Un Marco Formal para Transformaciones en la Ingeniería de Software Conducida por Modelos

Mg. Roxana Silvia Giandini

Directora de Tesis

Dra. Claudia Pons

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas

Facultad de Informática
Universidad Nacional de La Plata

- Noviembre 2007 -

Defensa de Tesis Realizada el día

- 18 de Febrero de 2008 -

Jurado de Tesis

Dr. Oscar Pastor López
Universidad Politécnica de Valencia, España

Dr. Mario Piattini Velthuis
Universidad de Castilla-La Mancha, España

Dr. Marcelo A. Falappa
Universidad Nacional del Sur, Argentina

A mi familia.

Especialmente a mis hijas Rocío y Paula,
compañeras de alegrías y tristezas.

A la memoria de mi hijo Iván, que está conmigo por siempre.

Agradecimientos

A Claudia Pons por su ayuda incondicional, su gran generosidad y por el estímulo brindado para poder llegar a la concreción de esta tesis. Por sobre todo, agradezco poder contar con su amistad.

A Gabriel Baum por el interés manifestado y por haber definido junto a sus amigos, un par de décadas atrás, la "teoría de problemas", formalismo utilizado como fundamento de este trabajo de investigación.

Índice General

Índice de Figuras

Capítulo
1

Introducción

Este capítulo describe la motivación y el contexto en el que se enmarca esta tesis; nuestra propuesta para los problemas a resolver y los objetivos planteados en este trabajo de investigación; una breve mención de los principales aportes a la Ingeniería de Software Conducida por Modelos y finalmente, la organización general de la tesis.

1.1 Motivación: MDE y la Transformación de Modelos

A lo largo de esta década la Ingeniería de Software Conducida por Modelos (MDE, acrónimo de *Model Driven Engeenering*) [1] se ha convertido en un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos y soportado por potentes herramientas que generan código a partir de modelos. MDE promete una mejora de la productividad y de la calidad del software generado debido a que se reduce el salto semántico entre el dominio del problema y de la solución; se reducen los tiempos de desarrollo y las herramientas de generación pueden aplicar frameworks, patrones y técnicas cuyo éxito se ha comprobado.

El paradigma MDE tiene dos ejes principales: - por un lado hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Para ello, el MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM (Platform Independent Model) y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM (Platform Specific Model); - por otro lado, los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software. Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico. La transformación entre modelos constituye el motor del MDE y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.
En los próximos apartados de esta sección, enumeramos los beneficios principales que aporta MDE al proceso de desarrollo de software y a continuación, presentamos un enfoque concreto (MDA) para esta metodología de desarrollo.

1.1.1 Beneficios de MDE sobre el desarrollo tradicional de software

Específicamente, enunciaremos los problemas más representativos que aparecen en el desarrollo tradicional de software, analizaremos la causa de estos problemas y veremos cómo MDE los trata, enfatizando fuertemente en la necesidad de contar con herramientas sólidas que automaticen el proceso de transformación de modelos.

El problema de la productividad

El proceso de desarrollo de software tradicional, es conducido a menudo por el diseño de bajo nivel y el código. Aún si el proceso es iterativo e incremental, los documentos y los diagramas se producen sólo en las fases tempranas (requerimientos, análisis). Los documentos y los diagramas correspondientes creados en las primeras fases pierden rápidamente su valor tan pronto como la codificación comienza. La conexión entre los diagramas y el código se pierde mientras se progresa la fase de codificación. En vez de ser una especificación exacta del código, los diagramas se convierten frecuentemente en dibujos con poca relación. Cuando un sistema cambia, la distancia entre el código, el texto y los diagramas producidos en las primeras fases, crece. Los cambios se hacen a menudo solo en el código, porque el tiempo para actualizar los diagramas y otros documentos de alto nivel no está disponible. También, el valor agregado de diagramas actualizados y documentos es cuestionable, porque cualquier nuevo cambio se encuentra en el código de todos modos. La idea de *Extreme Programming* (XP) [2] se volvió popular en muy poco tiempo. Una de las razones de esto es que reconoce el hecho de que el código es la fuerza impulsora del desarrollo del software. Afirma que las únicas fases en el proceso del desarrollo que son realmente productivas son la codificación y el testeo. Alistair Cockburn escribió en su libro de desarrollo ágil del software [3], que el acercamiento de XP soluciona solamente una parte del problema. Mientras un mismo equipo trabaja en el desarrollo del software, existe bastante conocimiento de alto nivel en sus cabezas para entender el sistema. Los problemas comienzan cuando se cambia parte del equipo, que sucede generalmente después de que se entrega del primer lanzamiento del software. La gente necesita mantener el software. Tener solamente el código hace muy difícil la tarea de mantenimiento de un sistema.
En **MDE** el foco está en el desarrollo de un PIM. Los PSMs necesarios son generados por una transformación desde un PIM. Por supuesto, es necesario que alguien defina la transformación, que es una tarea difícil y especializada. Pero esa transformación necesita ser definida sólo una vez, y puede ser aplicada en el desarrollo de muchos sistemas.
Ya que los desarrolladores necesitan enfocarse solo en los PIM, pueden trabajar independientemente de los detalles de las plataformas. Esos detalles técnicos serán agregados automáticamente por la transformación del PIM al PSM. Esto mejora la productividad de dos maneras:

- En primer lugar, los desarrolladores del PIM tienen menos trabajo que hacer ya que los detalles específicos de la plataforma no necesitan ser diseñados, sino que ya están en la definición de la transformación.

Pensando en el código, habrá mucho menos código que escribir ya que una gran cantidad de código será generado a partir del PIM.

- En segundo lugar, los desarrolladores pueden enfocarse en el PIM en vez de en el código. Esto hace que el sistema desarrollado se acerque más a las necesidades del usuario final, el cual tendrá mejor funcionalidad en menor tiempo.

Estas mejoras en la productividad se pueden alcanzar con el uso de herramientas que automaticen completamente la generación de un PSM a partir de un PIM; es decir herramientas que hagan automático el proceso de **transformación** de modelos.

El problema de la portabilidad

La industria del software tiene una característica especial que la diferencia de las otras industrias. Cada año, o cada menos tiempo, aparecen nuevas tecnologías y rápidamente llegan a ser populares (por ejemplo Java, Linux, XML, HTML, UML, NET, JSP, ASP, Flash[1], servicios de WEB, etc.). Muchas compañías necesitan seguir estas nuevas tecnologías por buenas razones:

- Los clientes las exigen (por ejemplo, interfaces de WEB).

- Son la solución para algunos problemas reales (por ejemplo, XML para el problema de intercambio o Java para el problema de portabilidad).

- La empresa que desarrolla la herramienta deja de apoyar las viejas tecnologías y se centra en las nuevas.

Las nuevas tecnologías ofrecen beneficios concretos para las compañías y muchas de ellas no pueden quedarse atrás, por lo tanto, tienen que utilizarlas muy rápidamente. El software ya existente puede seguir sin cambios, pero necesitará comunicarse con los nuevos sistemas que serán construidos usando una nueva tecnología.
Dentro de **MDE** la portabilidad se lleva a cabo enfocando en el desarrollo de PIMs que son independientes de la plataforma. El mismo PIM puede ser automáticamente transformado en muchos PSMs para diferentes plataformas. Por lo tanto, todo lo que se especifica en el nivel de PIM es completamente portable, solo depende de las herramientas de **transformación** disponibles.

El problema de la interoperabilidad

Los sistemas de software raramente funcionan en forma aislada. La mayoría de los sistemas necesitan comunicarse con otros que generalmente ya existen. Incluso cuando los sistemas se construyen completamente, usan a menudo múltiples tecnologías, a veces de versionados o épocas diferentes. Por ejemplo, un sistema WEB necesita usar una base de datos para almacenar información.

[1] Algunas de estas tecnologías están descriptas en el **Glosario de siglas y términos**.

En **MDE** se pueden generar muchos PSMs a partir del mismo PIM, y estos frecuentemente pueden estar relacionados. Estas relaciones se llaman *bridges* o puentes.

Como ejemplo, supongamos que transformamos un PIM (diagrama de clases UML) a dos PSMs, el primero a un modelo Java y el segundo a un modelo de base de datos relacional. Para la clase "Cliente" en el PIM, sabemos que clase Java genera y que tabla genera. Construir un puente entre ambos elementos es posible. Para retornar un objeto de la base de datos, se deberían leer los datos de la tabla Cliente e instanciar un objeto de la clase Cliente.

La interoperabilidad entre plataformas puede ser realizada por herramientas para transformación de modelos que generen, además de PSMs, puentes entre ellos y posiblemente entre otras plataformas.

El problema del mantenimiento y la documentación

La documentación fue siempre un punto débil en el proceso del desarrollo de software. La sensación de la mayoría de los desarrolladores es que su tarea principal es producir código. Generar documentación durante el desarrollo cuesta tiempo y retrasa el proceso; sin embargo ayudará en su tarea, a los encargados de integrar el proyecto en etapas finales. Así pues, escribir documentación se ve como algo para el futuro, no para el presente. Por otro lado, no hay incentivo para la documentación, con excepción del líder de proyecto, que sostiene que la documentación es necesaria. Por supuesto, los desarrolladores están equivocados. Su tarea es desarrollar sistemas que puedan cambiar y que se puedan mantener en el tiempo. A pesar de las sensaciones de muchos desarrolladores, escribir la documentación es una de sus tareas esenciales.

Una solución a este problema, a nivel de código, es la facilidad de generar la documentación directamente desde el código fuente, asegurándose de que este siempre actualizada. La documentación es parte del código y no algo separado. Esta solución, sin embargo, soluciona únicamente el problema de la documentación en niveles inferiores. La documentación de alto nivel (texto y diagramas) todavía necesita ser mantenida a mano. Dada la complejidad de los sistemas que se construyen, la documentación en un nivel más alto de la abstracción resulta obligatoria.

Con **MDE** los desarrolladores pueden enfocarse solamente en el PIM, el cual tiene un nivel más abstracto que el código. El PIM es usado entonces para generar PSMs, el cual posteriormente será usado para generar código. Por lo tanto, el modelo será una exacta representación del código. Así el PIM cumplirá la función de documentación de alto nivel necesaria en cualquier sistema de software.

La gran diferencia es que el PIM no se deja de lado luego de ser escrito. Los cambios hechos en el sistema serán hechos sobre el PIM y regenerando PSMs y código. Esta tarea requerirá su automatización mediante herramientas adecuadas para ello.

1.1.2 MDA: La Arquitectura Dirigida por Modelos

Han surgido varios enfoques dentro del ámbito de MDE, pero sin duda la iniciativa más conocida y extendida es la MDA, acrónimo de *Model Driven Architecture* [4] [5] [6], presentada por el consorcio OMG [7] en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos.

MDA propone el uso de un conjunto de estándares como Meta Object Facility MOF [8], UML [9], JMI [10] o XMI [11][2]. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos PIM y modelos PSM. En general, el código de una aplicación se puede generar a partir de un modelo PIM, mediante sucesivas transformaciones de modelos hasta llegar al código fuente. La idea es también poder implementar el modelo PIM de un sistema en diferentes plataformas (es decir generar varios modelos PSM), lo que nos llevaría a disponer de sistemas más robustos a los cambios tecnológicos, y por otro lado poder realizar las labores de modelado abstrayéndonos totalmente de los detalles de la tecnología que usemos como soporte.

Las transformaciones modelo-modelo y modelo-código juegan un papel crucial en MDA y son necesarios lenguajes de transformación especiales para escribir cada una de las definiciones de transformación que establecen las correspondencias (relaciones, *mappings*) entre dos lenguajes de modelado (o entre un lenguaje de modelado y un lenguaje de programación).

En la actualidad existen un buen número de herramientas y lenguajes que soportan MDA, entre las que destacan OptimalJ [12], ArcStyler [13] y AndroMDA [14]. Cada herramienta incorpora un mecanismo propio para el manejo de transformaciones. Una iniciativa de OMG es la definición de un lenguaje de transformaciones estándar denominado QVT [15], aún en proceso de estandarización.

Para llevar a cabo todo este proceso MDA define fases que a su vez marcan tres niveles de abstracción del sistema a desarrollar, como muestra la Figura 1-1:

- **Fase I. Creación de un *Modelo Independiente de la Plataforma*** (*Platform Independent Model* o *PIM*), es el modelo con el mayor nivel de abstracción del sistema, describe la funcionalidad del sistema pero omitiendo detalles de cómo y dónde va a ser implementado.
- **Fase II. Transformación del PIM a uno o varios *Modelos Específicos de Plataforma*** (*Platform Dependent Model* o *PSM*). Un PSM es el mismo modelo que el PIM pero describiendo el sistema de acuerdo a una plataforma dada, como .NET o J2EE.
- **Fase III. Generación del código a partir del PSM.** Consiste en hacer un *mapping* (transformación) del modelo PSM a la plataforma en la que está

[2] Estos estándares y demás acrónimos incluidos en la tesis, además de contar con referencia bibliográfica, están descriptos en el **Glosario de siglas y términos**.

especificado. Esto se puede realizar de forma automática ya que el PSM se encuentra muy ligado a dicha plataforma.

Sin embargo, pueden observarse en la práctica, situaciones donde existan más de tres niveles de abstracción. Pueden existir transformaciones intermedias entre modelos que repitan la Fase II (PIM/PSM), donde el modelo de salida de una transformación es la entrada para la siguiente, generando así una cadena hasta llegar a la generación de código [6].

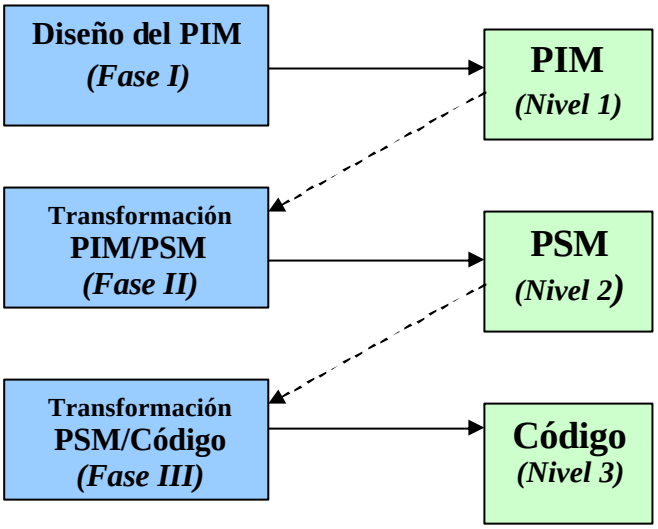

Figura 1-1. Pasos y niveles en el desarrollo MDA

1.2 Nuestra Propuesta: Un Marco Formal para Transformación de Modelos que Soporte Composición entre Transformaciones

Como vimos en la sección anterior, la iniciativa MDE y en particular la propuesta MDA, cubren un amplio espectro de áreas de investigación: lenguajes para la descripción de modelos, definición de lenguajes de transformación entre modelos, construcción de herramientas de soporte a las distintas tareas involucradas, aplicación de los conceptos en métodos de desarrollo y en dominios específicos, etc.

Actualmente, algunos de estos aspectos están bien fundamentados y se están empezando a aplicar con éxito, otros sin embargo están todavía en proceso de definición. En este contexto son necesarios esfuerzos que conviertan MDE y sus conceptos y técnicas relacionados en una propuesta coherente, basada en estándares abiertos, y soportada por técnicas y herramientas maduras. Las transformaciones entre modelos requieren de lenguajes específicos para su definición. Estos lenguajes deben tener base formal, por ejemplo tener un metamodelo que los sustente en su definición sintáctica y una formalización de su semántica; deben permitir también un tratamiento automatizado y ser compatibles (al igual que los modelos que transforman) con MOF.

Existe una visión más genérica sobre la metodología MDE, en la cual enfatizar sobre la diferencia entre modelos PIM y PSM no es el punto predominante. La clave para esta vista más genérica es que el proceso de desarrollo de software es implementado mediante una red de transformaciones que se combinan o componen en modos diversos. Las transformaciones pueden ser especificadas o implementadas usando diferentes herramientas y diferentes lenguajes. Pueden también manipularse como entidades de caja negra. La habilidad de organizar o componer diferentes transformaciones en manera flexible y confiable con el fin de producir el resultado requerido, es un desafío principal en MDE. Por lo tanto la definición de lenguajes de transformación debe incluir mecanismos de composición.

1.2.1 Objetivo de la Tesis

Por los motivos expuestos, este trabajo tiene como objetivo general:
Definir una base formal para lenguajes de transformación de modelos.

Este objetivo se logra mediante los siguientes sub-objetivos:

- Presentar un lenguaje para expresar transformaciones entre modelos que inspira su metamodelo inicial en el lenguaje QVT que es la especificación de OMG [2] para transformaciones, aún en proceso de definición. El lenguaje que proponemos está orientado a ser el mínimo para poder expresar relaciones y *queries* de transformación entre modelos.

- Definir formalmente la sintaxis y la semántica de dicho lenguaje minimal, para permitir validar consistencia entre una especificación del lenguaje (declarativa) y su implementación (operacional).

- Describir cómo la formalización propuesta puede ser aplicada como base para construir un fundamento matemático para el problema de **composición** de transformaciones, abarcando ambas dimensiones (declarativa y operacional).

1.2.2 Aportes de la Tesis

Fundamentalmente, el aporte principal del trabajo a la metodología MDE se resume en:

Contribuir a la madurez de los lenguajes de transformación de modelos

Específicamente, este aporte consiste en:

- Proponer mecanismos de validación de consistencia entre descripciones construidas a partir de lenguajes de transformación, a distintos niveles de abstracción (declarativa y operacional).

- Proveer definiciones precisas de operaciones de composición sobre transformaciones, útiles al desarrollo de software conducido por modelos.

- Permitir, bajo esta definición formal, la automatización del proceso construyendo herramientas CASE sólidas y precisas que mejoren, propicien y faciliten la aplicación y uso de esta metodología de desarrollo reciente.

1.3 Publicaciones

Algunos de los artículos publicados en congresos iberoamericanos e internacionales con referato, relacionados con el tema de este trabajo de tesis son:

"Un lenguaje para Transformación de Modelos basado en MOF y OCL"

Autores: Roxana Giandini, Claudia Pons. Publicado como full paper en la "XXXII Conferencia Latinoamericana de Informática (CLEI) 2006". Santiago, Chile. 20 al 25 de Agosto de 2006.

"A Minimal OCL-based Profile for Model Transformation"

Autores: Roxana Giandini, Gabriela Pérez, Claudia Pons.
Publicado en las actas de las VI Jornadas Iberoamericanas de Ingeniería de software e Ingeniería del Conocimiento (JIISIC´07) ISBN: 978-9972-2885-2-4. Lima, Perú, Febrero de 2007

"Composición de Transformaciones de Modelos en MDE basada en el Álgebra Relacional"

Autores: Roxana Giandini, Gabriela Pérez, Claudia Pons.
Memorias del X Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software (IDEAS´07) pág: 239-252. Editores: F. Losario, G. Horta Travassos, V. Pelechano, I. Díaz, A. Matteo. Mayo, 2007. Caracas, Venezuela ISBN: 978-980-325-323-3, Realizado en Isla de Margarita, del 7 al 11 de Mayo 2007

"**An Algebraic Approach for Composing Model Transformations in QVT**"

Autores: Claudia Pons, Roxana Giandini, Gabriela Pérez, Gabriel Baum.
In Proceedings of ATEM 2007 - 4th International Workshop on (Software) Language Engineering, at MoDELS 2007 Conference. Nashville, USA. October 2007

1.4 Organización del Trabajo

Los próximos capítulos de este trabajo se estructuran de la siguiente manera:

En el capítulo 2 introducimos el concepto de metamodelado para definición de lenguajes y presentamos con más detalle el concepto de transformación de modelos. Seguidamente introducimos los conceptos básicos que se utilizan en transformaciones de modelos MOF mediante la descripción de los elementos principales del lenguaje QVT. Finalmente se presenta un ejemplo de transformación de modelos.

El Capítulo 3 presenta nuestra definición del lenguaje SimpleQVT (SQVT), específicamente su sintaxis, tanto en su parte declarativa como operacional, para expresar transformaciones entre modelos.

En el capítulo 4 definimos la semántica -a nivel declarativo y a nivel operacional- de lenguajes para transformación de modelos. Dicha semántica está definida en base al formalismo de la *teoría intuitiva de problemas*, por lo que en las primeras secciones se introducen los conceptos básicos de dicho formalismo.

El capítulo 5 presenta un análisis sobre la necesidad de contar con mecanismos de composición entre transformaciones. Seguidamente enunciamos los mecanismos de composición que incluye el lenguaje QVT en su definición.

En el capítulo 6 presentamos una formalización de las operaciones de composición de transformaciones de modelos -a nivel declarativo y a nivel operacional-. Dicha formalización está definida en base el formalismo de la *teoría algebraica de problemas*, por lo que en la primera sección se introducen los conceptos algebraicos que dicho formalismo expresa en términos de problemas y soluciones.

El capítulo 7 presenta un análisis de los casos de aplicación de las operaciones de composición propuestas en el capítulo anterior.

En el capítulo 8 se presenta un prototipo de implementación para la propuesta, desarrollado como un plug-in de Eclipse.

El Capítulo 9 enuncia diversos trabajos relacionados con nuestro tema de investigación, mientras que el Capítulo 10 presenta las conclusiones finales, destacando las contribuciones principales al desarrollo de software conducido por modelos que brinda esta tesis, así como las líneas de trabajo a futuro que pueden extenderla.

Capítulo
2

Conceptos Básicos Sobre Transformación De Modelos

Dado que el objetivo del capítulo siguiente es definir un lenguaje simple para transformación de modelos, en la primera sección de este capítulo introducimos la técnica del metamodelado, utilizada para definir la sintaxis de lenguajes. En la segunda sección presentamos con más detalle el concepto de transformación y su definición. En la sección 3, mediante la descripción de los elementos principales del lenguaje QVT, introducimos los conceptos básicos que se utilizan en transformaciones de modelos MOF. Finalmente la sección 4 presenta un ejemplo de transformación de modelos.

2.1 Metamodelos y Definición de Lenguajes

El metamodelado es un mecanismo que permite construir formalmente lenguajes de modelado, como lo son el UML y el RDBMS.
La Arquitectura 4 capas de Modelado es la propuesta de OMG orientada a estandarizar conceptos relacionados al modelado, desde los más abstractos a los más concretos.
Los niveles definidos en esta arquitectura se denominan comúnmente: M3, M2, M1, M0:
El **nivel M3** (el meta-metamodelo) es el nivel más abstracto, donde se encuentra el MOF (*Meta Object Facility)* [8], un lenguaje para describir lenguajes de modelado. Un lenguaje de modelado usa MOF para definir formalmente la sintaxis abstracta de su conjunto de constructores de modelos (como el del lenguaje UML).

El **nivel M2** (el metamodelo), sus elementos son lenguajes de modelado, por ejemplo UML. Los conceptos a este nivel podrían ser Clase, Atributo, Asociación.

El **nivel M1** (el modelo del sistema), sus elementos son modelos de datos, por ejemplo entidades como “Persona”, “Auto”, atributos como “nombre”, relaciones entre estas entidades.

El **nivel M0** (instancias del sistema) modela al sistema real. Sus elementos son datos, por ejemplo “Juan López”, que vive en “Av. Libertador 345”

La Figura 2-1 muestra un ejemplo de esta jerarquía de metamodelado en 4 capas.

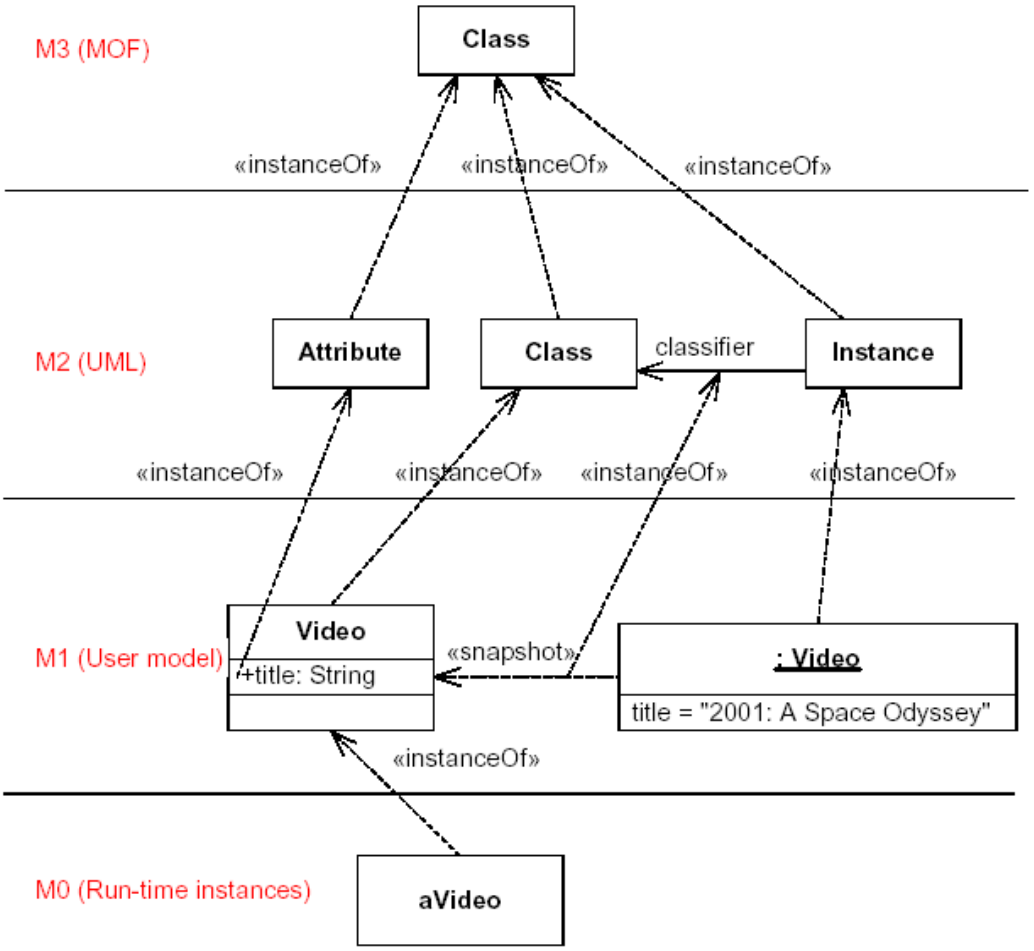

Figura 2-1. Ejemplo de la Arquitectura 4 capas de Modelado

En este contexto, respecto a la definición sintáctica de un nuevo lenguaje, existen dos opciones: una de ellas es crear el lenguaje definiendo su metamodelo en el nivel M2 y creando nuevas metaclases que lo implementen. Otra opción es implementarlo mediante el mecanismo estándar de extensión de UML (crear un *profile*) que consiste en definir estereotipos que extienden metaclases ya existentes en las especificaciones de OMG para representar los nuevos conceptos del lenguaje. Esta última opción es la que elegimos para la definición del lenguaje que presentaremos en el próximo capítulo. El próximo apartado presenta con más detalle especificaciones extensibles, ya existentes entre los estándares de OMG.

❖ MOF y la Infraestructura 2.0

En el nivel más abstracto de la Arquitectura 4 capas de Modelado se ubica MOF que es un Meta-metamodelo cerrado. En consecuencia, para definir lenguajes que incorporen la definición de nuevos conceptos, por ejemplo para transformaciones, a través de extensiones de especificaciones OMG ya existentes, no deberíamos extender a MOF, por ser una especificación cerrada. Analicemos entonces, otras

especificaciones, estrechamente vinculadas a MOF, que define OMG: una de estas especificaciones es la Infraestructura 2.0 [20] para lenguajes de modelado.
La Infraestructura es la especificación más simplificada que define los constructores básicos y conceptos comunes para lenguajes de modelado. Podemos decir que es independiente al lenguaje UML en sí. El metamodelo de UML se complementa con la especificación de la Superestructura [9], que define los constructores a nivel usuario de UML 2.0.
El caso de la Infraestructura es interesante por su definición recursiva respecto a MOF: por un lado, puede verse definida como instancia de MOF; por otro lado el MOF mismo se basa, o bien usa elementos del paquete principal, llamado *Core*, de la Infraestructura para su definición, situación que nos permite identificar a la Infraestructura como un meta-metamodelo. La Figura 2-2 ilustra como UML, CWM (*Common Warehouse Model*) y MOF dependen, para su definición, de este paquete común. El paquete *Core* puede ser considerado el núcleo arquitectural de MDA. La intención es reusar todo o parte de este paquete al definir otros metamodelos, brindando el beneficio de contar con sintaxis abstracta y semántica que ya han sido definidas.

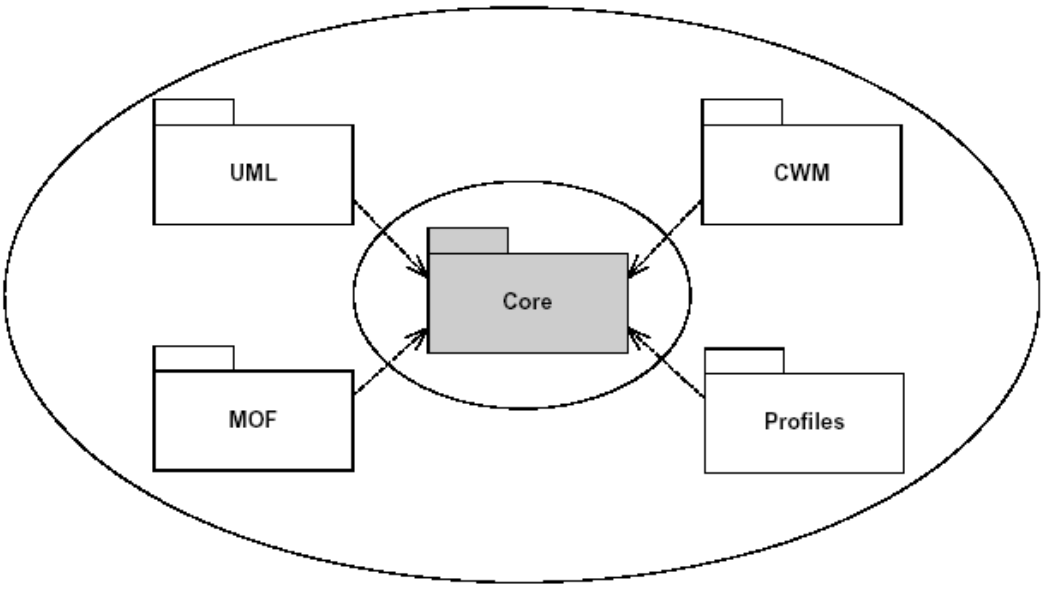

Figura 2-2. El rol del Paquete Core de la Infraestructura

Con el fin de facilitar su reuso, el paquete *Core* está subdividido en 4 paquetes: *PrimitiveTypes, Abstractions, Basic* y *Constructors*, como muestra la Figura 2-3. Cada uno de ellos a su vez, está también dividido en paquetes más refinados.
Por ejemplo, el paquete *Basic* representa unos pocos constructores usados como base para la definición de UML, MOF y otros metamodelos.
Las metaclases en *Basic* están especificadas usando cuatro diagramas:
- el diagrama de Tipos define metaclases abstractas que especifican elementos con nombre y con tipo (ver Figura 2-4). También especifica que cualquier elemento puede tener comentarios.

- el diagrama de Clases define los constructores (*Class, Property, Operation,* etc) para modelado basado en clases (ver Figura 2-5).
- el diagrama de Tipos de Datos define las metaclases para tipos de datos (ver Figura 2-6).
- el diagrama de Paquetes define los constructores relacionados con paquetes y su contenido (ver Figura 2-7).

En particular, la mayoría de las metaclases (*Class, Package, Operation,* etc.) que extenderemos al crear los *profiles* para transformaciones, pertenecen al paquete *Basic.*

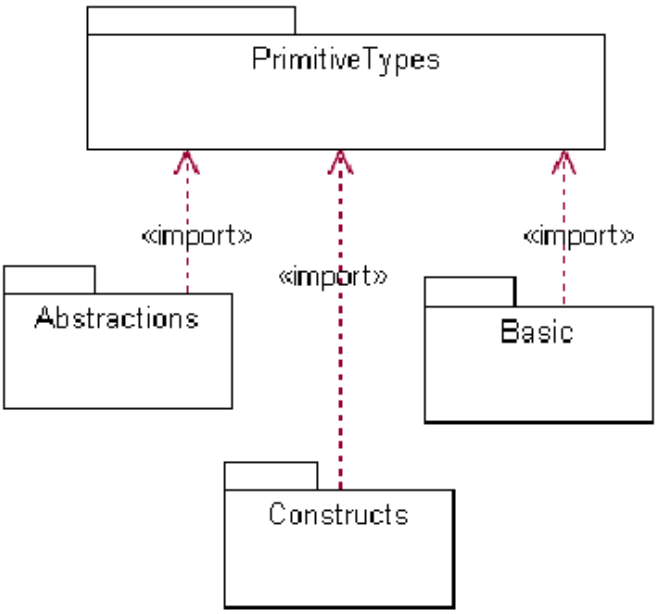

Figura 2-3. Sub-paquetes contenidos en el paquete Core de la Infraestructura

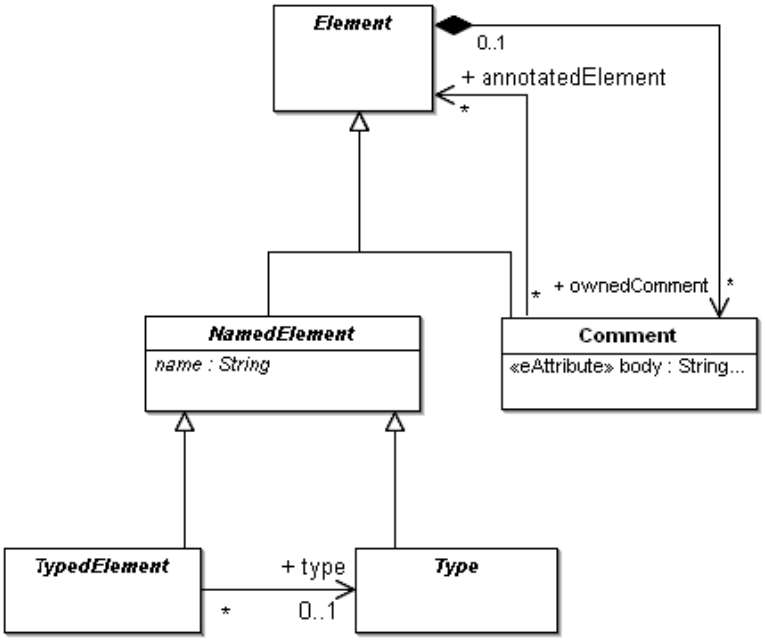

Figura 2-4. Diagrama de Tipos del paquete Basic

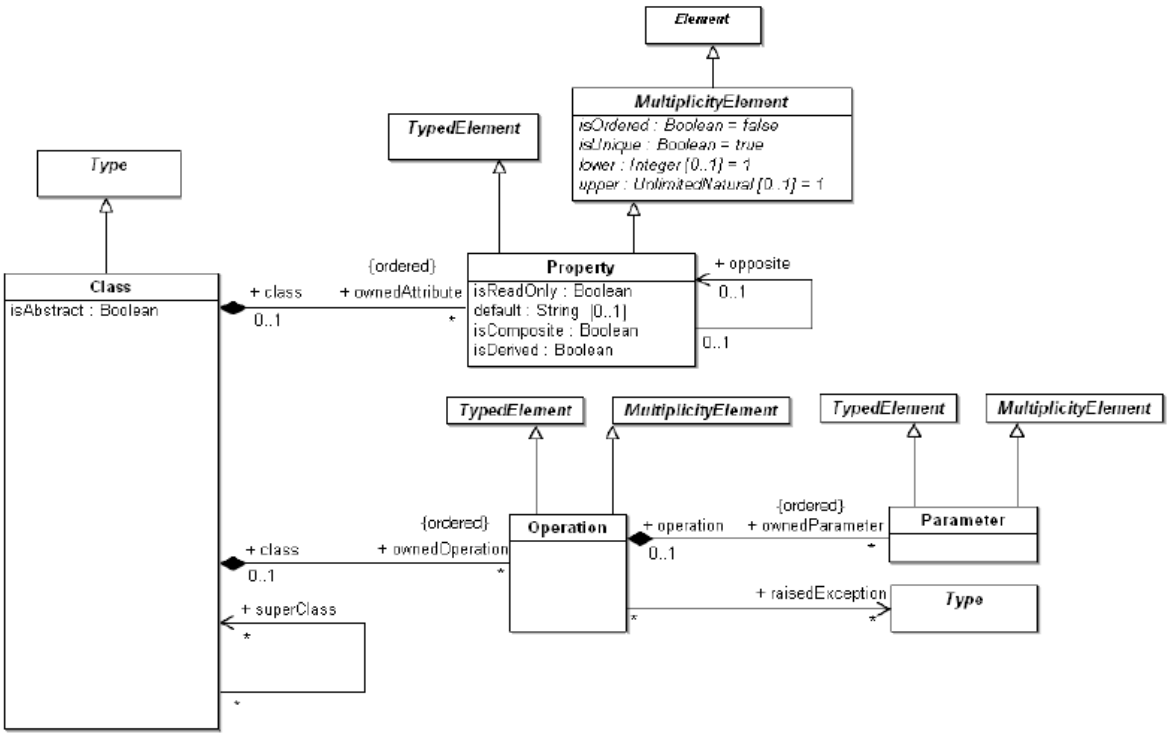

Figura 2-5. Diagrama de Clases del paquete Basic

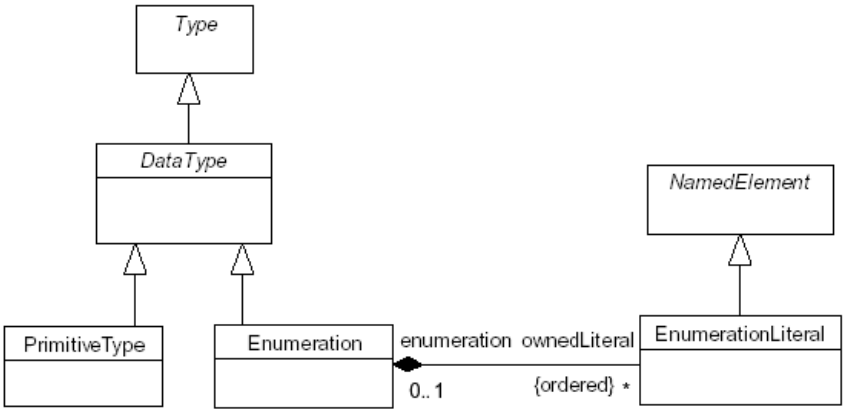

Figura 2-6. Diagrama de Tipos de Datos del paquete Basic

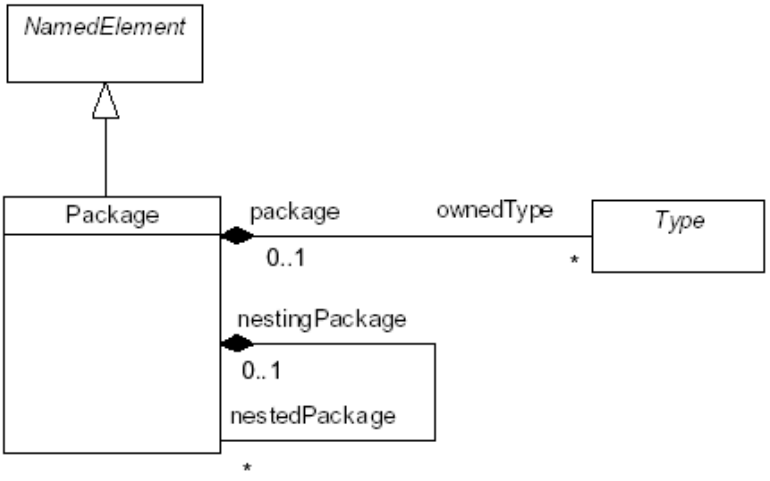

Figura 2-7. Diagrama de Paquetes del paquete Basic

En las próximas secciones se detallará la definición de transformación de modelos y los conceptos relacionados a esta definición, a través de la descripción de los elementos principales del lenguaje para transformaciones QVT.

