(related inst-modelo1 inst-Cliente owned-element)
(constrained inst-Cliente abstract-Cliente isAbstract)
(constraints (string= abstract-Cliente "false"))
(constrained inst-Cliente leaf-Cliente isLeaf)
(constraints (string= leaf-Cliente "false"))

; Individuo que representa al objeto unCarroDeCompras
(instance inst-unCarroDeCompras object)
(constrained inst-unCarroDeCompras name-of-unCarroDeCompras name)
(constraints (string= name-of-unCarroDeCompras ünCarroDeCompras"))
(related inst-modelo1 inst-unCarroDeCompras owned-element)
(related inst-unCarroDeCompras inst-CarroDeCompras instance-of)
```

Figura 3.3: *Abox* parcial (figura 3.1)

Este objeto también pertenece a `modelo1`, así que también se relaciona usando el rol `owned-element`. La definición de este rol impone la restricción de que los participantes sean instancias de `ModelElement`, lo cual se cumple en este caso porque el concepto `ModelElement` subsume `Class` y `Object`. En la última línea se registra la relación entre la clase `CarroDeCompras` y la instancia `unCarroDeCompras`, usando el rol `instance-of`.

El orden de aparición de los individuos en las definiciones de los individuos no importa, dado que los sistemas basados en DL pueden manejar información incompleta. Información adicional puede ser ingresada posteriormente, mientras sea consistente con la información que ya se conoce. Por ejemplo, se podría colocar la definición un individuo que representa a la clase `CarroDeCompras` a continuación de la definición del objeto que lo utiliza. Dadas las restricciones impuestas en la *Tbox*, el sistema deduce en este caso que el individuo `inst-CarroDeCompras` instancia al concepto `Class`, dado que el individuo `inst-unCarroDeCompras` asegura estar relacionado con este individuo a través del rol `instance-of`.

3.3. Razones que justifican el uso de DL

Son varias las razones que justifican el uso sistemas de representación de conocimientos basadas en variantes de DL para formalizar UML. La primera razón es que DL es decidible, es decir, dada la definición de un concepto, es posible determinar si esta definición es consistente o no con las definiciones de conceptos ya incluidas en el *Tbox*. Dada la definición de un individuo, también se puede saber cuál es la definición de concepto que mejor representa las cualidades conocidas del individuo. DL también ofrece subsunción de conceptos, es decir, se puede construir una jerarquía de los conceptos definidos en el sistema al clasificar sus definiciones, determinado cuáles son los conceptos más generales para cada concepto. La subsunción se usa para modelar las relaciones de generalización que existen entre las metaclases definidas por el metamodelo de UML. Además, DL soporta la semántica de mundo abierto. Esto permite la especificación de individuos incompletos, donde las especificaciones faltantes se deducen de las definiciones de los conceptos.

No sólo son teóricas las motivaciones para adoptar DL. Es importante notar que los motores de razonamiento ofrecidos por los sistemas basados en DL actuales son bastante eficientes, usando algoritmos de "tableau". Este es un punto crítico si se desea hacer algo práctico con el formalismo, dado que cualquier herramienta que sea ofrecida al usuario debe producir resultados dentro de límites de tiempo razonables. Además, los sistemas de hoy proveen interfaces de interconexión limpias, lo que facilita la inclusión de este tipo de sistema en herramientas propias. De esta forma, se puede aprovechar el poder de estos sistemas, por ejemplo, integrándolos con herramientas CASE para UML, así formalizando los modelos de usuario descritos en UML.

Para generalizar los ejemplos presentados, los siguientes teoremas demuestran que DL es lo suficientemente expresivo como para representar todo modelo de usuario bien formado descrito en UML.

Teorema 1. *El metamodelo de UML puede ser descrito en forma completa usando la lógica descriptiva.*

Demostración. El metamodelo de UML está definido en forma completa a través de diagramas de clases [29]. Y, según [7], los diagramas de clases pueden ser descritos en forma completa usando la lógica descriptiva. □

Teorema 2. *Todo modelo definido por el usuario puede ser descrito en forma completa usando lógica descriptiva.*

Demostración. Todos los modelos definidos por el usuario y los elementos incluidos en estos son instancias de las metaclases definidas en el metamodelo de UML [37]. Y, según el Teorema 1, el metamodelo de UML puede ser descrito en forma completo usando lógica descriptiva. □

De esta forma, el *Tbox* y la *Abox* pueden ser construidos usando la información contenida en el metamodelo de UML y los modelos definidos por el usuario, respectivamente. Los chequeos de consistencia se implementarán realizando consultas sobre la *Abox* para encontrar los conjuntos de individuos que cumplan con ciertas condiciones predefinidas.

Capítulo 4

Estrategia de Solución

En esta sección se propone una solución y se justifican las distintas decisiones tomadas para cumplir con los objetivos planteados. En particular, se analiza la selección de una herramienta para la visualización de los modelos en UML, la selección de un sistema de representación del conocimiento, el sistema operativo y el lenguaje de programación y finalmente, la estructura del framework implementado. La herramienta CASE de UML será utilizada para producir los modelos que serán revisados por el framework, el cual se encargará de hacer las traducciones correspondientes para ingresar los modelos de usuario al sistema de representación de conocimiento elegido.

4.1. Selección de la herramienta CASE para UML

Se escogió Poseidon for UML 2.0 (Standard Edition) [14] como la herramienta CASE para UML. Esta herramienta ofrece una interfaz gráfica donde se puede modelar diagramas en UML al hacer drag-and-drop de elementos, como clases y asociaciones. Este sistema tiene una gran base de usuarios, dado que, aun cuando la herramienta no sea opensource, se puede obtener una versión gratis (la versión Community) y una versión de evaluación (la versión Standard). Además, este sistema está implementado en Java, lo que hace que sea completamente portable. Es un sistema robusto, su versión actual es la 2.6, y nuevas versiones están disponibles en forma mensual.

Inicialmente, se pensó en implementar el framework como herramienta independiente, así que era necesario contar con una herramienta CASE para UML que generase archivos

en un formato estándar, como XMI (XML Metadata Interchange) [34], el formato estándar de intercambio de diagramas UML, y que después pudiese leer archivos en este formato y generar los modelos correspondientes. Esta fue la principal motivación para usar Poseidon como herramienta CASE para UML.

Esta herramienta también provee facilidades para la inclusión de expansiones. Al estudiar la API (Aplication Programming Interface) de programación de plug-ins disponible, se hizo evidente la posibilidad de implementar al framework como un plug-in para Poseidon. Esta API provee acceso a todos los elementos de los modelos de usuario, además de dar la posibilidad de incluir interfaces propias como menús y paneles dentro del ambiente de Poseidon. Estos plug-ins se implementan en Java, el lenguaje en que está implementado Poseidon.