2.2 ¿Qué es una Transformación?

En el Capítulo 1, las herramientas de transformación se nos presentan como cajas negras que toman un modelo como entrada y producen un modelo como salida. Cuando miramos dentro de una herramienta de transformación, podemos ver qué elementos están involucrados para realizar la transformación. En algún lugar dentro de la herramienta hay una definición que describe cómo un modelo debería ser transformado. A esta definición la llamamos definición de la transformación. La Figura 2-8 muestra la estructura de la herramienta de transformación "abierta".

Aquí vemos que hay una diferencia entre la transformación en si misma, es decir el proceso de generación de un nuevo modelo desde otro, y la definición de la transformación.

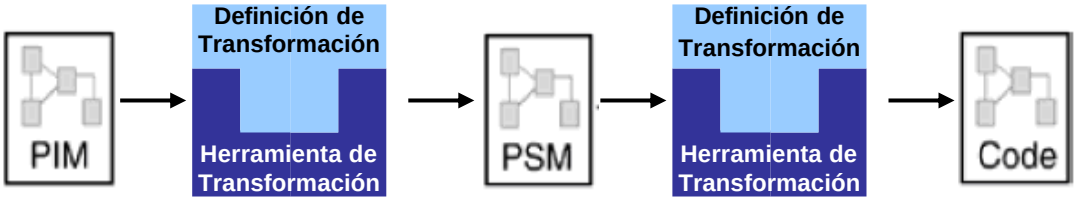

Figura 2-8. Definiciones de transformación dentro de herramientas

La herramienta de transformación usa la misma definición de transformación para cada transformación de cualquier modelo de entrada correspondiente al lenguaje fuente sobre el que se aplique, construyendo un modelo correspondiente al lenguaje destino. Podemos, por ejemplo, especificar una definición de transformación desde UML a Java, la cual describe como debería ser generado Java para un (o cualquier) modelo UML, como muestra la Figura 2-9.

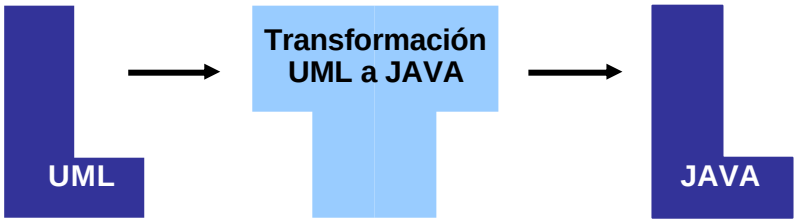

Figura 2-9. Las definiciones de transformación se definen entre lenguajes

En general podemos decir que una definición de transformación consiste de una colección de reglas de transformación, que son especificaciones no-ambiguas de la manera que un modelo (o parte de él) puede ser usado para crear otro modelo (o parte de él). Basándonos en estas observaciones, podemos definir, según Kleppe [4]:

Una transformación es la generación automática de un modelo destino desde un modelo fuente, de acuerdo a una definición de transformación.

Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo un modelo en el lenguaje fuente puede ser transformado en un modelo en el lenguaje destino.

Una regla de transformación es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

2.3 El lenguaje Estándar QVT para Transformación de Modelos

El lenguaje QVT (Query/View/Transformation) es el lenguaje estándar de OMG para transformaciones. Los conceptos, definiciones y ejemplos que siguen, están extraídos del manual de Especificación MOF 2.0 Query/View/Transformation [15]
Los requerimientos impuestos por la OMG para un lenguaje estándar para transformaciones, justifican el acrónimo Q/V/T:

1. *Query*: el lenguaje debe facilitar consultas *ad-hoc* para selección y filtrado de elementos de modelos. Generalmente se seleccionan elementos de modelos que son la fuente de entrada (inputs) de una transformación.

2. *View*: el lenguaje debe permitir la creación de vistas de metamodelos MOF, sobre los que se define la transformación.

3. *Transformation*: el lenguaje debe permitir la definición de transformaciones, a través de relaciones entre un metamodelo MOF fuente F, y un metamodelo MOF destino D, el cual es usado para generar un modelo destino, instancia que conforma a D, desde un modelo fuente, instancia que conforma a F.
Las siguientes subsecciones describen una visión general de QVT y sus componentes principales.

2.3.1 Descripción General de QVT

La especificación de QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles.
Esta especificación define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational, como puede observarse en la Figura 2-10. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios (Figura 2-11):
El paquete QVTBase define estructuras comunes para transformaciones. El paquete QVTRelation usa expresiones de patrones *template* definidas en el paquete QVTTemplateExp. El paquete QVTOperational extiende al QVTRelation, dado que usa el mismo framework para trazas. Usa también las expresiones imperativas definidas en el paquete ImperativeOCL. Todos los paquetes QVT dependen del paquete EssentialOCL de OCL 2.0 [16], y todos los paquetes del lenguaje dependen de EMOF (EssentialMOF) [8].

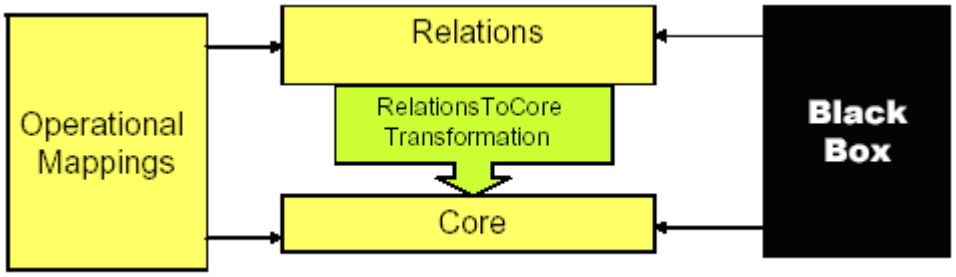

Figura 2-10. Relaciones entre metamodelos QVT

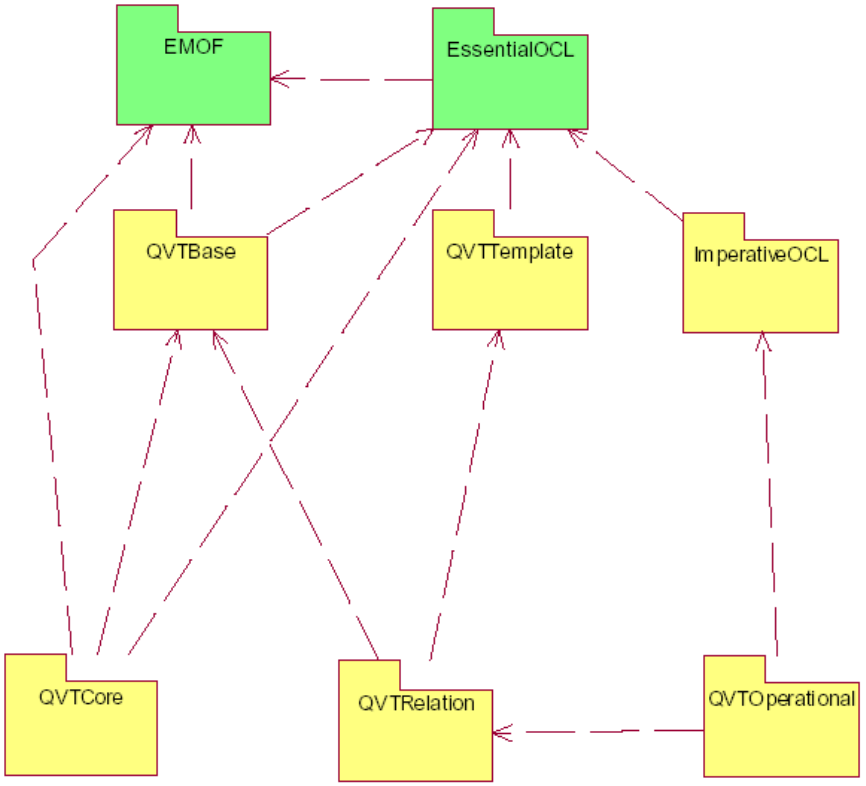

Figura 2-11. Dependencias de paquetes en la especificación QVT

En las siguientes secciones continuamos explicando la arquitectura de la parte declarativa y operacional del lenguaje.

2.3.2 QVT Declarativo

La parte declarativa de QVT está dividida en una arquitectura de dos niveles.
Las capas son:
• un metamodelo y lenguaje *Relations* amigable para el usuario, que soporta *pattern matching* complejo de objetos y creación de *template* para objetos. Las

trazas entre elementos del modelo involucrados en una transformación se crean implícitamente.
• un metamodelo y lenguaje *Core* definido usando extensiones de EMOF y OCL. Las clases traza se definen explícitamente como modelos MOF, y pueden crearse y borrarse como cualquier otro objeto.

❖ Lenguaje Relations

Es una especificación declarativa de las relaciones entre modelos MOF. Las relaciones pueden pedir que otras relaciones también se establezcan entre elementos particulares del modelo, que cumplen con los *pattern matching*.
En este lenguaje, una transformación entre modelos se especifica como un conjunto de relaciones que deben establecerse para que la transformación sea exitosa.
Los modelos tienen nombre, y los elementos que contienen deben ser de tipos correspondientes al metamodelo que referencian.
Un ejemplo:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {…
```

En esta declaración llamada "umlRdbms," hay dos modelos tipados: "uml" y "rdbms". El modelo llamado "uml" declara al paquete SimpleUML como su metamodelo y el modelo "rdbms" declara al paquete SimpleRDBMS como su metamodelo. Una transformación puede ser invocada para *chequear* consistencia entre dos modelos o para modificar un modelo *forzando* consistencia.
Cuando se fuerza consistencia, se elige el modelo destino; puede estar vacío o contener elementos a ser relacionados por la transformación. Los elementos que no existan serán creados para forzar el cumplimiento de la relación.

o Relaciones, Dominios y *Pattern Matching*

Las relaciones en una transformación declaran restricciones que deben satisfacer los elementos de los modelos. Una relación se define por dos o más dominios y un par de predicados *when* y *where*, que deben cumplirse entre los elementos de los modelos. Un dominio es una variable con tipo que puede tener su correspondencia en un modelo de un tipo dado. Un dominio tiene un patrón, que se puede ver como grafo de nodos de objetos. Un patrón se puede ver alternativamente como un conjunto de variables y un conjunto de restricciones que los elementos del modelo asocian a aquellas variables que lo satisfacen (*pattern matching*). Un patrón del dominio se puede considerar un *template* para los objetos y sus propiedades que deben ser *seteadas*, modificadas, o creadas en el modelo para satisfacer la relación. En el ejemplo de abajo, se declaran dos dominios que harán correspondencia entre elementos de los modelos "uml" y "rdbms" respectivamente. Cada dominio especifica un patrón simple: un paquete con un nombre, y un esquema con un nombre, en ambos la propiedad "name" asociada a la misma variable "pn" implica que deben tener el mismo valor:

```
relation PackageToSchema /* map each package to a schema */
{
        domain uml p:Package {name=pn}
        domain rdbms s:Schema {name=pn}
}
relation ClassToTable /* map each persistent class to a table */
{
        domain uml c:Class { namespace = p:Package {}, kind='Persistent', name=cn}
        domain rdbms t:Table { schema = s:Schema {}, name=cn, column = cl:Column {
name=cn+'_tid', type='NUMBER'}, primaryKey = k:PrimaryKey { name=cn+'_pk',
column=cl } }
when { PackageToSchema(p, s);}
where { AttributeToColumn(c, t); }
}
```

Relaciones top-Level

Una transformación contiene dos tipos de relaciones: topLevel y no topLevel. La ejecución de una transformación requiere que se puedan aplicar todas sus relaciones topLevel, mientras que las no topLevel se deben cumplir solamente cuando son invocadas, directamente o a través de una cláusula *where* de otra relación:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
        top relation PackageToSchema {…}
        top relation ClassToTable {…}
        relation AttributeToColumn {…}
}
```

Una relación topLevel tiene la palabra *top* para distinguirla sintácticamente. En el ejemplo de arriba, PackageToSchema y ClassToTable son relaciones topLevel, mientras que AttributeToColumn es una relación no topLevel.

o Claves y Creación de Objetos usando Patrones

Como ya mencionamos, una expresión *object template* provee un *template* para crear un objeto en el modelo destino. Cuando para un patrón válido en el dominio de entrada no existe el correspondiente elemento en la salida, entonces la expresión *object template* es usada como *template* para crear objetos en el modelo destino.
Por ejemplo, cuando ClassToTable se ejecuta con el modelo fuente "rdbms", la siguiente expresión *object template* sirve como *template* para crear objetos en el modelo destino "rdbms":

```
t:Table {
        schema = s:Schema {},
        name = cn,
        column = cl:Column {name=cn+'_tid', type='NUMBER'},
        primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}
```

El *template* asociado con *Table* especifica que un objeto tabla debe ser creado con las propiedades "schema", "name", "column" y "primaryKey" con los valores

especificados en la expresión *template.* Similarmente, los *templates* asociados con *Column, PrimaryKey*, etc, especifican cómo sus respectivos objetos deben ser creados.

- **Sintaxis Abstracta del Lenguaje Relations**

En esta sección presentamos los principales paquetes y algunas de las metaclases de cada uno de ellos que forman la sintaxis abstracta del lenguaje Relations. Para ver la descripción detallada de todas las metaclases, por favor consultar el manual de QVT [15].
La Figura 2-12 muestra el Paquete QVTBase para Transformaciones y Reglas, del cual detallamos la metaclase ***Transformation***:
.

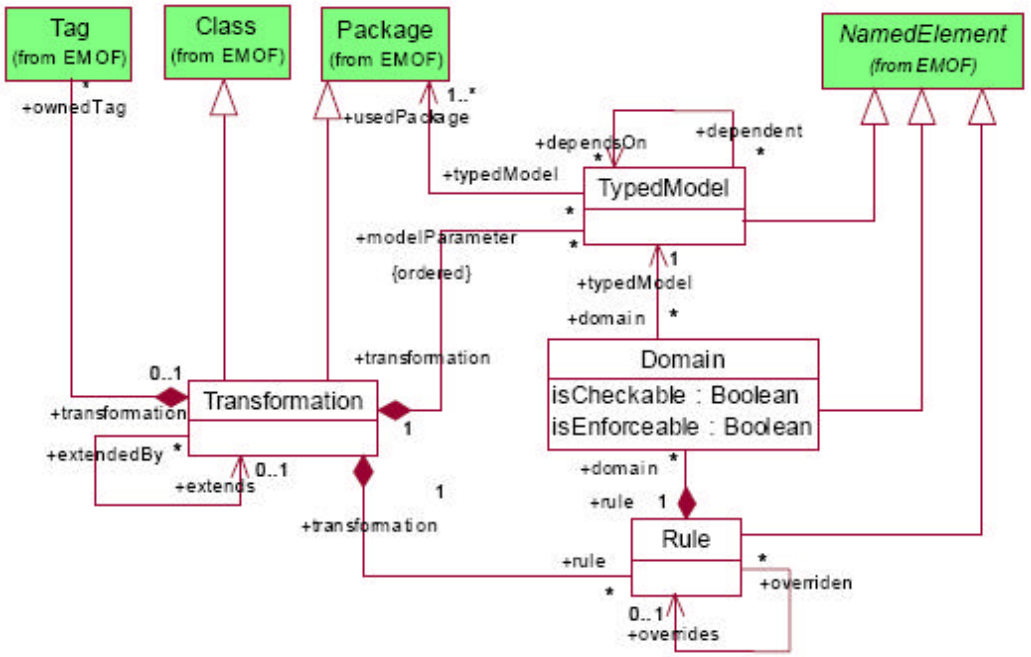

Figura 2-12. Paquete QVT Base – Transformaciones y Reglas.

Una ***transformation*** define cómo un conjunto de modelos puede ser transformado en otro. Contiene un conjunto de reglas (*rules)* que especifican su comportamiento en ejecución. Se ejecuta sobre un conjunto de modelos con tipo, especificados por un conjunto de parámetros (*typed model)* asociados con la transformación.

Superclases

`Package`
`Class`

Asociaciones

`modelParameter: TypedModel [*] {composes}`

El conjunto de modelos tipados que especifican los tipos de modelos que pueden participar en la transformación.

`rule: Rule [*] {composes}`

Las reglas contenidas en la transformación, las cuales juntas especifican el comportamiento en ejecución de la transformación.

`extends: Transformation [0..1]`

Una transformación puede ser extendida por otra. Las reglas de la transformación extendida están incluidas en la transformación que la extiende, para especificar el comportamiento en ejecución que queda en última instancia. La extensión es transitiva.

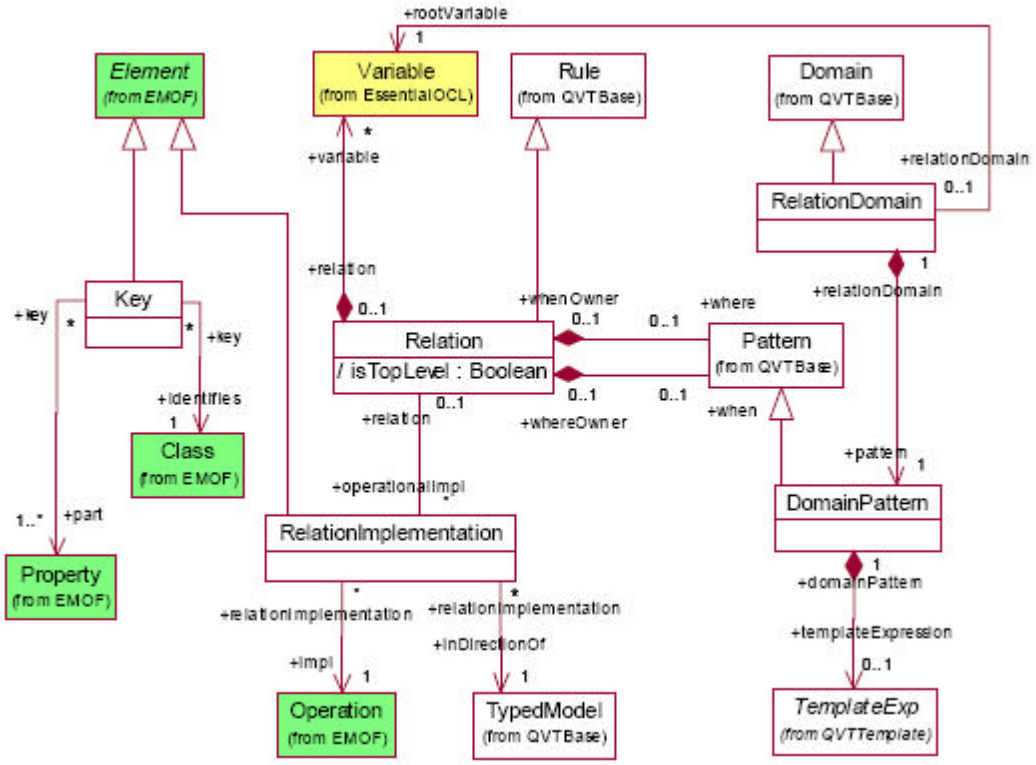

Figura 2-13. Paquete QVT Relations

La Figura 2-13 muestra el Paquete QVTRelations, del cual detallamos la metaclase ***Relation***:

Una ***relation*** es la unidad básica de especificación del comportamiento de la transformación en el lenguaje Relations. Se define por dos o más dominios que especifican los elementos de modelos a relacionar. Su cláusula **when** especifica las condiciones necesarias para que la relación se establezca, y su cláusula **where** especifica la condición que debe satisfacerse por los elementos del modelo que están siendo relacionados.

Superclases

`Rule`

Atributos

`/isTopLevel : Boolean`

Indica cuando la relación es top-level o cuando es una relación no top-level.

Asociaciones

`variable: Variable [*] {composes}`

El conjunto de variables en la relación. Este conjunto incluye todas las ocurrencias de variables en sus dominios y sus cláusulas **when** y **where**.

`/domain: Domain [*] {composes} (from Rule)`

El conjunto de dominios contenidos en la relación que especifican los elementos de modelo que participan de la relación. `Relation` hereda esta asociación de `Rule`, y está restringida a contener solamente `RelationDomains` vía esta asociación.

`when: Pattern [0..1] {composes}`

El patrón (como un conjunto de variables y predicados) que especifica la cláusula **when** de la relación.

`where: Pattern [0..1] {composes}`

El patrón (como un conjunto de variables y predicados) que especifica la cláusula **where** de la relación.

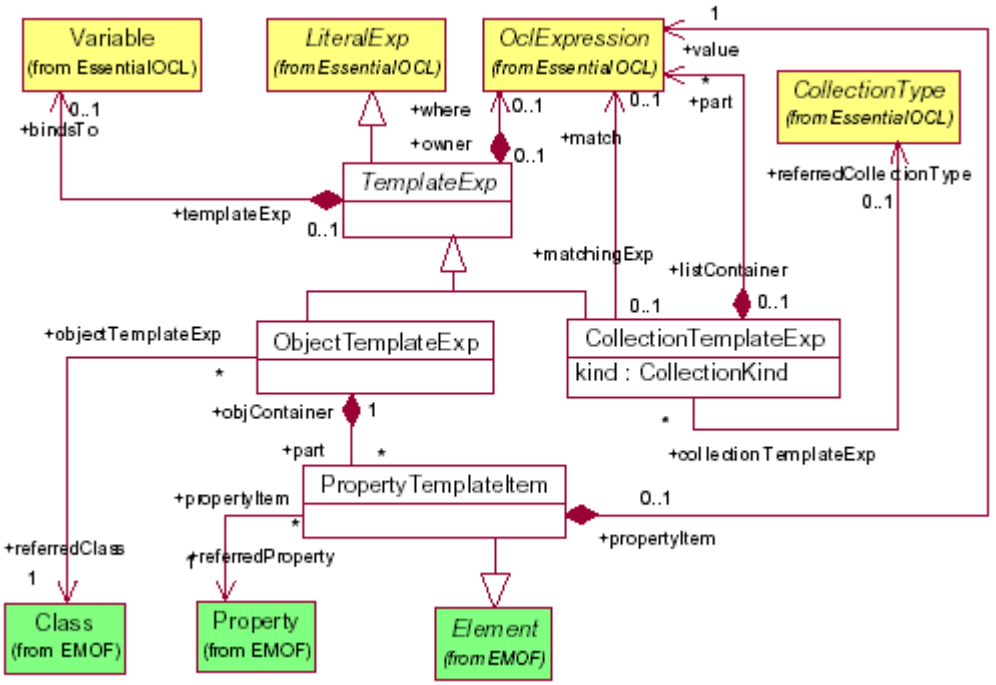

Figura 2-14. Paquete QVT Template

Finalmente, la Figura 2-14 presenta el Paquete QVT Template, del cual detallamos la metaclase ***TemplateExp***:

Una ***TemplateExp*** especifica un patrón de correspondencia con elementos de modelos definidos en una transformación. Los elementos del modelo pueden ligarse a variables y esta variable puede ser usada en otras partes de la expresión. Una *expresión template* puede corresponder tanto a elementos simples como a una colección de ellos, dependiendo si es una expresión *object template* (metaclase ObjectTemplateExp) o una expresión *collection template* (metaclase CollectionTemplateExp), las cuales son subclases de *TemplateExp*.

Superclases
LiteralExp
Associaciones
`bindsTo: Variable [0..1] {composes}`
La variable que refiere al elemento del modelo *macheado* por su expresión *template.*
`where: OclExpression [0..1] {composes}`
Expresión booleana que debe ser verdadera para que se liguen los elementos mediante la expresion *template.*

2.3.3 QVT Operacional

Además de sus lenguajes declarativos, QVT proporciona dos mecanismos para implementaciones de transformaciones: un lenguaje estándar, Operational Mappings, e implementaciones no-estandar *Black-box*. El lenguaje Operational Mappings se detallará en el apartado siguiente.
El mecanismo caja negra (*black-box)*, permite implementaciones **opacas** (escritas en otro lenguaje existente) de parte de transformaciones. Esto puede ser peligroso si la implementación tiene acceso a referencias de objetos en los modelos, pudiendo manipularlos arbitrariamente. Una implementación caja negra no tiene relación implícita con el lenguaje Relations, pero cada caja negra debería implementar una Relation, la cual es responsable de mantener las trazas entre los elementos relacionados.

❖ Lenguaje Operational Mappings

Este lenguaje se especificó como una forma estándar para proveer implementaciones imperativas. Proporciona extensiones OCL con efectos laterales que permiten un estilo más procedural, y una sintaxis que resulta familiar a los programadores.
Las operaciones *Mappings* pueden ser usadas para implementar una o más relaciones de una especificación del lenguaje *Relations.* Una transformación escrita usando solamente operaciones *Mapping* es llamada transformación Operacional.

o Sintaxis Abstracta del Lenguaje Operational Mappings

En esta sección presentamos los principales paquetes y algunas de las metaclases de cada uno de ellos que forman la sintaxis abstracta del lenguaje **Operational Mappings**. Similarmente al lenguaje Relations, para ver la descripción detallada de todas las metaclases, por favor consultar el manual de QVT [15].

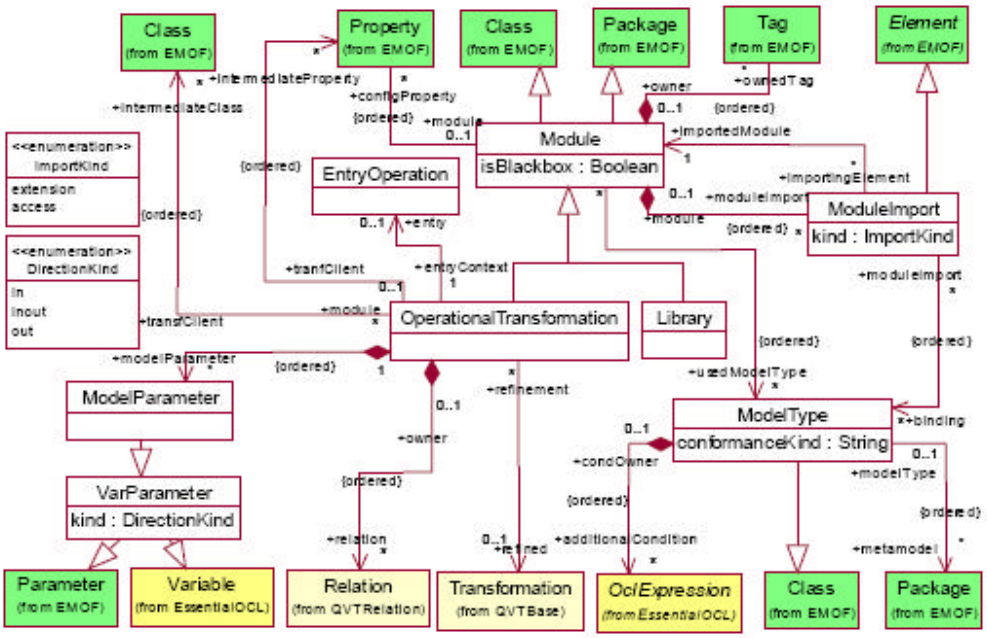

Figura 2-15. Paquete QVT Operational – Transformaciones Operacionales

La Figura 2-15 muestra el Paquete QVT Operational para Transformaciones Operacionales, del cual detallamos la metaclase ***OperationalTransformation***:

Una ***OperationalTransformation*** representa la definición de una transformación unidireccional, expresada imperativamente. Tiene una signatura indicando los modelos involucrados en la transformación y define una operación *entry*, llamada **main**, la cual representa el código inicial a ser ejecutado para realizar la transformación. La signatura es obligatoria, pero no así su implementación. Esto permite implementaciones *caja negra* definidas fuera de QVT.

Superclases

`Module`

Atributos

`/isBlackbox : Boolean (from Module)`

Indica que la transformación entera es opaca: no tiene operación de entrada ni operaciones *mapping*.

`/isAbstract : Boolean (from Class)`

Indica que la transformación sirve para la definición de otras transformaciones.

Asociaciones

`enforcedDirection : ModelParameter [0..1]`

Indica la dirección forzada de una transformación relacional refinada.

`entry : EntryOperation [0..1]`

Una operación actuando como punto de entrada para la ejecución de la transformación operacional. Es opcional porque una transformación operacional puede servir como base de otra transformación operacional.

`modelParameter: ModelParameter [*] {composes, ordered}`

Indica la signatura de esta transformación operacional. Un parámetro de modelo indica un tipo de dirección (in/out/inout) y un tipo dado por un tipo de modelo (ver la descripción de clase ModelParameter).

`refined : Transformation [0..1]`

Indica una transformación relacional (declarativa) que es refinada por esta transformación operacional.

`relation : Relation [0..*] {composes, ordered}`

El conjunto ordenado de definiciones de relaciones que tienen que están asociados con las operaciones *mapping* de esta transformación operacional.

La Figura 2-16 presenta la jerarquía de las Operaciones Imperativas del Paquete QVT Operational, de la que detallamos la metaclase ***MappingOperation***.

Una ***MappingOperation*** es una operación implementando un *mapping* entre uno o más elementos del modelo fuente, en uno o más elementos del modelo destino.

Puede tener sólo signatura o bien ser provisto de la definición de un cuerpo imperativo. En el primer caso, la operación es *caja negra.*

Una *mapping operation* siempre refina una relación, donde cada dominio se corresponde con un parámetro del *mapping.*

El cuerpo de una operación *mapping* se estructura en tres secciones opcionales.

La sección de inicialización es usada para código previo a instanciaciones de los elementos de salida. La intermedia, sirve para setear elementos de salida y la de finalización, para definir código que se ejecute antes de salir del cuerpo.

Las facilidades de reuso y composición entre operaciones *mapping,* se detallarán en el Capítulo 5.

Superclases

`ImperativeOperation`

Atributos

`isBlackbox: Boolean`

Indica si el cuerpo de la operación está disponible. Si *isBlackbox* es verdadero esto significa que la definición debería ser proporcionada externamente a fin de que la transformación pueda ser ejecutada.

Asociaciones

`inherited: MappingOperation [*]`

Indica la lista de operaciones *mappings* que son especializados.

`merged: MappingOperation [*]`

Indica la lista de operaciones *mappings* que son mezclados.

`disjunct: MappingOperation [*]`

Indica la lista de operaciones *mappings* potenciales para invocar

`refinedRelation: Relation [1]`

Indica la relación refinada. La relación define la guarda (*when*) y la poscondición para la operación *mapping*.

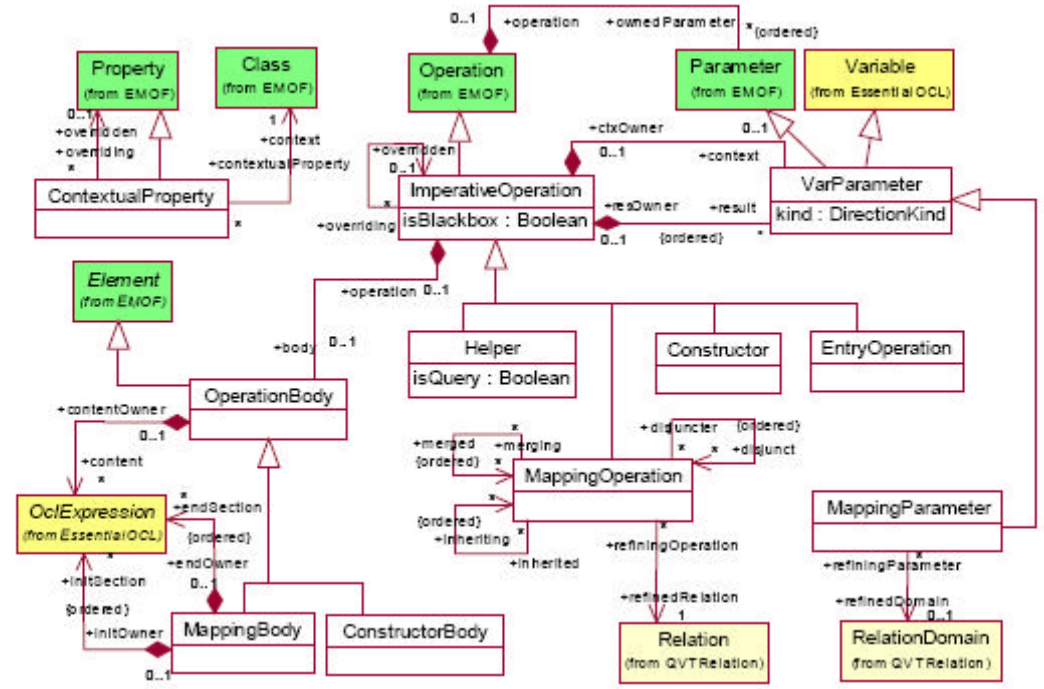

Figura 2-16. Paquete QVT Operational – Operaciones Imperativas

Finalmente, la Figura 2-17 muestra el Paquete OCL Imperativo que extiende a OCL proveyendo expresiones imperativas y manteniendo las ventajas de la expresividad de OCL.

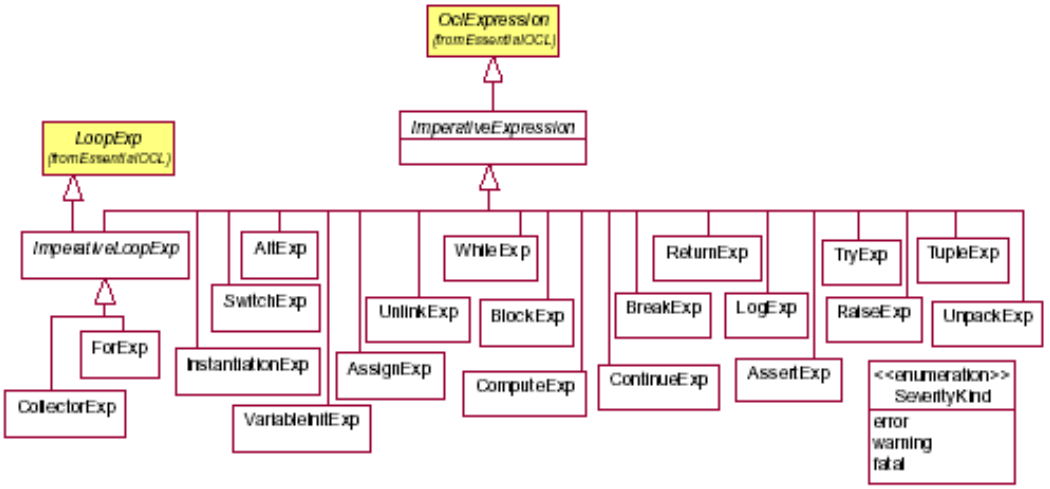

Figura 2-17. Paquete OCL Imperativo

ImperativeExpression es la raíz abstracta de esta jerarquía sirviendo como base para la definición de todas las expresiones con efectos laterales definidas en esta especificación. Tales expresiones son AssignExp, WhileExp, IfExp, entre otras. Podemos notar que en contraste con las expresiones OCL puras, libres de efectos laterales, las expresiones imperativas en general, no son funciones. Por ejemplo,

son constructores de interrupción de ejecución como break, continue, raise y return que producen un efecto en el flujo de control de las expresiones imperativas que las contienen.
La **Superclase** de *ImperativeExpression* es `OclExpression`.

2.4 Un Ejemplo de Transformación de Modelos

Para un mejor entendimiento del concepto de transformación de modelos, presentamos un ejemplo extraído del manual de QVT. Este ejemplo muestra la especificación textual de una transformación llamada UMLToRdbms que define una relación top-level llamada Class2Table que transforma clases UML (que cumplan la restricción de ser persistentes, indicado en la cláusula **when** de esta relación) a tablas del Modelo Relacional con el mismo nombre. Además la transformación tiene otra relación Attr2Col, la cual especifica que los atributos (no multivaluados con tipo de datos básico) de la clase, se corresponden con columnas del mismo nombre y tipo en la tabla correspondiente. Esta relación es invocada en la cláusula **where** de la relación Class2Table y significa que cada vez que Class2Table se cumple para una clase y una tabla, la relación Attr2Col para sus atributos y columnas respectivamente, se cumple también:

```
Transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS)
{
        Top Relation Class2Table {
        checkonly domain Uml c: Class {name = n }
        checkonly domain Rel t: Table {name = n }
        when { isPersistent = true }
        where { Attr2Col (c, t) } }
Relation Attr2Col {
        checkonly domain Uml c: Class {
        attribute = a: Attribute {name = an, type = p: DataType {name = dt}}}
        checkonly domain Rel t: Table {column: col:Column
        {name = an, type.name = dt } }
        when { not a.isMultivalued() }
        where { col.type = a.type } }
}
```

Gráficamente, una instanciación de esta transformación podría ser, por ejemplo, la que se muestra en la Figura 2-18 entre dos modelos concretos Uml1 (de UML) y Rel1 (de Rdbms), donde n = 'Persona' y un nombre de atributo podria ser 'nombre'. O sea, para la clase Persona del modelo Uml1, existe una Tabla en el modelo relacional Rel1 con el mismo nombre. Lo mismo sucede con los atributos: para cada atributo de la clase Persona existe una columna en la tabla Persona con el mismo nombre. O sea, para el atributo `nombre´, existe la columna `nombre´, ambos de tipo String.

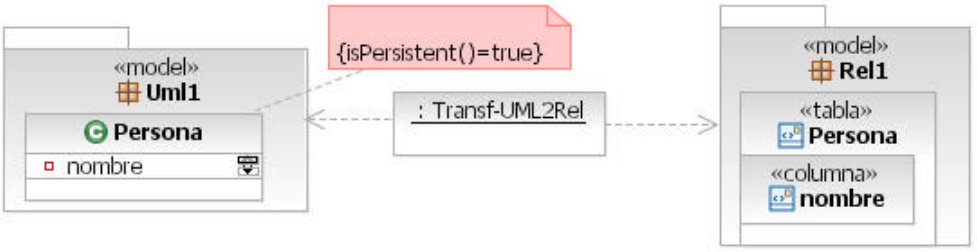

Figura 2-18. Una instanciación gráfica de la Transformación

2.5 Conclusión del Capítulo

En este capítulo hemos presentado la técnica del metamodelado, utilizada para definir la sintaxis de lenguajes. Seguidamente introducimos los conceptos básicos para transformación de modelos a través de la descripción del lenguaje estándar QVT para transformaciones, tanto en su parte declarativa como operacional. Mostramos en ambos niveles la sintaxis abstracta, o sea un metamodelo que lo sustenta. Otros lenguajes híbridos definidos a partir de los requerimientos pedidos en el estándar QVT, como ATL [17], Kent [18] y TefKat [19][3], (algunos de ellos ya cuentan con herramientas Case probadas), definen a su vez su propia sintaxis, obligando al usuario a aprender nuevos lenguajes. Además, al igual que sucede con QVT, carecen de una definición precisa de su semántica, tanto en su definición declarativa como en la operacional.
Consecuentemente, en el Capítulo 3 presentamos un lenguaje inspirado en QVT llamado SQVT (Simple/Query/View/Transformation) que incluye y adapta los conceptos mínimos para poder expresar transformaciones de modelos. Definiremos la sintaxis de SQVT extendiendo y utilizando especificaciones estándares ya existentes de OMG, lo cual facilitará su uso y aplicabilidad. Posteriormente, en el Capítulo 4 presentaremos una definición formal de la semántica de SQVT.

[3] Consultar **Glosario de siglas y términos** para obtener una descripción breve de estos lenguajes.

Capítulo
3

"SQVT": Una Simplificación de QVT basada en Estandares de OMG

En este capitulo presentamos el lenguaje SQVT, específicamente su sintaxis, tanto declarativa como operacional, para expresar transformaciones entre modelos. SQVT inspira inicialmente, la definición de sus metamodelos en el lenguaje QVT, estándar de OMG, presentado en el capítulo anterior. La idea de definir SQVT es presentar una propuesta amigable simplificando a QVT para facilitar su uso. Además, utilizando este lenguaje minimal, en el próximo capítulo, definiremos una semántica precisa de lenguajes de transformación de modelos.

3.1 SQVT: Un Lenguaje Simple para Transformaciones

El lenguaje que proponemos, llamado SQVT (Simple/Query/View/ Transformation), está orientado a ser el mínimo para poder expresar relaciones y *queries* de transformación entre modelos.
Sería deseable que los modeladores no tengan que aprender un nuevo lenguaje para crear transformaciones. Lo ideal es que se pudiesen utilizar los mismos lenguajes de modelado estándar, por ejemplo MOF y/o UML. Para solucionar este problema, SQVT está definido como una extensión de especificaciones estándares de OMG, complementada con el uso del lenguaje OCL. El lenguaje SQVT Relational (declarativo), es también visual para hacer más amigable su uso.

❖ Introducción informal de la propuesta

En el ejemplo de la sección 2.4 del Capítulo 2, se pueden observar ciertas cuestiones que generalmente ocurren en especificaciones escritas en QVT. Estas cuestiones pueden considerarse desventajas que dificultan el uso y entendimiento del lenguaje.
Algunas de ellas son:

- Las relaciones de transformación se expresan mediante expresiones *template* en cada uno de los dominios, entonces para concretar la transformación, se debe poducir un *pattern matching* entre estas expresiones. Además, las variables de los dominios pueden definirse recursivamente, por lo que el anidamiento de expresiones *template* y los *pattern matching* entre ellas puede propagarse generando especificaciones complicadas de entender.

- Los dominios aparecen precedidos por la palabra *checkonly*, lo cual indica que sólo se chequea consistencia entre modelos, no permitiendo la creación de nuevos elementos. Al respecto, considerando que en la práctica el proceso de transformación implica la generación de modelos,

sugerimos no utilizar los atributos para dominio definidos en QVT para permitir (*enforced*) o impedir (*checkonly*) la creación o eliminación de elementos (ver Capítulo 2, sección 2.3.2). Proponemos admitir, en general, la modificación de los modelos cuando sea necesario, para poder cumplir con la transformación.

- La relación Attr2Col sugiere una iteración sobre todos los atributos de la clase, que es pasada como parámetro, pero no lo indica explícitamente. Para denotar esto podemos usar el lenguaje OCL, quitándole ambigüedad.

- Otra desventaja de esta notación, puede verse en la relación Attr2Col, que transforma atributos en columnas, pero es invocada con parámetros Clase y Tabla. Resultaría más claro que cada relación tenga como parámetros los elementos que realmente participan de la transformación.

Consecuentemente, proponemos una nueva forma de expresar la misma transformación, teniendo en cuenta las desventajas mencionadas anteriormente, haciendo la notación más legible, siguiendo una sintaxis más amigable y maximizando el uso de OCL.

❖ Nuestra propuesta – Redefiniendo el ejemplo

En nuestra propuesta, expresamos en cada dominio sólo las variables que participan, dejando que las condiciones que se deben cumplir entre ellas, se especifiquen en la cláusula *where* de la relación. Es decir, ya no usamos expresiones *template* ni *pattern matching*. Tampoco usamos la palabra *checkonly* en los dominios, permitiendo así la creación de elementos, como fue discutido en el apartado anterior. Además, en la relación Class2Table expresamos en OCL explícitamente la iteración sobre los atributos de la clase e invocamos a la relación Attr2Col con los parámetros adecuados: Atributo y Columna. Entonces esta relación se define con las variables de dominio y codominio correspondientes: Atributos y Columnas respectivamente. Las condiciones previas que el atributo debe cumplir, van en la cláusula *when*, las de salida se incluyen en la cláusula *where* de la relación. El ejemplo anterior, escrito en la notación propuesta quedaría:

```
Transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS)  {

    Top Relation Class2Table {
    domain Uml c: Class
    domain Rel t: Table
    when { c.isPersistent = true }
    where { c.name = t.name and c.allAttribute-> forAll (a |
                            ( t.column -> exists (co | Attr2Col (a, co) )  }
    }

    Relation  Attr2Col {
```

```
        domain Uml a: Attribute
        domain Rel co: Column
        when { not a.isMultivalued() and a.type.oclIsKindOf ( DataType ) }
        where { co.type = a.type and co.name = a.name }
        }
}
```

En la sección siguiente se construye un perfil de la Infraestructura de UML [20] que provee una base formal a la notación propuesta. Dicho perfil está basado en la definición de nuevos estereotipos y maximiza el uso de OCL para evitar la creación de nuevos elementos. La definición de estereotipos y *constraints* es el mecanismo de extensión estándar de UML, el cual promueve estereotipar metaclases existentes en lugar de definir nuevas metaclases. A su vez, es recomendable minimizar la cantidad de nuevos estereotipos para mantener la simplicidad y reducir el tiempo de entrenamiento por parte de los usuarios, fomentando así su uso.

3.2 Definición de SQVT

En esta sección definimos el lenguaje para transformación de modelos SQVT. Previamente a ello, analizamos cómo y donde definirlo respecto a la Arquitectura 4 capas de Modelado descripta en el Capítulo 2.

En la capa M3 de esta Arquitectura, como ya dijimos, se ubica MOF, que representa un Meta-metamodelo cerrado sobre el que se instancian metamodelos. En consecuencia, el metamodelo para Transformaciones debería ubicarse en la capa M2, junto con el resto de los metamodelos (por ejemplo el metamodelo de UML, el de OCL, etc.).

Sin embargo, una instancia específica de transformación se debe ubicar en la capa M2 para poder relacionar metamodelos concretos (que se ubican en la capa M2) como el de UML y el de RDBMS, entre cuyas instancias (modelos tipados) se produce la transformación (por ejemplo una Class de UML y una Table de RDBMS). Es decir, los modelos que concretamente están involucrados en la transformación (capa M1) son parámetros para el lenguaje de transformación.

Por lo tanto, la definición del metamodelo para transformaciones puede pensarse en la capa M3 de esta Arquitectura ya que resulta lógico pensar que el metamodelo y su instanciación no pueden convivir en la misma capa, dado que representan distintos niveles de abstracción.

Siguiendo este razonamiento, si definimos el nuevo lenguaje para transformaciones como una extensión de elementos del paquete Core de la Infraestructura, su metamodelo se ubicará en el nivel M3, junto a la Infraestructura (ver Figura 3-1). Esto es posible dado que, como vimos en el Capítulo 2, MOF usa elementos del paquete Core para su definición, situación que nos permite identificar a la Infraestructura como un meta-metamodelo. Luego, una instancia del metamodelo para Transformaciones en la capa M2 relaciona elementos de metamodelos, mientras que en la capa M1 existirá un enlace de correspondencia entre modelos, instancias de dichos metamodelos, que son los que concretamente se transforman.

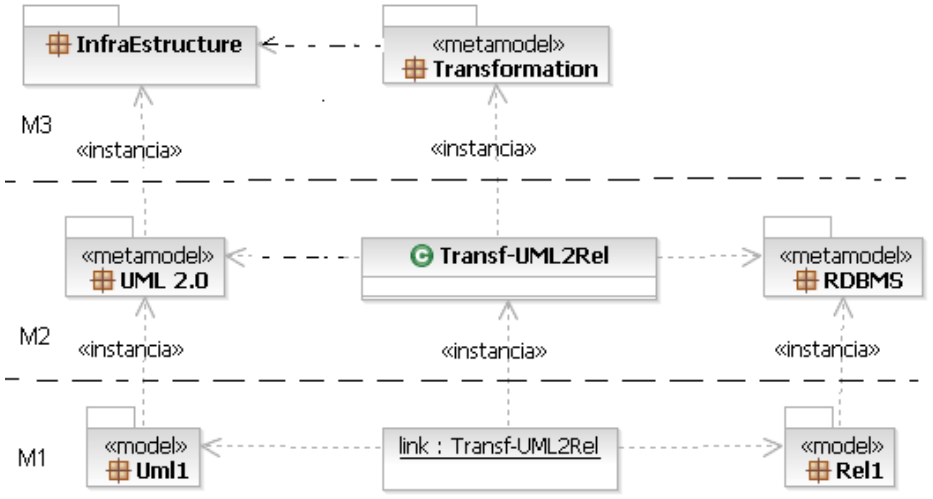

Figura 3-1. La Transformación de Modelos en la Arquitectura 4 capas

3.2.1 Sintaxis de SQVT Basada en OCL y definida como Extensión de la Infraestructura

En el documento de especificación de QVT, que es la propuesta estándar de OMG para transformaciones, el metamodelo se define como "extensión de MOF y de OCL [16]". Desde nuestro punto de vista, según lo analizado anteriormente y según las definiciones de OMG no es técnicamente correcto extender a MOF ya que representa un Meta-metamodelo cerrado sobre el que se instancian metamodelos. Por lo tanto, proponemos extender la Infraestructura 2.0 (definir un *profile*) y usar OCL 2.0 para implementar el metamodelo de las transformaciones.

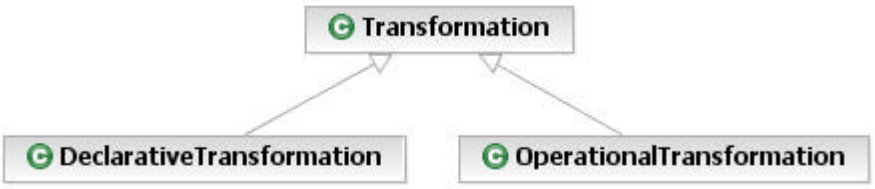

Figura 3-2. La jerarquía Transformation de SQVT

Nuestro lenguaje minimal para transformaciones, cuyo metamodelo inicial se inspira en el paquete Relations de QVT en su parte declarativa y en el paquete Operational de QVT en su parte imperativa, tendrá una raíz común a ambos metamodelos, la metaclase Transformation, con dos subclases, DeclarativeTransformation y OperationalTransformation como muestra la Figura 3-2. A su vez, cada una de ellas será raíz del metamodelo declarativo y operacional respectivamente.

3.3 Construcción del Perfil para SQVT Declarativo

Debido a lo expuesto en la sección anterior, nuestra propuesta es extender la Infraestructura, creando un perfil específico (instancia de la Clase Profile de la Infraestructura) que permita expresar declarativamente características particulares de la transformación de modelos.
La extensión se define en las siguientes subsecciones en cinco etapas, como se sugiere en otros trabajos que definen perfiles [21, 22, 23, 24].

Primera Etapa: Definición del metamodelo propuesto

El metamodelo que proponemos (ver Figura 3-3) minimiza al metamodelo del paquete Relations de QVT presentado en el capítulo 2, basándose en la sintaxis abstracta de dicho paquete con algunas adaptaciones con el propósito de simplificar su uso y entendimiento. Fueron suprimidas algunas clases abstractas (como Rule, RelationDomain, Pattern, PatternDomain, etc.) de la sintaxis abstracta de Relations QVT y agregadas otras necesarias (como Query y Helper) que QVT menciona pero no incluye al definir la sintaxis abstracta.
En nuestro metamodelo, un *helper* dentro de una relación, define operaciones adicionales para realizar navegación sobre los modelos que participan de la transformación. Los helpers pueden tener parámetros de entrada y pueden usar recursión. Se componen de declaraciones de variables y de expresiones que especifican dichas operaciones adicionales de consulta/navegación. La aplicación de un *helper* no produce efectos laterales.
Los dominios ya no se restringen con los atributos *isCheckeable* e *isEnforceable*.
Otra modificación al metamodelo QVT, es respecto a la metaclase *TemplateExp*. Como vimos en el capitulo anterior (Figura 2-14), representa expresiones complejas con el propósito de definir arbitrariamente *templates* de objetos y/o colecciones para *pattern matching* e instanciación. En el documento QVT, la jerarquía TemplateExp está especificada por nuevas metaclases agregadas al metamodelo OCL.
En SQVT proponemos separar la definición de variables, que puede ser anidada, de las expresiones usadas para *pattern matching* en los modelos que se incluirán en la cláusula *where* de la relación.
Específicamente, para representar definiciones de variables en dominio y codominio, proponemos usar la metaclase VariableExp, una OCLExpression que tiene una referencia a una variable tipada (VariableDeclaration), a través de la asociación referredVariable (ver Figura 3-3). Así, reutilizando metaclases OCL, ya no es necesario definir nuevas metaclases ni nuevos estereotipos para el *profile* propuesto en cuanto a expresiones *template*.
Por su parte, los *queries* especifican transformaciones entre instancias, al igual que las relaciones, pero con un formato más simple. La idea es que son funciones libres de efectos laterales, con parámetros formales y una OCLExpression como cuerpo.

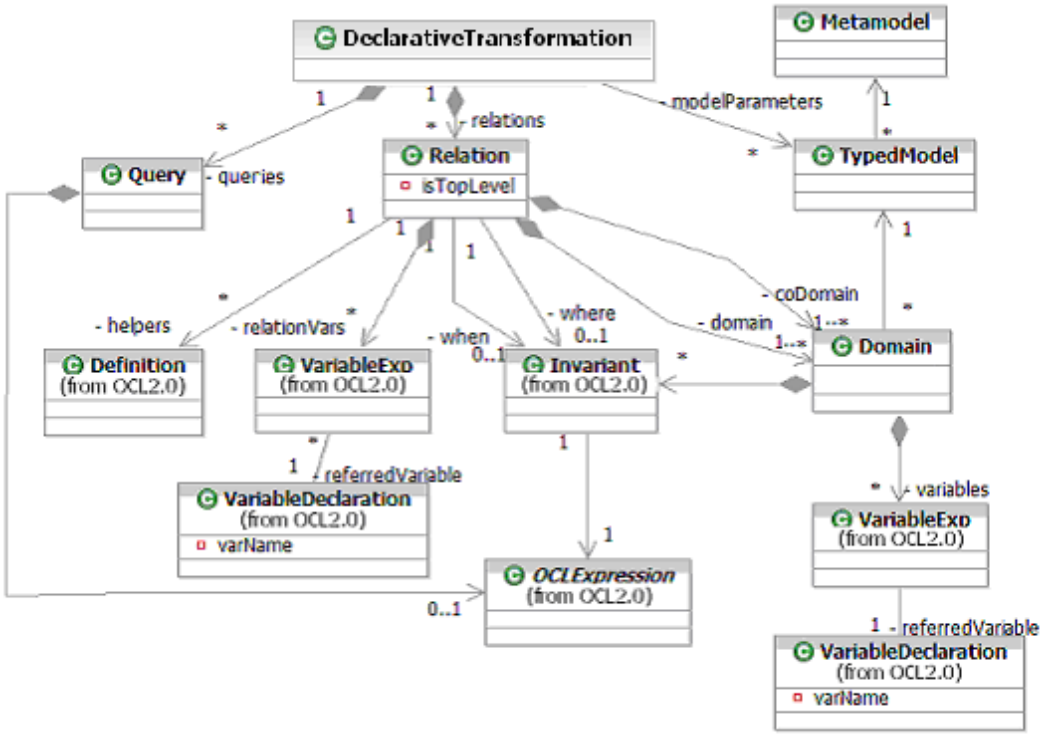

Figura 3-3. Metamodelo SQVT Declarativo

Segunda Etapa: Definición de los estereotipos correspondientes al metamodelo propuesto

Por cada metaclase y relaciones entre metaclases del paso anterior, definimos un estereotipo nuevo. Los estereotipos en la Infraestructura a diferencia que en UML, son simplemente clases y los llamados taggedValues de UML son Property (ownedAttribute). La Relationship entre el estereotipo y la metaclase que es su base se denomina Extension y es subclase de Association.
Notacionalmente no hay diferencia para describir estereotipos en la Infraestructura y en UML. La próxima subsección muestra a los nuevos estereotipos con sus bases.
En cuanto a las asociaciones entre las metaclases propuestas, hemos decidido no estereotiparlas para evitar usar notación excesiva al instanciar el perfil. Para no perder información, decidimos añadir restricciones al definir los conceptos que se asocian.

Tercera Etapa: Identificación de las bases para los estereotipos.

En esta subsección se identifican las metaclases de la Infraestructura que pueden ser extendidas para representar el metamodelo propuesto.
Según este metamodelo, una *DeclarativeTransformation* podría definirse como una Class formada por los modelos que transforma y por las relaciones y *queries* definidos entre ellos. Una *Relation* puede también representarse extendiendo la metaclase Class, contiene dominios, condiciones que la restringen y *helpers*.

Domain agrega propiedades a los modelos y del mismo modo que las anteriores, puede extender a Class.
El estereotipo Query, puede tener base en Operation. Su atributo isQuery ya tiene representación en Operation; isQuery es un metaAtributo booleano de Operation.
En general, los modelos se pueden representar mediante paquetes. Por lo tanto, TypedModel y Metamodel se definen con base en Package ya que especifican respectivamente a los modelos que participan en la transformación y a los metamodelos correspondientes de los cuales son instancias cada uno de ellos.
Para la relación entre Transformation y Query (que tiene base en *Operation*), el dueño de la operación <<Query>> debe ser una clase <<Transformation>>.
Finalmente, resta definir bases para las expresiones (*helpers*, *when y where*) que restringen una *relation* de nuestro metamodelo. En los lenguajes de modelado basados en MOF, en muchos de los casos que se utiliza el término *expression*, puede usarse una expresión escrita en OCL. El significado de la evaluación de la expresión OCL depende de su contexto y del objeto al que referencia dentro del modelo. Dado que en la sintaxis abstracta de OCL una OCLExpression se define recursivamente, necesitamos una nueva metaclase para representar la raíz del árbol sintáctico abstracto que representa una OCLExpression. Esta metaclase es llamada ExpressionInOCL. En el metamodelo OCL, una ExpresionInOCL aparece como subclase de Expression, pero no para todas las Expression se permite definir una especificación (*body*) en un determinado lenguaje. Por esto creemos que ExpressionInOCL debería ser subclase de OpaqueExp de la Infraestructura, ya que ésta si tiene un atributo *body* (que sería la OCLExpression) y un atributo *language* el cual debería tener para este caso el valor “OCL”. Un estereotipo ya existente para una ExpressionInOCL es <<definition>>, que restringe a la ExpressionInOCL indicando que debe estar definida en el contexto de un Classifier y puede definir operaciones adicionales y contener expresiones Let. Esto hace a una espresión OCL <<definition>> apropiada para modelar *helpers*.
Respecto a las cláusulas *when* y *where*, por ser expresiones Booleanas, pueden representarse especializando al estereotipo existente <<invariant>> (que extiende a la metaclase ExpressionInOCL). Una ExpressionInOCL <<invariant>> es una expresión cuyo cuerpo es una OCLExpression Booleana restringiendo a un Classifier. En nuestro metamodelo, estas cláusulas restringen una clase <<relation>> y pueden referenciar a otras relaciones y helpers.
La Figura 3-4 muestra los estereotipos definidos con su metaclase extendida, conformando así el perfil DeclarativeModelTransformation.

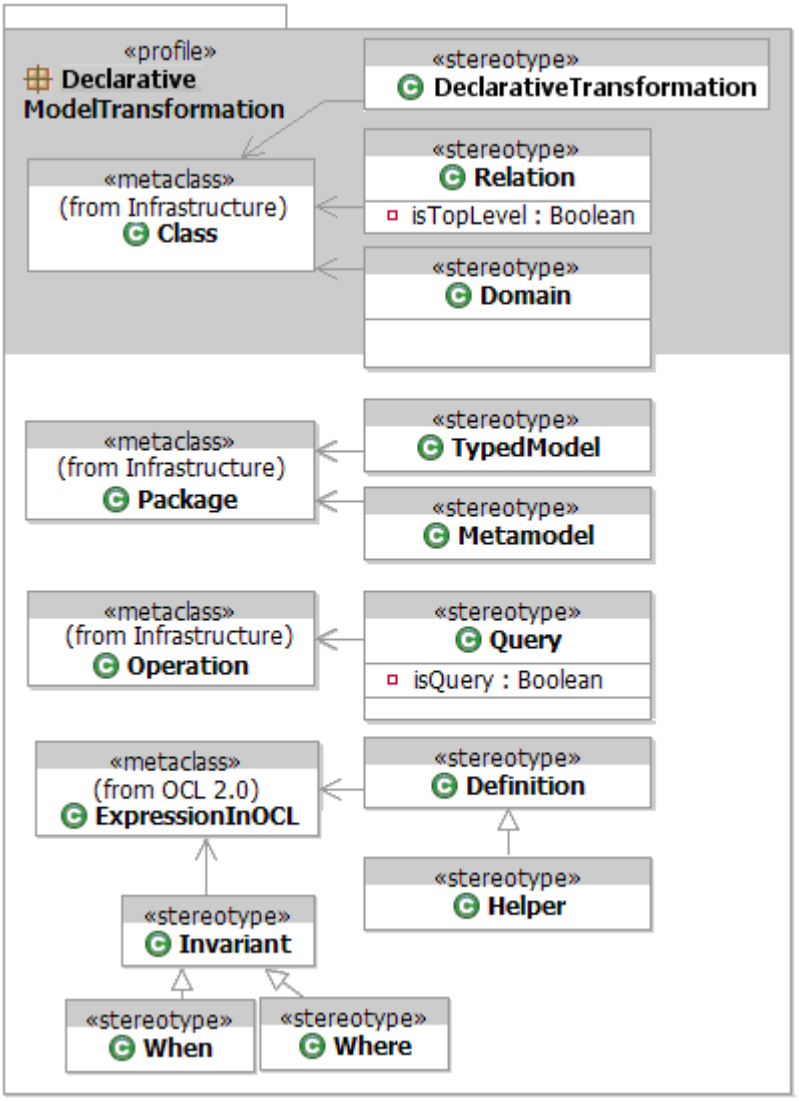

Figura 3-4. Perfil DeclarativeModelTransformation

Cuarta Etapa: Definición de atributos para los estereotipos.

Por cada atributo presente en el metamodelo propuesto, se define un atributo en el estereotipo correspondiente. La Figura 3-4 muestra los estereotipos con sus atributos, por ejemplo el atributo *IsTopLevel* de tipo Boolean en <<Relation>>.

Quinta Etapa: Definición en forma completa de la extensión propuesta

La extensión propuesta constituye una instanciación de la clase Profile dentro del paquete Profile de la Infraestructura 2.0. Este paquete se relaciona con el paquete Core y especifica entre otras, a las metaclases Profile, Stereotype y Extension.
Presentamos aquí una definición formal de los nuevos estereotipos mencionados previamente, indicando a que metaclase extienden (cual es su base), sus atributos, las restricciones que deben cumplirse al utilizarlos expresadas en OCL y las operaciones adicionales, de consulta y navegación.

Para enfatizar gráficamente algunos de los conceptos definidos en el *profile*, introducimos nuevos iconos. Este es el caso de los conceptos *Transformation* y *Relation*.

STEREOTYPE DeclarativeTransformation

Extended metaclass: Class

Graphical Notation: una ocurrencia de *DeclarativeTransformation* es mostrada por una figura hexagonal conteniendo el nombre de la transformación y el estereotipo <<transformation>>.

Constraints:

[1] Una *transformation* debe contener relaciones o queries y debe tener al menos dos modelos como parámetros (un *input* y un *output*).

```
(self.relations()->notEmpty() or  self.queries() -> notEmpty())
and self.modelParameters() -> size() > 1
```

[2] El conjunto de parámetros de modelo de una *transformation* debe ser igual al conjunto de modelos tipados del dominio y codominio.

```
self.modelParameters()=self.relations()-> collect (relation: Class
| relation. typedModels()) -> flatten() -> asSet()
```

[3] Una *transformation* puede solamente relacionarse con *relations* o *typedModels.*

```
self.relatedElements()->forAll(re| re.stereotype.name='relation'
or re.stereotype.name='typedModel')
```

Additional Operations

[1] La operación de consulta *relations()* retorna todas las relaciones asociadas con la transformación.

Class:: relations(): Set (Class)

```
relations = self.owningPackage.ownedMember-> select ( r:Class |
r.stereotype.name = 'relation' and self.owningPackage.ownedMember
-> exists (a:Association | a.memberEnd -> includes (e: Property |
e.type = self and e.opposite.type = r))
```

[2] La operación de consulta *queries()* retorna todos los *queries* asociados con la transformación.

Class:: queries(): Set (Operation)

```
queries = self.ownedOperations->select (o:Operation |
o.stereotype.name='query')
```

[3] La operación de consulta *modelParameters()* retorna todos los *typedModels* asociados con la transformación.

Class:: modelParameters (): Set (Package)

```
modelParameters = self.owningPackage.ownedMember-> select ( m:
Package | m.stereotype.name = 'typedModel' and
self.owningPackage.ownedMember -> exists (d:DirectedRelationship |
d.source = self and d.target = m))
```

[4] La operación de consulta *relatedElements*() retorna todos los *elementos* asociados con la transformación.
Class:: relatedElements(): Set (NamedElement)

```
relatedElements = self.owningPackage.ownedMember-> select ( e:
NamedElement | self.owningPackage.ownedMember -> exists (a:
Association | a.memberEnd -> includes (p: Property | p.type = self
and p.opposite.type = e))-> union (self.owningPackage.ownedMember-
> select( e: NamedElement | self.owningPackage.ownedMember ->
exists(d:DirectedRelationship | d.source = self and d.target = e))
```

STEREOTYPE Relation

Extended metaclass: Class

Graphical Notation: una ocurrencia de *relation* es mostrada por un pentágono conteniendo el nombre de la relación y el estereotipo <<relation>>.

OwnedAttributes:

- isTopLevel : Boolean

Especifica cuando la *relation* es una relación top level. El valor por defecto es falso. Si isTopLevel es falso, la *relation* debe ser llamada explícitamente por otra para ejecutarse.

Constraints:

[1] Una *relation* es parte sólo de una *transformation*.

```
self.transformation() -> size() = 1
```

[2] Una *relation* debe estar definida en al menos un *domain* y un *coDomain.*

```
self.domain() -> size() >= 1 and self.coDomain() -> size() >= 1
```

[3] Una *relation* **puede** tener una cláusula *when*. Sin embargo, **debe** tener una cláusula *where.*

```
self.when -> size ()<= 1 and self.where -> size ()= 1
```

Additional operations

[1] La operación de consulta *domain*() retorna todos los *domains* asociados con la *relation* a través del nombre de rol *domain.*
Class:: domain (): Set (Class)

```
domain = self.owningPackage.ownedMember -> select ( d:Class |
d.stereotype.name = 'domain' and self.owningPackage.ownedMember ->
exists (a:Association | a.memberEnd -> includes (e: Property |
e.type = self and e.opposite.type = d and e.opposite.name =
'domain')))
```

[2] La operación de consulta *coDomain*() retorna todos los *domains* asociados con la *relation* a través del nombre de rol *coDomain.*
Class:: coDomain(): Set (Class)

```
coDomain = self.owningPackage.ownedMember-> select ( d:Class |
```

```
d.stereotype.name = 'domain' and self.owningPackage.ownedMember ->
exists (a:Association |
a.memberEnd -> includes (e: Property | e.type = self and
e.opposite.type = d and e.opposite.name = 'codomain')))
```

[3] La operación de consulta *transformation*() retorna todas las *transformations* asociadas con la *relation.*
Class:: transformation (): Set (Class)

```
transformation = self.owningPackage.ownedMember-> select ( t:Class
| t.stereotype.name = 'transformation' and
self.owningPackage.ownedMember ->
exists (a:Association | a.memberEnd -> includes (e: Property |
e.type = self and e.opposite.type = t)))
```

[4] La operación de consulta *typedModels*() retorna todos los *type models* asociados con la *relation* (*type models* del dominio y del codominio).
Class:: typedModels(): Set (Package)

```
typedModels = self.domain().typedModel -> union
(self.coDomain().typedModel)
```

[5] La operación de consulta *when*() retorna la cláusula *when* que restringe a la *relation*.
Class:: when(): Set (Constraint)

```
when = self.constraint -> select(c | c.stereotype.name = 'when')
```

[6] La operación de consulta *where*() retorna la cláusula *where* que restringe a la *relation*.
Class:: where():Set (Constraint)

```
where = self.constraint -> select(c | c.stereotype.name = 'where')
```

[7] La operación de consulta *helpers*() retorna todos los *helpers* definidos en la *relation*.
Class:: helpers():Set (Constraint)

```
helpers = self.constraint -> select(c | c.stereotype.name =
'helper')
```

[8] La operación de consulta *relationVars*() retorna todas las *variables* definidas en la *relation.*
Class:: relationVars():Set (VariableDeclaration)

```
relationVars = self.constraint-> select ( c:Constraint |
c.body.bodyExpression.oclIsKindOf(VariableExp))-> collect
(c:Constraint | c.body.bodyExpression.referredVariable)
```

STEREOTYPE Domain

Extended metaclass: Class

Constraints:

[1] Un *domain* es parte de una sola *relation*.

```
self.relation() -> size() = 1
```

[2] Un *domain* debe referenciar a un único *TypedModel* (existe una única *DirectedRelationship* entre *Domain* and *TypedModel*).

```
self.owningPackage.ownedMember-> select(d:DirectedRelationship |
d.source = self and d.target.stereotype.name = 'typedModel')->
size()=1
```

[3] Un *domain* debe contener al menos una declaración de variables.

```
self.variables() -> size()>0
```

Additional operations

[1] La operación de consulta *relation*() retorna todas las *relations* asociadas con el *domain*.

Class:: relation(): Set (Class)

```
relation  = self.owningPackage.ownedMember-> select ( d:Class |
d.stereotype.name = 'relation' and self.owningPackage.ownedMember
->exists (a:Association | a.memberEnd-> includes (e : Property|
e.type = self and e.opposite.type = d))
```

[2] La operación de consulta *variables*() retorna todas las variables definidas en el *domain*.

Class:: variables(): Set (VariableDeclaration)

```
variables  = self.constraint-> select ( c:Constraint |
c.body.bodyExpression.oclIsKindOf(VariableExp))-> collect
(c:Constraint | c.body.bodyExpression.referredVariable)
```

STEREOTYPE TypedModel

Extended metaclass: Package

Constraints:

[1] Un *typed model* debe ser instancia de sólo metamodelo, o sea debe ser el *source* de una DirectedRelationship con stereotipo <<type>> y *target* un paquete <<metamodel>>.

```
self.metamodel() -> size() = 1
```

[2] Un *typed model* debe contener solamente instancias del metamodelo con el que se relaciona.

```
self.ownedMember-> forAll (e| self.metamodel()-> includes
(e.type))
```

Additional operations

[1] La operación de consulta *metamodel*() retorna todas los metamodelos asociados con el *typedModel.*

Package:: metamodel (): Set (Package)

```
metamodel = self.owningPackage.ownedMember->select
(d:DirectedRelationship | d.source=self and
d.target.stereotype.name = 'metamodel') ->collect
(d:DirectedRelationship| d.target) )
```

STEREOTYPE Query

Extended metaclass : Operation

Constraints:

[1] Una operación <<query>> debe ser de consulta.
```
self.isQuery=true
```
[2] Una operación <<query>> debe estar definida en una clase <<transformation>>.
```
self.class-> notEmpty() and self.class.stereotype.name =
'transformation'
```

STEREOTYPE When/Where (subclass of Invariant)

Extended metaclass: ExpressionInOCL

Constraints:

[1] Una cláusula *when/where* debe restringir una *relation*.
```
self.constraint.constrainedElement.stereotype.name = 'relation'
```

STEREOTYPE Helper (subclass of Definition)

Extended metaclass: ExpressionInOCL

Constraints:

[1] Un *helper* debe restringir una *relation*.
```
self.constraint.constrainedElement.stereotype.name = 'relation'
```

3.4 Construcción del Perfil para SQVT Operacional

En forma similar a como fue definido el perfil declarativo, nuestra propuesta es definir un perfil específico para SQVT operacional, como extensión de la Infraestructura, que permite expresar características particulares de las transformaciones operacionales.
Como ya vimos, la extensión se define en cinco etapas:

Primera Etapa: Definición del metamodelo propuesto

El metamodelo que proponemos minimiza al metamodelo del paquete Operational de QVT, basándose en la sintaxis abstracta de dicho paquete con algunas adaptaciones, como muestra la Figura 3-5. Las Figuras 3-6, 3-7 y 3-8 presentan los metamodelos que completan la definición para transformaciones operacionales: las ImperativeOperation y las ImperativeExpression de OCL respectivamente.

Estos metamodelos fueron mínimamente modificados para simplificar QVT y también para poder definir correctamente la semántica de SQVT Operacional. Cada modificación se presenta con su correspondiente justificación.