Se decidió implementar el enfoque de plug-in para Poseidon, dado que esto haría más amigable el uso del framework, dado que los usuarios interactuaría con el framework a través de una interfaz conocida. El plug-in requiere el uso de la versión Standard de Poseidon, dado que la versión Community no permite la carga de plug-ins.

4.2. Selección de Sistema de Representación del Conocimiento

Existen varios Sistemas de Representación del Conocimiento (SRC), como se menciona en el capítulo 3. En [41, 43] se usó Loom [22] como SRC. Este SRC es altamente expresivo, pero tiene problemas de clasificación, es decir, dada la alta expresividad de la variante de DL usada y el tipo de algoritmo de clasificación, existen inferencias que el sistema no puede deducir [25]. Este sistema está implementado en Lisp, lo que hace difícil la inclusión de esta herramienta en otras, dado que debe hacerse a través de librerías propietarias. Este sistema ofrece un mecanismo poderoso de recuperación de individuos, lo que se aprovechó en [41] en la implementación de los predicados de chequeo de consistencia. El desarrollo de este sistema se interrumpió.

Para este trabajo se eligió trabajar con Racer 1.7 [15] como SRC. Las principales razones que motivaron esta decisión son que Racer ofrece un lenguaje de definición de terminología muy expresivo como también algoritmos de clasificación muy eficientes basados en cálculo de "tableau". Este sistema ofrece la variante de DL conocida como $\mathcal{ALCQHIR}+$, la cual también se conoce como $\mathcal{SHIQ}$ [20]. Este DL es la lógica básica $\mathcal{ALC}$, incluyendo también

restricciones numéricas cualificadas, jerarquías de roles y roles transitivos. También soporta la especificación de axiomas terminológicos generales. Se puede definir *Tboxes* que definan relaciones de subsunción entre conceptos. Racer también puede manejar definiciones múltiples para un concepto, como así también definiciones cíclicas.

El mecanismo de recuperación de individuos ha sido mejorado, imitando el mecanismo ofrecido por Loom. Antes, sólo se ofrecía un conjunto de funciones que hacen consultas estándares al *Tbox* y *Abox*. Estas funciones todavía están disponibles, pero se incluyó un mecanismo flexible de recuperación de individuos, el cual permite la construcción de consultas complejas con comodines [16]. Este será usado de la misma forma que antes se utilizaba el mecanismo de recuperación de Loom. Una comparación más extensa se puede encontrar en [41].

Usar Racer también ofrece la ventaja de que este sistema está implementado con una arquitectura Cliente / Servidor, lo que hace fácil incluir este sistema en herramientas propias. Racer se ejecuta en forma independiente, y esta pendiente al envio de axiomas, aserciones y consultas de parte de clientes. Ya existe una implementación de cliente para Racer en Java, la que fue aprovechada en la implementación en Java del plug-in para Poseidon. Esto hace que sea fácil programar la comunicación entre la interfaces gráficas programadas en Java y el motor de razonamiento.

4.3. Selección del ambiente de trabajo

Las herramientas seleccionadas no están restringidas a un sistema operativo en particular, por lo que el framework estará disponible para Linux, Windows y Mac OS X. El lenguaje de programación es Java, dado que el framework será implementado como un plug-in para Poseidon, sistema implementado en Java 1.4.2. También influye en esta decisión la existencia de JRacer, interfaz de cliente que será usada para implementar las interacciones con Racer.

4.4. MCC

Se construyó MCC (Model Consistency Checker)[1], un framework que permite la revisión de modelos en UML 2.0 a través del formalismo ofrecido por un sistema de representación

[1]Este sistema se puede bajar desde `http://www.dcc.uchile.cl/~jsimmond/mcc.html`

del conocimiento. Este incluye ya 6 diferentes chequeos de consistencia. Esta herramienta es fácilmente extensible y nuevos chequeos se pueden agregar siguiendo la misma estrategia utilizada en la implementación de los chequeos ya existentes. MCC es un plug-in para la herramienta CASE para UML Poseidon, dado que de esta forma se podía aprovechar la interfaz existente como una forma de ofrecer una herramienta amigable. Además, se ofrece una herramienta formal, basada en el poder de razonamiento ofrecido por Racer. La herramienta para UML y el motor de DL son ambos sistemas robustos. APIs bien definidas son usadas para integrar ambas herramientas.

La figura 4.1 muestra las tres componentes que forman parte del framework:

- Interfaz visual de consulta: interfaz que provee acceso fácil a los predicados de detección de inconsistencia existentes, de forma encapsulada. La configuración del plug-in y el manejo del *Tbox* también están disponibles a través de esta interfaz.
- Extracción de aserciones: genera las aserciones que se necesitan para poblar la *Abox*, basado en un modelo de usuario en UML.
- Procesamiento de consultas: actúa como el canal de comunicación entre los módulos de Interfaz visual de consulta y Extracción de aserciones y Racer.

Esta figura también muestra las relaciones que existen entre MCC, Poseidon for UML y Racer.

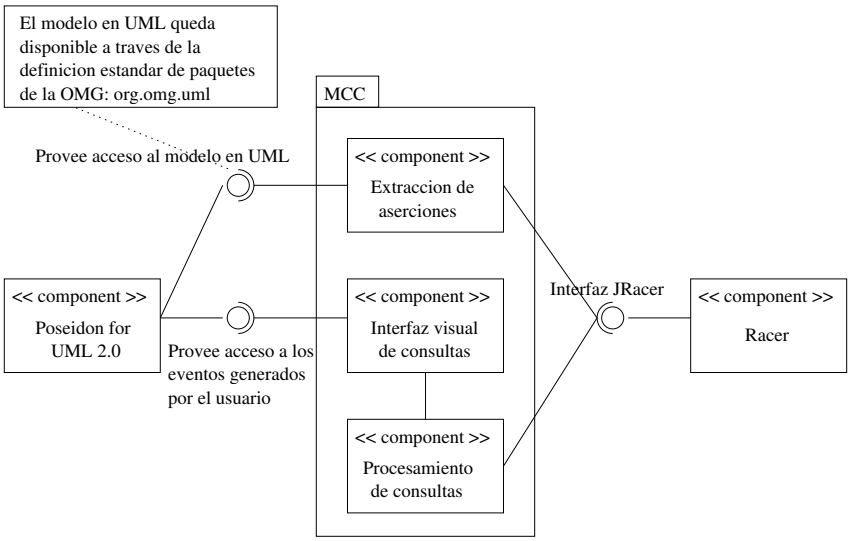

Figura 4.1: MCC components

Para usar la herramienta, el plug-in de MCC debe ser cargado en Poseidon y el motor de Racer debe iniciarse. Una vez hecho esto, cualquier modelo creado por el usuario puede ser revisado. Al iniciarse el plug-in, el *Tbox* se envía a Racer, estableciendo la terminología para los diagramas en UML. Cualquier modelo abierto en Poseidon se carga en Racer usando el componente de Extracción de aserciones. Se puede llevar a cabo revisiones de consistencia eligiendo distintos problemas de consistencia de una lista pre-establecida de revisiones. Estos se ofrecen en forma amigable, y el usuario nunca estará en contacto con el formalismo de DL. Cuando se elige un chequeo de consistencia, la componente de Procesamiento de consultas se contactará con Racer, para llevar a cabo inferencias basadas en el conocimiento ya existente en el *Abox* (cargado por la componente de Extracción de aserciones). La componente de Procesamiento de consultas es la encargada de procesar la salida generada por Racer para generar mensajes que serán entendidos por el usuario.

Capítulo 5

Análisis y Diseño

En este capítulo se presenta el diseño final del sistema desarrollado. Para esta tarea se hará uso de los distintos diagramas que posee UML en las distintas etapas de análisis y diseño, para obtener el diagrama de clases final. El comportamiento más relevante es modelado usando diagramas de estado y secuencia.

5.1. Modelo de Casos de Uso

El principal requerimiento es el siguiente: se desea desarrollar una herramienta que permita hacer un análisis de consistencia de los modelos en UML elaborados por el usuario. Estos modelos se generan usando la herramienta Poseidon, y toda interacción con el usuario será a través de la interfaz de este programa.