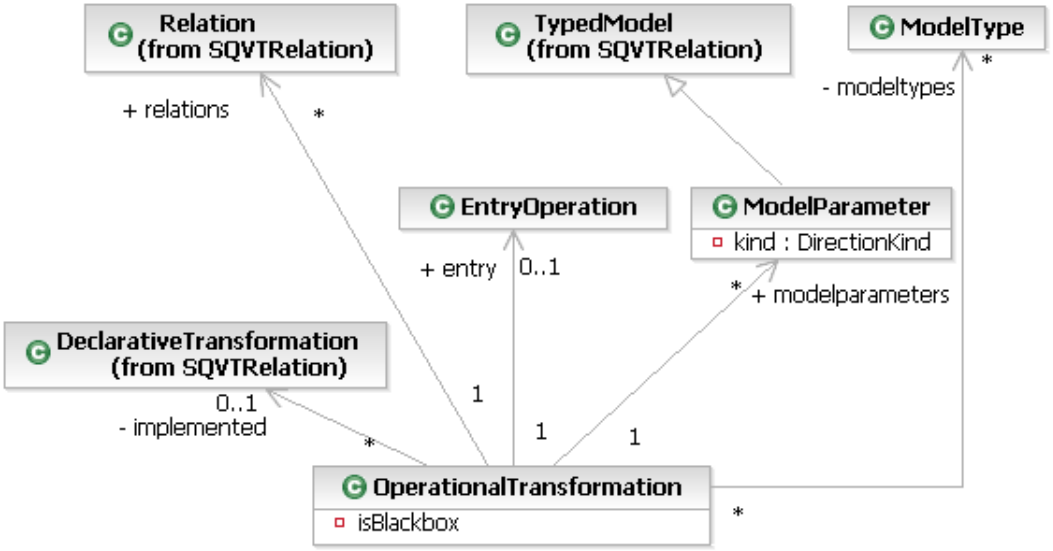

Figura 3-5. Metamodelo de las transformaciones operacionales

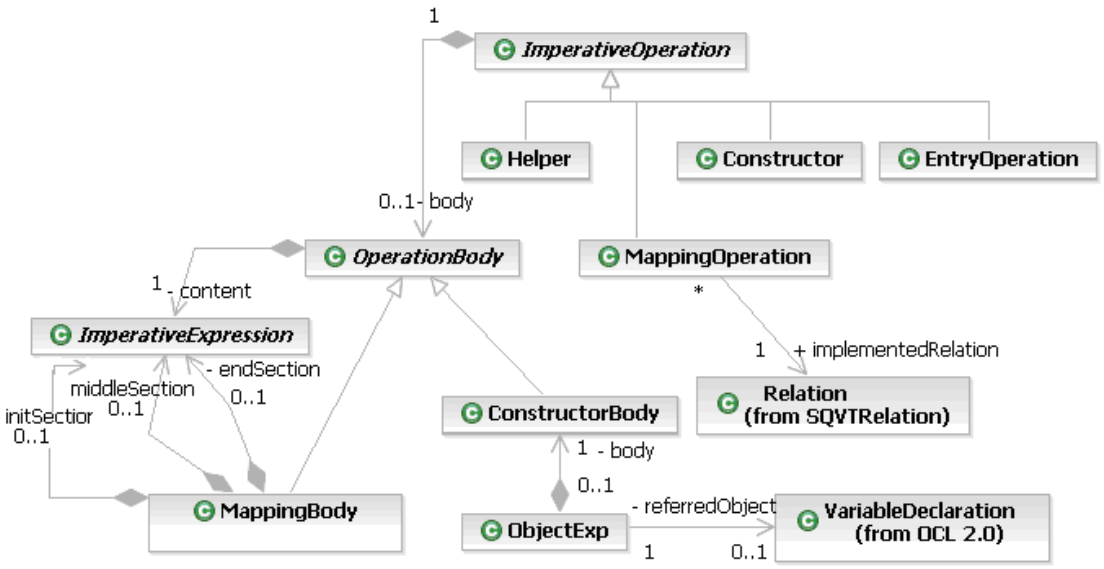

Figura 3-6. Metamodelo de las operaciones imperativas

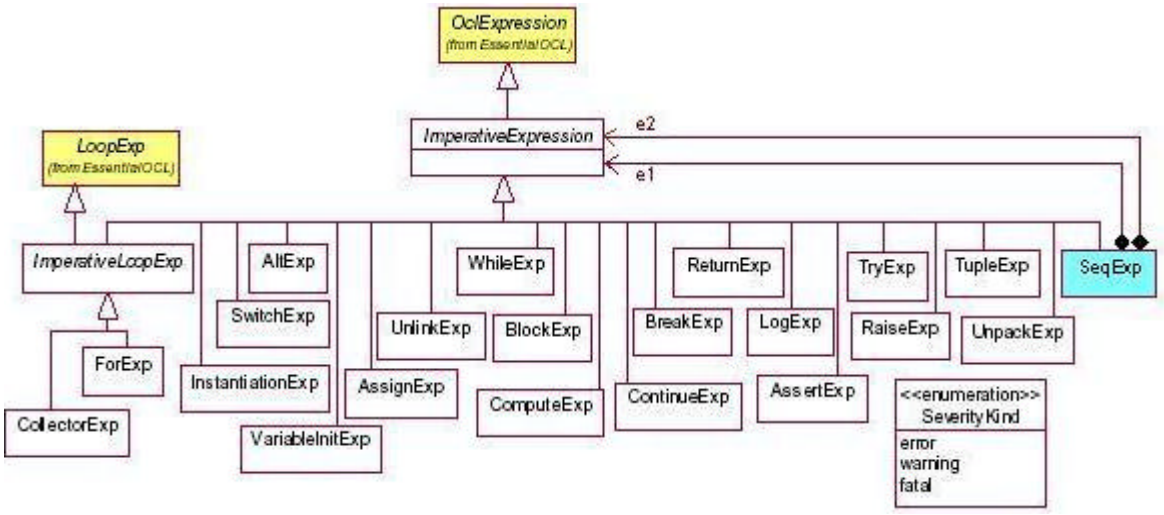

Figura 3-7. Metamodelo de las expresiones imperativas

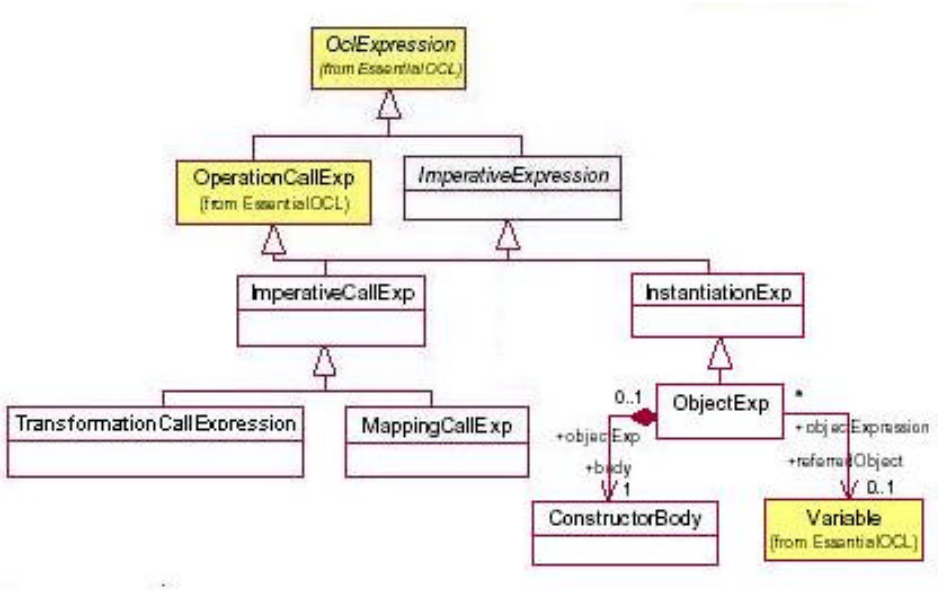

Figura 3-8. Expresiones imperativas para invocación e instanciación

Al igual que en QVT Operacional, una OperationalTransformation (Figura 3-5) puede ser no blackbox por lo que tendrá arranque de ejecución en la operación entryOperation. Una OperationalTransformation se puede componer de: EntryOperation, Constructor, MappingOperation y Helper, subclases de ImperativeOperation. Una ImperativeOperation (Figura 3-6) tiene parámetros y *body*, de tipo OperationBody. El *body*, a su vez, contiene una ImperativeExpression (en QVT original aparecen OclExpression como body),

subclases de OclExpression. Entre otras AssignExp, ifExp, whileExp, etc., son subclases de ImperativeExpression (Figura 3-7).
Al metamodelo de las expresiones imperativas que propone QVT, resulta necesario agregar o modificar las siguientes expresiones:

SeqExp subclass de ImperativeExpression
Representa la expresión secuencia, o sea está formada por dos expresiones de tipo ImperativeExpression, con un ; en el medio.
Sea e: **SeqExp,** se le pueden pedir las 2 expresiones que la forman e.e1 y e.e2, como muestra la Figura 3-7.
Justificación: la secuencia de expresiones fue **agregada** a la jerarquía ImperativeExpression para poder expresar la secuencia como una expresión más y así poder definir su semántica mediante la función semántica de las expresiones imperativas.

NullExp subclass de ImperativeExpression
Representa la expresión vacía, que al ejecutarse no produce cambios en el ambiente de ejecución ni en la memoria, o sea, no produce cambios de estado.
Justificación: la expresión nula fue **agregada** a la jerarquía ImperativeExpression para poder tratarla como una expresión más y definir su semántica mediante la función semántica de las expresiones imperativas. Esta expresión representa la ausencia de alguna expresión o sub-expresión opcional y evita que se produzca un error al aplicar la función semántica de expresiones.

ImperativeCallExp: subclass de OperationCallExp y de **ImperativeExpression**.
Representa la invocación a otras operaciones (Figura 3-8). El metamodelo de QVT Operacional la propone, pero sólo como subclase de OperationCallExp.
O sea, por ser OperationCallExp se le puede pedir:
Sus *arguments* (de tipo OCLExpression [*]) (parámetros reales)
Su *source*, que representa a su expresión receptora, de tipo OCLExpression, su *referredOperation* (Operation a quien llama, en este caso de la jerarquía ImperativeOperation), la cual tiene sus parámetros formales y los podemos obtener con operaciones adicionales ya definidas[4].
Justificación: Consideramos que debe ser también subclase de ImperativeExpression para poder definir su semántica mediante la función semántica de las expresiones imperativas.
QVT propone dos subclases para **ImperativeCallExp:**
MappingCallExp: representa la invocación a una MappingOperation. La estructura de esta expresión difiere del resto de las expresiones imperativas. Puede tener initSection, middleSection y endSection.
TransformationCallExp: representa la invocación explícita a una transformación (a la cual referencia) desde el body de una expresion imperativa.
La operación **ImperativeCallExp** genera la ejecución del resto de las ImperativeOperations. La operación EntryOperation es especial respecto al resto de las ImperativeOperation. No tiene parámetros y en su *body* habrá llamadas (ImperativeCallExp) que provocarán la ejecución de las otras

[4] El perfil que presentamos más adelante, incluye operaciones adicionales para ImperativeOperation

ImperativeOperations de la transformación o bien la ejecución de otra transformación (a través de una TransformationCallExp). Es decir, el resto de las operaciones no se ejecutarán aisladamente, sino a través de las ImperativeCallExp.
Finalmente, consideramos en nuestro metamodelo a la expresion imperativa ObjectExp para instanciar o crear nuevos objetos (ya definida por QVT). ObjectExp tiene body, de clase ConstructorBody, y referencia a una VariableDeclaration de OCL, como muestra la Figura 3-8.

Segunda Etapa: Definición de los estereotipos correspondientes al metamodelo propuesto

Por cada metaclase del paso anterior, definimos un estereotipo nuevo. La próxima etapa muestra a los nuevos estereotipos con sus bases. Para algunas metaclases (particularmente aquellas que pertenecen a jerarquías de OCL ya existentes o bien propuestas por QVT Operational) no definimos estereotipos para guardar uniformidad con la definición del resto de la jerarquías existentes. Este es el caso de la jerarquía ImperativeExpression donde fueron agregadas o adaptadas algunas metaclases.

Tercera Etapa: Identificación de las bases para los estereotipos.

En esta etapa se identifican las metaclases de la Infraestructura que pueden ser extendidas para representar el metamodelo propuesto en la primera etapa.
Según este metamodelo, una *OperationalTransformation* podría ser una Class compuesta por su entryOperation, los modelos que transforma, y por los mappingOperation, Constructors y Helpers.
El estereotipo abstracto ImperativeOperation, puede tener base en Operation, al igual que los estereotipos concretos que lo especializan (EntryOperation, Constructor, MappingOperation y Helper).
El estereotipo OperationBody, al igual que los estereotipos que lo especializan (MappingBody y ConstructorBody), pueden tener base en Behavior. Un elemento Behavior representa el método contenido en las propiedades con comportamiento (es decir propiedades correspondintes a la metaclase BehavioralFeature).
El estereotipo ModelParameter que especifica a los modelos que participan en la transformación, se puede representar como sub-estereotipo de TypedModel, del profile declarativo (Relational) con base en Package.
La Figura 3-9 muestra los estereotipos definidos con su metaclase extendida, conformando así el perfil OperationalModelTransformation.

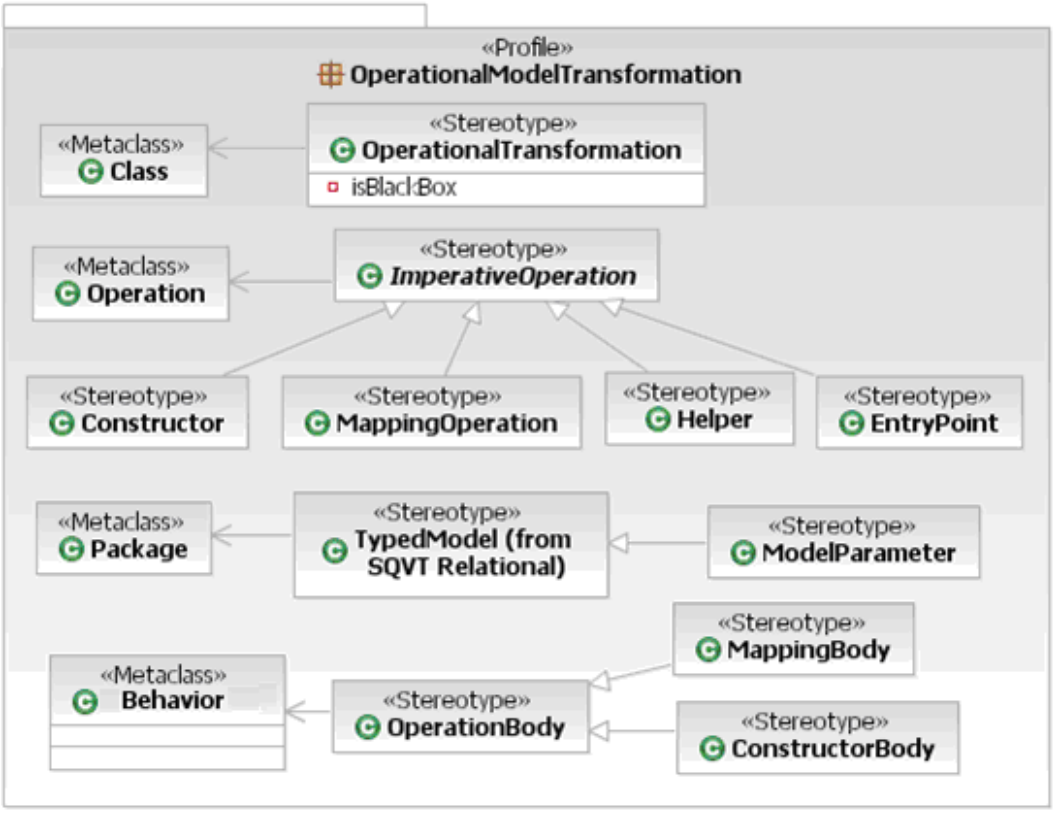

Figura 3-9. Perfil OperationalModelTransformation

Cuarta Etapa: Definición de atributos para los estereotipos.

Por cada atributo presente en el metamodelo propuesto, se define un atributo en el estereotipo correspondiente. La Figura 3-9 muestra los estereotipos con sus atributos, por ejemplo *IsBlackbox* de tipo Boolean en <<OperationalTransformation>>.

Quinta Etapa: Definición en forma completa de la extensión propuesta

La extensión propuesta constituye (como en el caso del perfil declarativo) una instanciación de la clase Profile de la Infraestructura 2.0.
Presentamos aquí una definición formal de los nuevos estereotipos mencionados previamente, indicando a qué metaclase extienden (cual es su base), sus atributos, las restricciones que deben cumplirse al utilizarlos expresadas en OCL y las operaciones adicionales, de consulta y navegación.

STEREOTYPE OperationalTransformation

Extended metaclass: Class

Graphical Notation: una ocurrencia de *OperationalTransformation* contiene el nombre de la transformación y el estereotipo <<operationalTransformation>>.

OwnedAttributes:

- isBlackbox: Boolean

Constraints:

[1] Una *operationalTransformation* debe contener mappingOperations, constructores o helpers y debe tener al menos dos modelos como parámetros (un *input* y un *output*) o un inout.

```
(self.mappingOperations()->notEmpty() or self.helpers()->
notEmpty()) or self.constructs()->notEmpty and
self.modelParameters() -> size() > 1
```

[2] Una *operationalTransformation* no blackbox debe contener un único entryOperation para arrancar su ejecución

```
self.isBlackbox = false implies self.entryOperation ()->size() = 1
```

[3] Una *operationalTransformation* puede implementar a lo sumo una transformation del SQVTRelation

```
self.implemented()->size()<= 1
```

[4] El conjunto de parámetros de modelo de una *operationalTransformation* debe ser igual al conjunto de modelos tipados que intervienen en las mappingOperations, constructores, helpers y entryOperation.

```
self.modelParameters()= (self.mappingOperations-> union
(self.constructors) -> union->(self.helpers) -> union (self.
entryOperation))-> collect (o| o.ownedParameter()) -> flatten() ->
asSet()
```

[5] El conjunto de parámetros de modelo de una *operationalTransformation* debe ser igual al conjunto de modelos tipados definidos en la transformación declarativa que implementa, si ésta existe.

```
self.implemented -> size()=1 implies self.modelParameters()=
self.implemented.modelParameters
```

[6] Una *operationalTransformation* puede solamente ser dueña de mappingOperations, constructores, helpers o entryOperation.

```
self.ownedOperation-> forAll(o| o.stereotype.name=‘
entryOperation´ or o.stereotype.name=‘ mappingOperation´ or
o.stereotype.name=‘ constructor´ or o.stereotype.name=‘ helper´)
```

[7] Una *operationalTransformation* puede solamente relacionarse con modelParameters, con una transformation, y relations (del SQVTRelation)

```
self.relatedElements()->forAll(re| re.stereotype.name=‘relation’
or re.stereotype.name=‘ modelParameter’or
re.stereotype.name=‘transformation’)
```

[8] El conjunto de relaciones referenciadas por una operationalTransformation debe incluir al conjunto de las relaciones que implementa mediante sus mappingOperations.

```
self.relations()-> includesAll(self.mappingOperations -> collect
(m | m.implementedRelation))
```

Additional Operations

[1] La operación de consulta *mappingOperations ()* retorna todas las operaciones mapping asociadas con la transformación.
Class:: mappingOperations (): Set (Operation)
```
mappingOperations = self.ownedOperations->select (o:Operation |
o.stereotype.name='mappingOperation')
```

[2] La operación de consulta *helpers()* retorna todos los *helpers* asociados con la transformación.
Class:: helpers(): Set (Operation)
```
helpers = self.ownedOperations->select (o:Operation|
o.stereotype.name='helper')
```

[3] La operación de consulta *constructors()* retorna todos los *constructores* asociados con la transformación.
Class:: constructors(): Set (Operation)
```
   constructors = self.ownedOperations->select (o:Operation |
   o.stereotype.name='constructor')
```

[4] La operación de consulta *entryOperation()* retorna la operación de entrada asociada con la transformación.
Class:: entryOperation(): Set (Operation)
```
   entryOperation = self.ownedOperations->select (o:Operation |
   o.stereotype.name='entryOperation´)
```

[5] La operación de consulta *implemented()* retorna la transformación declarativa asociada con la transformación operacional.
Class:: implemented(): Set (Class)
```
implemented = self.owningPackage.ownedMember-> select ( t:Class |
r.stereotype.name = 'transformation'and
self.owningPackage.ownedMember -> exists (a:Association |
a.memberEnd -> includes (e: Property | e.type = self and
e.opposite.type = t))
```

[6] La operación de consulta *relations()* retorna las relaciones declarativas asociadas con la transformación operacional.
Class:: relations(): Set (Class)
```
relations = self.owningPackage.ownedMember-> select ( r:Class |
r.stereotype.name = 'relation' and self.owningPackage.ownedMember
-> exists (a:Association | a.memberEnd -> includes (e: Property |
e.type = self and e.opposite.type = r))
```

[7] La operación de consulta *modelParameters()* retorna todos los *modelParameters* asociados con la transformación.
Class:: modelParameters (): Set (Package)
```
modelParameters = self.owningPackage.ownedMember-> select ( m:
Package | m.stereotype.name = 'modelParameter' and
self.owningPackage.ownedMember -> exists (d:DirectedRelationship |
d.source = self and d.target = m))
```

[8] La operación de consulta *relatedElements*() retorna todos los *elementos* asociados con la transformación.
Class:: relatedElements(): Set (NamedElement)

```
relatedElements = self.owningPackage.ownedMember-> select ( e:
NamedElement | self.owningPackage.ownedMember -> exists (a:
Association | a.memberEnd -> includes (p: Property | p.type = self
and p.opposite.type = e))-> union (self.owningPackage.ownedMember-
> select( e: NamedElement | self.owningPackage.ownedMember ->
exists(d:DirectedRelationship | d.source = self and d.target = e))
```

STEREOTYPE ImperativeOperation (Abstract)

Extended metaclass: Operation

OwnedAttributes:

- `body`: **OperationBody**

Constraints:

[1] Una operación de la jerarquía <<imperativeOperation>> debe estar definida en una clase <<operationalTransformation>>.

```
self.class-> notEmpty() and self.class.stereotype.name =
'operationalTransformation'
```

[2] Una operación de la jerarquía <<imperativeOperation>> contiene un *body* perteneciente a la jerarquia <<OperationBody>>.

```
self.body-> notEmpty() and self.body.stereotype.allParent.name ->
includes(`operationBody´)
```

STEREOTYPE OperationBody

Extended metaclass: Behavior

Constraints:

[1] Un OperationBody contiene una expresión OCL perteneciente a la jerarquía ImperativeExpression.

```
self.content.type.oclIsKindOf(ImperativeExpression)
```

STEREOTYPE MappingBody (subclass of OperationBody)

Extended metaclass: Behavior
Graphical Notation: cada sección del MappingBody es mostrada con la keyword correspondiente "**init", "middle" y** "**end"** delante**,** encabezando la sección.

Constraints:

[1] Un MappingBody contiene a lo sumo una sección initSection, una seción middleSection y una seción endSection, las tres pertenecientes a la jerarquia ImperativeExpression de OCL

```
 (self.initSection -> size()=1 implies
self.initSection.OclIsKindOf(ImperativeExpression))and
```

```
(self.middleSection -> size()=1 implies
self.middleSection.type.oclIsKindOf(ImperativeExpression))
and (self.endSection -> size()=1 implies
self.endSection.type.oclIsKindOf(ImperativeExpression))
```

STEREOTYPE ConstructorBody (subclass of OperationBody)

Extended metaclass: Behavior

Constraints:

[1] Un ObjectExp contiene un body de tipo ConstructorBody.

```
self.body.stereotype.name = <<constructorBody>>
```

STEREOTYPE EntryOperation (subclass of ImperativeOperation)

Extended metaclass : Operation

[1] El nombre de una operación <<entryOperation>> es "**main".**

```
self.name = 'main'
```

STEREOTYPE Constructor (subclass of ImperativeOperation)

Extended metaclass: Operation

Graphical Notation: la signatura de una operación <<**constructor**>> es mostrada como cualquier otra operación excepto que la keyword **constructor** debe aparecer encabezando la signatura.

Constraints:

[1] El *body* de una operación <<**constructor**>> debe corresponder al estereotipo <<**constructorBody**>>

```
self.body.stereotype.name = <<constructorBody>>
```

STEREOTYPE Helper (subclass of ImperativeOperation)

Extended metaclass : Operation

Graphical Notation: la signatura de una operación <<**helper**>> es mostrada como cualquier otra operación excepto que la keyword **query** debe aparecer encabezando la signatura.

Constraints:

[1] Una operación helper no debe producir efectos laterales

```
self.isQuery = true
```

STEREOTYPE MappingOperation (subclass of ImperativeOperation)

Extended metaclass: Operation

Graphical Notation: la signatura de una operación <<**mappingOperation** >> es mostrada como cualquier otra operación excepto que la keyword **mapping** debe aparecer encabezando la signatura.

Constraints:
[1] El *body* de una operación << **mappingOperation**>> debe corresponder al estereotipo <<m**appingBody**>>

```
self.body.stereotype.name = <<mappingBody>>
```

[2] Una **mappingOperation** debe implementar una relation del SQVT Relation

```
self.implementedRelation()->size() = 1 and
self.implementedRelation().stereotype.name='relation'
```

Additional operations

[1] La operación de consulta *implementedRelation()* retorna la relation declarativa asociada con la operación mapping
Operation:: implementedRelation(): Set (Class)

```
implementedRelation = self.owningPackage.ownedMember-> select (
r:Class | r.stereotype.name = 'relation' and
self.owningPackage.ownedMember-> select( e: NamedElement |
self.owningPackage.ownedMember -> exists(d:DirectedRelationship |
d.source = self and d.target = e))
```

STEREOTYPE ModelParameter (subclass of TypedModel - from SQVT Relation)

Extended metaclass: Package

OwnedAttributes:
- kind: DirectionKind = {in, out, inout}

3.5 Aplicación de Ambos Perfiles al Ejemplo

Retomando el ejemplo definido mediante una notación textual en el Capítulo 3, donde marcamos las adaptaciones a QVT que proponemos en SQVT, en esta sección presentamos el uso de ambos perfiles de SQVT (declarativo y operacional) definidos más arriba, instanciando dicho ejemplo.

3.5.1 Ejemplo en SQVT Declarativo

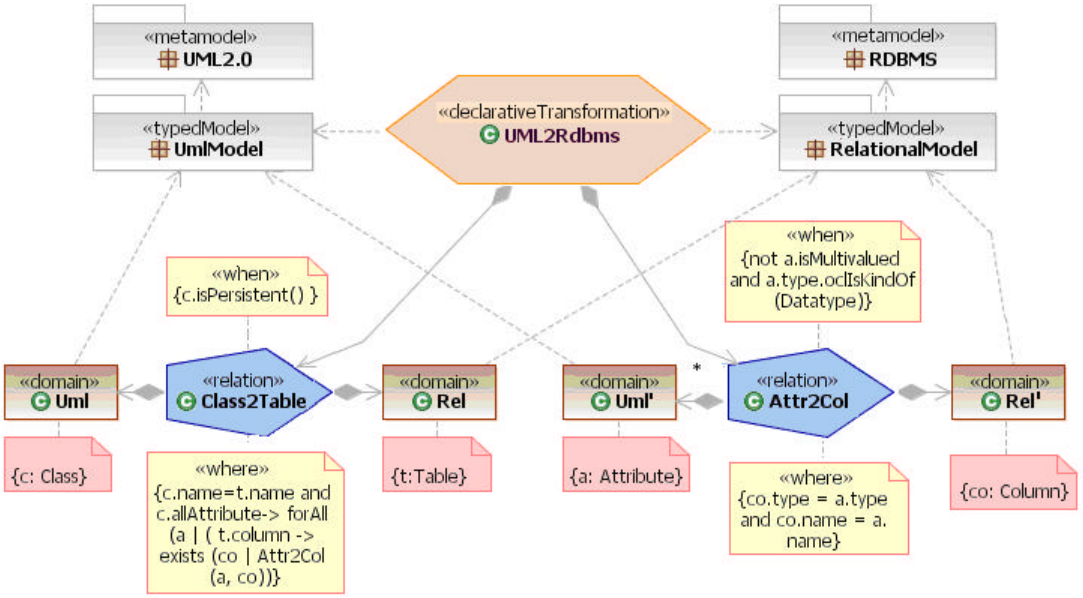

Figura 3-10. Instanciación del profile Declarativo para el ejemplo.

La Figura 3-10 muestra este ejemplo utilizando la notación gráfica y estereotipos propuestos en SQVT declarativo. Este modelo presenta una aplicación de la transformación que define reglas de transformación sobre instancias genéricas de las metaclases (clases y tablas, atributos y columnas). Podemos observar que una nueva instanciación de esta transformación, será sobre elementos particulares de modelos concretos a nivel M1 de la arquitectura 4 capas.

La transformación tiene dos modelos como parámetros (umlModel y RelationalModel) y dos relaciones (class2Table y Attr2Col).

Las cláusulas *when* y *where* del ejemplo son expresiones OCL estereotipadas con <<when>> y <<where>> respectivamente que restringen las relaciones Class2Table y Attr2Col. Ambas relaciones tienen un dominio (Uml) y un codominio (Rel). Como dominio y codominio son estructuralmente idénticos y la única diferencia entre ellos es el rol que cumplen en la relación, es posible representar a ambos mediante una clase estereotipada con <<domain>>. Las variables dentro de los dominios también se expresan en OCL con la metaclase VariableDeclaration que, como vimos en el Capítulo 3, sección 3.3, se corresponde con la metaclase TemplateExp de QVT.

3.5.2 Ejemplo en SQVT Operacional

La transformación operacional que presentamos, implementa a la transformación declarativa de la sección anterior. Las mappingOperation, que implementan relaciones de la transformación declarativa, no se distinguen por niveles, a diferencia de las relaciones (que pueden ser topLevel). Por lo tanto, la transformación debe tener una entryOperation llamada main para arrancar la ejecución.

Dentro del mapping, la keyword *refines* indica a que relación (declarativa) se está implementando, (self.implementedRelation). De esa relación se toman las cláusulas *when* y *where* que deben cumplirse para ejecutar el *mapping*.

```
<<operationalTransformation>> UML2Rdbms (in uml:UML2.0,out rel:
RDBMS)
-- la operación entry selecciona las clases del modelo uml y
-- aplica la transformación sobre cada clase
      <<entryOperation>> main()
      {
      uml.objectOfType(Class) -> map Class2Table( );
      }

-- mapea una clase persistente en una tabla, con una columna por
-- cada atributo simple de la clase
       <<mappingOperation>> mapping Class:: Class2Table( ): Table
      refines Class2Table
      {
      middle {
      -- inicializa la tabla resultante
      if self.isPersistent() then
      result.name := self.name;
      result.column:= self.allAtributes-> map Attr2Col( ) endif;}
      }

-- mapea cada atributo simple en una columna del mismo tipo
       <<mappingOperation>> mapping Atribute:: Attr2Col ( ):
       Column
      refines Attr2Col
      {
      middle {-- inicializa la columna
      if not self.isMultivalued and
      self.type.OclIsKindOf(DataType) then
      result.name := self.name;
      result.type:= self.type endif; }
      }
```

3.6 Conclusión del Capítulo

En este capítulo hemos presentado el lenguaje SQVT para expresar transformación de modelos cuyo metamodelo inicial está inspirado en el lenguaje QVT.

Respecto a la definición sintáctica de un lenguaje, existen dos opciones: una de ellas es crear un nuevo lenguaje basado en un metamodelo, lo que implica la definición de nuevas metaclases. Otra opción es utilizar el mecanismo estándar de extensión de UML que consiste en extender metaclases existentes mediante estereotipos.

Nuestra propuesta se basa en utilizar especificaciones ya existentes en OMG; por un lado, mediante la definición de estereotipos, extendemos la Infraestructura 2.0, una especificación más abstracta que UML e independiente de él y, por otro lado utilizamos el lenguaje OCL para expresar patrones de la transformación, puesto que después de analizar estos conceptos, concluimos que no es necesario crear nuevos elementos para modelarlos.

Pretendemos que el lenguaje SQVT sea minimal en el conjunto de metamodelos que permiten expresar relaciones y *queries* de transformación de modelos. Mantener simplicidad y reducir el tiempo de entrenamiento del usuario son las ventajas principales de esta propuesta minimal. Además, puesto que la propuesta maximiza el uso del OCL y actualmente existen varias herramientas de modelado que soportan OCL, el trabajo de desarrollo de una herramienta para soportar este lenguaje de transformación, se facilita considerablemente.
En el capitulo siguiente, presentamos el formalismo de la *teoría de problemas;* basándonos en este formalismo definimos la semántica de SQVT, tanto a nivel declarativo como operacional.

Capítulo
4

La Teoría Intuitiva de Problemas como Base Formal para Lenguajes de Transformación de Modelos

En este capítulo presentamos la semántica -a nivel declarativo y a nivel operacional- de lenguajes para transformación de modelos. Dicha semántica está definida en base al formalismo de la *teoría intuitiva de problemas*, por lo que en las primeras secciones se introducen los conceptos básicos de dicho formalismo. Definir en forma precisa la semántica de estos lenguajes, nos permitirá probar corrección entre sus niveles declarativo y operacional.

4.1 El Formalismo Básico: Los Conceptos de Problema y Solución

En esta sección resumiremos los conceptos principales que comprende la *teoría de problemas* [25] [27], la cual está basada en las ideas intuitivas desarrolladas por Pólya [26].
En los últimos años, la teoría de problemas ha sido usada como fundamento para cálculos para derivación de programas, tales como las álgebras Fork [28].

Los conceptos de problema y solución

Un *problema* para esta teoría es una cuadrupla **P** = < **D,R,q,I**> donde **D** es el *dominio de datos* y **R** es el *dominio de resultados* (ambos subconjuntos de un conjunto fijo U que denominaremos *universo del discurso*), mientras que **q** es una relación binaria **D**x**R**, la cual es la *especificación del problema*, es decir un elemento **d** del domino de datos **D** y un elemento **r** del dominio de resultados **R** están en la relación **q** si y sólo si **r** es un resultado esperado o aceptable para **d** en el problema. En otras palabras, **q** es lo que Pólya denomina la condición del problema. Sobre el cuarto elemento de la cuadrupla hablaremos más adelante.

Por ejemplo, si queremos derivar código Java desde diagramas de clase UML, el dominio de datos consistirá en clases UML, el dominio de resultado será el conjunto de programas Java y la condición **q** relacionará cada clase **d** de UML perteneciente a **D** con algunos elementos en **R**, cada uno de los cuales será una implementación en Java aceptable para la clase **d** de UML.
Podemos estar interesados en derivar código sólo para clases persistentes, que es obviamente un subconjunto del dominio de todas las clases UML. En este caso, diremos que el subconjunto obtenido desde las clases persistentes es el *conjunto de instancias de interés* de nuestro *problema*, que es el cuarto elemento, **I**, de nuestro cuadrupla. El diagrama de conjuntos que representa un *problema* se muestra en la Figura 4-1.

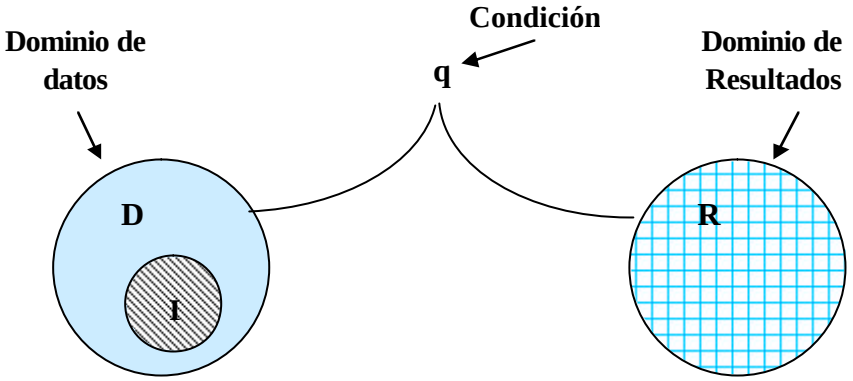

Figura 4-1. Diagrama de un Problema

Gráficamente, un problema también puede ser visualizado usando coordenadas Cartesianas, donde **D** es un subconjunto de las abscisas y **R** es un subconjunto de las ordenadas.
Diremos que un *problema* es *viable* si y sólo si para cada dato **d** del *conjunto* **I** *de instancias de interés* existe al menos un elemento **r** perteneciente al *dominio de resultados* **R**, tal que el par **<d,r>** pertenezca a la *condición* **q**. Es decir, **q** debe estar definida para todo el *conjunto de instancias de interés*:

(condición de viabilidad) $(\forall \mathbf{d})\,(\,\mathbf{d} \in \mathbf{I} \rightarrow (\exists r)\,(\,r \in \mathbf{R} \wedge \mathbf{q}(\mathbf{d}, r)\,)\,)$

Ahora bien, ¿qué significa una *solución* para el *problema* **P**? Parece natural pensar que es una subregión de **q**, tal que a cada punto **d** en **I** le asigne uno o más puntos **r** del *dominio de resultados*, por ejemplo a cada clase UML persistente se le asignará sus correspondientes programas Java (posiblemente, más de uno).
Sin embargo hay en esta definición de *solución* un hecho un tanto molesto: **q** misma es una *solución*. Así, todo *problema viable*, una vez enunciado, ya está automáticamente solucionado.
Por otra parte, ¿qué significa el cuantificador existencial en la *condición de viabilidad* enunciada más arriba? Simplemente significa que para un dato dado existe algún resultado relacionado con él por la *condición*, o lo que es lo mismo, para un dato dado **d** existen pares <**d**, **r**> en la *condición del problema*, y debemos elegir cuáles nos interesan. La elección de algún arco de la *condición* que contenga un dato dado no parece poder generalizarse a un método efectivo (en algún sentido *razonable*) de obtención de *soluciones* para *problemas*.

Por ejemplo, si consideramos la derivación de código Java desde diagramas de clase UML, un típico algoritmo del tipo *British Museum* que toma elementos del conjunto de programas Java y verificar cuales de ellos implementa la clase UML, no parece ser el método más satisfactorio para resolver el problema de derivación de código. Entonces, contar con un procedimiento eficaz para construir el resultado es muy distinto a tener que elegir puntos en una relación.

En nuestro ejemplo de derivación de código, usar el algoritmo de *British Museum* es muy distinto a conocer un algoritmo para obtener código desde clases UML. De este modo, la teoría de problemas considera que una solución debe ser una función de **D** en **R** que satisface la condición **q** para el *conjunto de instancias de interés*. Pero, también una solución debería cumplir con la propiedad α, que es llamada *contexto de admisibilidad.*
Es importante notar que la propiedad α puede ser extensional, como por ejemplo: *continua, derivable, creciente, calculable,* etc.; o intencional, como: *eficiente, elegante, fácil,* etc.
En particular, estamos interesados en que la solución sea calculable mediante un algoritmo de computación eficiente (de complejidad aceptable, no más que polinomial). Llamemos α-*solución* a las funciones que tienen tales características y llamemos Ω_P al conjunto de todas las α-soluciones de un problema **P.**

4.2 Lenguajes Declarativos vs. Lenguajes Imperativos en la Teoría de Problemas

Los problemas así como las soluciones, son expresados por medio de sentencias escritas en un lenguaje dado que tiene su propia sintaxis y semántica. En los párrafos siguientes introducimos algunas consideraciones simples sobre *lenguajes declarativos* y *lenguajes imperativos*, desde la perspectiva de la *teoría de problemas*, indicando la diferencia entre aspectos sintácticos y semánticos.

Lenguajes Declarativos para la descripción de Problemas

Los *problemas* se expresan por medio de *sentencias* escritas en un lenguaje declarativo L_D que tiene su propia sintaxis y una semántica dada por una función $\mathtt{m}$. El rol de esta función semántica $\mathtt{m}$ es permitir dar un significado a las *sentencias* de problemas, asociando cada *sentencia* **Spec,** escrita en el lenguaje L_D, al *problema* **P** = $\mathtt{m}$[**Spec**] especificado por la *sentencia*.
Así, debemos distinguir *declaraciones* de *problemas*. Un *problema* es *un objeto matemático abstracto e ideal.* Por otra parte, una *sentencia* es un objeto lingüístico concreto, en el sentido de que su texto consiste de un grupo de símbolos (o diagramas). La conexión entre ellos ocurre por medio de la función semántica $\mathtt{m}$ que nos permite definir problemas desde sus *sentencias*.

Lenguajes Imperativos para la descripción de Soluciones

Las *soluciones* se expresan por medio de *programas*, ahora escritos en un lenguaje algorítmico dado L_A que, además de su sintaxis, tiene semántica dada por una función $\mathtt{u}$. El rol de esta función es, como en caso de $\mathtt{m}$ para problemas, asociar cada (texto de) *programa* **Impl**, escrito en lenguaje L_A, a la α*-función* $\mathtt{d}$ = $\mathtt{u}$ [**Impl**] calculada por el programa cuya restricción es definida por la precondición expresada en el texto **Impl.**
Como en caso de los problemas, es importante distinguir *programas* de *funciones*: una *función* es un objeto matemático abstracto e ideal, mientras un *programa* es un objeto lingüístico concreto (en el sentido de que consiste de un conjunto de

símbolos o diagramas). La conexión entre ambos ocurre por medio de la función semántica u. O sea, un programa es una descripción de un algoritmo que calcula una *a-función*.