En la figura 5.1 se observa el diagrama de casos de uso para el sistema. El caso de uso *configurar MCC*, implica que se deberá contar con elementos que permitan definir la configuración del sistema. El caso de uso *cargar Tbox* está asociado a la iniciación del sistema de representación del conocimiento, al cargar los axiomas terminológicos que definen al dominio modelado. El conjunto de axiomas terminológicos se obtiene de un archivo que se distribuye con el framework.

El caso de uso *cargar Abox* se refiere a los modelos individuales creados por el usuario. Estos se obtienen a través de la interfaz definida por Poseidon, en la forma de objetos. El plug-in genera una traducción automática de estos objetos y puebla el dominio a través del caso de uso *cargar Tbox*. El *Tbox* y *Abox* se cargan en forma independiente, para permitir

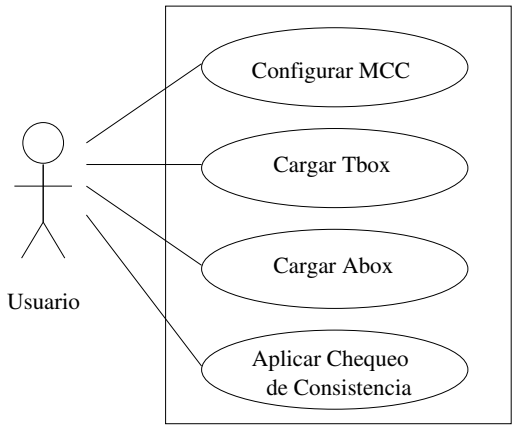

Figura 5.1: Modelo de casos de uso del sistema

a futuro la carga de varios modelos simultáneos, lo que permitiría hacer chequeos entre modelos.

El último caso de uso es *Aplicar chequeo de consistencia*. Esto involucra además, la selección de un chequeo de consistencia. Actualmente, los chequeos se presentan como una lista predefinida, pero se planea a futuro permitir la parametrización de los chequeos y la generación de chequeos propios. El usuario debe preocuparse de que se haya hecho la carga correspondiente del *Abox* antes de activar los chequeos. La razón de esta separación es para que el modelo sea traducido una sola vez, permitiendo la posterior aplicación de cualquier análisis de consistencia.

5.2. Diagrama de Clases

En la figura 5.2 se puede ver el diagrama de clases final. Por razones de espacio sólo se han incluido los atributos y métodos más importantes para la comprensión del diseño del sistema. Algunas asociaciones no figuran en forma explícita en el diagrama, dado que se utiliza reflexión para generar la relación en forma dinámica durante la ejecución del framework. En este diagrama se ha omitido las clases correspondiente a JRacer, dado que esta librería tiene su propia documentación.

A continuación se dará una breve explicación de qué hace cada clase, dando algunos detalles de sus atributos y métodos y comentando además las relaciones que existen entre las distintas clases.

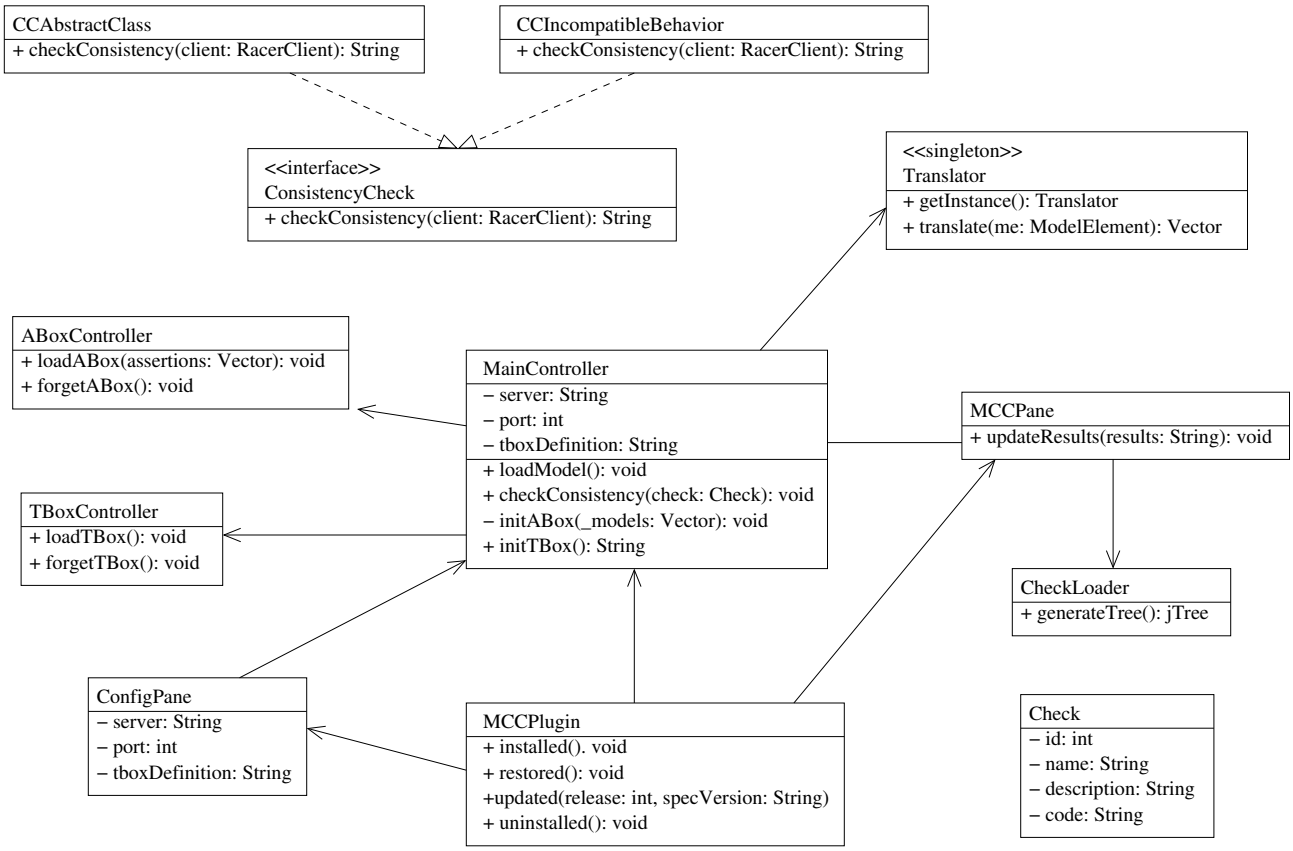

Figura 5.2: Diagrama de clases del sistema

MCCPlugin es la clase que se encarga de iniciar el plug-in al cargarse este a través de la interfaz de Poseidon. Al instalarse el plug-in, esta clase inicia la interfaz gráfica y crea una instancia del controlador principal, asociando la interfaz gráfica al controlador, para que los eventos generados en la interfaz gráfica puedan ser procesados por el controlador y este pueda mandar peticiones de actualización de contenido a la interfaz gráfica.

- installed(): método que es invocado al iniciarse el plug-in desde el manejador de plug-ins de Poseidon.

MainController esta es la clase principal del plug-in. Es asociada a una instancia del panel de configuración y una instancia del panel principal. Todos los eventos generados por el usuario se envían a este controlador, el que se encarga de delegar el trabajo a los objetos correspondientes. Está relacionado con un objeto singleton de la clase Translator, la cual se encarga de traducir los objetos UML a DL.

- server: atributo que almacena el nombre del servidor donde se encuentra corriendo Racer. El valor se obtiene a través del panel de configuración.
- port: atributo que almacena el puerto del servidor donde Racer esta pendiente a peticiones de información. El valor se obtiene a través del panel de configuración.
- tboxDefinition: atributo que almacena el nombre del archivo donde se encuentra la definición del *Tbox* para UML 2.0. Este archivo debe encontrarse en el servidor donde está corriendo Racer. El valor se obtiene a través del panel de configuración.
- loadModel(): método que obtiene el modelo actualmente activo en Poseidon e invoca al método privado initABox(_models: Vector). El evento que activa la invocación de este método es generado desde el panel principal.
- initABox(_models: Vector): método que se encarga de obtener las traducciones de los objetos (utilizando el objeto de tipo Translator asociado al controlador) y enviarlas a Racer (a través del controlador de *Abox* asociado).
- initTbox(): método que activa la iniciación del *Tbox*, a través del controlador de *Tbox* asociado.
- checkConsistency(check: Check): este método recibe un objeto del tipo Check, objeto que encapsula la presentación de los diferentes chequeos de consistencia,

y usa reflexión para generar una instancia del chequeo de consistencia correspondiente. A esta instancia se le pide que revise la consistencia del modelo de acuerdo a su definición. La salida de esta invocación será enviada a la interfaz principal.

TBoxController esta clase se preocupa de la inicialización y destrucción de la *Tbox* en Racer.

- loadTBox(): método que carga la definición del *Tbox* para UML 2.0 en el servidor Racer activo.

ABoxController esta clase se preocupa de la inicialización y destrucción de la *Abox* en Racer.

- loadABox(): método que carga el conjunto de aserciones correspondientes al modelo actual en el servidor Racer activo.

MCCPane la interfaz principal del plug-in. Es creada por la clase MCCPlugin, y asociada al controlador principal. Todo evento generado a través de esta interfaz es manejado por el controlador principal. Los eventos que genera esta interfaz son: iniciar la carga del modelo actual en el servidor Racer activo y la activación de chequeos de consistencia.

- updateResults(results: String): método que actualiza la interfaz gráfica, desplegando el mensaje recibido para informar al usuario de algún resultado.

ConfigPane esta es la interfaz de configuración del plug-in. A través de esta interfaz se define los parámetros del servidor Racer activo (servidor y puerto de comunicación) y la ruta al archivo de definiciones terminológicas del *Tbox*. El evento de inicialización del *Tbox* se genera desde esta interfaz. El controlador principal utiliza la configuración establecida a través de esta interfaz.

Translator esta clase se preocupa de la traducción de los objetos generados por Poseidon a aserciones en DL. Es una clase singleton, cuya única instancia es asociada al controlador principal. El método principal de la clase está asociado a la traducción de `ModelElements`. Este método delega a métodos específicos a cada tipo de objeto. Toda nueva traducción debe ser agregada a esta clase.

- translate(me: ModelElement): método que delega la traducción del elemento, dependiendo de su tipo dinámico. Si no existe un método al cual delegar la traducción, se genera una excepción.

CheckLoader clase auxiliar que toma el archivo XML que tiene la configuración de los chequeos actuales de consistencia y genera un JTree, elemento principal para la interacción con el usuario. El archivo XML incluye información como la categoría, el nombre, la descripción y la clase encargada del chequeo de consistencia, para cada chequeo de consistencia. Esta información se usa para generar elementos del tipo Check, con los cuales se puebla el JTree.

- generateTree(): método que genera el JTree usado en la interfaz principal.

Check clase utilitaria que encapsula la información disponible para un chequeo de consistencia.

- id: atributo que almacena un número de identificación para el chequeo.
- name: atributo que almacena el nombre del chequeo. Es utilizado en la generación del JTree de la interfaz principal.
- description: atributo que almacena la descripción del chequeo. Es desplegada en la interfaz principal cada vez que el chequeo es seleccionado en el JTree.
- code: atributo que almacena el nombre de la clase que implementa el chequeo. Esta clase se instancia usando reflexión cada vez que se activa este chequeo.

ConsistencyCheck interfaz que debe ser implementada por todo chequeo de consistencia.

- checkConsistency(client: RacerClient): método general que revisa la consistencia asociada a la clase que implementa el método. Una clase puede subdividir la responsabilidad del chequeo al implementar métodos auxiliares, pero este método es el que debe retornar un String con el resultado de la aplicación de la revisión de consistencia, en un formato amigable al usuario.

5.3. Diagramas de Secuencia

En la figura 5.3 se puede ver el diagrama de secuencia para la inicialización del plug-in. El flujo mostrado se produce cuando se carga el plug-in en Poseidon, a través de su interfaz de manejo de plug-ins. Se inicializa el controlador principal del plug-in, además de crearse la interfaz gráfica. Se asocian las interfaces con el controlador, para que los eventos generados desde las interfaces puedan ser manejados por el controlador.

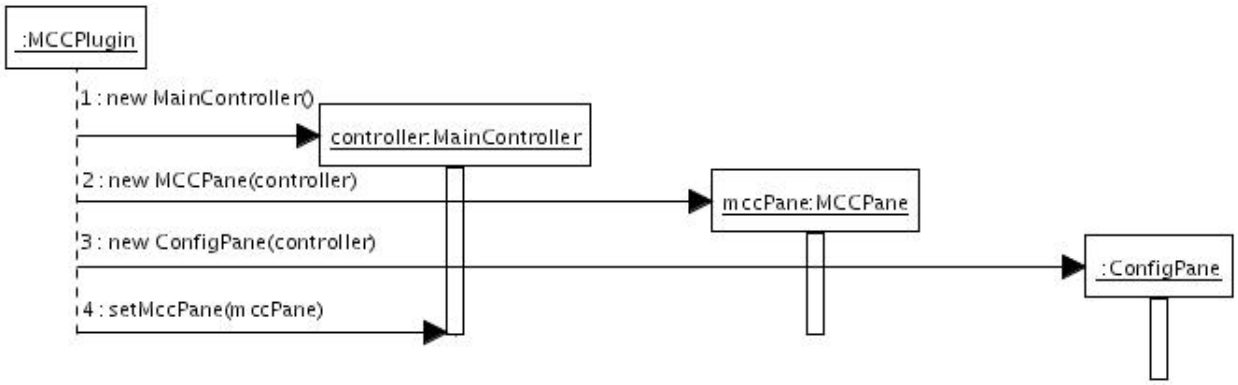

Figura 5.3: Diagrama de secuencia: inicialización del plug-in

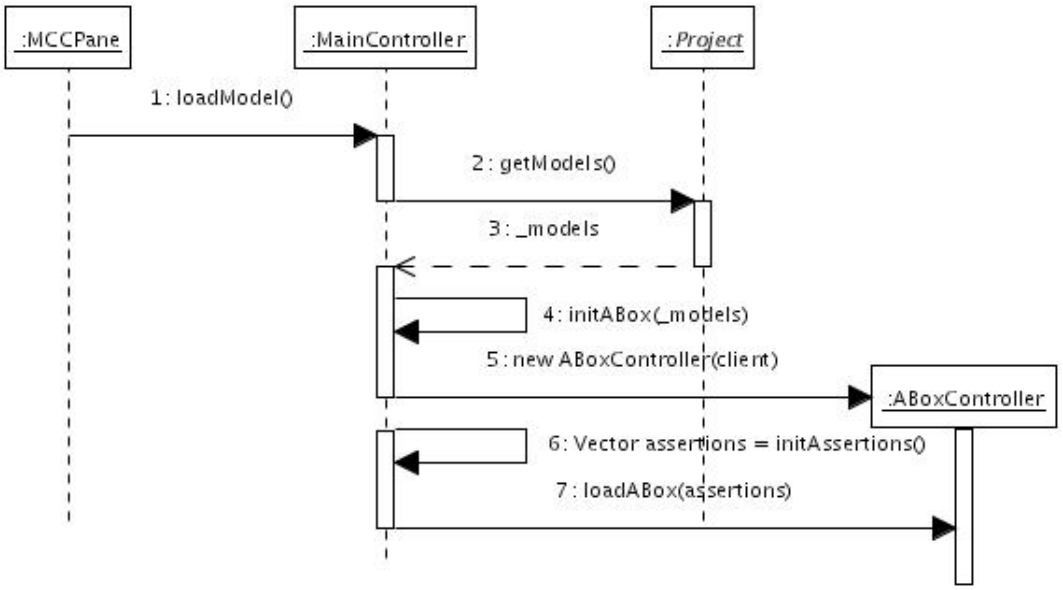

Figura 5.4: Diagrama de secuencia: generación de *ABox*

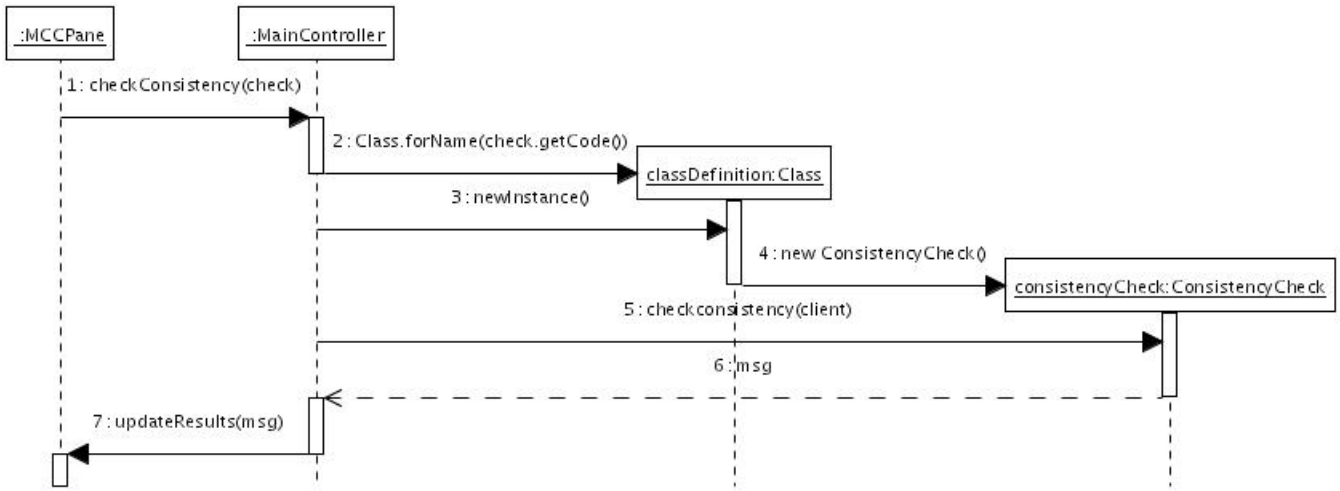

Figura 5.5: Diagrama de secuencia: revisión de consistencia

En la figura 5.4 se puede ver el diagrama de secuencia correspondiente a la generación del *ABox* a partir de un modelo abierto en Poseidon. El usuario inicia esta secuencia de eventos al oprimir un botón en la interfaz principal. La interfaz delega el evento al controlador principal. Este obtiene los modelos activos en Poseidon a través de la API de plug-in de Poseidon. Se crea una nueva instancia del controlador de la *Abox* para manejar el proceso de carga del modelo. De los modelos se obtiene una lista de todos los elementos, en la forma de objetos. Cada elemento es traducido, generándose una lista de aserciones correspondientes, y estas listas de aserciones son finalmente enviadas al controlador del *Abox*, donde son cargadas en el servidor Racer activo.

En la figura 5.5 se puede ver el diagrama de secuencia correspondiente a la revisión de consistencia. El usuario elige un chequeo desde la interfaz principal del plug-in. Esto genera un evento, el cual es procesado por el controlador principal, el cual recibe un objeto representando al chequeo de consistencia elegido por el usuario. Utilizando la información disponible en este objeto, el controlador crea una instancia de la clase encargada de revisar la consistencia para este chequeo. Se delega a esta instancia la responsabilidad de revisar la consistencia. Al finalizar el chequeo, esta clase retorna un mensaje, el cual se envía a la interfaz gráfica. Es la responsabilidad de cada chequeo de consistencia de que el mensaje enviado a la interfaz sea amigable y entendible.

5.4. Diagrama de Estado

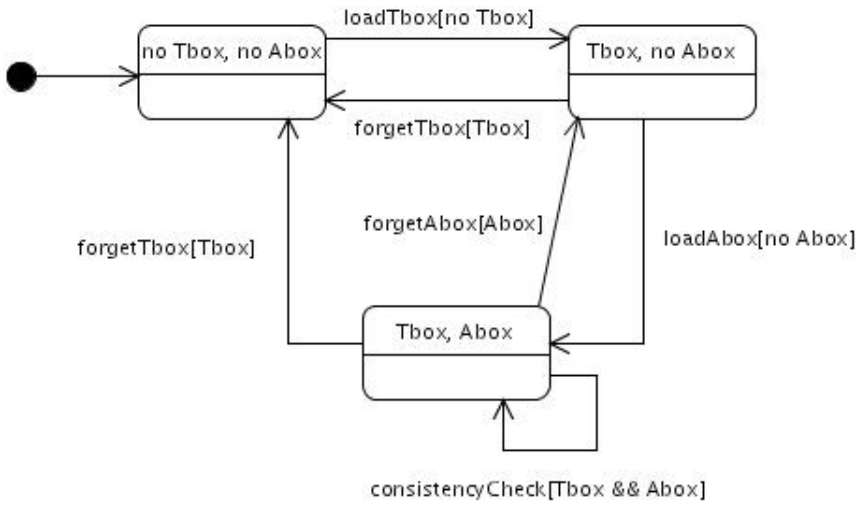

Figura 5.6: Diagrama de estado para la clase MainController

La figura 5.6 muestra el diagrama de estado para la clase MainController. Este controlador es el encargado de coordinar los estados del *Tbox* y la *Abox* en el servidor Racer activo. Por ejemplo, al iniciarse el sistema, es obligatorio iniciar el *Tbox*. Para poder llevar a cabo cualquier revisión de consistencia, es necesario que el modelo activo en Poseidon sea cargado al *Abox*. Esto deja al sistema en un estado válido para realizar cualquier número de revisiones independientes.

Capítulo 6

Implementación

En este capítulo se dan detalles de la solución implementada. Se analiza la interfaz gráfica, el mecanismo de traducción y se especifican los detalles de los chequeos implementados.

6.1. Interfaz Gráfica

La interfaz gráfica de la herramienta se divide en dos paneles, los cuáles están disponibles en el sector de paneles de la interfaz de Poseidon. El primer panel es bastante simple, siendo la interfaz de configuración del plug-in (figura 6.1). El segundo panel es más complejo, al ser la interfaz principal del framework. Esta interfaz puede ser vista en la figura 6.2.

El panel de control de la configuración consta de tres cajas de texto y tres botones. Las cajas de texto son para recibir los valores de configuración para la conexión y uso de Racer. Esto incluye el nombre y puerto del servidor donde esta aplicación se encuentra activa, como también la ruta al archivo que define la terminología para UML 2.0. El primer botón, *Set Configuration*, manda los valores que hay en las cajas de texto al controlador principal. El segundo botón, *Load Tbox*, genera el evento de carga de la *Tbox*, evento que es delegado por el controlador principal al controlador del *Tbox*. El último botón, *Forget Tbox*, también genera un evento que es delegado al controlador del *Tbox*, borrando el conocimiento acumulado y las definiciones terminológicas existentes en el sistema.

El panel principal consiste de un JTree, dos áreas de texto y dos botones. Para iniciar el uso del plug-in, el modelo actualmente activo en Poseidon debe ser traducido y cargado en el servidor de Racer activo. Para generar esta secuencia de acciones, se debe oprimir el

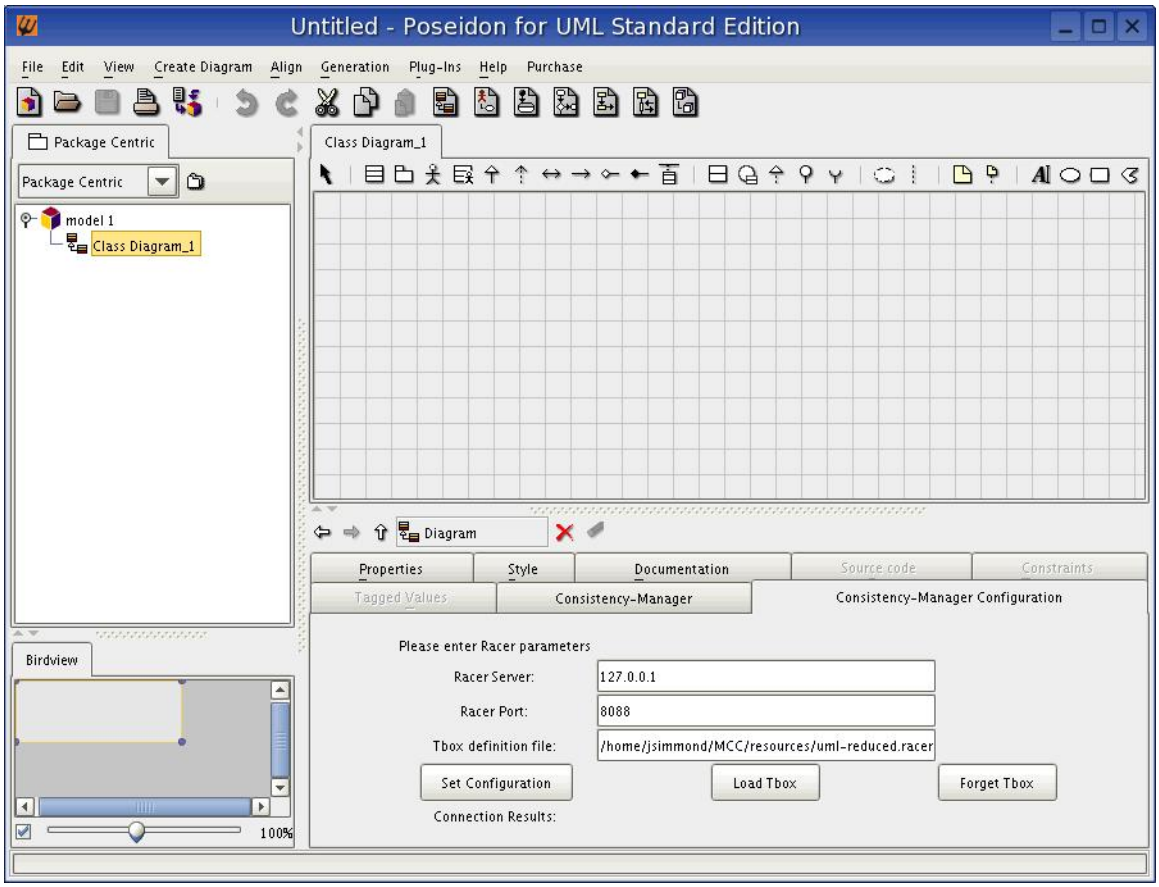

Figura 6.1: Panel de Configuración del plug-in

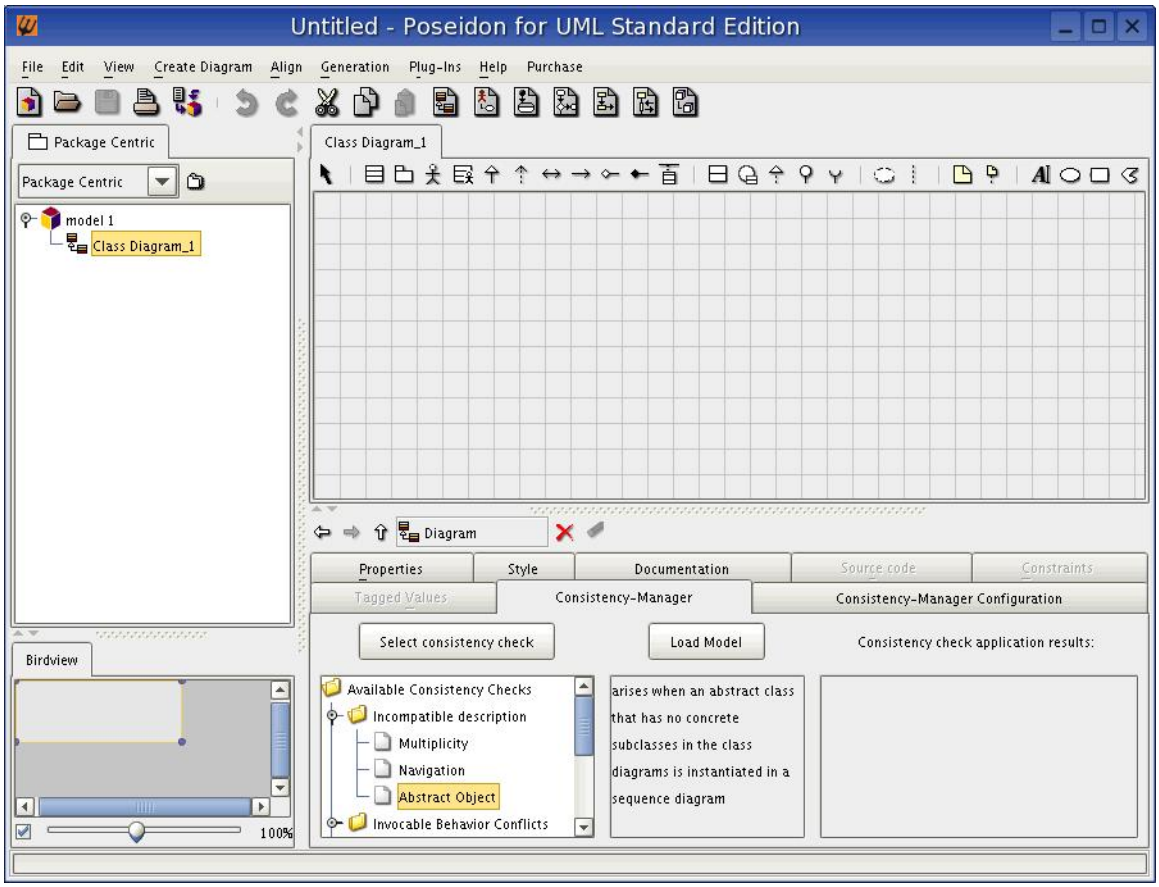

Figura 6.2: Interfaz principal del plug-in

botón *Load Model*. La interfaz delega este evento al controlador principal. La secuencia de acciones generadas por este evento se puede ver en la figura 5.4. Una vez cargado el modelo, es posible iniciar el análisis del modelo.

El JTree es una componente Swing, permitiendo una definición propia de los valores de los nodos. En este caso, se construye utilizando un archivo XML de configuración de los chequeos de consistencia. El archivo XML incluye información como la categoría, el nombre, la descripción y la clase encargada del chequeo de consistencia, para cada chequeo de consistencia. Si se elige un nodo del árbol, se despliega la descripción asociada al chequeo en la primera área de texto. Esta caja también se usa para informar al usuario del estado de chequeo: si está disponible o no. Para activar el análisis de consistencia, es necesario elegir un nodo implementado del árbol y presionar el botón *Select consistency check*. Todo resultado del análisis queda disponible en la segunda caja de texto. El texto de esta caja se acumula, por lo que es posible acumular los resultados de distintos análisis.

6.2. Traducciones

El controlador principal es el encargado de realizar las traducciones. El código correspondiente al método encargado de realizar las traducciones se puede ver en la figura 6.3. Para esto, primero es necesario obtener todos los elementos de modelo que pertenecen al modelo actual. Esto se consigue usando la API de programación de plug-in de Poseidon, concretamente haciendo el siguiente llamado:

```
Vector allElements = new Vector(pro.getUml1SmFactory().getCorePackage().getModelElement().refAllOfType());
```

A continuación, es necesario definir un nombre único para cada elemento que existe en el modelo. Inicialmente, se pensó en usar el nombre de cada elemento para definir el nombre de la instancia en forma directa. Esto no funcionó porque elementos de distintos tipos pueden tener el mismo nombre, y existen elementos anónimos, es decir, no tienen nombre. Para solucionar este problema, inicialmente se genera un nombre único para cada elemento. De esta forma, al traducirse el objeto, se utilizan los nombres de instancias predefinidas. De esta forma se garantiza que todas las relaciones están correctamente armadas.

Finalmente, se pide la traducción de cada objeto. Para poder acceder a todos los elementos del diagrama, se manejan directamente como `ModelElements`. Entonces, el traductor,

al recibir una petición de traducción, inicialmente procesa al elemento como si fuese un `ModelElement`, para después delegar al método específico al consultar el tipo dinámico del objeto. El vector de aserciones correspondientes a todos los elementos del modelo es enviado al controlador del *Abox* para que sea cargado en el servidor Racer activo.

En la figura 6.4, se presenta un ejemplo de un método traductor, en este caso, el correspondiente a los elementos de tipo *Clase*.

Es importante notar de que no se incluye la traducción de los atributos y las operaciones de la clase en el método traductor. Esto se debe a que estos elementos también son elementos de modelo, por lo que también hay métodos traductores específicos para ellos. Por ejemplo, una operación sabe a qué clase pertenece, una asociación sabe a qué clases relaciona, etc. Cada vez que se quiere relacionar una instancia con otra, se obtiene el objeto relacionado y se pide el nombre de este objeto al controlador principal.

6.3. Chequeos implementados

En [41], se ha definido un conjunto de 18 relaciones de consistencia entre los diagramas de clase, de secuencia y de estado que no son reforzadas por la definición del metamodelo de UML. De esta lista, cinco chequeos de consistencia se han implementado como parte de una primera versión de MCC: objeto abstracto, comportamiento no compatible (estado versus secuencia), multiplicidades, instancias sin definición de clase y conflictos observables de comportamiento (entre máquinas de estado relacionadas por herencia). Las definiciones de estas relaciones de consistencia se han modificado para cumplir con la nueva especificación de UML (2.0).

- Objeto abstracto: se genera cuando se instancia una clase abstracta que no tiene hijos.
- Comportamiento no compatible: se genera cuando la secuencia ordenada de acciones recibida por un objeto en un diagrama de secuencia no existe como una traza de eventos en la máquina de estado asociada a la clase del objeto.
- Multiplicidad: los conflictos de multiplicidad se generan cuando los links entre los objetos en un diagrama de secuencia no respetan las restricciones de multiplicidad impuestas por las asociaciones correspondientes.