4.3. La Teoría Intuitiva de Problemas como un Fundamento para Lenguajes de Transformación de Modelos

El Lenguaje QVT, que fue introducido en el capítulo 2, es un objeto lingüístico concreto para expresar transformaciones de modelos con una naturaleza híbrida (declarativa / imperativa), por lo que permite a los desarrolladores expresar *problemas* así como *soluciones* en el dominio de las transformaciones de modelos. En esta sección explicaremos la conexión entre el lenguaje QVT estándar y la teoría de problemas.

4.3.1 QVT Declarativo vs. QVT imperativo

Recordemos brevemente los conceptos principales de QVT, ya descriptos más detalladamente en las secciones 2.2 y 2.3 del Capítulo 2:

❖ La componente declarativa de QVT (específicamente el ***Lenguaje Relations***) permite la creación de especificaciones declarativas de relaciones entre modelos MOF. En QVT declarativo una transformación define cómo un conjunto de modelos puede ser transformado en otro. Contiene un conjunto de relaciones, que son las unidades básicas de especificación de comportamiento de la transformación en el lenguaje Relations.
Una relación se define por dos o más dominios que especifican los elementos de modelos que deben estar relacionados, una cláusula *when* que especifica las condiciones bajo las cuales la relación tiene que aplicarse, y una cláusula *where* que especifica la condición que deben satisfacer los elementos de los modelos que están siendo relacionados.

❖ Además del lenguaje declarativo existe un lenguaje operacional (***Operational Mappings***) para invocar las implementaciones imperativas de transformaciones.
Este lenguaje provee extensiones de OCL con efectos laterales que permiten un estilo más procedural, y una sintaxis concreta similar a la sintaxis de lenguajes de programación imperativos. Una transformación operacional consiste principalmente de una lista de *mapping operations*, operaciones que implementan una correspondencia entre elementos del modelo fuente y elementos del modelo destino. Una operación *mapping* siempre es el refinamiento de una *relation* (declarativa) la cual es dueña de las cláusulas *when* y *where* que la operación debe cumplir.

Así, cuando nos referimos al lenguaje de transformación QVT, debemos tener en cuenta dos clases diferentes de construcciones lingüísticas:
- las declaraciones de relaciones escritas en el lenguaje QVT declarativo, que denotan problemas en términos de la teoría de problemas,

- y las descripciones de *mappings* escritas en el lenguaje QVT operacional, que denotan soluciones en términos de la teoría de problemas. La figura 4-2 muestra la conexión entre los dos niveles lingüísticos de QVT y la teoría de problemas.
La función *m* define la semántica de expresiones declarativas en términos de *problemas*, mientras que la función u define la semántica de construcciones imperativas en términos de *soluciones*.
En las siguientes secciones presentamos la definición formal de ambas funciones semánticas.

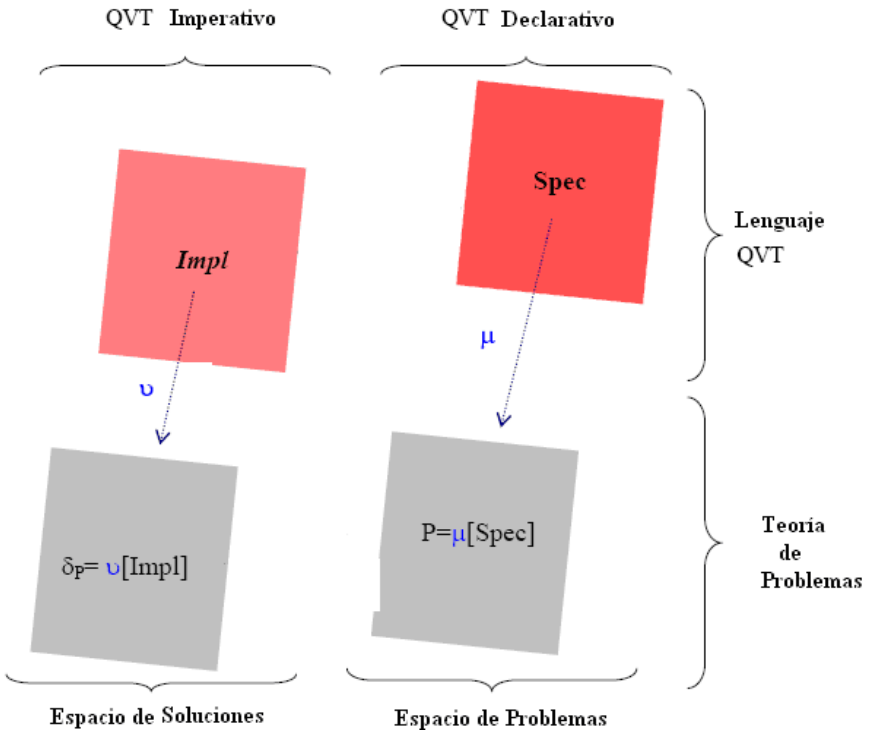

Figura 4-2. Semántica de QVT en términos de la Teoría de Problemas.

4.3.2 Semántica de Lenguajes para Transformación de Modelos

Dado que el lenguaje SQVT presentado en el Capítulo 3, minimiza al lenguaje QVT, definiremos las funciones semánticas sobre este lenguaje minimal para transformaciones. En forma trivial, pueden extenderse estas funciones semánticas al lenguaje QVT estándar.
Para la definición de las funciones semánticas elegimos utilizar el lenguaje de especificación OCL [16] en vez de algún otro lenguaje de lógica de primer orden, por ser un lenguaje estándar ampliamente conocido por la comunidad dedicada al área del modelado; es además suficientemente expresivo y de uso amigable.

Semántica de SQVT Relational (Declarativo)

En esta sección presentamos la semántica del lenguaje declarativo para especificar transformaciones, cuya sintaxis abstracta está especificada mediante

metamodelos. La Figura 4-3 nos recuerda el metamodelo de las transformaciones declarativas propuesto: SQVT Relational, que ya fuera presentado en el Capítulo 3.

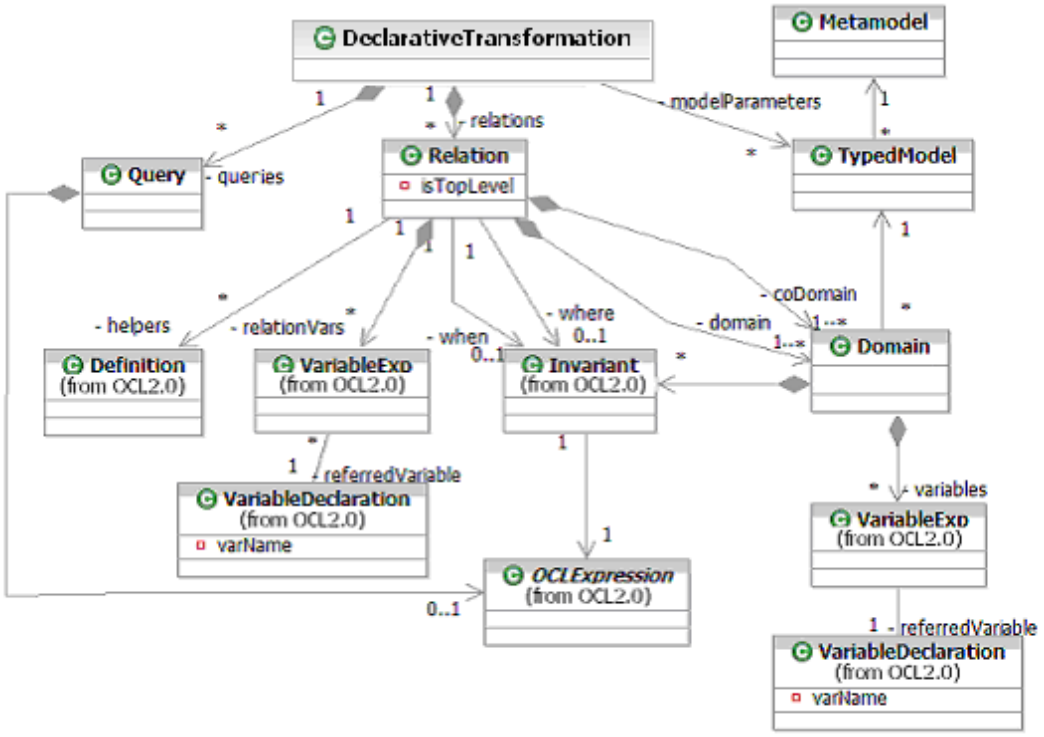

Figura 4-3. Metamodelo de las transformaciones declarativas.

Como ya analizamos, una transformación **T** se compone básicamente por relaciones, por lo tanto la semántica de **T** se definirá en términos de la semántica de sus relaciones componentes.
Analicemos la función semántica ***μ*** aplicada sobre **T**, una especificación de transformación escrita en SQVT Relational. Esta función interpreta al lenguaje relacional en el espacio de problemas de la Teoría de Problemas.

μ: DeclarativeTransformation -> PROBLEM

Veamos primero como se define la semántica de cada *Relation* de la transformación y luego como definirla para toda la *Transformation*.

μ: Relation -> PROBLEM

El dominio semántico ***PROBLEM*** es una tupla <D, R, q, I> y lo especificamos en OCL, de la siguiente forma:

PROBLEM = TupleType (dom: Set(NamedElement), res: Set(NamedElement), q: Set (TupleType (d: NamedElement, cod: NamedElement)), i: Set(NamedElement))

Sea r una Relation:

μ (r).dom = r.domain().variables.referredVariable.type.allInstances

El dominio de datos de la relación es el conjunto de todas las instancias del tipo de la variable[5] definida en el dominio

μ (r).res = r.coDomain().variables.referredVariable. type.allInstances

El dominio de resultado de la relación es el conjunto de todas las instancias del tipo de la variable definida en el coDominio.

μ (r).q = μ (r).dom -> product (μ (r).res) -> select (Tuple { d , cod } | r.where().bodyExpression [r. domain().variables.referredVariable.varName / d.name] [r. coDomain().variables.referredVariable.varName / cod.name])

La condición q de la relación es el conjunto de pares de instancias del dominio y codominio respectivamente que satisfacen la cláusula *where* de la relación, aplicando la sustitución de nombres de variables correspondiente.

μ (r).i = μ (r).dom -> select (d | r.when().bodyExpression [r. domain().variables.referredVariable.varName / d.name])

El dominio de interés de la relación es el conjunto de instancias de la variable definida en el dominio que satisfacen la cláusula *when* de la relación, aplicando la sustitución de nombres de variables correspondiente.

Luego de haber definido la semántica de las expresiones de tipo Relation, estamos en condiciones de definir la semántica de las expresiones de tipo DeclarativeTransformation, de la siguiente forma:

μ: DeclarativeTransformation -> PROBLEM

Se define:

μ (t).dom = t.relations() -> select (r : Relation | r.isTopLevel) -> collect (r: Relation | μ(r).dom) -> flatten( ) -> asSet
μ (t).res = t.relations () -> select (r : Relation | r.isTopLevel)-> collect (r: Relation | μ(r).res) -> flatten( ) -> asSet
μ (t).q = t.relations () -> select (r : Relation | r.isTopLevel)-> collect (r: Relation | μ(r).q) -> flatten( ) -> asSet
μ (t).i = t.relations ()-> select (r : Relation | r.isTopLevel) -> collect (r: Relation | μ(r).i) -> flatten( ) -> asSet

Es decir, cada componente de la semántica de la especificación de una transformación, es la unión de las componentes correspondientes de cada relación topLevel de la transformación.

[5] Asumimos, por simplicidad, que dominio y coDominio de cada relación están compuestos por una única declaración de variable, es decir r.domain().variables -> size() = 1 and r.coDomain().variables -> size() = 1

Semántica de SQVT Operacional

En esta sección definimos la semántica del lenguaje SQVT operacional para describir transformaciones, cuya sintaxis abstracta está especificada mediante metamodelos (ver Figuras 4-4 y 4-5 donde se recuerdan los metamodelos principales: OperationalTransformation e ImperativeOperation)[6] y cuya estructura fuera explicada en el capítulo anterior. Para la definición de esta semántica, utilizamos notación funcional, que complementa el uso de OCL, para lograr una mayor expresividad.

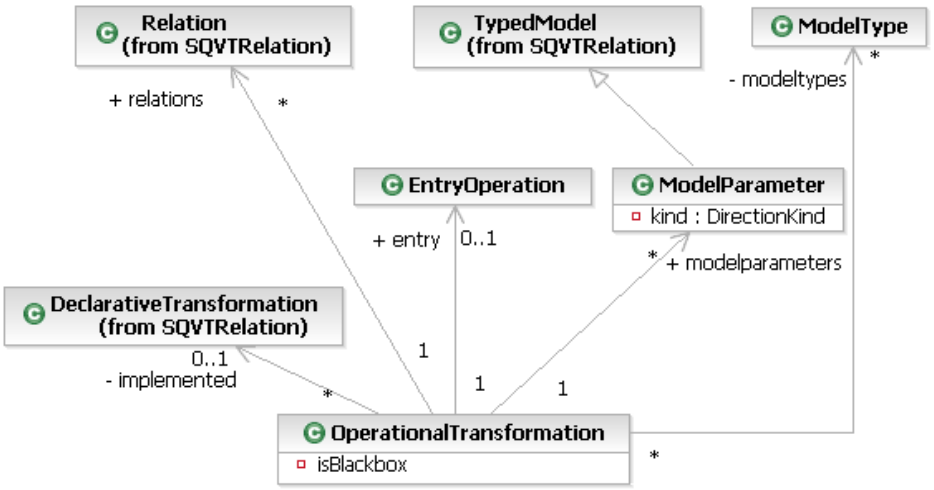

Figura 4-4. Metamodelo de las transformaciones operacionales

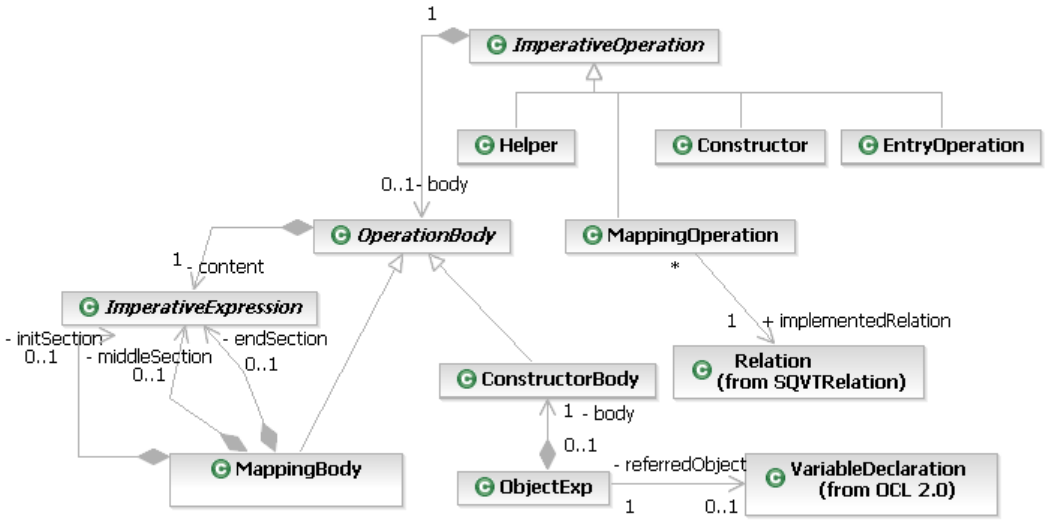

Figura 4-5. Metamodelo de las operaciones imperativas

[6] El metamodelo de las Expresiones Imperativas puede encontrarse en el capítulo 3, sección 3.4.

La semántica se define sólo para transformaciones no *blackbox* o sea, las que tienen arranque de ejecución en la operación entryOperation. Esta operación genera la ejecución del resto de las ImperativeOperations. La operación EntryOperation es especial respecto al resto de las ImperativeOperations. No tiene parámetros y en su *body* habrá llamadas (ImperativeCallExp) que provocarán la ejecución de las otras ImperativeOperations de la transformación o bien la ejecución de otra transformación (a través de una TransformationCallExp de la jerarquía ImperativeCallExp). Es decir, el resto de las operaciones no se ejecutarán aisladamente, sino a través de las ImperativeCallExp.

En los párrafos siguientes, definimos los dominios semánticos y funciones necesarias para definir la función semántica *u* para transformaciones de modelos.

Dominios semánticos
Consideramos los siguientes dominios:
Bvalue, dominio primitivo para valores básicos (por ejemplo: bool, nat)
Oids, dominio primitivo para identificadores de valores almacenables
Vars, dominio primitivo para nombres de variables
Evalue = Bvalue + Oids, dominio para valores expresables (es decir valores básicos e identificadores)
Svalue, dominio para valores almacenables (objetos)
Env = [Vars -> Evalue] dominio para el ambiente de variables. Un ambiente es una función que liga nombres de variables con valores expresables.
Store = [Oids -> Svalue] dominio para el almacenamiento (memoria) de objetos. Una memoria es una función inyectiva que liga identificadores de objetos con objetos.
*S = (**Env X Store)*** dominio para estados que representan la asignación de valores a variables. Cada estado está representado por el ambiente de ejecución (***Env***) y la memoria (***Store***).

Función semántica *I* para expresiones OCL

Utilizaremos la función semántica ***I*** definida en el "Appendix A – Semantics" de [16] para evaluar expresiones OCL originales, libres de efectos laterales. Esta función no es aplicable a la sub-jerarquía de las *OclExpression* con raíz en ***ImperativeExpression***, que producen cambios de estado.

La función se define:
I : OclExpression -> (Store X Env) -> Evalue

Es decir, ***I*** considera a los estados representados por el par Store X Env, en ese orden**.** La función u que definimos a continuación, considera a *S =(**Env X Store)***

Función semántica *u* para el lenguaje de transformaciones de modelos

La función semántica *u* se aplica a las construcciones (expresiones) del lenguaje de las transformaciones de modelos. Se especifica la semántica de dichas expresiones como funciones de estados iniciales en estados finales.

La función *u* está definida sobre la estructura jerárquica del metamodelo:

u : OperationalTransformation -> ***(Env X Store) -> (Env X Store)***
u : ImperativeOperation -> ***(Env X Store) -> (Env X Store)***
u : OperationBody -> ***(Env X Store) -> (Env X Store)***
u : ImperativeExpression -> ***(Env X Store) -> (Env X Store)***

Sean δ: ***Env***, σ: ***Store***

u (ot: OperationalTransformation) (δ,σ) = *u* (ot.entry) (δ,σ)
La semántica de la construcción OperationalTransformation es el resultado de aplicar *u* a la operación "main" (de clase EntryOperation) de la transformación, obtenida por la asociación *entry*.

u (entry: EntryOperation) (δ,σ) = *u* (entry. body) (δ,σ)
La semántica de una operación EntryOperation es el resultado de aplicar *u* al cuerpo de dicha operación, obtenida por la asociación *body*.

u (body: OperationBody) (δ,σ) = *u* (body.content) (δ,σ)
La semántica del cuerpo de una operación es el resultado de aplicar *u* al contenido del cuerpo de dicha operación, obtenida por la asociación *content*.

El contenido de un *body* se corresponde con una instancia de la jerarquía de ImperativeExpression. Consecuentemente, a partir de este punto, *u* es la función semántica aplicada a las ImperativeExpression.
En general, la ejecución de las ImperativeExpression produce cambios en el Env, no en el Store, salvo cuando se crean objetos, por ejemplo con la expresión ObjectExp.

u (e: NullExp) (δ,σ) = (δ,σ)
La semántica de la expresión NullExp es una función que no produce modificaciones ni en el Env, ni en el Store.

u (e: SeqExp) (δ,σ) = *u* (e.e2) (*u* (e.e1) (δ,σ))
La semántica de la expresión SeqExp es el resultado de aplicar *u*, en primer instancia, a la primer expresión componente de e, es decir *e1*, y luego, en ese estado modificado, aplicar *u* a la segunda componente de e, *e2*.

u (e:AssigExp) (δ,σ) = (δ [e.left / ***I*** (e.value) (σ, δ)], σ)
La semántica de la expresión AssigExp es una función que solamente modifica el Env, ligando la parte izquierda de e con el resultado de evaluar, a través de la función ***I***, la expresión dada por e.value.

u (e:IfExp) (δ,σ) = *u* (e.thenExpression) (δ,σ), ***I*** (e.condition) (σ, δ) = tt
= *u* (e.elseExpression) (δ,σ), ***I*** (e.condition) (σ, δ) = ff

La expresión IfExp funciona como la IfExp de OCL. La única diferencia es que ahora, thenExpression y elseExpression son ImperativeExpression.

u (e:WhileExp) (δ,σ) = (δ,σ) , $\quad$ $\boldsymbol{I}$ (e.condition) (σ, δ) = ff
$\quad$ = u (e) (u (e.body) (δ,σ)), $\boldsymbol{I}$ (e.condition) (σ, δ) = tt

La expresión WhileExp representa la repetición condicional, cuya definición es recursiva y depende de la evaluación de e.condition, a través de la función ***I***.

u (e:ImperativeCallExp) (δ,σ) =
$\quad$ u (e.referredOperation.body) (δ [self / $\boldsymbol{I}$ (e.source) (σ, δ)] [p_i / a_i], σ)

La expresión ImperativeCallExp funciona como la OperationCallExp de OCL. La diferencia es que ahora, la operación invocada que es obtenida por la asociación *referredOperation*, es una ImperativeExpression. En el Env se asocia la pseudovariable *self* con el resultado de evaluar la expresión receptora (e.*source*) y donde:

[p_i / a_i] es la función de sustitución de cada parámetro formal por su correspondiente argumento real. A cada argumento debe aplicársele la función semántica ***I***, para obtener su valor.
El índice i es tal que:
i = 1.. (e.referredOperation.parameter-> size())
p_i = e.referredOperation.parameter -> at(i).name
a_i = ***I*** (e. arguments -> at(i)) (σ, δ)

Particularmente, la invocación a una MappingOperation se define con una MappingCallExp (subclase de ImperativeCallExp) debido a que su estructura difiere del resto de las operaciones imperativas, como mencionamos en el Capítulo 3, puede tener initSection, middleSection y endSection. Por otro lado, TransformationCallExp es usada para invocar explícitamente a otra transformación desde el *body* de una ImperativeOperation y es también subclase de ImperativeCallExp.
Para la invocación del resto de las ImperativeOperation, se utiliza la clase ImperativeCallExp.

u (e:MappingCallExp) (δ,σ) = u (e.referredOperation.body.endSection)
(u (e.referredOperation.body.middleSection)
(u (e.referredOperation.body.initSection) (δ [self / $\boldsymbol{I}$ (e.source) (σ, δ)] [p_i / a_i], σ)
))

La semántica de una expresión MappingCallExp es el resultado de aplicar incrementalmente u a cada sección (desde la inicial a la final) del cuerpo de la operación referenciada.

u (e:TransformationCallExp) (δ,σ) = u (e.referredTransformation) (δ,σ)

La semántica de una expresión TransformationCallExp es el resultado de aplicar *u* a la transformación referenciada por la llamada, obtenida por la asociación *referredTransformation*.

Una expresión ObjectExp (ver Figura 4-5) es usada para instanciar o crear nuevos objetos. ObjectExp tiene un *body* y referencia a una VariableDeclaration (*referredObject*) de OCL, que tiene varName y type.

u (e:ObjectExp) (δ,σ) = (δ [e.referredObject.varName / id], σ [id / obj])

Donde la función next_id retorna un identificador no usado en la memoria, sea id = next_id (σ).
El identificador es ligado a la variable referenciada por la expresión, en el ambiente (δ). La nueva instancia, obj, se liga al nuevo identificador en la memoria. Obj es una instancia correspondiente al tipo de la variable referenciada, es decir:
e.referredObject.type.allInstances -> includes (obj) y cumple con *e.body*, que conforma un ConstructorBody y define la inicialización de la instancia.

4.3.3 Corrección de QVT en sus dos Niveles

Teniendo la semántica del lenguaje QVT expresada en términos de la teoría de problemas, estamos capacitados para verificar formalmente si una implementación QVT es correcta respecto a su especificación QVT o no. Tal condición de corrección se ilustra en la Figura 4-6 y se define como sigue:

Definición 1. (Condición de corrección de QVT)

Sea **Spec** una especificación de transformación escrita en QVT Declarativo, y sea **Impl** una implementación escrita en QVT Operacional. **Impl** es una implementación para **Spec** si y sólo si la función $d_{P=m}$**[Impl]** es una solución para el problema **P=m[Spec]**.

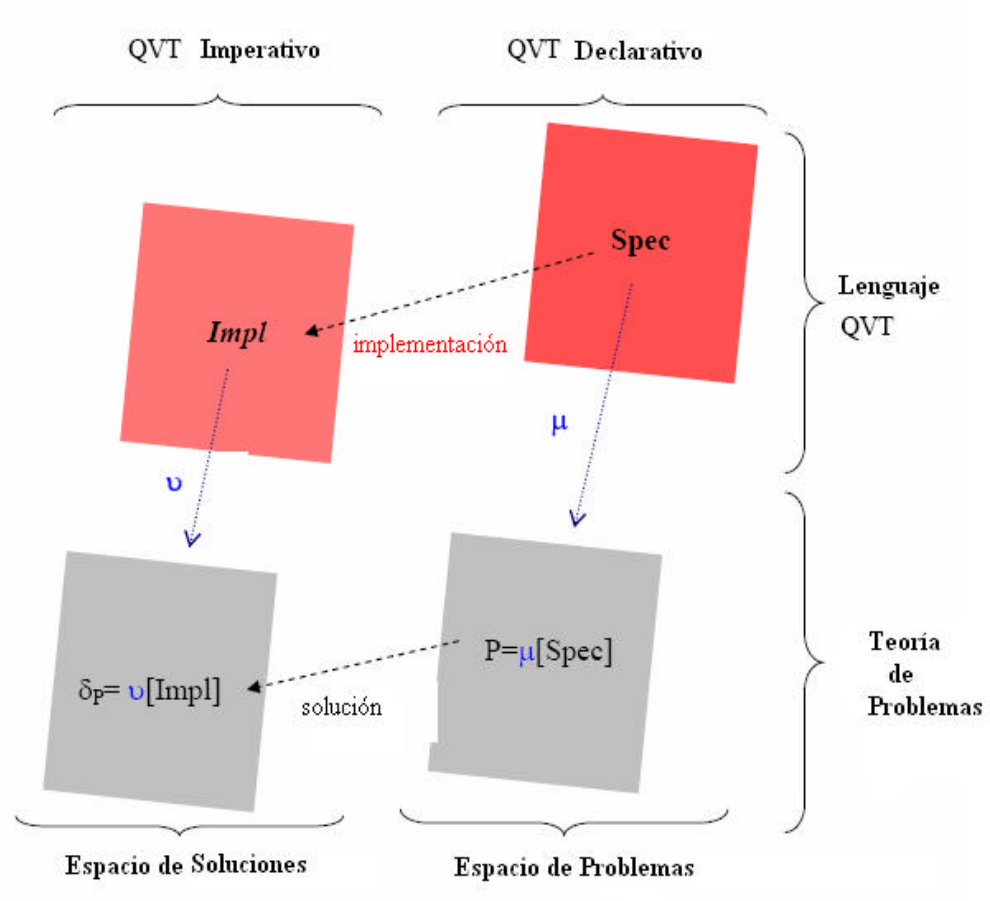

Figura 4-6. Corrección de QVT entre ambos niveles

4.4 Conclusión del Capítulo

Tanto el lenguaje para transformación de modelos QVT como otros basados en él que definen sus propios constructores, carecen de una semántica formalmente definida. Contar con la definición semántica precisa de un lenguaje lo hace más sólido y permite que probemos el cumplimiento de ciertas propiedades y condiciones de corrección.

En este capítulo hemos presentado el formalismo de la teoría de problemas; basándonos en esta teoría definimos la semántica (tanto a nivel declarativo como operacional) de los lenguajes para transformación de modelos y consecuentemente pudimos enunciar condiciones de corrección entre ambos niveles de expresión.

En el capítulo siguiente presentaremos mecanismos de composición existentes en lenguajes para transformación. Luego, aplicando la teoría algebraica de problemas, completaremos y formalizaremos estos mecanismos que carecen de definiciones precisas tanto declarativas como operacionales y formalizaremos la relación entre los dos niveles de composición.

Capítulo
5

Conceptos Básicos sobre Composición de Transformaciones

Las transformaciones pueden ser especificadas o implementadas usando diferentes herramientas y diferentes lenguajes. Pueden también manipularse como entidades de caja negra. La habilidad de organizar o componer diferentes transformaciones en manera flexible y confiable con el fin de producir el resultado requerido, es un desafío principal en MDE. En la primera sección de este Capítulo presentamos un análisis sobre la necesidad de contar con mecanismos de composición entre transformaciones. Seguidamente introducimos mecanismos de composición que incluye en su definición, el lenguaje QVT.

5.1 Análisis y Motivación del Concepto de Composición de Transformaciones

Supongamos que queremos definir una transformación que transforme todos los elementos que forman un diagrama de clases UML en elementos Java. Para simplificar esta tarea podría dividirse y asignarse subtareas a distintos grupos de trabajo. Cada uno de estos grupos desarrollará las reglas o relaciones necesarias para transformar un subconjunto de elementos de UML. En este caso, llamemos T1 y T2 a dos de estas transformaciones parciales. La cuestión que se plantea ahora es como componer estas transformaciones para obtener la transformación requerida inicialmente. Surge así la necesidad de definir operadores para combinar transformaciones. En este caso podemos llamar *unión* a la operación que se aplica a dichas transformaciones parciales para obtener una transformación más completa. Por ejemplo, la Figura 5-1 muestra inicialmente dos transformaciones, T1 y T2. T1 transforma algunos elementos del metamodelo UML en elementos Java. Análogamente, T2 transforma otros elementos del metamodelo UML en elementos Java. Llamamos $T1 \cup T2$ a la transformación resultante de unir estas dos transformaciones, la cual deberá contener las relaciones necesarias para transformar los mismos elementos fuente, en los mismos elementos destino de ambas transformaciones originales.

Supongamos ahora que se requiere definir una transformación UML2Rel que convierta elementos UML en elementos del Modelo Relacional. Contamos con $T1 \cup T2$ (definida entre UML y Java), y tenemos disponible otra transformación T3, que se aplica entre Java y el Modelo Relacional. Para obtener UML2Rel (llamada $T1 \cup T2 ; T3$ en la Figura 5-1) existen al menos dos posibilidades: la primera es creando una nueva transformación e inicializándola definiendo las relaciones necesarias para transformar cada uno de los elementos; la segunda es realizar la composición secuencial de las transformaciones ya existentes mediante

la aplicación de una operación. Como en el caso de la unión, esto resultará mucho más beneficioso.
Por último, supongamos que al modelo UML es necesario implementarlo en más de una plataforma, para cubrir tanto la representación de los elementos del dominio del problema como la persistencia de dichos elementos. En consecuencia, es deseable mantener ambas opciones de transformación. Las plataformas para este caso son el Modelo Relacional y un Modelo Orientado a Objetos. Para esto, contamos con una transformación T4 que convierte elementos UML en elementos del Modelo Orientado a Objetos. Se requiere por lo tanto, definir una transformación que permita especificar relaciones que conviertan elementos UML tanto en elementos del Modelo Relacional como en elementos del Modelo Orientado a Objetos. Resulta entonces necesario de un tercer operador, el cual llamaremos *Fork*, que produce como resultado la transformación compuesta llamada $T1 \cup T2 ; T3 \nabla T4$ en la Figura 5-1. Esta operación permitirá ampliar el codominio de la transformación resultante mediante la combinación de los metamodelos de las transformaciones originales. Por ejemplo, para una transformación de clases UML a elementos del Modelo Relacional, compuesta con otra transformación de clases UML a elementos del Modelo Orientado a Objetos, genera una relación que tenga como dominio a las clases UML y como codominio a las tuplas formadas por elementos del Modelo Relacional y elementos del Modelo Orientado a Objetos. Para poder aplicarse, las transformaciones a componer deben tener el mismo metamodelo como dominio.
Mediante el desarrollo de este ejemplo se puede ver que la utilización de operadores de composición entre transformaciones tiene la ventaja de evitar la especificación de una nueva transformación, mediante la reutilización de elementos ya definidos. Aún en este simple ejemplo, podemos reconocer tres diferentes tipos de composición entre transformaciones.

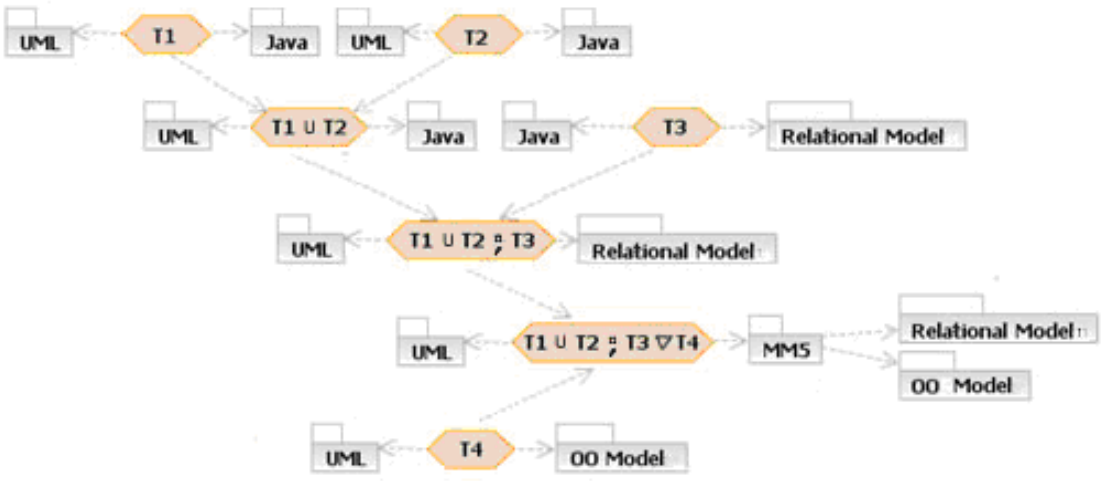

Figura 5-1. Red de transformaciones

5.2 Mecanismos de Composición en QVT

Existen varias propuestas para transformación de modelos que ofrecen formas de composicionalidad (ver [29] para un panorama general), basadas en composiciones internas o externas de transformaciones.

El lenguaje QVT en su parte operacional, cuya especificación y elementos básicos fueron introducidos en el Capítulo 2, sección.2.3, permite que las *mappingOperations* puedan ser combinadas mediante llamadas, o utilizando facilidades de reuso: herencia, *merge* y disyunción. Kleppe en [30] llama a esto composición interna (o *fine-grained*) de transformaciones.
Por otro lado, la combinación de transformaciones como entidades de caja negra, es llamada composición externa (o *coarse-grained*) de transformaciones. Al respecto, QVT operacional también incluye un conjunto de constructores de programación para expresar encadenamiento secuencial de transformaciones. Este conjunto de constructores ofrece la posibilidad de escribir *loops*, controles *if-then-else*, pasaje de parámetros a las transformaciones y la posibilidad de recuperar la salida de una transformación y pasarla como entrada a la transformación siguiente.
En las siguientes secciones presentamos en detalle los mecanismos de composición de QVT.

5.2.1 Composiciones Internas (*fine-grained*)

El lenguaje proporciona dos facilidades de reuso al nivel de *mapping operations*: herencia y *merge* de *mappings*. También incluye la disyunción entre *mappings.* Como ya mencionamos, estos 3 mecanismos son considerados composiciones internas de transformaciones.

Herencia

Una operación *mapping* puede heredar desde otra operación *mapping*. En términos de semántica de ejecución, el *mapping* heredado es ejecutado después de la sección de inicialización (init) del *mapping* que lo hereda. El ejemplo que sigue ilustra el uso de herencia de *mappings*. El *mapping* que crea columnas foráneas RDBMS hereda al *mapping* definido para crear columnas "comunes".

```
mapping Attribute::attr2Column (in prefix:String) : Column {
        name := prefix+self.name;
        kind := self.kind;
        type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR'
endif;
        }
mapping Attribute::attr2ForeignColumn (in prefix:String) : Column
inherits leafAttr2OrdinaryColumn {
        kind := "foreign";
        }
```

Merge de *mappings*

En una transformación, una operación *mapping* puede también declarar una lista de operaciones *mapping* que complementa su ejecución: esto es un *mapping merge*. En términos de ejecución, la lista ordenada de *merged mappings* es ejecutada en secuencia, después de la sección **end**.
El ejemplo siguiente muestra una definición de transformación que usa *mapping merging*. Este estilo de escritura permite definir una especificación modular donde se puede incluir en la transformación a lo sumo una operación *mapping* por cada regla definida en lenguaje natural.

```
// Regla 1: una Class de UML debe ser transformada en una JavaClasss y una
Table del Modelo Relacional.
// El nombre de la JavaClass y de la Table es el mismo que el de la Class.
mapping Class::class2JavaClass2Table () : jc: JavaClasss, t: Table
merges class2Table, class2JavaClass
{
}
// Regla 2: Si la Class es persistente, sus atributos pasan a ser columnas de la
Table.
mapping Class::class2Table () : t:Table {
when {self.isPersistent();} {
object t:{ name := self.name; columns := self.attribute->map attribute2column();
}
}
// Regla 3: los atributos de la Class pasan a ser campos de la JavaClass.
mapping Class:: class2JavaClass () : jc: JavaClass {
object jc:{ name := self.name; ownedFields := self.attribute->map
attribute2field();};
}
```

Un *merged mapping* no es invocado si la guarda no se satisface. Esto ocurre en la Regla 2 para instancias de *Class* que no son persistentes.