```
private static Vector initAssertions() {

    // Vector que alamacena la traducciones de todos los elementos
    // del modelo actual
    Vector assertions = new Vector();

    // Uso de la API de Poseidon para obtener los elementos del modelo
    Vector allElements = new Vector(pro.getUml1SmFactory().getCorePackage().getModelElement().refAllOfType());

    // contador que se usara para generar nombres unicos de instancias
    int counter = 0;

    // generacion de nombres unicos para los elementos
    for (Enumeration e_allElements = allElements.elements();e_allElements.hasMoreElements(); ){

        ModelElement element = (ModelElement) e_allElements.nextElement();

        // existen dos casos: el elemento tiene un nombre o es anonimo
        String name = element.getName();
        if (name != null){
            name = name.replace(' ', '_');
            name += ¨_¨ + counter;
            counter++;
        }
        else if (name == null || name.equals()){
            name = ¨anon¨;
            name += ¨_¨ + counter;
            counter++;
        }

        // se guarda el nombre de la instancia. Estos nombres seran usados
        // en la etapa de traduccion, al relacionar elementos
        setInstanceName(element, name);
    }

    // generacion de traduccion para cada elemento
    for (Enumeration e_allElements = allElements.elements();e_allElements.hasMoreElements(); ){

        ModelElement element = (ModelElement) e_allElements.nextElement();
        try{
            // se pide la traduccion al traductor y se agrega al vector de aserciones
            Vector translation = translator.translate(element);
            assertions.add(translation);
        }
        catch(Exception e){
            System.out.println(¨Missing translation¨);
        }
    }

    return assertions;
}
```

Figura 6.3: El método initAssertions(), responsable de obtener los objetos que representan al modelo y activar al traductor

```
/**
* UmlClass
* @param umlClass
* @return
*/

public Vector translate(UmlClass umlClass) {

    // se pide el nombre de instancia asignado a la clase
    String name = MainController.getInstanceName(umlClass);

    // se pide el nombre de instancia del modelo asociado a la clase
    String modelName = MainController.getInstanceName(umlClass.getNamespace());

    // se piden datos asociados a la clase
    boolean isAbstract = umlClass.isAbstract();
    boolean isLeaf = umlClass.isLeaf();
    Vector assertions = new Vector();

    // se generan las aserciones correspondientes

    // se genera la instancia
    assertions.add(¨(instance inst-¨ + name + ¨ class)\n ¨);

    // a la instancia se le asocia el nombre de la clase
    assertions.add(
        ¨(constrained inst-¨ + name + ¨ name-of-¨ + name + ¨ name)\n ¨);
    assertions.add(
        ¨(constraints (string= name-of-¨ + name + ¨ \¨¨ + umlClass.getName() + ¨ \¨))\n ¨);

    // se relaciona la clase con la instancia de modelo correspondiente
    assertions.add(
        ¨(related inst-¨ + modelName + ¨ inst-¨ + name + ¨ owned-element)\n ¨);

    // se agrega mas informacion acerca de la clase
    assertions.add(
        ¨(constrained inst-¨ + name + ¨ abstract-¨ + name + ¨ isAbstract)\n ¨);
    assertions.add(
        ¨(constraints (string= abstract-¨ + name + ¨ \¨¨ + isAbstract + ¨ \¨))\n ¨);
    assertions.add(
        ¨(constrained inst-¨ + name + ¨ leaf-¨ + name + ¨ isLeaf)\n ¨);
    assertions.add(
        ¨(constraints (string= leaf-¨ + name + ¨ \¨¨ + isLeaf + ¨\¨))\n ¨);