Disyunción de *mappings*

Una mapping operation puede ser definida como una disyunción de una lista ordenada de *mappings*. Esto significa que una invocación de la operación se resuelve sobre la selección del primer *mapping* cuya guarda (coincidencia de tipo y cláusula *when*) se satisface. Si no se satisface ninguna guarda, se retorna el valor *null*.
Un ejemplo:

```
mapping UML::Feature::convertFeature () : JAVA::Element
disjuncts convertAttribute, convertOperation, convertConstructor() {}
mapping UML::Attribute::convertAttribute : JAVA::Field {
        name := self.name;
}
```

```
mapping UML::Operation::convertConstructor : JAVA::Constructor {
when {self.name = self.namespace.name;}
        name := self.name;
}
mapping UML::Operation::convertOperation : JAVA::Constructor {
when {self.name <> self.namespace.name;}
        name := self.name;
}
```

5.2.2 Composiciones Externas *(coarse-grained)*

La composición *coarse-grained* de transformaciones es una cualidad esencial en transformaciones "reales" grandes. El lenguaje permite instanciar e invocar transformaciones explícitamente.
Imaginemos que la transformación *Uml2Rdbms* requiere que el modelo fuente *uml* sea un modelo "limpio" donde todas las asociaciones redundantes fueron eliminadas. Necesitaremos extender la definición de la transformación previa invocando "facilidades de limpieza" *in-place* sobre el modelo UML antes de aplicar la transformación *Uml2Rdbms.* Esto puede lograrse mediante la siguiente definición:

```
transformation CompleteUml2Rdbms(in uml:UML,out rdbms:RDBMS)
access transformation UmlCleaning(inout UML),
extends transformation Uml2Rdbms(in UML,out RDBMS);
main() {
        var tmp: UML = uml.copy();
        var retcode := (new UmlCleaning(tmp)).transform(); // performs the
"cleaning"
        if (not retcode.failed())
        uml.objectsOfType(Package)->map packageToSchema()
        else raise "UmlModelTransformationFailed";
}
```

En este ejemplo vemos el uso de los mecanismos de reuso *access* y *extends*.
Una importación *access* se asemeja a una importación tradicional de paquetes, pero aquí la semántica de *extends* combina la importación de paquetes con la herencia entre clases. Este ejemplo también ilustra lo siguiente:
(i) permitir la ejecución *in place* de transformaciones, como *UmlCleaning*,
(ii) permitir instanciar explícitamente una transformación (a través del operador ***new***), mientras que la operación ***transform*** es usada para invocar la concreción de la transformación sobre la instancia y la operación ***failed***() es usada para testear cuando la transformación produce un error.
(iii) permitir invocar operaciones *high level* sobre modelos, como la facilidad de *cloning* (operación ***copy***).

Características avanzadas: definición dinámica y paralelismo

Cuando tratamos con procesos MDA complejos, puede ser útil permitir definir transformaciones que usan definiciones de transformaciones computadas automáticamente. Esto permite realmente que una definición de transformación sea la salida (output) de otra definición de transformación, ya que un modelo QVT puede ser representado como un modelo. El lenguaje provee una operación predefinida *'asTransformation'* que permite hacer una conversión a una definición de transformación retornando una instancia de la transformación correspondiente. La implementación de esta operación *'asTransformation'* requiere típicamente "compilar" la definición de transformación dinámicamente.
El ejemplo que sigue ilustra un uso posible de esta facilidad. La transformación *Pim2Psm* transforma un modelo PIM en un modelo PSM. Para lograr esto, el modelo PIM inicial es en primer lugar, marcado o anotado automáticamente usando un conjunto ordenado de paquetes UML que define las reglas de transformación para inferir las anotaciones, sobre la base de un formalismo arbitrario orientado a UML. Cada paquete UML definiendo una transformación, es transformado en su correspondiente especificación QVT. Al ejecutarse en secuencia, cada definición de transformación QVT resultante agrega un conjunto propio de anotaciones al modelo PIM.
Finalmente, el PIM con anotaciones es convertido a modelo PSM usando la transformación AnnotatedPim2Psm.

```
transformation PimToPsm(inout pim:PIM, in transfSpec:UML, out psm:PSM)
access UmlGraphicToQvt(in uml:UML, out qvt:QVT)
access AnnotatedPimToPsm(in pim:PIM, out psm:PSM);
main() {
transfSpec->objectsOfType(Package)->forEach(umlSpec:UML) {
var qvtSpec : QVT;
var retcode := new UmlGraphicToQvt(umlSpec,qvtSpec).transform();
        if (retcode.failed()) {
        log("Generation of the QVT definition has failed",umlSpec); return;};
        if (var transf := qvtSpec.asTransformation()) {
        log("Instanciation of the QVT definition has failed",umlSpec); return;}
        if ( transf.transform(pimModel,psmModel).failed()) {
        log("failed transformation for package spec:",umlSpec); return;}
        }
        }
```

Otra característica avanzada es el lanzamiento en paralelo de varias transformaciones.
Esto es útil cuando no hay restricciones de secuencia entre un conjunto de transformaciones *coarse-grained*. En términos de ejecución, la invocación de una transformación simplemente se comporta como el *fork* de un proceso para completar la tarea. La sincronización se hace esperando las variables retornadas por la ejecución de la transformación.
El ejemplo siguiente es un requerimiento "mágico" de transformación a PSM, el cual descompone un modelo de requerimiento en dos modelos PIM intermedios (uno para la GUI, otro para el comportamiento) y luego hace un *merge* de los dos

modelos PIM en un modelo PSM ejecutable. Los dos modelos PIM son generados en paralelo.

```
transformation Req2Psm (inout pim:REQ, out psm:PSM)
access Req2Pimgui(in req:REQ, out pimGui:PIM)
access Req2Pimbehavior(in req:REQ, out pimBehavior:PIM),
access Pim2Psm(in pimGui:PIM, in pimBehavior:PIM, out psm:PSM);
main() {
var pimGui : PIM := PIM::createEmptyModel();
var pimBehavior : PIM := PIM::createEmptyModel();
var tr1 := new Req2Pimgui(req, pimGui);
var tr2 := new Req2Pimbehavior(req, pimBehavior);
// dispara la transformación PIM GUI (en paralelo)
var st1 := tr1.parallelTransform();
// dispara la transformación PIM Behavior (en paralelo)
var st2 := tr2.parallelTransform();
self.wait(Set{st1,st2}); // espera la sincronización de ambas
if (st1.succeeded() and st2.succeeded())
then // crea el modelo ejecutable
        new Pim2Psm(pimGui,pimBehavior,psm).transform() endif
}
```

5.3 Conclusión del Capítulo

A pesar del hecho que QVT ofrece dos perspectivas de modelado – esto nos permite especificar ***que*** hace la transformación (QVT declarativo) y también ***cómo*** se concreta su ejecución (QVT operacional) – muchos de los trabajos actuales sobre transformación de modelos parecen tener esencialmente naturaleza operacional y ejecutable. Las descripciones ejecutables son necesarias desde el punto de vista de la implementación, pero desde el punto de vista conceptual, las transformaciones pueden también ser vistas como modelos descriptivos (ver [31] para un mayor análisis de esta aserción) que establecen solamente las propiedades que una transformación tiene que cumplir y omiten detalles de ejecución. En particular, respecto al problema de composición, muchas propuestas se enfocan solamente en los aspectos operacionales de la composición sin considerar su parte descriptiva, como vimos en este capítulo.

Tal visualización parcial del problema de composición es útil para ofrecer una solución razonable a un amplio rango de necesidades prácticas. Sin embargo, esto no cubre en absoluto el espectro completo del problema.

Consecuentemente, en el capitulo siguiente, describimos cómo la *teoría algebraica de problemas* puede ser aplicada como base para construir un fundamento matemático para el problema de la composición de transformaciones abarcando ambas dimensiones (declarativa y operacional).

Capítulo
6

La Teoría Algebraica de Problemas como Base Formal para la Composición de Transformaciones

En este capitulo presentamos una formalización de las operaciones de composición de transformaciones de modelos -a nivel declarativo y a nivel operacional-. Dicha formalización está definida en base el formalismo de la *teoría algebraica de problemas*, por lo que en la primera sección se introducen los conceptos algebraicos que dicho formalismo expresa en términos de problemas y soluciones (estos conceptos básicos de la *teoría de problemas* fueron introducidos en el Capítulo 4). En las siguientes secciones, se aplica esta teoría algebraica a la composición de transformaciones.

6.1 Operaciones sobre Problemas y Operaciones sobre Soluciones

Una estrategia fundamental para el desarrollo de *programas*, y en realidad para la resolución de *problemas* en general, es el paradigma de *dividir –para- conquistar*. Es decir, descomponer un *problema* en *subproblemas*, cuyas *a-soluciones* se recombinan en una *a-solución* para el *problema* original. Desde el punto de vista de la teoría de problemas, descomponer significa expresar el problema dado en términos de operaciones sobre problemas componentes, mientras que la recombinación se hace por medio de las operaciones correspondientes sobre soluciones.

Las operaciones sobre problemas son aquellas del Algebra Fork [28], un álgebra obtenida extendiendo el Algebra de Relación con un nuevo operador llamado ∇ (*fork).*

El Algebra de Relación [32] es una estructura algebraica definida con la intención de capturar las propiedades matemáticas de relaciones binarias. Es una extensión propia del álgebra Booleana de dos elementos. Matemáticamente, un Álgebra de Relación es un álgebra $A= (P(V), \cup, \cap, \varnothing, V, \neg, ;, \hookleftarrow, 1)$, tal que $(P(V), \cup, \cap, \varnothing, V, \neg)$ es un álgebra Booleana y $(P(V), ;, 1)$ es un monoide.

Las operaciones del Algebra Fork se definen en la teoría de conjuntos, como sigue:

- Unión o disyunción:	$R \cup S = \{(x,y) \setminus xRy \vee xSy\}$
- Intersección o *join*:	$R \cap S = \{(x,y) \setminus xRy \wedge xSy\}$
- Relación vacía:	$\varnothing = \{\}$
- Relacion Universal	$V = UxU$
- complemento	$\neg R = \{(x,y) \setminus xVy \wedge \neg xRy\}$
- secuencia	$R ; S=\{(x,y) \setminus \exists z.\ xRz \wedge zSy\}$
- conversa o inversa:	$R\hookleftarrow = \{(x,y) \setminus yRx\}$

- Relación identidad: $1 = \{(x,x) \setminus x \in U\}$
- *fork*: $R \nabla S = \{(x,y*z) \setminus xRy \wedge xSz\}$[7]

La definición de estas operaciones se aplica a problemas y soluciones de un modo completamente directo**.** En las siguientes secciones describiremos a dos de ellas detalladamente: la operación de unión y la operación de composición secuencial.

6.1.1 Unión de Problemas y Soluciones para la Unión de Problemas

Definición 2. Sean P y Q problemas; el problema $P \cup Q$ se define como el problema cuyas componentes son la unión de las componentes de P y Q. Es decir:

$D_{P \cup Q} = D_P \cup D_Q$
$R_{P \cup Q} = R_P \cup R_Q$
$q_{P \cup Q} = q_P \cup q_Q$
$I_{P \cup Q} = I_P \cup I_Q$

La figura 6-1 muestra un ejemplo de unión de problemas. Los problemas involucrados podrían tener diferente dominio de datos, dominio de resultados e instancias de interés, aunque estos conjuntos no necesariamente sean disjuntos. La unión de problemas es conmutativa ($P \cup Q = Q \cup P$), asociativa ($P \cup (Q \cup R) = (P \cup Q) \cup R$) e idempotente ($P \cup P = P$). Existe el elemento neutro llamado 0, que es el problema obtenido desde el conjunto vacío: $0 = \langle \varnothing, \varnothing, \varnothing, \varnothing \rangle$.

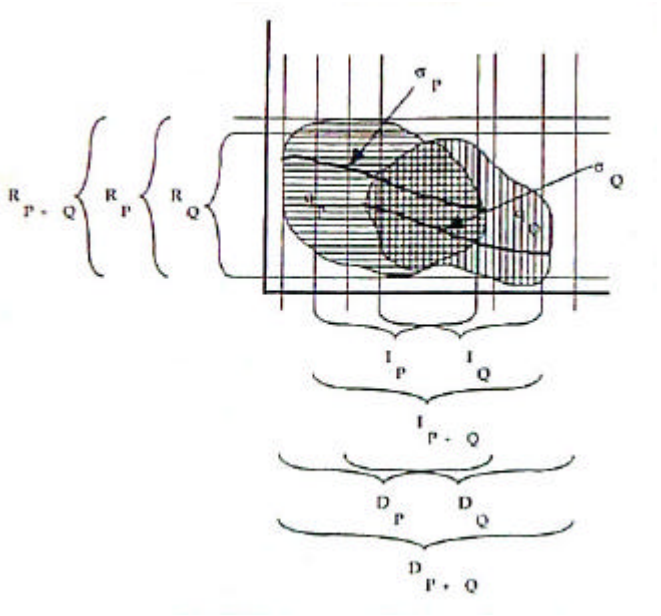

Figura 6-1. Unión de problemas

[7] La función * debe ser inyectiva. Intuitivamente construye pares de elementos. El conjunto base U es cerrado bajo*.

❖ a -soluciones para la unión de problemas

Analicemos la unión de problemas desde el punto de vista de sus α- *soluciones*. El tratamiento de este comportamiento es muy importante pues si descomponemos un problema S en dos subproblemas P y Q, tal que S = P∪Q, y si ya conocemos α−soluciones δ_P y δ_Q para P y Q respectivamente, entonces nos interesaría ser capaces de calcular α−soluciones para S en términos de P y Q.

¿Qué dice la teoría de problemas sobre el *espacio de soluciones* de la unión a partir de los *espacios de soluciones* de los términos?

Por ejemplo, sean P, Q y S los problemas en la figura 6-1, tal que S=P∪Q. Tomemos dos α−soluciones para estos problemas, $\delta_P \in \Omega_P$ y $\delta_Q \in \Omega_Q$. Si realizamos el *join* de estas soluciones -considerando funciones como conjuntos de pares- entonces el resultado no es una función porque cada elemento que pertenece al conjunto $I_P \cap I_Q$ se asociará a dos resultados (uno desde δ_P y otro desde δ_Q). Así, el concepto de unión de problemas no puede ser ampliado a la unión de soluciones de un modo directo.

Tenemos que definir una operación de unión para ser aplicada a soluciones. Esta operación $\delta_P \underline{\cup} \delta_Q$ es básicamente el *join* de δ_P and δ_Q más una elección en los puntos donde ambas están definidas, o sea:

Definición 3. Sean δ_P y δ_Q funciones; la función $\delta_P \underline{\cup} \delta_Q$ se define como
$(\delta_P \underline{\cup} \delta_Q) = \lambda d.$ `if` $\delta_P(d) = \perp$ `then` $\delta_Q(d)$ `else` $\delta_P(d)$

Consecuentemente, podemos enunciar el siguiente lema ilustrado en la Figura 6-2:

Lema 1 (unión de problemas y unión de soluciones):
Si δ_P es una solución para P y δ_Q es una solución para Q entonces $\delta_P \underline{\cup} \delta_Q$ es una solución para P∪Q.

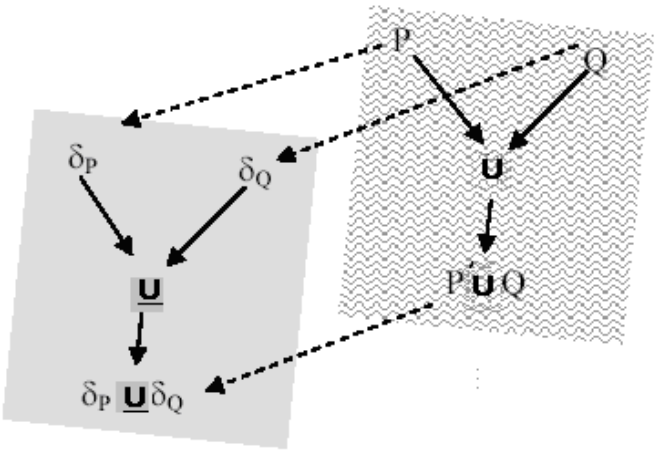

Figura 6-2. Unión de problemas y unión de soluciones

6.1.2 Composición Secuencial de Problemas y Soluciones para la Composición Secuencial de Problemas

Definición 4. Sean P y Q problemas; el problema P;Q se define como el problema cuyas componentes son:

$D_{P;Q} = D_P$. El dominio de datos es el dominio de datos del primer factor.
$R_{P;Q} = R_Q$. El dominio de resultados es el dominio de resultados del segundo factor.
$q_{P;Q} = q_P ; q_Q$. Donde $q_P ; q_Q = \{(x,y) / (\exists z) ((x,z) \in q_P \wedge (z,y) \in q_Q \}$
La condición para la composición es la composición secuencial de ambas condiciones de los factores.
$I_{P;Q} = I_P \cap D(q_P ; q_Q)$.
El conjunto de instancias de interés de la composición es la intersección del conjunto de instancias de interés del primer factor con el dominio de la condición de la composición. La composición secuencial de problemas no es conmutativa; es asociativa y distributiva a izquierda (pero no a derecha) con respeto a la unión.
Para ilustrar, la Figura 6-3 muestra la composición secuencial de los problemas P y Q (para un mejor entendimiento, la representación de Q está rotada 90^o).

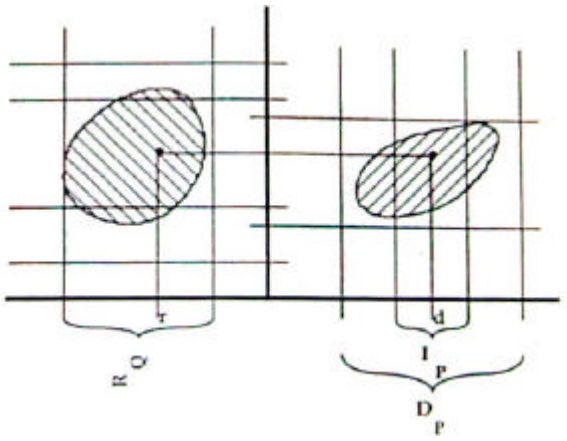

Figura 6-3. Composición secuencial de problemas

❖ **a-soluciones para la composición secuencial de problemas**

Sean P, Q los problemas en la Figura 6-3 y el problema S, tal que S = P ;Q. Tomemos dos α–soluciones para estos problemas, $\delta_P \in \Omega_P$ y $\delta_Q \in \Omega_Q$. Así, necesitamos una operación correspondiente para ser aplicada a dichas soluciones con el fin de construir α–soluciones para S. La operación $\delta_P \underline{;} \delta_Q$ se define de la siguiente manera:

Definición 5. Sean δ_P y δ_Q funciones; la función $\delta_P \underline{;} \delta_Q$ se define como:
$\delta_P \underline{;} \delta_Q (e) = \delta_Q (\delta_P (e))$

Consecuentemente, podemos enunciar el siguiente lema ilustrado en la Figura 6-4:

Lema 2 (Composición secuencial de problemas y soluciones):
Si δ_P es una solución para P y δ_Q es una solución para Q y $R_P \subseteq D_Q$ y $I_P\ q_P \subseteq I_Q$ entonces δ_P ; δ_Q es una solución para P ; Q.

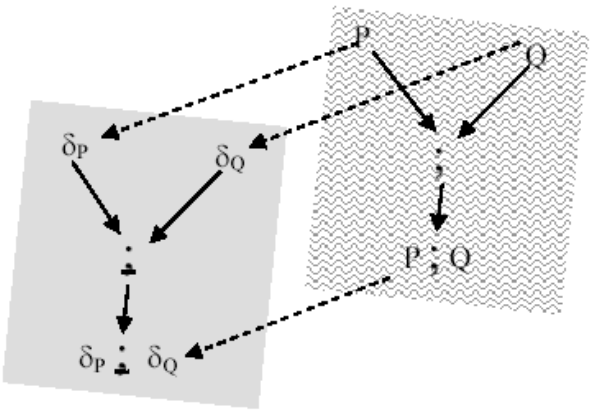

Figura 6-4. Composición secuencial de problemas y soluciones

6.2 La Teoría Algebraica de Problemas como Fundamento para Composición en Lenguajes de Transformaciones

Retomemos la idea presentada en el Capítulo 4, sección 4.2, donde planteamos la distinción entre *sentencias* de lenguajes y *problemas* de la teoría de problemas. Dijimos que los problemas se expresan por medio de *sentencias* escritas en un lenguaje declarativo L_D que tiene su propia sintaxis y una semántica dada por una función m, que permite dar un significado a las *sentencias* de problemas.

Ahora nos planteamos: ¿cuál es la relación entre construcciones lingüísticas de L_D y las operaciones de la teoría algebraica de problemas? A fin de contestar esta pregunta, supongamos que L_D es un lenguaje de especificación, por ejemplo un lenguaje lógico de primer orden. Es deseable que las operaciones sobre problemas de la teoría algebraica de problemas pudieran ser expresadas en L_D. En la lógica, tenemos un modo de combinar sentencias para obtener el efecto de la unión de problemas: el conectivo de disyunción $\vee$. Análogamente, la composición secuencial de problemas puede ser interpretada por medio de la composición de relaciones.

Por otro lado, también vimos en el Capítulo 4, que las *soluciones* se expresan por medio de *programas*, escritos en un lenguaje algorítmico dado L_A que, además de su sintaxis, tiene semántica dada por una función u. El rol de esta función es asociar cada *programa* **Impl**, escrito en lenguaje L_A, a la *α-función* d = u [**Impl**]. La conexión entre ambos ocurre por medio de la función semántica u. O sea, un programa es una descripción de un algoritmo que calcula una *α-función*. Entonces, consideremos ahora el léxico del lenguaje L_A. Supongamos que L_A es un lenguaje de programación habitual, como Java. Es deseable que las operaciones sobre α-soluciones estén expresadas en L_A. Por ejemplo, en Java, tenemos un modo de combinar programas con el fin de obtener el efecto de la

operación ; entre α-soluciones: la construcción de composición de comandos ";". Análogamente, la operación È puede ser interpretada por medio del comando condicional `if-then-else`.

6.2.1 QVT Declarativo vs. QVT Imperativo

Retomemos ahora los conceptos vertidos en el Capítulo 4, sección 4.3, donde presentamos al lenguaje de transformación QVT remarcando sus dos diferentes construcciones lingüísticas:
- las declaraciones de *relaciones* escritas en el lenguaje QVT declarativo, que denotan problemas en términos de la teoría de problemas,
- y las descripciones de *mappings* escritas en el lenguaje QVT operacional, que denotan soluciones en términos de la teoría de problemas.
En dicha sección, la figura 4-2 nos mostraba la conexión entre los dos niveles lingüísticos de QVT y la teoría de problemas, a través de las funciones semánticas m y u. Al respecto, la **Definición 1** (Capítulo 4, sección 4.3.3) enunciaba la *condición de corrección* de QVT entre sus dos construcciones lingüísticas (declarativa y operacional)

Ahora, avanzando un paso más, esta definición formal del lenguaje QVT en términos de teoría algebraica de problemas proporciona un claro fundamento para construir una estructura algebraica que soporte composiciones de transformación de modelos. De esta manera, el lenguaje QVT tendrá la capacidad para expresar problemas a solucionar así como sus propias descomposiciones, como están definidas por la teoría algebraica de problemas. O sea, el lenguaje QVT declarativo proveerá constructores lingüísticos para interpretar las operaciones sobre problemas (es decir, ∪, ;, etc.), mientras que el lenguaje operacional QVT proveerá los constructores correspondientes en cuanto a las operaciones sobre soluciones (es decir, ∪, ;, etc.). Por ejemplo, necesitamos una operación $\cup_{QVT}$ para realizar la unión de dos transformaciones declarativas, mientras por otra parte necesitamos una operación $\cup_{QVT}$ para realizar la unión de dos transformaciones operacionales.
La Tabla 1 muestra la lista de operaciones y sus correspondientes (esperadas) materializaciones en los lenguajes QVT (Declarativo y Operacional).
Las últimas tres operaciones de la Tabla (la transformación vacía, la identidad y la universal), se definen en forma trivial. Las operaciones complemento e intersección no las tratamos por no ser comúnmente usadas en la composición de problemas.

Tabla 1. Materialización del álgebra de problemas y del álgebra de soluciones en QVT

Operación	Algebra de Problemas	QVT Declarativo	Algebra de Soluciones	QVT Operacional
unión	$\cup$	$\cup_{QVT}$	$\underline{\cup}$	$\underline{\cup}_{QVT}$
secuencia	$;$	$;_{QVT}$	$\underline{;}$	$\underline{;}_{QVT}$
fork	∇	∇_{QVT}	$\underline{\nabla}$	$\underline{\nabla}_{QVT}$
intersección	$\cap$	$\cap_{QVT}$	$\underline{\cap}$	$\underline{\cap}_{QVT}$
inversa	$\hookleftarrow$	$\hookleftarrow_{QVT}$	$\underline{\hookleftarrow}$	$\underline{\hookleftarrow}_{QVT}$
complemento	$\neg$	$\neg_{QVT}$	$\underline{\neg}$	$\underline{\neg}_{QVT}$
vacío	$\varnothing$	$\varnothing_{QVT}$	$\underline{\varnothing}$	$\underline{\varnothing}_{QVT}$
identidad	1	1_{QVT}	$\underline{1}$	$\underline{1}_{QVT}$
universal	V	$\underline{V}_{QVT}$	$\underline{V}$	$\underline{V}_{QVT}$

En las siguientes secciones definimos intuitiva y formalmente, tanto a nivel problema (QVT Declarativo) como a nivel solución (QVT Operacional), las operaciones de unión (*no-determinismo*), composición secuencial (*secuenciación*) y *fork* (*paralelismo*) de transformaciones. Estas operaciones representan los tres operadores combinatorios comúnmente conocidos en la historia de la computación. También definimos la operación inversa para transformaciones.

6.2.2 Expresando Composición de Problemas en QVT

Como vimos en el Capítulo 5, no existen mecanismos para combinar transformaciones declarativas escritas en el lenguaje QVT Declarativo. Para cubrir esta carencia, hemos especificado formalmente una interpretación en QVT declarativo, para las operaciones en la Tabla 1 cuyo tratamiento fue justificado al presentar la Tabla en la sección anterior. Es decir, en las próximas secciones presentamos la unión ($\cup_{QVT}$), la composición secuencial ($;_{QVT}$), la operación *fork* (∇_{QVT}) y la operación inversa ($\hookleftarrow_{QVT}$) para transformaciones declarativas.

6.2.2.1 La unión de transformaciones declarativas

Para un mejor entendimiento de la unión de transformaciones, introducimos la especificación de dos transformaciones T1 y T2, y la transformación $T1 \cup_{QVT} T2$ resultante de aplicar la operación $\cup_{QVT}$ sobre ellas. Las especificaciones de transformaciones que usaremos en lo que sigue, fueron extraídas del Catálogo de Lano [33] y adaptadas para el desarrollo del ejemplo de composición de transformaciones presentado en la Figura 5-1 del Capítulo 5:

```
Transformation T1 (Uml:UML2.0,Java:JAVA)  {
TopLevel Relation
PersistClass2ClassWithKey {
```

```
  domain Uml c: Class
  domain Java jc: JavaClass
  when {c.isPersistent}
  where {c.name =jc.name and
   jc.ownedFields-> exists (jf: JavaField | jf.name = "primaryKey"
   and jf.type = "Integer") and jc. ownedMethods -> exists (jm:
   JavaMethod | jm.name = "getPrimaryKey" and jm.returnType =
   "Integer")  }
}}

Transformation T2 (Uml:UML2.0,Java:JAVA)  {
TopLevel Relation Class2Class {
  domain Uml c: Class
  domain Java jc: JavaClass
  when { not c.isPersistent }
  where { c.name = jc.name and
   c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
   exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods ->
   exists (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and
   Attr2Setter (p, jm2))  }
   }
Relation Attr2Getter{
   domain Uml a: Property
   domain Java jm: JavaMethod
      when { }
      where { 'get' +  p.name.firstUpperCase() = jm.name  }
}
Relation Attr2Setter{
   domain Uml p: Property
   domain Java jm: JavaMethod
   when { ! p.isReadOnly }
  where {jm.name = 'set' + p.name.
    firstToUpperCase () and jm.ownedParameter -> first().name =
    'a' + p.name. firstToUpperCase ()  and jm.ownedParameter->
    first().type = p.type}
   }
Query firstToUpperCase(string: String) : String
      { self.substring(1,1).toUpperCase()+
self.substring(2,self.size())}
}
```

Ambas transformaciones, T1 y T2 se aplican entre un modelo UML y un modelo JAVA. T1 define una única relación *topLevel*, llamada *PersistClassWithKey*. El dominio de datos de esta relación es el conjunto de clases UML. El dominio de resultados de esta relación es el conjunto de clases Java. La cláusula *when* de esta relación determina su *conjunto de instancias de interés*, el cual está restringido a todas las clases persistentes. Cuando aplicamos la transformación, por cada clase UML, se generará una clase Java con el mismo nombre. Un nuevo atributo de identidad de tipo *Integer* llamado *primaryKey*, se agrega a la clase Java. También se agrega una nueva operación llamada *getPrimaryKey* para permitir recuperar este atributo.

La transformación T2 define una única relación *topLevel*, *Class2Class*, cuyo dominio de datos es también el conjunto de clases UML y cuyo conjunto de instancias de interés está restringido a todas las clases que no sean persistentes. El dominio de resultado está integrado por clases Java. *Class2Class* invoca a otras

dos relaciones, *Attr2Getter* y *Attr2Setter*. Cuando la relación Class2Class es aplicada, por cada atributo definido en la clase original, un método *getter* y un método *setter* son agregados en el modelo Java resultante. Además de estas relaciones, la transformación incluye un *query* que convierte un *string* dado en otro con el primer carácter en mayúscula

La idea intuitiva respecto a la unión de dos transformaciones T1 y T2 es que la transformación resultado contiene la unión de las relaciones definidas en los factores. Las relaciones que no son *topLevel* aparecen en la transformación resultado sin modificación, mientras que las relaciones *topLevel* tienen que ser combinadas.

Así, *Attr2Getter* y *Attr2Setter*, que no son *topLevel*, serán parte del resultado. Lo mismo pasa con el *query* definido en la segunda transformación. Por otra parte, la operación de unión es aplicada entre las relaciones *topLevel PersistClass2ClassWithKey* y *Class2Class*, para combinarlas.

La unión de relaciones se define como sigue:

- el dominio de datos de la relación resultado es la unión de los dominios de datos de ambos factores;
- el dominio de resultado de la relación resultado es la unión de los dominios de resultado de ambos factores;
- el conjunto de instancias de interés de la relación resultado es la unión de ambos conjuntos de instancias de interés (esto se consigue construyendo la disyunción entre las cláusulas *when* de cada factor);
- la condición del problema de la relación resultado es la unión de ambas condiciones (esto se consigue construyendo la disyunción entre las dos conjunciones formadas por las cláusulas *when* y *where* de cada factor).

El resultado de aplicar la operación de unión entre T1 y T2 es el siguiente:

```
Transformation T1∪QVTT2 (Uml:UML2.0, Java:JAVA)
{
TopLevel Relation
PersistClass2ClassWithKeyÈQVTClass2Class
      {
    domain Uml c: Class
    domain Java jc: JavaClass
    when { not c.isPersistent or c.isPersistent }
   where
   {
   (not c.isPersistent AND c.name = jc.name and
   c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields-
   > exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods
   -> exists (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and
   Attr2Setter (p, jm2))  )
   OR
   ( c.isPersistent AND c.name =jc.name and jc.ownedFields->
   exists (jf: JavaField | jf.name = “primaryKey” and jf.type =
   “Integer”) and jc. ownedMethods -> exists (jm: JavaMethod |
   jm.name = “getPrimaryKey” and jm.returnType = “Integer”)
   }
  }
```

```
Relation Attr2Getter{ … }
Relation Attr2Setter{ … }
Query firstToUpperCase(string: String):String {…}
}
```

Detengámonos en notar que la unión de transformaciones declarativas podría introducir un no-determinismo en la transformación resultado. Esta situación ocurre cuando los conjuntos de instancias de interés de T1 y T2 no son disjuntos (que no es el caso del ejemplo anterior).

❖ Formalización de la unión de transformaciones declarativas

En esta sección presentamos la definición formal de las operaciones para realizar la unión de transformaciones declarativas. La definición está dada por *pre* y *post* condiciones expresadas en OCL, en el contexto de las metaclases *Transformation* y *Relation* del metamodelo de SQVT Declarativo (por simplicidad), como sigue:

Definición 6. *(Unión de transformaciones declarativas)*

```
Context Transformation:: ∪QVT(t2: Transformation): Transformation

Post:
 --El nombre de la transformación resultado es la concatenación
 --de los nombres de ambos factores con la palabra '∪QVT'
 --intercalada.
 result.name = self.name.concat ('∪QVT').concat (t2.name)

 --El conjunto de modelos tipados que especifican los tipos de
 --modelos que pueden participar en la transformación, es la
 --unión de ambos conjuntos.
 result.modelParameter=
     self.modelParameter -> union(t2.modelParameter)

 --El conjunto de relaciones no topLevel del resultado es la
 --unión de ambos conjuntos de relaciones no topLevel de cada
 --factor.
 result.relations->select(not isTopLevel)=
     (self.relations ->union (t2. relations)) ->select (not
 isTopLevel)

 --La operación Unión es aplicada sobre las  relaciones topLevel
 --para obtener su combinación.
 (self. relations ->union (t2. relations))->select(isTopLevel) -
 >forAll(r1,r2| result.relations ->
 select(isTopLevel)->includes(r1 ∪QVT r2)  )


Context Relation:: ∪QVT (r2: Relation) : Relation

Post:

 --El nombre de la relación resultado es la concatenación de los
 --nombres de ambos factores con la palabra '∪QVT' intercalada.
 result.name = self.name.concat ('∪QVT').concat (r2.name)
```

```
--El dominio de la relación resultado es la unión de los
--dominios de ambos factores.
result.domain.variables = self.domain.variables ->union
(r2.domain.variables)

--El codominio de la relación resultado es la unión de los
--codominios de ambos factores.
result.coDdomain.variables = self.coDomain.variables ->union
(r2.coDomain.variables)

--La cláusula when está formada por la disyunción entre las
--cláusulas when de cada factor. El query getExpression retorna
--la expresión correspondiente a la cláusula when, si está
--especificada, o la expresión true en caso contrario.
result.when.bodyExpression =
self.when.getExpression‘or’r2.when.getExpression

--La cláusula where se obtiene por la disyunción entre las dos
--conjunciones formadas por las cláusulas when y where de los
--factores
result.where.bodyExpression = (self.when.getExpression ‘and’
self.where.getExpression) ‘OR’ (r2.when.getExpression ‘and’
r2.where.getExpression)

--Los helpers se obtienen por la unión de los helpers de ambos
--factores
result.helpers = self.helpers ->union (r2.helpers)

--Las variables globales se obtienen por la unión de las
--variables globales de ambos factores.
result. relationVars = self.relationVars -> union (r2.
relationVars)
```

Al unir dos relaciones, suponemos que los nombres de variables de dominio de la relación receptora y de r2 coinciden así como los de las variables de codominio de ambas relaciones. En caso contrario, las variables de r2 deben renombrarse para coincidir, modificando las ocurrencias de dichas variables en las cláusulas *when/where* cuando sea necesario.

6.2.2.2 La composición secuencial de transformaciones declarativas

La idea intuitiva de componer en secuencia dos transformaciones es generar una transformación compuesta cuya aplicación sea equivalente a aplicar primero una transformación T1 y a ese resultado, aplicarle una segunda transformación T2. Solo los elementos del modelo de entrada de T1 que tengan como salida elementos del modelo de entrada de T2, pertenecerán a la transformación compuesta. Aquellos que no tengan elementos que los conecten no pertenecerán al resultado.
Para un mayor entendimiento de la composición secuencial introducimos la transformación T3 definida con una única relación *topLevel* llamada *Class2Table*. El dominio de datos es el conjunto de clases UML. El dominio de resultado es el conjunto de clases JAVA. La cláusula *when* de la relación es vacía, por lo que no hay restricciones al definir el dominio de interés. *Class2Table* invoca a la relación

Attr2Col que se aplica a cada atributo simple (indicado por la cláusula *when*) introduciendo en la clase JAVA una columna con igual nombre y tipo.

```
Transformation T3 (Java: JAVA, Rel: RDBMS)  {
TopLevel Relation Class2Table {
      domain Java jc: JavaClass
      domain Rel t: Table
      when { }
      where {jc.name = t.name and jc.ownedFields -> forAll
(jf:JavaField | t.column -> exists (co | Attr2Col (jf, co))
   Relation  Attr2Col {
   domain Java jf: JavaField
   domain Rel co: Column
   when {not jf.isMultivalued()}
   where { co.type = jf.type and co.name = jf.name }
}
```

Como en otros casos de composición, al componer dos transformaciones, se analizan las relaciones que las componen. Siguiendo con el ejemplo planteado en el capítulo 5, si componemos en secuencia la transformación obtenida más arriba llamada $T1 \cup_{QVT} T2$ con T3, por cada relación de la transformación $T1 \cup_{QVT} T2$, se estudia su conexión con cada relación de T3. Se aplicará la operación de composición secuencial pero ahora entre relaciones; en este ejemplo las relaciones que cumplen con la condición para ser encadenadas son *PersistentClass2ClassWithKey* $\dot{E}_{QVT}$ *Class2Class* de $T1 \cup_{QVT} T2$ y *Class2Table* de T3. Se genera así una nueva relación *topLevel* que convierte clases UML en tablas del Modelo Relacional.

La composición secuencial de relaciones se define como sigue:

-el dominio de datos de la relación resultado es el dominio de la primer relación;
-el dominio de resultado de la relación resultado es el codominio de la segunda relación;
-el conjunto de instancias de interés de la relación resultado es el de la primer relación.
-la condición del problema de la relación resultado es la condición del problema de la segunda relación, donde también se debe tener en cuenta la condición del problema de la primer relación y el conjunto de instancias de interés de la segunda relación.

Es decir, en la relación compuesta, aunque los elementos intermedios no aparecen ni en el dominio de datos ni en el dominio de resultado, las condiciones entre ellos deben seguir cumpliéndose, por lo que la cláusula *where* de la primera relación debe cumplirse. Por esta razón debe formar parte de la cláusula *where* de la transformación resultante. Lo mismo ocurre con la cláusula *when* de la segunda relación. Estas condiciones intermedias aparecen especificadas mediante una expresión LET de OCL dentro de la cláusula *where* de la relación resultado.

En este caso, la composición secuencial resultará en la siguiente transformación:

```
Transformation T1∪QVTT2;QVTT3 (Uml: UML2.0, Rel: RDBMS)  {
```

```
TopLevel Relation PersistentClass2ClassWithKey∪QVTClass2Class;QVT
Class2Table {
      domain Uml c: Class
      domain Rel t: Table
      when {not c1.isPersistent or c1.isPersistent }
      where {
LET jc: Class =
(not c.isPersistent AND c.name = jc.name and
c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
 exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods ->
 exists (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and
 Attr2Setter (p, jm2))  )
OR
( c.isPersistent AND c.name =jc.name and jc.ownedFields-> exists
 (jf: JavaField | jf.name = "primaryKey" and jf.type = "Integer")
 and jc. ownedMethods -> exists (jm: JavaMethod | jm.name =
 "getPrimaryKey" and jm.returnType = "Integer")
AND true
IN
jc.name = t.name and jc.ownedFields -> forAll (jf:JavaField |
t.column -> exists (co | Attr2Col (jf, co)) }
```

❖ Formalización de la composición secuencial de transformaciones declarativas

En esta sección presentamos la definición formal de las operaciones para realizar la composición secuencial de transformaciones declarativas. Como en la formalización de la Unión, la definición está dada por *pre* y *post* condiciones expresadas en OCL, en el contexto de las metaclases *Transformation* y *Relation* del metamodelo de SQVT Declarativo, como sigue:

Definición 7. *(Composición secuencial de transformaciones declarativas)*

```
Context Transformation:: ;QVT(t2: Transformation): Transformation

Post:

 --El nombre de la transformación resultado es la concatenación
 --de los nombres de ambos factores con la palabra  ';QVT'
 --intercalada.
 result.name = self.name.concat (';QVT').concat (t2.name)

 --El conjunto de modelos tipados que participan en la
 --transformación es la unión de los modelos de entrada de la
 --receptora con los de salida de t2.
 result.modelParameter= self.relations.domain.typedModel->
 union(t2. relations.coDomain.typedModel)

 --La composición secuencial es aplicada sobre las relaciones
 --topLevel de la receptora con las de t2, que sea posible
 --combinar. Las no topLevel se incluirán en las cláusulas
 --correspondientes para completar los cálculos intermedios.
 (self.relations ->select (isTopLevel) ->forAll(r1 |
 t2.relations->select (r2 | isTopLevel and r1.connectedWith(r2))
 -> forAll(r2 | result.relations ->select (isTopLevel)->
 includes(r1 ;QVT r2) ) )
```

donde la definición booleana *connectedWith* retorna verdadero si la relación receptora se puede componer en secuencia con la relación parámetro. Dos relaciones se pueden conectar o combinar cuando coincide el metamodelo del codominio de la relación receptora con el metamodelo del dominio de la relación parámetro. También debe coincidir (o ser subtipo) el tipo de variable del codominio de la relación receptora con el tipo de variable del dominio de la relación parámetro.

```
Context Relation
def: connectedWith(r2: Relation): Boolean =
self.codomain.typedModel.metamodel = r2.domain.typedModel. metamodel
and self.coDomain.variables.type.oclIsKindOf(r2.domain.variables.type)
```

```
Context Relation:: ;QVT (r2: Relation) : Relation

Post:
--El nombre de la relación resultado es la concatenación de los
--nombres de ambos factores con la palabra ';QVT' intercalada.
result.name = self.name.concat (';QVT').concat (r2.name)

--El metamodelo del dominio de la relación resultado coincide -
--con el metamodelo del dominio de la relación receptora
result.domain.typedModel.metamodel =
self.domain.typedModel.metamodel

-- El metamodelo del codominio de la relación resultado coincide
--con el metamodelo del codominio de la relación parámetro.
result.coDomain.typedModel.metamodel =
r2.coDomain.typedModel.metamodel

--Las variables de dominio coinciden con las variables de la
--relación receptora. Las variables del codominio coinciden con
--las variables de la relación parámetro.
result.domain.variables = self.domain.variables
result.coDomain.variables = r2.coDomain.variables

--La cláusula when de la relación resultado es la cláusula when
--de la relación receptora.
result.when.bodyExpression = self.when. getExpression

--La cláusula where de la relación resultante se forma con el
--where de la relación parámetro. Como parte de esta cláusula
--se define, diante una expresión LET de OCL, la conjunción de
--la cláusula where de la relación receptora con la cláusula
--when de la relación parámetro.
result.where. bodyExpression =
`LET´ self.codomain.variables. first.referredVariable =
self.where.getExpression `and´ r2.when. getExpression
`IN´ r2.where.getExpression

--Los helpers se obtienen por la unión de los helpers de ambos
--factores
result.helpers = self.helpers ->union (r2.helpers)

--Las variables globales se obtienen por la unión de las
--variables globales de ambos factores.
```

```
result. relationVars = self.relationVars -> union (r2.
relationVars)
```

Al componer en secuencia dos relaciones, suponemos que los nombres de variables de dominio de la relación r2 coinciden con los de las variables de codominio de la relación receptora. En caso contrario, se deben renombrar como se especificó en la unión.

6.2.2.3 La operación *fork* de transformaciones declarativas

Esta operación tiene como precondición que las transformaciones se apliquen sobre el mismo metamodelo de entrada. La idea intuitiva al componer dos transformaciones T1 y T2 mediante un *fork* es que la transformación compuesta mantenga como entrada los elementos de entrada en común de T1 y T2 y como salida los elementos de salida de ambas transformaciones. Es decir, componer dos transformaciones para lograr codominios de transformación más amplios, para un mismo elemento del dominio. Como en los otros casos, debemos analizar como componer las relaciones. La idea es que se compongan aquellas relaciones de cada transformación que se apliquen sobre el mismo dominio. El codominio se forma con los codominios de ambas relaciones, dando así la posibilidad de que a un mismo elemento del dominio, le correspondan dos elementos en el codominio.
Continuando con el ejemplo del Capitulo 5, consideremos la transformación T4 que convierte mediante una relación *topLevel*, clases UML a clases del Modelo Orientado a Objetos:

```
Transformation T4 (Uml: UML2.0,oodbms: OODBMS)  {
TopLevel Relation Class2OODBMSClass {
      domain Uml c: Class
      domain oodbms c2: Class
      when { }
      where {c.name = c2.name and c. ownedAttributes -> forAll (p:
Property |  if (p.type.oclIsKindOf(Datatype)) then c2.attributes -
> exists (a: Attribute | p.name = a.name and p.type = a.type) and
c2.attributes -> exists (a:Attribute | a.name = "OID" and a.type =
String) }
```

En la transformación resultante, el metamodelo de entrada coincide con el metamodelo de las transformaciones originales, mientras que el metamodelo de salida se define como un metamodelo representado por una tupla compuesta por los metamodelos de salida de las transformaciones originales. Los modelos parámetros de salida de ambas transformaciones, deberán formar también una tupla, ya que ambos se mantienen.
Como en otros casos de composición, al componer dos transformaciones, se analizan las relaciones que las componen. En el *fork* de transformaciones se componen las relaciones *topLevel* definidas en las transformaciones originales. Las relaciones restantes aparecerán sin modificaciones en la transformación resultante.

La composición *fork* entre relaciones se define como sigue:

-el dominio de la relación resultado es el dominio de la primer relación (ambas deben coincidir en dominio).
-el codominio de la relación resultado es la tupla compuesta por los codominios de ambas relaciones.
-el conjunto de instancias de interés de la relación resultado es la intersección de ambos conjunto de instancias de interés (esto se logra mediante la conjunción de las cláusulas *when* de ambas relaciones)
-la condición del problema de la relación resultado es la composición de ambas condiciones (esto se logra mediante la conjunción entre las dos implicaciones formadas por el *when* y el *where* de cada relación)

Siguiendo con el ejemplo planteado en el Capítulo 5, en este caso, la aplicación de la operación *fork* entre las transformaciones $T1 \cup_{QVT} T2;_{QVT} T3$ y T4 resulta en la siguiente transformación:

```
Transformation T1∪QVTT2;QVTT3∇QVTT4(Uml:UML2.0, tupleM: TupleType
(m1:RDBMS, m2:OODBMS))
{
TopLevel Relation PersistentClass2ClassWithKey∪QVTClass2Class;QVT
Class2Table∇QVTClass2OODBMSClass {
      domain Uml c: Class
      domain tupleM td: TupleType(t: Table, c2: Class)
      when { not c.isPersistent or c.isPersistent AND true }
      where { not c.isPersistent or c.isPersistent implies
LET jc: Class =
(not c.isPersistent AND c.name = jc.name and
c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
 exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods ->
 exists (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and
 Attr2Setter (p, jm2))  )
OR
( c.isPersistent AND c.name =jc.name and jc.ownedFields-> exists
 (jf: JavaField | jf.name = "primaryKey" and jf.type = "Integer")
 and jc. ownedMethods -> exists (jm: JavaMethod | jm.name =
 "getPrimaryKey" and jm.returnType = "Integer")
AND true
IN
jc.name = t.name and jc.ownedFields -> forAll (jf:JavaField |
      t.column -> exists (co | Attr2Col (jf, co))
AND
      true implies c.name = c2.name and c. ownedAttributes ->
forAll (p: Property |  if (p.type.oclIsKindOf ( Datatype)) then
c2.attributes -> exists (a: Attribute | p.name = a.name and p.type
= a.type) and and c2.attributes -> exists (a:Attribute | a.name =
"OID" and a.type = String) }
}
```

❖ Formalizando la operación *fork* de transformaciones declarativas

En esta sección ilustramos la definición formal de las operaciones para realizar la composición *fork* de transformaciones declarativas. Como en las formalizaciones anteriores, la definición está dada por *pre* y *post* condiciones expresadas en OCL,

en el contexto de las metaclases *Transformation* y *Relation* del metamodelo de SQVT Declarativo.
Para poder representar el codominio de salida de la transformación resultante, que como vimos en la definición intuitiva, mantiene ambos codominios originales con sus respectivos metamodelos (por lo que el metamodelo resultante en este caso es una tupla), fue necesario adaptar la clase *Metamodel* de SQVT Declarativo cuyo modelado fue presentado en la Figura 3-3 del Capítulo 3. Dicha adaptación puede verse en la Figura 6-5. Como se observa en la figura, ahora contamos con metamodelos simples y otros formados por una tupla, por lo que éstos últimos también son subclase de TupleType, el cual es un tipo predefinido de OCL.

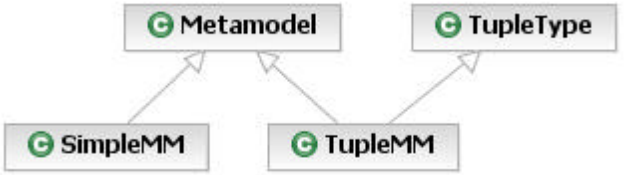

Figura 6-5. Clase *Metamodel* de SQVT Declarativo adaptada

Hechas estas aclaraciones la definición es la que sigue:

Definición 8. *(fork de transformaciones declarativas)*

```
Context Transformation:: ∇QVT(t2: Transformation): Transformation

Post:

 --El nombre de la transformación resultado es la concatenación
 --de los nombres de ambos factores con la palabra
 '∇QVT'intercalada.
 result.name = self.name.concat ('∇QVT').concat (t2.name)

 --El conjunto de modelos tipados que participan en la
 --transformación es la unión del modelo de entrada de la
 --receptora con los de salida (formando una tupla) de ambas
 --transformaciones.
 result.modelParameter= self.relations.domain.typedModel ->
 union( Tuple { self.relations.coDomain.typedModel,
 t2.relations.coDomain.typedModel})

 --La operación fork es aplicada sobre las relaciones topLevel
 --de la receptora con las de t2, que sea posible combinar. Las
 --no topLevel se incluirán sin modificaciones en el resultado.
 (self.relations ->select (isTopLevel) ->forAll(r1 |
 t2.relations->select (r2 | isTopLevel and
 r1.connectedByForkWith(r2))
 -> forAll(r2 | result.relations ->select (isTopLevel)->
 includes(r1 ∇QVT r2) ) )
```

donde la definición booleana *connectedByForkWith* retorna verdadero si la relación receptora se puede componer por *fork* con la relación parámetro. Dos relaciones se pueden conectar o

combinar cuando coincide el metamodelo del dominio de la relación receptora con el metamodelo del dominio de la relación parámetro. También debe coincidir el tipo de variable del dominio de la relación receptora con el tipo de variable del dominio de la relación parámetro.

```
Context Relation
def:
connectedByForkWith (r2: Relation): Boolean =
self.domain.typedModel.metamodel = r2.domain.typedModel.metamodel and
self.domain.variables.type = r2.domain.variables.type
```

Context Relation:: ∇_{QVT} (r2: Relation) : Relation

Post:

```
--El nombre de la relación resultado es la concatenación de los
--nombres de ambos factores con la palabra 'QVT'intercalada.
result.name = self.name.concat ('QVT').concat (r2.name)

-- El metamodelo del dominio de la relación resultado coincide
--con el metamodelo del dominio de la relación receptora
result.domain.typedModel.metamodel =
self.domain.typedModel.metamodel

--El nombre del metamodelo del codominio de la relación
--resultado es la palabra 'tupleM'
result.codomain.typedModel.metamodel.name= 'tupleM'

--El metamodelo del codominio de la relación resultado es el
--metamodelo resultante de construir una tupla con el codominio
--de la relación receptora y el codominio de la relación
--parámetro.
result.coDomain.typedModel.metamodel = Tuple {
self.coDomain.typedModel.metamodel,
r2.coDomain.typedModel.metamodel}

--Las variables de dominio coinciden con las variables de la
--relación receptora.
result.domain.variables = self.domain.variables

--Las variables del codominio forman una tupla con las
--variables del codominio de ambas relaciones originales
result.coDomain.variables.referredVariable.varName = `td´
result.coDomain.variables.referredVariable.type = TupleType
(self.coDomain.variables.referredVariable,
r2.coDomain.variables.referredVariable)

-- La cláusula when de la relación resultado es la conjunción de
--las cláusulas when de ambas relaciones
result.when.bodyExpression = self.when. getExpression `and´
r2.when.getExpression

--La cláusula where de la relación resultante se forma con una
--conjunción entre las implicaciones formadas por las cláusulas
--when y where de cada relación.
result.where.bodyExpression = (self.when.getExpression
`implies´ self.where.getExpression ) `and´
(r2.when.getExpression
```

```
   `implies´ r2.where.getExpression)

   --Los helpers se obtienen por la unión de los helpers de ambos
   --factores
   result.helpers = self.helpers ->union (r2.helpers)

   --Las variables globales se obtienen por la unión de las
   --variables globales de ambos factores.
   result. relationVars = self.relationVars -> union (r2.
   relationVars)
```

Al componer en *fork*, suponemos que los nombres de variables de dominio de la relación r2 coinciden con los de las variables de dominio de la relación receptora. En caso contrario, se deben renombrar como se especificó en la unión.

6.2.2.4 La operación inversa de transformaciones declarativas

Otra de las operaciones que forma parte del álgebra de problemas, es la operación inversa. Supongamos que se la queremos aplicar a la transformación T1 que transforma elementos UML a JAVA. Su inversa deberá transformar elementos Java en elementos UML. Todas las relaciones de T1 deberán invertirse para lograr este fin, tanto las *topLevel* como las otras.

La operación inversa de relaciones se define como sigue:

-el dominio de datos de la relación resultado es el dominio de resultados de la relación receptora.
-el dominio de resultados de la relación resultado es el dominio de datos de la relación receptora.
-el conjunto de instancias de interés de la relación resultado coincide su domino de datos (cláusula *when* vacía, indica valor *true*)
-la condición del problema de la relación resultado es la conjunción formada por la condición que cumple el conjunto de instancias de interés de la relación receptora (cláusula *when*) y la condición del problema la relación receptora (cláusula *where*).

En este caso, la operación inversa resultará en la siguiente transformación:

```
Transformation T1_QVT (Java:JAVA, Uml:UML2.0)  {
TopLevel Relation
PersistClass2ClassWithKey_QVT {
  domain Java jc: JavaClass
  domain Uml c: Class
    when { }
  }
where {c.isPersistent and c.name =jc.name and
   jc.ownedFields-> exists (jf: JavaField | jf.name = “primaryKey”
and jf.type = “Integer”) and jc. ownedMethods -> exists (jm:
JavaMethod | jm.name = “getPrimaryKey” and jm.returnType =
“Integer”)  }
}
```

❖ Formalización de la operación inversa de transformaciones declarativas

En esta sección presentamos la definición formal de las operaciones para realizar la inversa de transformaciones declarativas. La definición está dada en OCL, en el contexto de las metaclases *Transformation* y *Relation* del metamodelo de SQVT Declarativo, como sigue:

Definición 9. *(Inversa de transformaciones declarativas)*

```
Context Transformation:: ↵QVT(): Transformation

Post:

 --El nombre de la transformación resultado es la concatenación
 --del nombre de la transformación con la palabra '↵QVT'.
 result.name = self.name.concat ('↵QVT')

 --El conjunto de modelos tipados que participan en la
 --transformación resultado es mismo que el de la receptora
 result.modelParameter= self.modelParameter

 -- la inversa es aplicada sobre todas las relaciones de la
 --receptora.
 result.relations = self.relations ->collect (r | r ↵QVT )


Context Relation:: ↵QVT () : Relation

Post:
 --El nombre de la relación resultado es la concatenación del
 --nombre de la relación con la palabra '↵QVT'.
 result.name = self.name.concat ('↵QVT')

 --El dominio de la relación resultado es el codominio de la
 --relación receptora
 result.domain = self.coDomain

 --El codominio de la relación resultado es el dominio de la
 --relación receptora.
 result.coDdomain = self.domain

 -- La cláusula when de la relación resultado es siempre true.
 result.when.bodyExpression = 'true'

 -- La cláusula where de la relación resultante es la conjunción
 --del when y el where de la relación receptora.
 result.where. bodyExpression = self.when.getExpression `and´
 self.where.getExpression

 --Los helpers coinciden con los helpers de la relación
 --receptora.
 result.helpers = self.helpers

 --Las variables globales coinciden con las variables globales
 --de la relación receptora.
```

```
result. relationVars = self.relationVars
```

6.2.3 Expresando composición de soluciones en QVT

Como vimos en el Capítulo 5, los mecanismos de composición de QVT Operacional están definidos en dos niveles: la composición *coarse-grained* es la capacidad de combinar varias transformaciones completas (eventualmente pueden ser *blackBox*), mientras que la composición *fine-grained* es la capacidad de combinar transformaciones parciales (como las operaciones *mapping*).

Para el propósito de nuestro trabajo sólo consideraremos las capacidades de la composición *coarse-grained*. La composición de transformaciones *coarse-grained* es una propiedad esencial en transformaciones extensas. Para conseguir tal composición el lenguaje QVT Operacional permite que invoquemos transformaciones explícitamente, usando el operador *new()* y la operación *transform()***.** Este mecanismo de invocación se combina con el uso de los mecanismos de reutilización *access* y *extends*. Un *access* se comporta como una importación de paquete tradicional, mientras que *extends* combina la importación de paquete y el paradigma de herencia entre clases. Estos mecanismos fueron explicados con más detalle en el Capítulo 5.

Sin embargo, estas capacidades no proveen un mecanismo *blackBox* limpio para realizar la composición de transformaciones ya que continúa siendo necesario escribir el código para la transformación compuesta en vez de solamente aplicar una operación de composición. En las siguientes sub-secciones trataremos detalladamente estas cuestiones considerando las mismas operaciones tratadas para QVT Declarativo.

6.2.3.1 La unión de transformaciones operacionales

Supongamos que tenemos dos transformaciones operacionales ImplT1 e ImplT2 que implementan las transformaciones declarativas T1 y T2 respectivamente, donde T1 y T2 son las transformaciones especificadas en la sección anterior.

Quisiéramos ser capaces de calcular una implementación **Impl** para la transformación compuesta $T1 \cup_{\mathbf{QVT}} T2$ en términos de ImplT1 e ImplT2. Es decir, necesitamos constructores lingüísticos para combinar ImplT1 e ImplT2 que cumplan con la semántica de la unión de soluciones establecida por el Lema 1 (sección 6.1.1). Tal combinación puede ser realizada por el uso del constructor `if-then` que permite realizar la elección de la transformación adecuada para ser aplicada según las propiedades del elemento fuente de la transformación (en nuestro ejemplo la elección depende del valor del atributo *isPersistent*). Esto puede ser expresado del siguiente modo:

```
transformation Impl(in uml:UML2.0,out java:JAVA)
access Impl_T1(), Impl_T2();
main()
{
      var returncode := (new Impl_T1(uml,java)).transform();
```

```
      if (returncode.failed())
            then (new Impl_T2(uml,java)).transform() endif
 }
```

En vez de usar el constructor `if-then`, sería deseable contar con un mecanismo de más alto nivel para realizar la unión de transformaciones operacionales. Llamemos $\underline{\grave{E}}_{\mathbf{QVT}}$ a tal operador, es decir:

$$Impl = Impl_{T1}\ \underline{\grave{E}}_{\mathbf{QVT}}\ Impl_{T2}$$

❖ Formalización de la unión de transformaciones operacionales

La definición formal de la operación $\underline{\cup}_{\mathbf{QVT}}$ está dada por *pre* y *post* condiciones expresadas en OCL, en el contexto de la metaclase *OperationalTransformation* de SQVT Operacional, como sigue:

Definición 10. *(Unión de transformaciones operacionales)*

```
Context OperationalTransformation:: ∪QVT(t2: OperationalTransformation):
OperationalTransformation

Post:
--El nombre de la transformación resultado es la concatenación de
--los nombres de ambos factores con la palabra  '∪QVT'
intercalada.
result.name = self.name.concat ('∪QVT').concat (t2.name)

--El resultado es blackBox si y sólo si algún factor es blackBox.
result.isBlackbox = self.isBlackbox or t2.isBlackbox

--El resultado es abstract si y sólo si algún factor es abstract.
result.isAbstract = self.isAbstract or t2.isAbstract

--El cuerpo del resultado consiste de una expresión if-then que
--produce la composición coarse-grained no-deterministica de ambas
--transformaciones; donde transf1=self.name y transf2=t2.name y
--cada pi es el nombre del elemento self.modelParameter->at(i) y
--cada qi es el nombre del elemento t2.modelParameter->at(i)
result.entry.body= 'main()
                   {
                   var returnCode := (new
                   transf1(p1,..,pn)).transform();
                   if (returnCode.failed())
                   then (new transf2(q1,..,qm)).transform()endif
                    }'

--Los parámetros resultan de la unión de los parámetros de cada
--factor.
result.modelParameter=self.modelParameter->
union(t2.modelParameter)

--La transformación declarativa refinada por la transformación
--operacional resultado es la unión de las transformaciones
```

```
--declarativas refinadas de cada factor.
result.implemented = self.implemented ∪QVT t2.implemented
```

6.2.3.2 La composición secuencial de transformaciones operacionales

Supongamos ahora que contamos con la transformación operacional $\mathbf{Impl_{T3}}$ que implementa a la transformación declarativa T3, que convierte elementos Java en elementos del modelo Relacional, especificada en la sección previa. Siguiendo el ejemplo desarrollado en la composición de transformaciones declarativas, imaginemos el requerimiento de que el modelo Java de salida de la transformación operacional **`Impl`**, obtenida más arriba, sea convertido en forma directa (transparente) a un Modelo Relacional.
Para esto, como vimos en el Capítulo 5, la especificación de QVT sugiere extender la definición de la transformación **`Impl`** invocando *in-place* a la transformación $\mathbf{Impl_{T3}}$, que se aplica sobre el modelo Java de salida, obtenido al ejecutar la transformación $\mathbf{Impl_{T2}}$. Esto puede obtenerse con la siguiente definición:

```
transformation CompositeImpl (in uml: UML2.0,out rel: RDBMS)
      access Impl_T3(in java:JAVA, out rel:RDBMS) ;
      extends Impl(in uml:UML2.0,out java:JAVA);
      main()
      {
        var returncode:=(new Impl(uml, java)).transform();

        if (not returncode.failed())
           then (new Impl_T3(java, rel)).transform() endif
      }
```

De manera similar a la situación anterior, sería mejor contar con un instrumento de más alto nivel para realizar la composición secuencial de transformaciones evitando así la escritura de código para la transformación compuesta. Llamemos **`¿QVT`** a tal operador, es decir:

```
CompositeImpl = Impl ¿QVT Impl_T3
```

❖ **Formalización de la composición secuencial de transformaciones operacionales**

La definición formal de la operación **`¿QVT`** está formalmente expresada en OCL, en el contexto de la metaclase *OperationalTransformation* de SQVT Operacional, como sigue:

Definición 11. *(Composición secuencial de transformaciones operacionales)*

```
Context OperationalTransformation:: ¿QVT(t2: OperationalTransformation) :
OperationalTransformation
```

Post:

```
--El nombre de la transformación resultado es la concatenación de
--los nombres de ambos factores con la palabra  ';QVT' intercalada.
result.name = self.name.concat (';QVT').concat (t2.name)

--El resultado es blackBox si y sólo si algún factor es blackBox.
result.isBlackbox = self.isBlackbox or t2.isBlackbox

--El resultado es abstract si y sólo si algún factor es abstract.
result.isAbstract = self.isAbstract or t2.isAbstract

--El cuerpo del resultado consiste en una expresión if-then que
--produce la composición secuencial coarse-grained de ambas
--transformaciones. Donde transf1= self.name, transf2= t2.name y
--cada pi es el nombre del elemento self.modelParameter->at(i) y
--cada qi es el nombre del elemento t2.modelParameter->at(i)
result.entry.body= 'main()
                   {
                   var returnCode := (new
                   transf1(p1,..,pn)).transform();
                   if (not returnCode.failed())
                   then (new transf2(q1,..,qk)).transform()endif
                   }'
--Los parámetros de entrada del resultado son los del primer
-- factor. Los parámetros de salida del resultado son los del
--segundo factor.
result.modelParameter =  self.inputs -> union (t2.outputs)

--La transformación declarativa refinada por la transformación
--operacional resultado es la composición secuencial de las
--transformaciones declarativas refinadas de cada factor.
result.implemented = self.implemented ;QVT t2.implemented
```

```
Context OperationalTransformation
def:
inputs: Set(ModelParameter) =
      self.modelParameter -> select(p | p.kind=DirectionKind::in or
                                        p.kind= DirectionKind::inout)
outputs:Set(ModelParameter) =
      self.modelParameter -> select(p| p.kind=DirectionKind::out or
                                        p.kind= DirectionKind::inout)
```

6.2.3.3 La operación *fork* de transformaciones operacionales

Supongamos ahora que contamos con la transformación operacional $\mathbf{Impl_{T4}}$ que implementa a la transformación declarativa T4, que convierte elementos UML en elementos de un modelo Orientado a Objetos, especificada en la sección previa. Siguiendo nuevamente con el ejemplo desarrollado en la composición de transformaciones declarativas, imaginemos el requerimiento de que los elementos de entrada (clases UML) de la transformación operacional **`CompositeImpl`**, obtenida en la sección anterior, que coincidan con los elementos de entrada de $\mathbf{Impl_{T4}}$, sean convertidos, por una única transformación, tanto al elemento correspondiente del Modelo Relacional, como al elemento correspondiente del modelo Orientado a Objetos, manteniendo ambos resultados.
Para esto, como vimos en el Capítulo 5, la especificación de QVT sugiere invocar *in-place* a las transformaciones **`CompositeImpl`** y $\mathbf{Impl_{T4}}$, produciendo el

lanzamiento en paralelo de ambas transformaciones. Esto puede obtenerse con la siguiente definición:

```
transformation ParallelImpl (in uml: UML2.0, out tupleM: TupleType
(rel:RDBMS, oodbms: OODBMS))
      access CompositeImpl (in uml: UML2.0, out rel: RDBMS);
      access Impl_T4 (in uml: UML2.0, out oodbms: OODBMS);
      main()
      {
         var st1 :=(new CompositeImpl(uml, rel
         )).parallelTransform();
         var st2 :=(new Impl_T4 (uml, oodbms)).parallelTransform();
            self.wait(Set{st1,st2});
            if (st1.succeeded() and st2.succeeded())
                  then tupleM:= Tuple {rel, oodbms} endif
      }
}
```

De manera similar a las situaciones anteriores, sería más beneficioso contar con un instrumento de más alto nivel para realizar la composición *fork* de transformaciones evitando así la escritura de código para la transformación compuesta. Llamemos $\underline{\nabla}_{\mathbf{QVT}}$ a tal operador, es decir:

```
ParallelImpl = CompositeImpl ∇QVT Impl_T4
```

❖ Formalización de la operación *fork* de transformaciones operacionales

La definición formal de la operación $\underline{\nabla}_{\mathbf{QVT}}$ está formalmente expresada en OCL, en el contexto de la metaclase *OperationalTransformation* de SQVT Operacional, como sigue:

Definición 12. *(fork de transformaciones operacionales)*

```
Context OperationalTransformation:: ∇QVT(t2: OperationalTransformation) :
OperationalTransformation

  Post:

--El nombre de la transformación resultado es la concatenación de
--los nombres de ambos factores con la palabra  '∇QVT'
--intercalada.
result.name = self.name.concat ('∇QVT').concat (t2.name)

--El resultado es blackBox si y sólo si algún factor es blackBox.
result.isBlackbox = self.isBlackbox or t2.isBlackbox

--El resultado es abstract si y sólo si algún factor es abstract.
result.isAbstract = self.isAbstract or t2.isAbstract

--Los parámetros de entrada del resultado son los del primer
-- factor. Los parámetros de salida del resultado son los de ambos
--factores, formando una tupla.
result.modelParameter =  self.inputs ->union(TupleType(
```

```
self.outputs.name:self.outputs.metamodel,
t2.outputs.name:t2.outputs.metamodel))

--El cuerpo del resultado consiste en dos expresión que
--producen la composición fork de ambas transformaciones a través
--de ejecuciones en paralelo. Donde transf1= self.name,
--transf2= t2.name y cada pi es el nombre del elemento
--self.modelParameter->at(i) y cada qi es el nombre del elemento
--t2.modelParameter->at(i). Si las ejecuciones fueron exitosas, se
--construye la tupla de salida con los modelos correspondientes.
result.entry.body= 'main()
                {
                var st1:=(new
            transf1(p1,..,pn)).parallelTransform();
                var st2:=(new
            transf2(q1,..,qk)).parallelTransform();
            self.wait(Set{st1,st2});
            if (st1.succeeded() and st2.succeeded())
            then tupleM:= Tuple {self.outputs.name,
            t2.outputs.name}  endif
                }'


--La transformación declarativa refinada por la transformación
--operacional resultado es la composición fork de las
--transformaciones declarativas refinadas de cada factor.
result.implemented = self.implemented ∇QVT t2.implemented
```

6.2.3.4 La operación inversa de transformaciones operacionales

Operacionalmente, no puede definirse automáticamente la inversa de una transformación. Hay que construir una nueva transformación imperativa, que implementará a la especificación declarativa, la cual puede definirse automáticamente aplicando la operación $\hookleftarrow_{QVT}$, como ya vimos.

6.2.4 Sincronizando los Operadores de Composición en ambos Niveles

En esta sub-sección mostramos una situación sobre el beneficio que podríamos obtener contando con la maquinaria de composición precisamente definida y sincronizada en ambos niveles de QVT.
Considerando que el lenguaje QVT declarativo fue extendido en la sección anterior, con un constructor lingüístico llamado "$\cup_{QVT}$" para interpretar la unión de problemas, y dado que el lenguaje QVT operacional ha sido extendido con un constructor correspondiente en cuanto a la unión de soluciones, llamado "$\underline{\cup}_{QVT}$", estamos en condiciones de probar el siguiente lema, que se ilustra en la parte superior de la Figura 6-6:

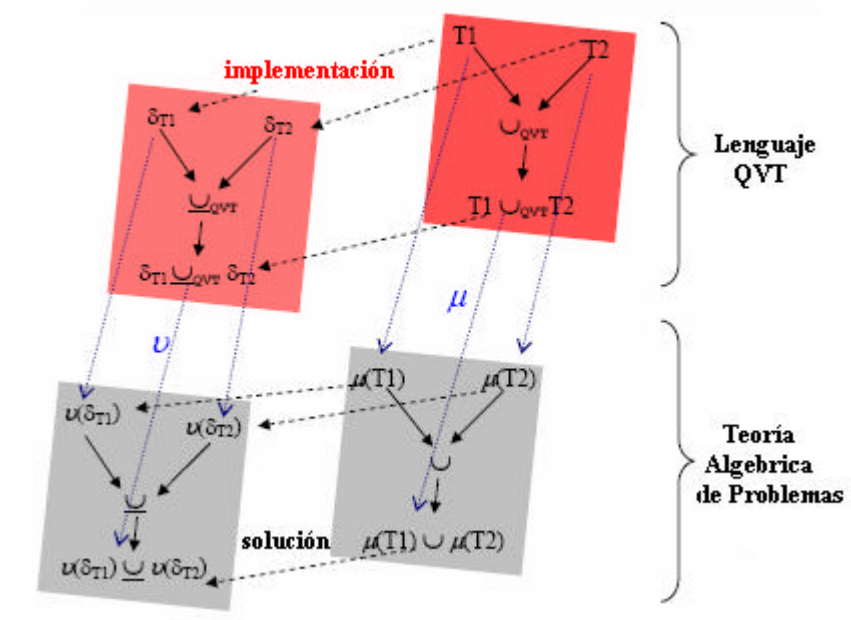

Figura 6-6. Unión de problemas y unión de soluciones en QVT.

Lema 3 *(Unión de problemas y unión de soluciones)*:
Sean T_1 y T_2 transformaciones declarativas. Si δ_{T1} es una implementación para T_1 y δ_{T2} es una implementación para T_2 entonces $\delta_{T1} \cup_{QVT} \delta_{T2}$ es una implementación para $T_1 \cup_{QVT} T_2$.

Prueba:
Partiendo de la hipótesis que δ_{T1} es una implementación para T1 y δ_{T2} es una implementación para T2 y usando la semántica formal del lenguaje QVT definida en el Capítulo 4, construimos la siguiente cadena de deducciones (que son ilustradas en la Figura 6-6):

[1]. $u(\delta_{T1})$ es una solución para m(T1) (por Definición 1 (Capítulo 4), porque δ_{T1} es una implementación para T1)
[2]. $u(\delta_{T2})$ es una solución para m(T2) (por Definición 1, porque δ_{T2} es una implementación para T2)
[3]. $u(\delta_{T1}) \cup u(\delta_{T2})$ es una solución para $m(T1) \cup m(T2)$ (por lema 1)
[4]. $u(\delta_{T1}) \cup u(\delta_{T2}) = u(\delta_{T1} \cup_{QVT} \delta_{T2})$ (por corrección de la función u)
[5]. $m(T1) \cup m(T2) = m(T1 \cup_{QVT} T2)$ (por corrección de la función m)
[6]. $u(\delta_{T1} \cup_{QVT} \delta_{T2})$ es una solución para $m(T1 \cup_{QVT} T2)$ (reemplazando [4] y [5] en [3])
[7]. Luego, $\delta_{T1} \cup_{QVT} \delta_{T2}$ es una implementación para $T_1 \cup_{QVT} T_2$ (por Definición 1).

Resultados similares pueden probarse para la interpretación en QVT de cada operación de la teoría algebraica de problemas.

6.3 Conclusión del Capítulo

En este capítulo hemos introducido las definiciones de operaciones sobre problemas y operaciones sobre soluciones en la teoría algebraica de problemas. Luego, hemos definido la aplicación de este formalismo al álgebra de las transformaciones de modelos.
Específicamente definimos intuitiva y formalmente las operaciones de unión, composición secuencial, la operación *fork* y la operación inversa para transformaciones. Algunas de estas operaciones pueden definirse y automatizarse tanto declarativa como operacionalmente, otras (como el caso de la operación Inversa) pueden especificarse formalmente en su aspecto declarativo, pero no puede automatizarse su construcción imperativa, lo cual no quita poder expresivo al álgebra de las transformaciones.
Definir en forma precisa la composición de transformaciones, nos permitió sincronizar los dos niveles de QVT es decir, es posible combinar automáticamente y de forma sincronizada, tanto especificaciones como implementaciones definidas en QVT.

En el capítulo siguiente presentamos un análisis de los casos de aplicación de las operaciones de composición. Seguidamente, describiremos un prototipo de implementación para composición de transformaciones.

Capítulo
7

Análisis de los Casos de Aplicación de las Operaciones del Algebra

En este capítulo presentamos un análisis de los casos en que las operaciones para composición de transformaciones definidas en el capítulo anterior pueden ser aplicadas en la práctica obteniendo resultados coherentes o útiles. También señalamos los casos en que la operación no puede aplicarse por que las transformaciones no cumplen con las condiciones necesarias.

7.1 Casos de Aplicación de las Operaciones del Algebra

En el Capítulo 4 vimos que el lenguaje declarativo para transformaciones puede interpretarse, a través de la función semántica m, en el espacio de problemas de la Teoría de Problemas, donde un *Problema* es una tupla <D, R, q, I>, y donde:

D es el dominio de datos de la transformación

R es el dominio de resultado (codominio) de la transformación

q es la condición de la transformación

I es el conjunto de instancias de interés de la transformación

Sean T1 y T2 transformaciones; considerando esta interpretación, podemos expresarlas mediante estas 4 componentes que representan un *problema*:

m (T1) = < D1, R1, q1, I1>
m (T2) = < D2, R2, q2, I2>

En base a esta representación, analizaremos los casos de aplicación de las operaciones del álgebra, considerando dominios, codominios y conjuntos de instancias de interés de las transformaciones sobre las que se apliquen.

7.1.1 Operación Unión

La unión representa la división del dominio de datos (de entrada) por casos (*divide and conquer*).

En la unión se toman en cuenta los dominios de las transformaciones y sus conjuntos de instancias de interés. La operación no presenta restricciones respecto al codominio de las transformaciones.
Analizando los dominios y los conjuntos de instancias de interés, los casos que se pueden considerar son:

Caso 1: Dominios coincidentes

Como dijimos, la unión es una representación de los distintos casos en que se puede dividir un dominio de datos. Por lo tanto, deberíamos poder contar con un único dominio para poder tratarlo por casos y que tenga utilidad unirlos.
Este es el **Caso 1** que planteamos para la unión: los dominios de ambas transformaciones coinciden, representando un único dominio en común que puede ser analizado y tratado por casos.
Dentro del **Caso 1,** debemos analizar que sucede con los conjuntos de instancias de interés de las transformaciones. Entonces el **Caso 1 a)** analiza que sucede cuando los dominios coinciden y los conjuntos de instancias de interés no tienen intersección, mientras que el **Caso 1 b)** trata con dominios coincidentes pero los conjuntos de instancias de interés tienen intersección no vacía.

a) Conjuntos de instancias de interés disjuntos.

En este sub-caso los dominios coinciden y los conjuntos de instancias de interés no tienen intersección.

Es decir, para T1 y T2,
$D1=D2$ y $I1 \cap I2=\varnothing$

Aquí, la aplicación de la unión, $T1 \cup T2$, amplía el dominio de interés.
Se aplica cuando el dominio del problema fue tratado en casos excluyentes y los unimos.