    return assertions;
}
```

Figura 6.4: El método traductor para los elementos del tipo Clase

- Instancias sin definición de clase: se generan cuando un objeto en un diagrama de secuencia es instancia de una clase que no existe en ninguno de los diagramas de clase.
- Conflictos observables de comportamiento: cuando existe una relación de herencia entre dos máquinas de estado, al esconder los eventos nuevos en la máquina de estado de la subclase, se debe observar de que cada traza de la máquina de estado de la subclase debe pertenecer al conjunto de trazas de la máquina de estado de la superclase.

Como se mencionó en el capítulo 2, la especificación de UML 2.0 introdujo varios cambios, especialmente con respecto al desarrollo basado en componentes. En el conjunto anterior de predicados, este tipo de diagrama no fue considerado. Para probar la extensibilidad del framework, se decidió tratar de incluir un predicado que revisase algún aspecto de consistencia de los diagramas de componente. El mecanismo de traducción fue extendido, agregándose el soporte para elementos como componentes, puertos y conectores. El nuevo predicado fue incluido en el archivo XML de configuración del plug-in. Se comprobó de que el framework funcionase de forma adecuada después de hacer estos cambios, probándose que el mecanismo de extensión del sistema para incluir nuevos elementos y nuevos predicados funciona adecuadamente.

A continuación se presenta el nuevo predicado introducido, con respecto a la existencia de los servicios ofrecidos por una componente.

Se dice que un servicio no está disponible cuando existen servicios que pertenecen a la interfaz ofrecida de la componente que no son implementados como operaciones públicas por ninguno de los clasificadores internos de la componente.

MODEL(m) - m es un modelo en UML

COMP(m) - es el conjunto de todos los componentes que aparecen en el modelo m

COS(i) - es el conjunto de servicios ofrecidos de la componente i

Class(i) - es el conjunto de clasificadores internos de la componente i

ClPO(j) - es el conjunto de operaciones públicas del clasificador j

$$
\begin{aligned}
\textit{Servicios no disponibles}(m) \;\Leftrightarrow\; & MODEL(m) \wedge \\
& \exists i | i \in COMP(m) \wedge \\
& COS(i) \not\subseteq \bigcup_{j | j \in Class(i)} ClPO(j)
\end{aligned}
$$

Capítulo 7

Ejemplos de uso

En este capítulo se presentan dos ejemplos de uso del framework. El primer ejemplo presentado es de comportamiento no compatible y el segundo de servicios no disponibles.

7.1. Comportamiento no compatible

El modelo representado por los diagramas presentados en las figuras 7.1, 7.2 y 7.3 especifica la estructura y el comportamiento para un sistema de cajeros automáticos (los detalles completos de este ejemplo están disponibles en [41]). El comportamiento especificado para la clase `ATM` en el diagrama de secuencia no es compatible con la definición de la máquina de estado para esta clase. Esto se debe a que la secuencia de acciones que recibe la instancia de la clase `ATM` en la figura 7.2 no existe como una traza de la máquina de estado - las operaciones darDinero y expulsarTarjeta se reciben en orden cambiado.

Usando la herramienta Poseidon, el usuario debe primero especificar el modelo en UML, en este caso, modelando los diagramas de clase, secuencia y estado correspondientes. Después de iniciarse el plug-in, se puede proceder a revisar la consistencia del modelo. Esto se hace a través del panel *Consistency-Manager*, el cual se encuentra disponible en la parte inferior de la pantalla de Poseidon. El usuario debe activar la traducción automática del modelo primero, oprimiendo el botón *Load Model*. En la figura 7.4, el usuario selecciona el chequeo de consistencia correspondiente y lo activa. Los resultados del análisis se muestran en este mismo panel.

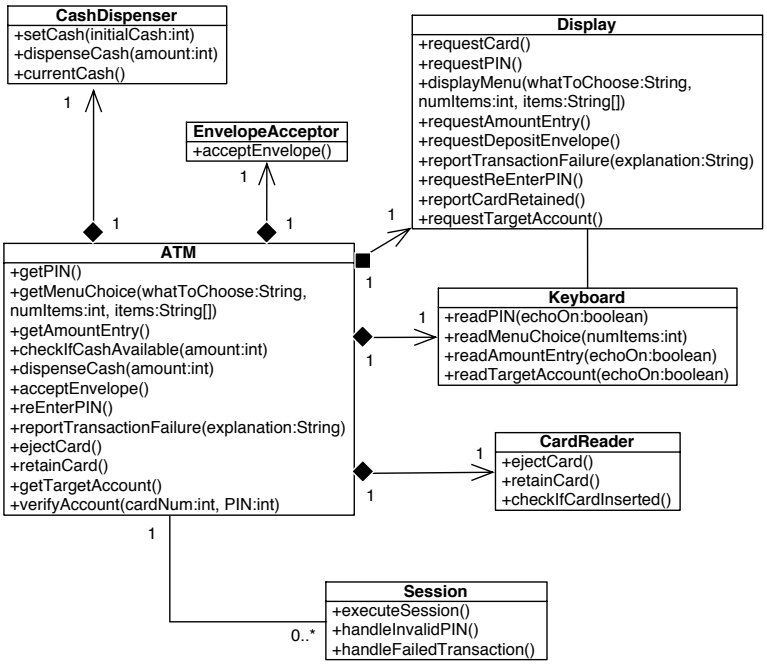

Figura 7.1: Comportamiento no compatible: diagrama de clase

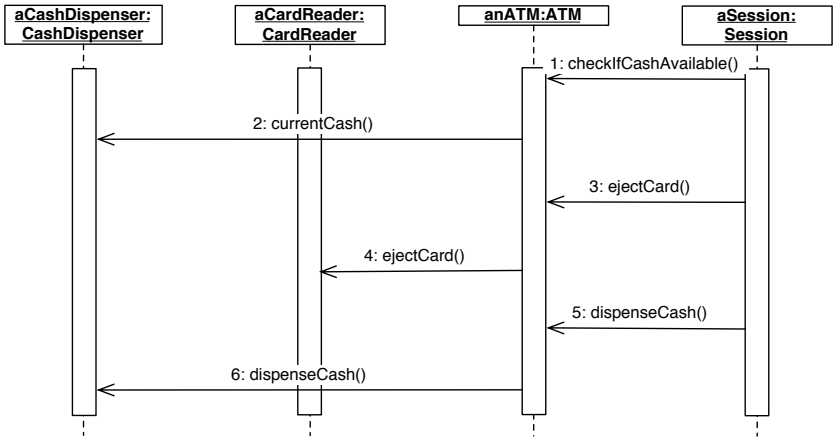

Figura 7.2: Comportamiento no compatible: diagrama de secuencia

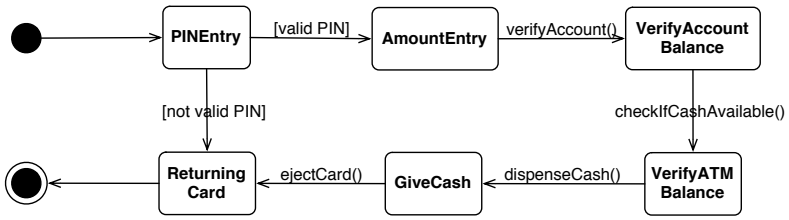

Figura 7.3: Comportamiento no compatible: diagrama de estado (clase ATM)

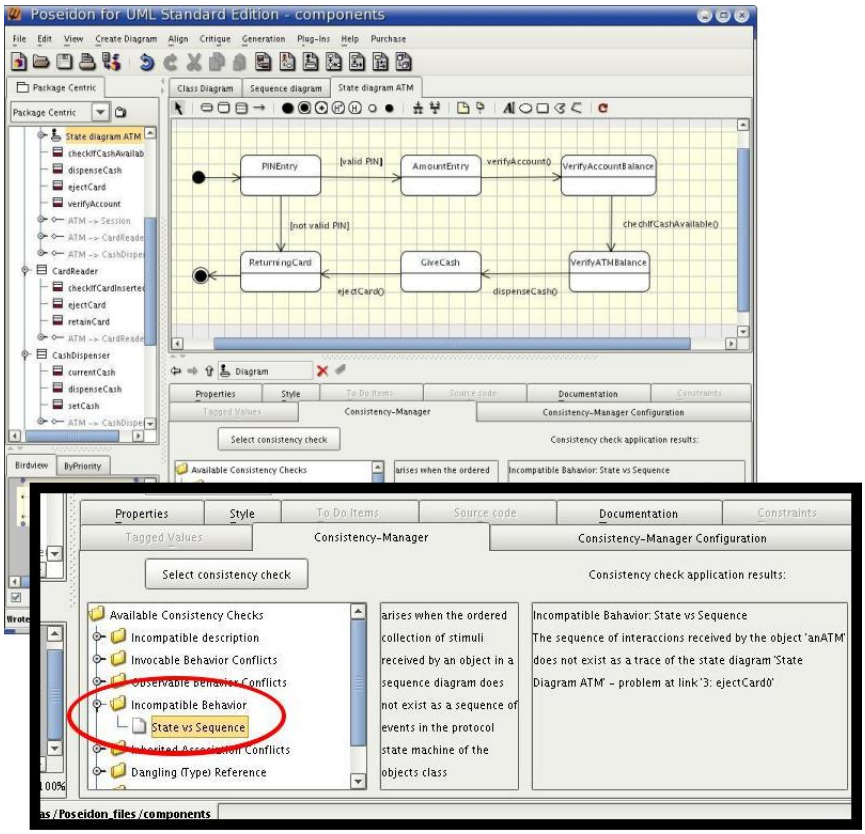

Figura 7.4: Elegir y aplicar la revisión de consistencia

7.2. Servicios no disponibles

A continuación, se muestra como se puede usar MCC para detectar casos de servicios no disponibles basándose en los diagramas de componentes y los de clases. El ejemplo que se muestra en la figura 7.5 fue tomado de la especificación de la superestructura de UML 2.0 [29, 37]. La componente `Order` tiene dos clasificadores internos, `OrderHeader` y `LineItem`. En esta versión del modelo, todavía no se han especificado operaciones para estas clases. La inconsistencia se produce porque `OrderEntry`, la interfaz ofrecida por la componente, ofrece la operación `getOrderEntry`. Esta operación no es implementada como método público por ninguna de las clases interiores de la componente.

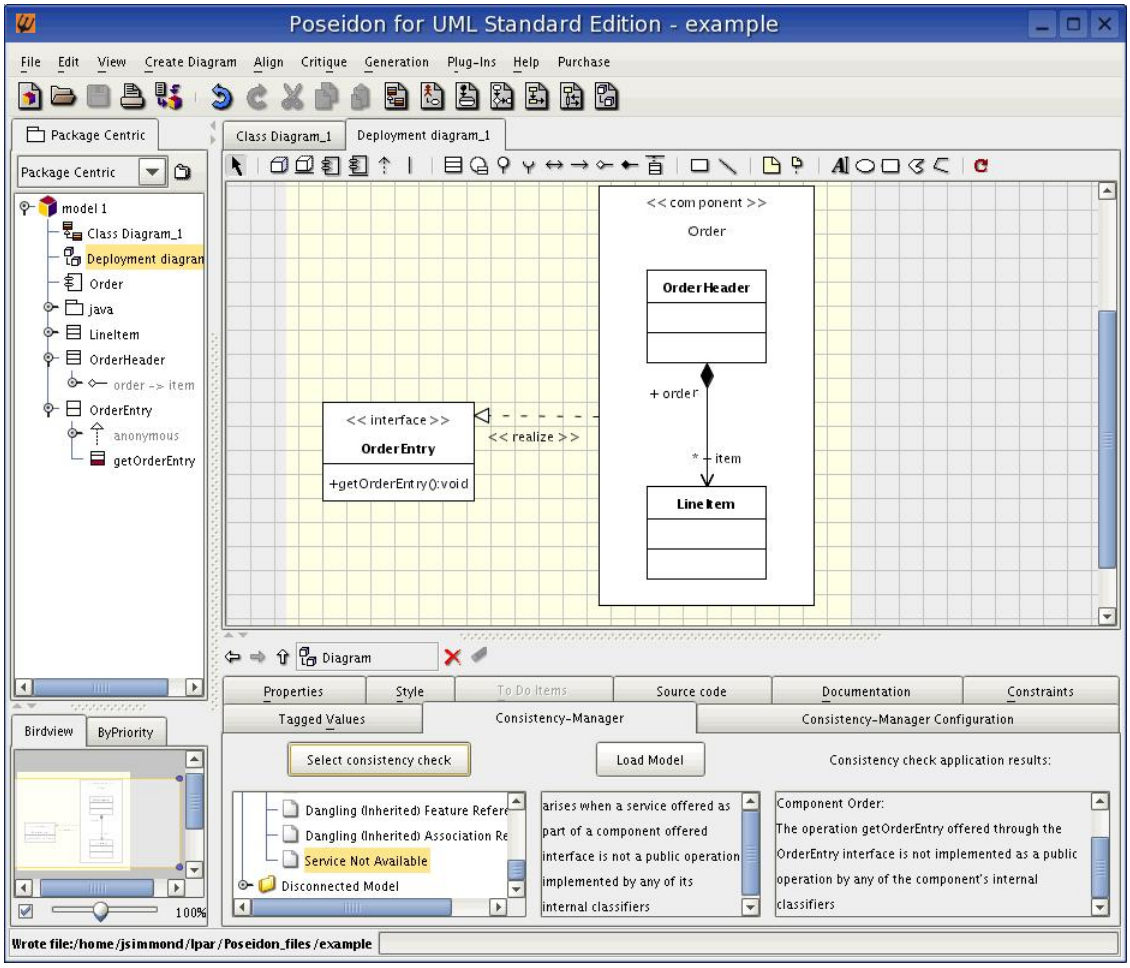

Figura 7.5: Ejemplo de servicios no disponibles

Capítulo 8

Discusión y Conclusiones

Dado el tamaño y la complejidad de los sistemas desarrollados hoy en día, se hace necesario el uso de herramientas de apoyo al proceso. Entre estas herramientas está UML, un lenguaje que permite representar al sistema modelado a través de varios diagramas complementarios. La nueva versión de este lenguaje (2.0) ha cobrado importancia al introducir nuevos elementos y refinar elementos existentes, para permitir su uso en la especificación de arquitecturas y sistemas basados en componentes. El estándar de este lenguaje especifica qué se puede modelar con este y cómo debe hacerse, pero no obliga a que los distintos diagramas de un modelo sean consistentes. Esto puede ser fatal dado que cada nueva vista introduce nuevas posibilidades de inconsistencia.

Las razones de por qué UML es permisivo al respecto es porque no todas las inconsistencias son indeseadas. Es común empezar con un diseño de alto nivel, refinándolo a medida que se avanza en el proceso de desarrollo. Inicialmente, no se han establecido todas las responsabilidades, así que los primeros diagramas son incompletos y posiblemente inconsistentes. Por ejemplo, podrían haber operaciones en el diagrama de secuencia que todavía no se han agregado al diagrama de clase, asociaciones faltantes, ciclos infinitos con respecto a las multiplicidades de las asociaciones, etc. Hacer revisiones manuales de la consistencia de los modelos no es una solución satisfactoria, por la posible complejidad y tamaño de estos.

Esto llevó a la idea de implementar un framework que pusiese en práctica los resultados obtenidos en [41]. Esta tesis estableció que usando Description Logics como formalismo, es posible detectar problemas de consistencia en diagramas en UML. El sistema DL escogido para llevar a cabo el trabajo fue Racer. Las razones por las cuales se escogió este sistema

fueron varias. Racer ofrece un lenguaje de especificación de terminología muy expresivo, tiene algoritmos completos, decidibles y eficientes para la clasificación, y el sistema ofrece interfaces de conexión fáciles de usar. Además, se presenta una demostración formal de que UML 2.0 puede ser formalizado usando DL.

Inicialmente se pensó en implementar una herramienta independiente, la cual recibiera los diagramas en UML a través de archivos en formato XMI (el formato estándar de intercambio de modelos en UML), para después traducirlos a DL para razonar acerca de los diagramas en Racer. Si se hubiese seguido este enfoque, el framework hubiera tenido que manejar la carga y descarga de estos archivos, reimplementando funcionalidad ya disponible en varias herramientas CASE para UML. Se estudiaron algunas herramientas CASE para UML, llegando a la conclusión de que sería ventajoso implementar el framework como plug-in para Poseidon para UML. Poseidon es una herramienta madura, con un buen mecanismo de inclusión de plug-ins, lo que da acceso a todos los elementos de los modelos generados en la aplicación. Una ventaja adicional de este enfoque es que el sistema es completamente portable, dado que existen versiones de Poseidon y Racer para Linux, Mac OS X y Windows.

Dada la importancia de sistemas basados en componentes, se experimentó con la extensión de la herramienta para incluir análisis de consistencia de diagramas de componentes con respecto a los diagramas ya considerados. Se definió la inconsistencia *Servicio no disponible*, que ocurre cuando la interfaz ofrecida por una componente incluye algún servicio que no esta implementado como una operación pública por alguna de las clases interiores de la componente. Esta extensión fue un éxito, agregándose las traducciones necesarias para el razonamiento acerca de diagramas de componentes e implementándose la nueva revisión de consistencia.

El trabajo realizado hasta ahora permite hacer análisis formales de modelos en UML, a través de una interfaz amigable, usando el mismo formalismo para analizar un conjunto de variados problemas conocidos de consistencia. La traducción de los modelos generados por el usuario queda disponible a través del framework. Esto da pie para varias posibles extensiones al sistema, lo que permite identificar los siguientes puntos como trabajo futuro:

- Poseidon actualmente trabaja con sólo un modelo a la vez. Sería interesante agregar soporte para varios modelos, relacionándolos a través de versiones, para permitir la revisión de consistencia durante el proceso de evolución del diseño de un sistema.
- Actualmente la herramienta ofrece sólo 6 revisiones de consistencia, de un conjunto

inicial de 18 mencionados en [41] más la revisión *Servicio no disponible* que fue desarrollada en este trabajo.

- Finalmente, los modelos traducidos pueden ser usados para llevar a cabo otros tipos de análisis, como la generación de estadísticas de calidad del modelo y la detección de refactorings [11], técnica con la cual también se experimento en [41].

Bibliografía

[1] Carlos Areces. *Logic Engineering. The Case of Description and Hybrid Logics*. PhD thesis, ILLC, University of Amsterdam, 2000.

[2] argouml.tigris.org. ArgoUML, August 2004. `http://argouml.tigris.org`.

[3] F. Baader, D. McGuinness, D.Ñardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[4] Grady Booch. *Object-oriented Analysis and Design with Applications, 2nd edition*. Benjamin Cummings, 1993.

[5] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[6] R. Brachman and H. Levesque. The tractability of subsumption in frame-based description languages. *AAAI-84*, pages 34–37, 1984.

[7] Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. A formal framework for reasoning on UML class diagrams. In *Proc. of the 13th Int. Sym. on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2002.

[8] Mariela Cortés, Marcus Fontoura, and Carlos de Lucena. Using Refactoring and Unification Rules to Assist Framework Evolution. *UPGRADE*, 5(2):49–55, April 2004.

[9] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, March 2000.

[10] Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla and Cris Kobryn, editors, *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada.

[11] Martin Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.

[12] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition).* Addison-Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems.* Addison-Wesley, 1994.

[14] gentleware.com. Poseidon, August 2004. `http://www.gentleware.com/products/poseidonPE.php3`.

[15] Volker Haarslev and Ralf Möller. RACER, April 8 2003. `http://www.fh-wedel.de/~mo/racer/`.

[16] Volker Haarslev, Ralf Möller, Ragnhild Van Der Straeten, and Michael Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In *Proceedings of the International Workshop in Description Logics 2004 (DL2004)*, 2004.

[17] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[18] Ian Horrocks. FaCT, September 2004. `http://www.cs.man.ac.uk/~horrocks/FaCT/`.

[19] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 161–180. Springer-Verlag, 1999.

[20] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17).* Springer-Verlag, 2000.

[21] IBM. Rational Software, October 2004.
http://www-306.ibm.com/software/rational/.

[22] ISI. Loom, April 2003.
http://www.isi.edu/isd/LOOM/LOOM-HOME.html.

[23] Ivar Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[24] Boris Litvak, Shmuel Tyszberowicz, and Amiram Yehudai. Consistency Validation of UML Diagrams. In *Correctness of Model-based Software Composition (CMC) Workshop, ECOOP*, 2003.

[25] Robert MacGregor. Retrospective on Loom, August 13 1999.
http://www.isi.edu/isd/LOOM/papers/macgregor/Loom_Retrospective.html.

[26] Carnegie Mellon. Carnegie Mellon Software Engineering Institute, June 7 2004.
http://www.sei.cmu.edu/str/descriptions/cbsd.html.

[27] Michael Moors. Rose Model Checker, October 2004.
http://www.rationalrose.com/modelchecker/index.htm.

[28] Marvin Minsky. Form and content in computer science (1970 ACM Turing lecture). *J. ACM*, 17(2):197–215, 1970.

[29] Object Management Group. UML 2.0 Infrastructure Specification. 03-09-15.pdf, September 2003.
http://www.omg.org/docs/ptc/.

[30] Object Management Group. UML 2.0 OCL Specification. 03-10-14.pdf, October 2003.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[31] Object Management Group. UML 2.0 OCL Superstructure. 03-10-14.pdf, October 2003.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[32] Object Management Group. UML Profile for Enterprise Distributed Object Computing Specification. 03-09-11.pdf, September 2003.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[33] Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.
http://www.omg.org/uml.

[34] Object Management Group. XML Metadata Interchange XMI. 03-05-02.pdf, May 2003.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[35] Object Management Group. MOF 2.0 Core and UML 2.0 Infrastructure Finalization Task Force. 04-01-13.pdf, January 2004.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[36] Object Management Group. UML 2.0 Superstructure Finalization Task Force. 04-01-11.pdf, January 2004.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[37] Object Management Group. UML 2.0 Superstructure Specification. 04-05-02.pdf, May 2004.
http://www.omg.org/docs/ptc/.

[38] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, 1991.

[39] T. Schfer, A. Knapp, , and S. Merz. Model Checking UML State Machines and Collaborations. *In Electronic Notes in Theoretical Computer Science*, 47:1–13, 2001.

[40] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[41] Jocelyn Simmonds. Consistency Maintenance of UML Models with Description Logics. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, 2003.

[42] Jocelyn Simmonds. MCC, December 2004.
http://www.dcc.uchile.cl/~jsimmond/mcc/mcc.html.

[43] Jocelyn Simmonds, Tom Mens, and Ragnhild Van Der Straeten. Maintaining consistency between UML models using description logic. In *LMO 2004, Langages et Modèles à Objets*, 2004.