Por ejemplo:
D1= clases UML
D2=clases UML, los dominios de T1 y T2 coinciden

I1= clases UML persistentes
I2= clases UML no persistentes, los conjuntos de instancias de interés son disjuntos.
En este ejemplo, cada transformación se aplicará sobre los elementos correspondientes, determinísticamente, ya que T1 y T2 no se aplican a elementos en común.

En problemas recursivos, este caso puede ser útil si definimos a los dominios como caso base y caso recursivo respectivamente, es decir:
D1=caso base
D2=caso recursivo

b) Conjuntos de instancias de interés con elementos en común.

En este sub-caso los dominios coinciden y existe intersección entre los conjuntos de instancias de interés.

Es decir, para T1 y T2,
$D1=D2$ y $I1 \cap I2 \neq \varnothing$

Al aplicar la unión, en este caso, se agrega no-determinismo en los elementos que forman la intersección. Es decir, no se puede determinar que transformación se aplicó en los datos comunes, pudiendo producir distintos resultados sobre los mismos datos de entrada. En general, no tiene aplicación práctica.
Por ejemplo:
D1= clases UML
D2=clases UML, los dominios de T1 y T2 coinciden

I1= clases UML persistentes
I2= clases UML concretas
En este ejemplo, los conjuntos de instancias de interés pueden tener elementos en común, o sea, clases UML que son persistentes y concretas a la vez. Entonces no se sabrá cuál de las transformaciones, si T1 o T2, se aplica a estos elementos en común.

Caso 2: Dominios diferentes

El **Caso 2** que planteamos para la unión se presenta cuando los dominios de ambas transformaciones son diferentes, por lo tanto, las transformaciones no surgieron de dividir un dominio de datos en casos, consecuentemente resulta poco aplicable unir dominios heterogéneos.

Es decir, para T1 y T2,
$D1 \neq D2$

En general, no tiene aplicación práctica. Estaríamos construyendo una transformación sin cohesión, que transforma indistintamente elementos heterogéneos.

Por ejemplo:
D1= clases UML
D2= tablas RDBMS

7.1.2 Operación Composición Secuencial

Al operador de Composición Secuencial le interesa analizar si es posible combinar el codominio de T1 (R1) y el dominio de T2 (D2), que son los elementos que se deben componer en secuencia. Los casos que se pueden presentar son:

Caso 1: El codominio de T1 coincide con el dominio de T2

Este es el caso natural, donde la operación puede aplicarse. Es el caso donde existe conexión entre las dos transformaciones.

Es decir,
$R1 \subset D2$
Cuando el codominio de T1 está incluido o es el mismo que el dominio de T2, el resultado de combinar ambas transformaciones es el esperado.

Para analizar este caso, llamemos T3 a la transformación resultante de aplicar la secuenciación, y veamos su dominio y codominio:

Sea T3 = T1 ; T2, luego
m (T3) = < D3, R3, q3, I3>

Consideremos distinguir dos sub-casos según el efecto que produce la aplicación de la operación en el codominio de la transformación T3:

a) Efecto incremental

Se aplican las dos transformaciones en cadena y el efecto puede verse como incremental (se obtiene el efecto de ambas).
Por ejemplo, supongamos que T1 transforma Clases UML en Clases UML donde fue agregado un método setter por cada atributo de la clase, y que T2 transforma Clases UML en Clases UML donde fue agregado un método getter por cada atributo de la clase, es decir:

D1= clases UML
R1= clases UML (con métodos setters)
D2= clases UML
R2= clases UML (con métodos getters),

Entonces, en la transformación T3, resultante de aplicar la composición secuencial,
T3 = T1 ; T2 se obtendrá en el codominio el efecto incremental de ambas y se mantendrá en dominio de T1.
Es decir:
D3= clases UML
R3= clases UML (con métodos setters y getters)

Podemos notar que si en este ejemplo aplicamos la unión ($T1 \cup T2$) tendríamos una transformación no-determinística que toma una clase y no sabemos si la transformará a una clase UML con métodos setters o a una clase UML con métodos getters. Evidentemente, no se logra el efecto incremental de la secuenciación.

b) Efecto oculto

También puede ocurrir que el paso intermedio al componer dos transformaciones, quede 'oculto'.

Por ejemplo, si T1 transforma Clases UML en Clases JAVA y T2 transforma Clases JAVA en Tablas de Modelo Relacional (RDBMS), es decir:
D1= clases UML
R1= clases JAVA
D2= clases JAVA
R2= tablas RDBMS

Entonces, en la transformación resultante de aplicar la composición secuencial,

T3 = T1 ; T2 se obtendrá en el codominio el efecto final de aplicar T2.
D3= clases UML
R3= tablas RDBMS

Podemos notar que el modelo JAVA intermedio que se obtiene al aplicar T1, y que es entrada de T2, queda oculto en el resultado final de la composición.

Caso 2: El codominio de T1 no coincide con el dominio de T2

Este es el caso donde no existe conexión entre algunos elementos de las dos transformaciones.
Es decir,
$R1 \not\subset D2$
Cuando el codominio de T1 no está incluido en el dominio de T2, el resultado de combinar ambas transformaciones puede no ser el esperado, ya que habrá elementos a los que no se le aplicará la segunda transformación.

7.1.3 Operación *fork*

La operación *fork* representa la computación en paralelo. El operador *fork* requiere que coincidan los dominios de entrada de ambos elementos a combinar. Produce como resultado un par (x, y) que generalmente debe poder re-combinarse. Por ejemplo, 2 números que luego de obtenidos, puedan sumarse o compararse o, en el caso de las transformaciones, dos modelos que puedan mezclarse *(merge*) o bien compararse. Por lo tanto, deben analizarse tanto los datos de entrada de ambas transformaciones, como los datos resultantes (codominio) de aplicar la operación.

Caso 1: Los dominios de T1 y T2 coinciden, pero los conjuntos de instancias de interés son disjuntos.

En este caso no existen elementos comunes de interés, a pesar de que los dominios de T1 y T2 coinciden.

Es decir, para T1 y T2,
D1=D2
$I1 \cap I2 = \varnothing$

Por ejemplo:
D1= clases UML
D2=clases UML, los dominios de T1 y T2 coinciden
I1= clases UML persistentes
I2= clases UML no persistentes, los conjuntos de instancias de interés son disjuntos.
En este ejemplo, no existen elementos en común donde aplicar el *fork*.

En general este caso no tiene aplicación práctica. Los conjuntos de instancias de interés deben tener elementos en común para que la operación pueda aplicarse sobre ellos.

<u>Caso 2</u>: Los dominios de T1 y T2 coinciden y los conjuntos de instancias de interés tienen elementos en común.

Este es el caso aplicable en la práctica. Existen elementos en común en T1 y T2, sobre los que se puede aplicar el operador.

Es decir, para T1 y T2,
D1=D2
$I1 \cap I2 \neq \varnothing$

Sea la transformación resultante $T3 = T1 \nabla T2$, luego
m (T3) = < D3, R3, q3, I3>

En este caso, pueden analizarse en sub-casos, los codominios de ambas transformaciones y el codominio de la transformación resultante T3:

a) Codominios diferentes

Puede ocurrir que los codominios de T1 y T2 (R1 y R2 respectivamente) sean diferentes.

Es decir, para T1 y T2,
D1=D2, $I1 \cap I2 = \varnothing$ y
$R1 \neq R2$

La operación puede aplicarse; se obtiene como codominio un par con modelos de distinto tipo, que en general, no llegan a mezclarse en la práctica, pero puede tener sentido comparar ambos modelos resultantes. Puede suceder también que tenga sentido seguir transformando cada modelo obtenido individualmente.

Por ejemplo, supongamos que T1 transforma Clases UML en Tablas del Modelo Relacional y T2 transforma Clases UML en Clases del Modelo Orientado a Objetos (OODBMS), es decir:

D1= clases UML
R1= tablas RDBMS
D2= clases UML
R2= clases OODBMS

Entonces, en la transformación resultante de aplicar la operación *fork*,

T3 = T1∇T2 , se obtendrá como codominio el par formado por ambos codominios originales,
D3= clases UML
R3= (tablas RDBMS, clases OODBMS)

b) Codominios coincidentes

En este escenario, los codominios de T1 y T2 (R1 y R2 respectivamente) coinciden.

Es decir, para T1 y T2,
D1= D2, $I1 \cap I2 = \varnothing$ y
R1= R2

La operación puede aplicarse; se obtiene como codominio un par con elementos del mismo tipo, que tiene sentido re-combinar en la práctica.

Por ejemplo (ver Capítulo 5, ejemplo de Paralelismo), dos modelos PIM (uno para la GUI, otro para el comportamiento de un sistema) son generados en paralelo, mediante dos transformaciones T1 y T2, desde un modelo de requerimientos que fue dividido en dos partes. Cada parte es el modelo de entrada de T1 y T2 respectivamente.
Luego se puede hacer un *merge* de los modelos PIM obtenidos logrando un sólo modelo PIM que posteriormente podrá transformarse en un modelo PSM. Es decir:

D1= req:REQ
R1= pimGui:PIM
D2= req:REQ
R2= pimBehavior:PIM

Entonces, en la transformación resultante de aplicar la operación *fork*,

T3 = T1∇T2, se obtendrá como codominio el par formado por ambos codominios originales, que en este caso son del mismo tipo.

D3= req:REQ
R3= (pimGui:PIM, pimBehavior:PIM)

Podemos notar que en este caso, tiene utilidad práctica aplicar un *merge* sobre los componentes del par de salida, a través de aplicar una composición secuencial con la transformación constante *merge*, es decir:

Sea T4 = T3 ; *merge*
m (T4) = < D4, R4, q4, I4>

Entonces, el dominio de T4 será el de T3 y el codominio será un único modelo PIM resultante de aplicar el *merge*:
D4= req:REQ
R4= pim:PIM

La transformación constante *merge se* define como la transformación que dado un par de modelos tipados, produce el *merge* de ambos:

merge = { ((x, y), z) \ z.packageMerge.mergedPackage = Set {x, y }}

En el caso de las transformaciones, la operación *merge* se aplica sobre un par de modelos tipados. En el metamodelo SQVT los modelos tipados que estamos mezclando se representan con paquetes. En consecuencia, la operación fue definida en términos del metamodelo de UML, donde existe la metaclase packageMerge, la cual es subclase de DirectedRelationship con estereotipo <<merge>> y produce el *merge* entre paquetes (instancias de la metaclase Package).
Específicamente,
z.packageMerge.mergedPackage colecta los paquetes que fueron *mergeados* en el paquete receptor (z) a través de la asociación packageMerge.

Caso 3: Los dominios de T1 y T2 no coinciden.

Finalmente, consideremos el caso donde los dominios de T1 y T2 son distintos. No existe dominio en común y menos aún elementos en común en ambas transformaciones.
Es decir, para T1 y T2,
D1≠D2

No se puede aplicar la operación. Es precondición que coincidan los dominios de ambas transformaciones.

7.2 Conclusión del Capítulo

Este análisis realizado sobre los casos en que puedan aplicarse o tenga sentido práctico aplicar las operaciones del álgebra propuestas, representa un aporte

inicial a la definición de metodologías para la aplicación de la composición y/o descomposición de transformaciones. Constituye una ayuda tanto para el diseñador de transformaciones, como para el encargado de seleccionar que transformaciones componer y sirve al ensamblador en su actividad de composición.
En el próximo capítulo se presenta un prototipo de implementación para composición de transformaciones.

Capítulo
8

Prototipo de Implementación

Con el fin de asistir a las actividades de desarrollo de transformaciones construimos un prototipo de implementación que permita a los desarrolladores editar y almacenar transformaciones QVT atómicas. En base a este prototipo, ellos podrán construir transformaciones más complejas aplicando las operaciones de composición presentadas en este trabajo. Las transformaciones resultantes serán automáticamente "calculadas" por la herramienta. El prototipo está siendo implementado como una extensión de ePlatero [55] [56], un plug-in de código abierto para la plataforma Eclipse [57], desarrollado por nuestro grupo de investigación, que corre sobre el *framework* para metamodelado EMF [58].
En las siguientes secciones, daremos una visión general del plug-in ePlatero y una descripción del prototipo llamado Calculador de Composiciones de Transformaciones.

8.1 Implementación de la Herramienta ePlatero

Es importante contar con una herramienta automatizada que evalue las propiedades de los modelos y sirva de soporte para las técnicas de modelado. Es común encontrar editores UML; sin embargo hay pocas herramientas que permiten la evaluación de reglas semánticas. ePlatero es una herramienta CASE educativa que soporta el proceso de desarrollo de software dirigido por modelo utilizando notación gráfica con fundamento formal. Dicha herramienta es un plug-in para la plataforma Eclipse, y puede interoperar con herramientas que soporten MDA.
La arquitectura de plug-ins de Eclipse puede verse en la Figura 8-1. Permite entre otras cosas enriquecerlo con herramientas que pueden ser útiles para el desarrollo de software. Para ello, la plataforma está estructurada como un conjunto de subsistemas, los cuales son implementados en uno o más plug-ins que corren sobre un *runtime engine* (motor en tiempo de ejecución). Dichos subsistemas definen puntos de extensión para facilitar la extensión de la plataforma.

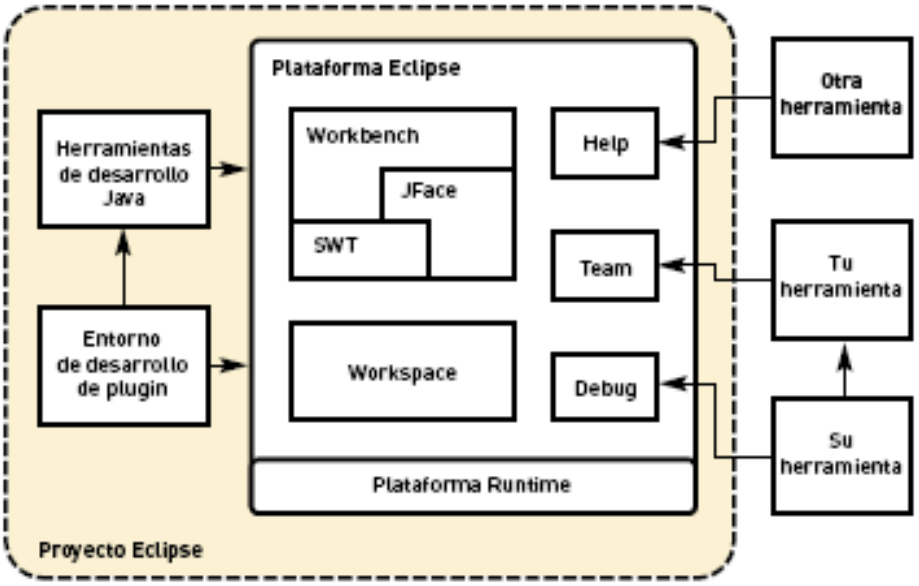

Figura 8-1. Arquitectura de Eclipse, para el desarrollo de plug-ins.

En los siguientes apartados se describe brevemente: la Arquitectura de ePlatero, el proyecto Eclipse Modeling Framework y su importancia para el desarrollo de herramientas compatibles con MOF y finalmente se presentan algunos módulos de ePlatero.

❖ Arquitectura de ePlatero

La arquitectura de ePlatero respeta el estándar de Eclipse para el desarrollo de plug-ins.

Consta de varios módulos, como muestra la Figura 8-2, los principales son:

- Coordinador
- Editor UML
- Editor de Metamodelos MOF
- Analizador léxico / Parseador
- Repositorio del proyecto
- Evaluador de fórmulas OCL
- Editor de fórmulas OCL

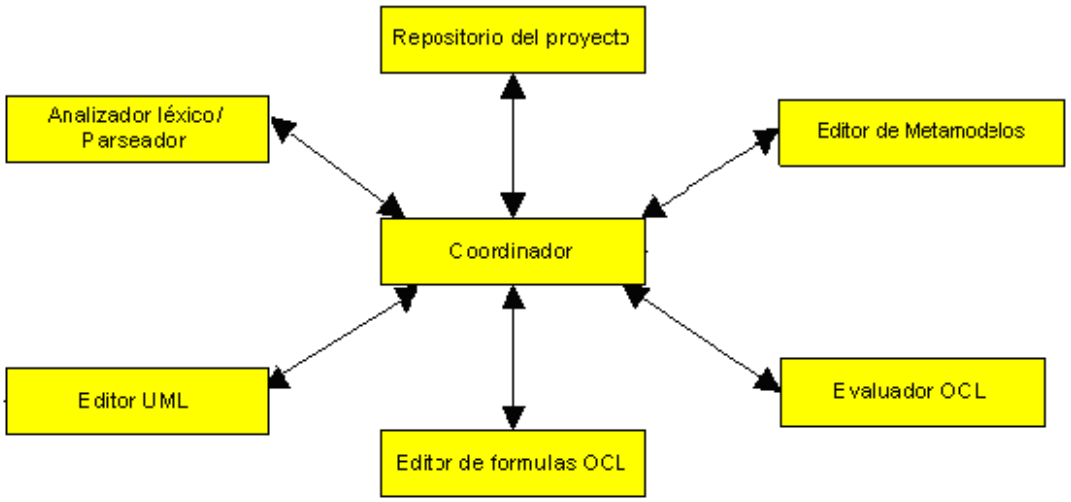

Figura 8-2. Arquitectura de ePlatero

❖ **Eclipse Modeling Framework**

El Eclipse Modeling Framework (EMF) es un framework de modelado para Eclipse y generador de código que es muy útil para el desarrollo de herramientas y aplicaciones.

EMF consiste en dos frameworks:

- **Core**: Provee la generación básica y soporte a la *runtime* para crear clases Java para un modelo.
- **Edit**: Extiende y se construye sobre el framework **Core**. Añade soporte para la generación de clases adaptadoras que observan y comandan la edición de un modelo, e incluso provee un editor básico de modelo.

EMF comenzó como una implementación de la especificación de MOF, pero luego evolucionó en base a la experiencia adquirida en la construcción de un conjunto de herramientas basadas en EMF. Sin embargo, soporta la lectura y escritura de serializaciones MOF, y está basado en un meta-metamodelo llamado Ecore equivalente a su progenitor. EMF al igual que MOF respeta el estándar de XMI facilitando de este modo el intercambio de modelos en diferentes herramientas compatibles con MOF.

Un modelo *ecore* se puede generar a partir de:

1. Diagramas de clases UML
2. Interfaces Java
3. Esquemas XML

Con cualquiera de la tres alternativas mencionadas anteriormente se utiliza un *wizard* (asistente) para transformar el modelo en un modelo *ecore* y un modelo *generator*. Este último permite generar el código de implementación JAVA correspondiente.

En el caso particular de ePlatero, se utilizó el plug-in UML2 que es una implementación realizada con EMF del metamodelo UML 2.0.
Actualmente se está usando EMF para generar cualquier metamodelo MOF que defina los tipos de modelos de entrada y salida para transformaciones.

❖ Descripción de los módulos de ePlatero

En este apartado describiremos brevemente los siguientes módulos: el Editor de UML, el Editor de fórmulas OCL, el Analizador léxico y sintáctico y el Evaluador de OCL. Pera ver el detalle de los pasos a seguir en el uso de cada módulo, sugerimos consultar la Guía de Usuario incluida en la página de ePlatero [56].

o Descripción del Editor UML

El Editor de UML permite la edición de diagramas UML. Para crear un modelo, se debe crear un nuevo Project de Eclipse.
Siguiendo los pasos del *wizard*, podemos fácilmente crear un nuevo modelo formado por dos archivos: el diagrama de clase (la vista del modelo), con extensión ePlatero, en esta caso llamado Flights y el modelo mismo, en un archivo uml, como se muestra en la Figura 8-3. El editor gráfico proporciona todos los recursos para que la herramienta nos permita crear un archivo uml. Mediante la técnica *drag and drop* (seleccionar y arrastrar) se pueden crear los diferentes elementos desde la paleta de edición que aparece en el panel derecho de la pantalla que muestra el archivo uml.
El editor cuenta con un menú de ayuda. Todos los cambios realizados en el modelo, a través del editor, serán salvados también en el archivo uml. Estos dos archivos deben estar sincronizados para que el modelo sea válido. También pueden definirse atributos (de algún tipo) en la clase. Estos tipos deben estar definidos previamente a realizar el agregado del atributo a la clase. Algo similar pasa con las operaciones.

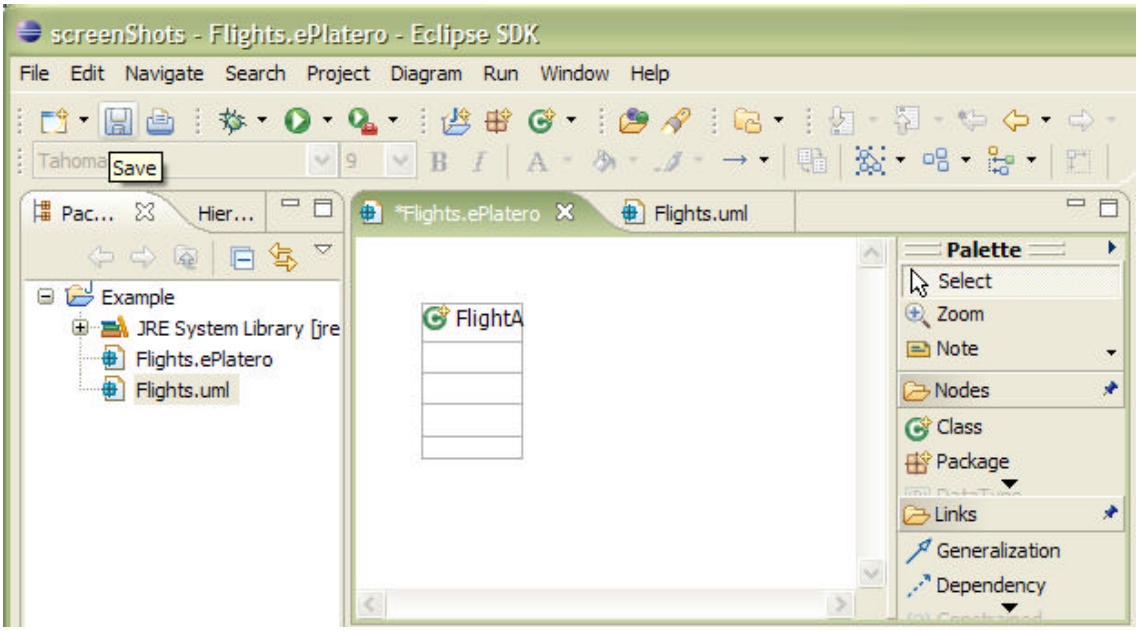

Figura 8-3. Editor de UML

o Descripción del editor de fórmulas OCL

El editor de fórmulas de OCL (Figura 8-4) se utiliza para editar los archivos OCL que contienen las reglas a evaluar sobre el modelo. Posee operaciones de edición, *syntax highlighting* (ayuda sintáctica que presenta opciones válidas respecto al contexto, durante la escritura), asistencia y corrección de errores. Posee una vista que despliega la estructura de los archivos OCL.

En Eclipse, un editor es una parte principal del área de trabajo y tiene su propio punto de extensión que permite implementar cualquier editor personalizado.

Para el desarrollo del Editor OCL se utilizó el *framework* JFace Text proporcionado por Eclipse. Dicho *framework* aporta un editor que permite la edición de documentos de texto independientemente del dominio.

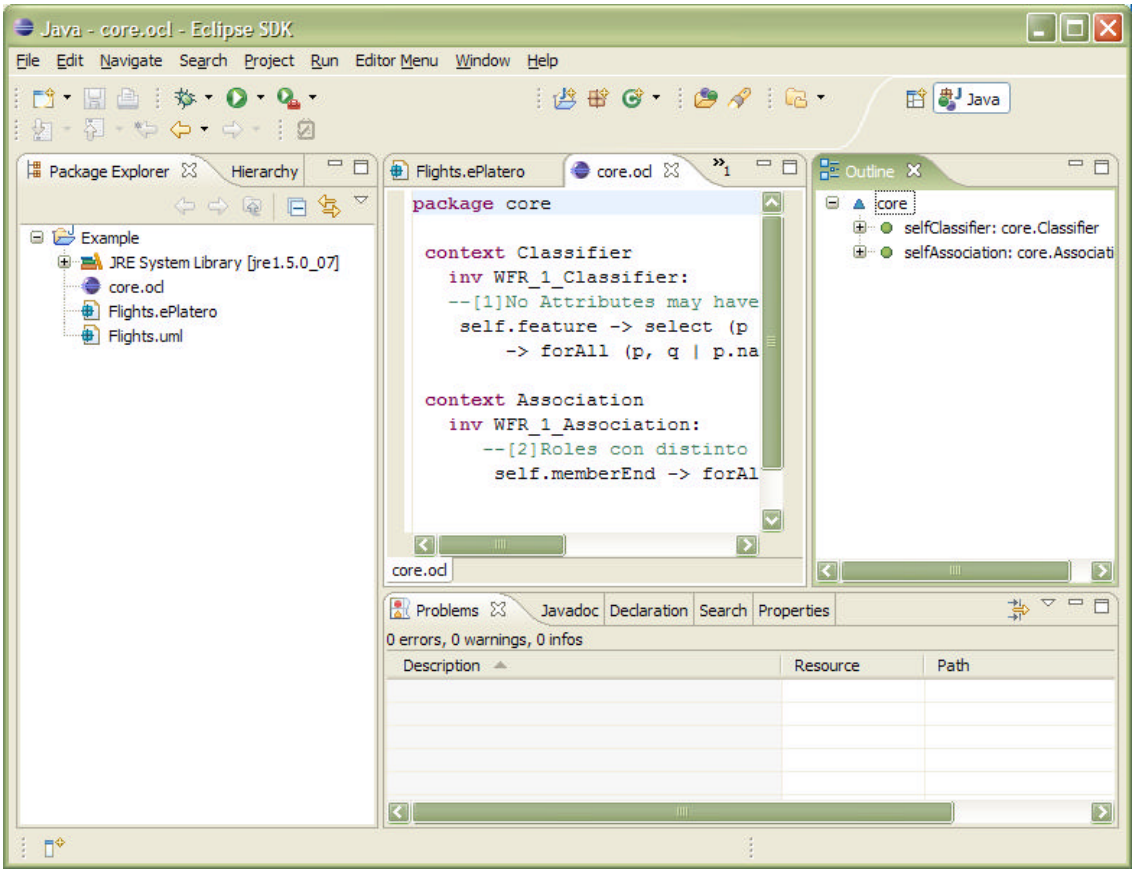

Figura 8-4. Editor de fórmulas OCL

En el *framework* Jface Text, la clase AbstractTextEditor es la clase base del editor de texto predefinido. Para implementar un editor de texto se debe subclasificar dicha clase. El AbstractTextEditor trabaja sobre el contenido del modelo del documento.

o Descripción del Analizador léxico y sintáctico

El analizador léxico recibe el contenido del archivo OCL, que es un archivo de texto con extensión ocl. En este proceso se agrupan los diferentes caracteres del flujo de entrada en *tokens*. Los *tokens* son los símbolos léxicos del lenguaje. Estos están identificados con símbolos y suelen contener información adicional (como el archivo en el que están, la línea donde comienzan, etc.). Una vez identificados, son transmitidos al analizador sintáctico.
En la fase de análisis sintáctico se aplican las reglas sintácticas del lenguaje analizado al flujo de *tokens*. En caso de no haberse detectado errores, el intérprete representará la información codificada en el código fuente en un Árbol de Sintaxis Concreta (CST), que es una representación arbórea de los diferentes patrones

sintácticos que se han encontrado al realizar el análisis, salvo que los elementos innecesarios (signos de puntuación, paréntesis) son eliminados.

La especificación de OCL 2.0 define la sintaxis abstracta (AS), concreta (CS) y la transformación de la CS a la AS. Para ver en detalle la definición de ambas sintaxis, puede consultarse el documento de especificación de OCL [16]. La sintaxis concreta permite a los modeladores escribir expresiones en forma textual.

o Descripción del Evaluador OCL

El evaluador de OCL es el responsable de realizar el análisis semántico de las fórmulas OCL, parseadas anteriormente. El evaluador actual de ePlatero permite evaluar los invariantes de los tipos y meta tipos, es decir evalúa los invariantes del modelo y reglas de buena formación del metamodelo. En la Figura 8-5 se ilustra cómo el editor de fórmulas OCL presenta los errores de evaluación. Los comentarios son utilizados para clarificar las restricciones y/o para que el usuario pueda comprender rápidamente cuales son los errores semánticos. Específicamente, la Figura 8-5 muestra un caso en el que la Clase FlightA tiene dos atributos con el mismo nombre, isCanceled. Cuando el archivo core.ocl es evaluado, podrán verse los errores a través de comentarios de restricciones que no se cumplen en el modelo. La Figura 8-6 muestra la restricción (Regla de Buena Formación de la metaclase Classifier) que no se cumple en el modelo.

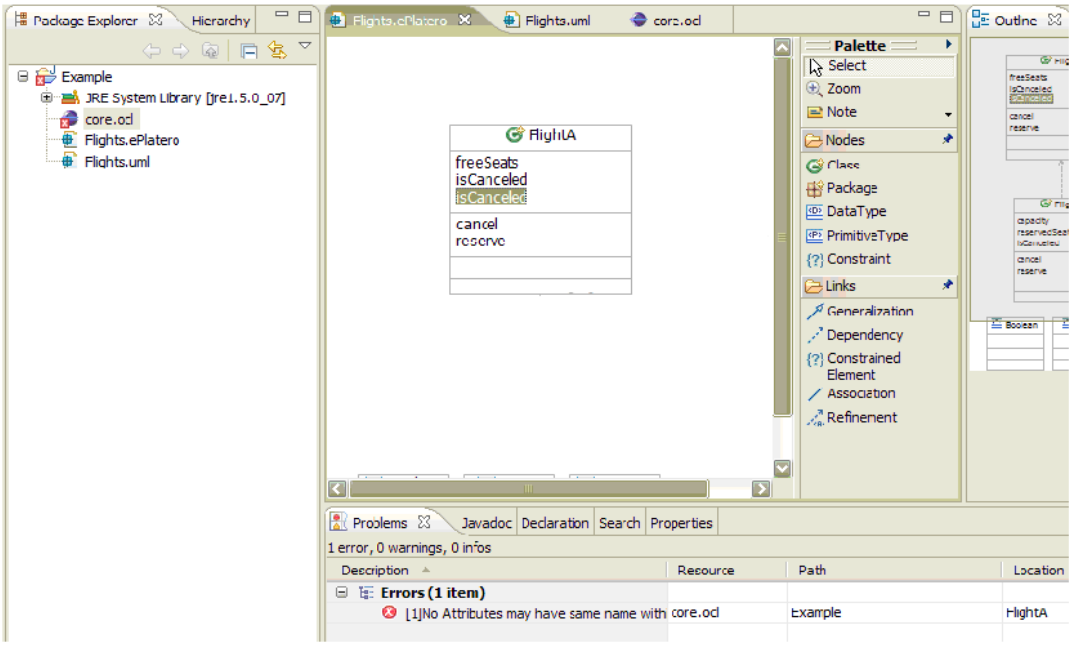

Figura 8-5. Evaluador OCL

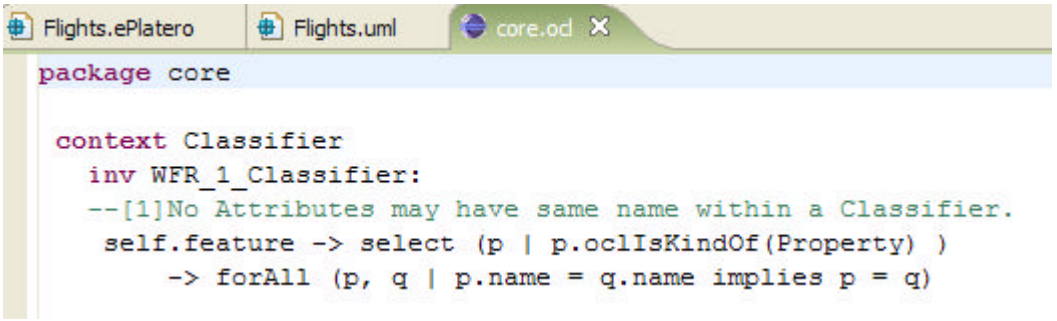

Figura 8-6. Evaluador OCL - Restricciones

8.2 El prototipo Calculador para Composición de Transformaciones

Las figuras 8-7 a 8-10 muestran las pantallas más relevantes del calculador para composiciones. La Figura 8-7 muestra la pantalla donde se realiza la selección, desde un repositorio de transformaciones, de las transformaciones declarativas que serán compuestas; la pantalla de la Figura 8-8 muestra la aplicación de una operación algebraica sobre las transformaciones seleccionadas; en este caso la unión de transformaciones declarativas. En el panel inferior de la figura, podemos visualizar una vista previa del resultado de la composición, una vez aplicada la operación.
Similarmente, pero a nivel operacional, la Figura 8-9 muestra la selección desde el repositorio de transformaciones, de las transformaciones operacionales a componer, mientras que la Figura 8-10 muestra la aplicación de la operación de unión de transformaciones operacionales.

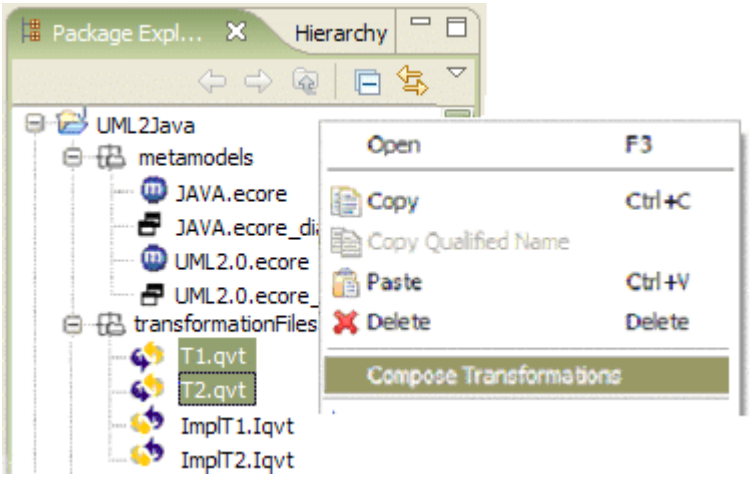

Figura 8-7. Selección de Transformaciones Declarativas

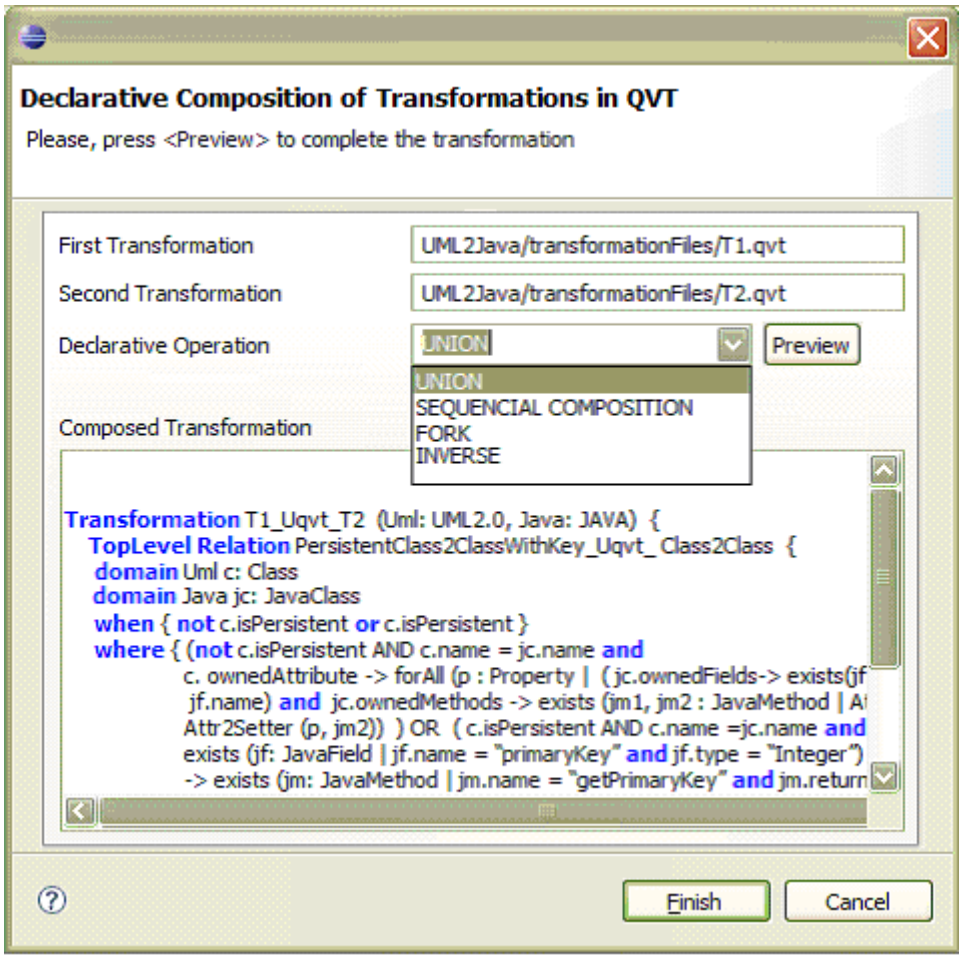

Figura 8-8. Aplicación de la unión para Transformaciones Declarativas

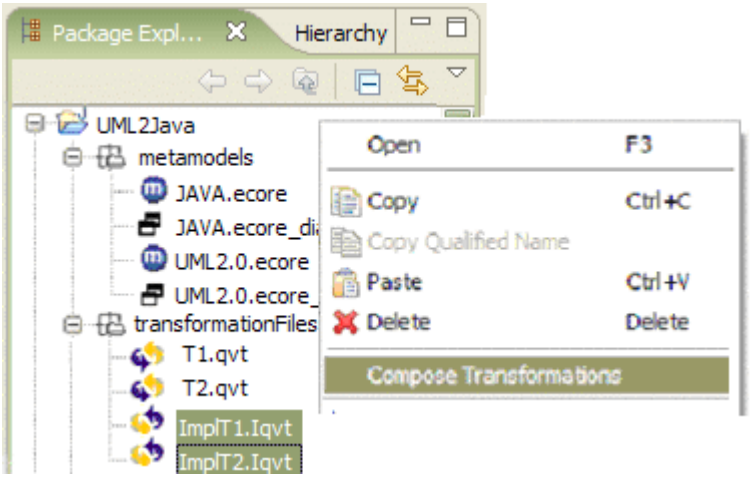

Figura 8-9. Selección de Transformaciones Operacionales

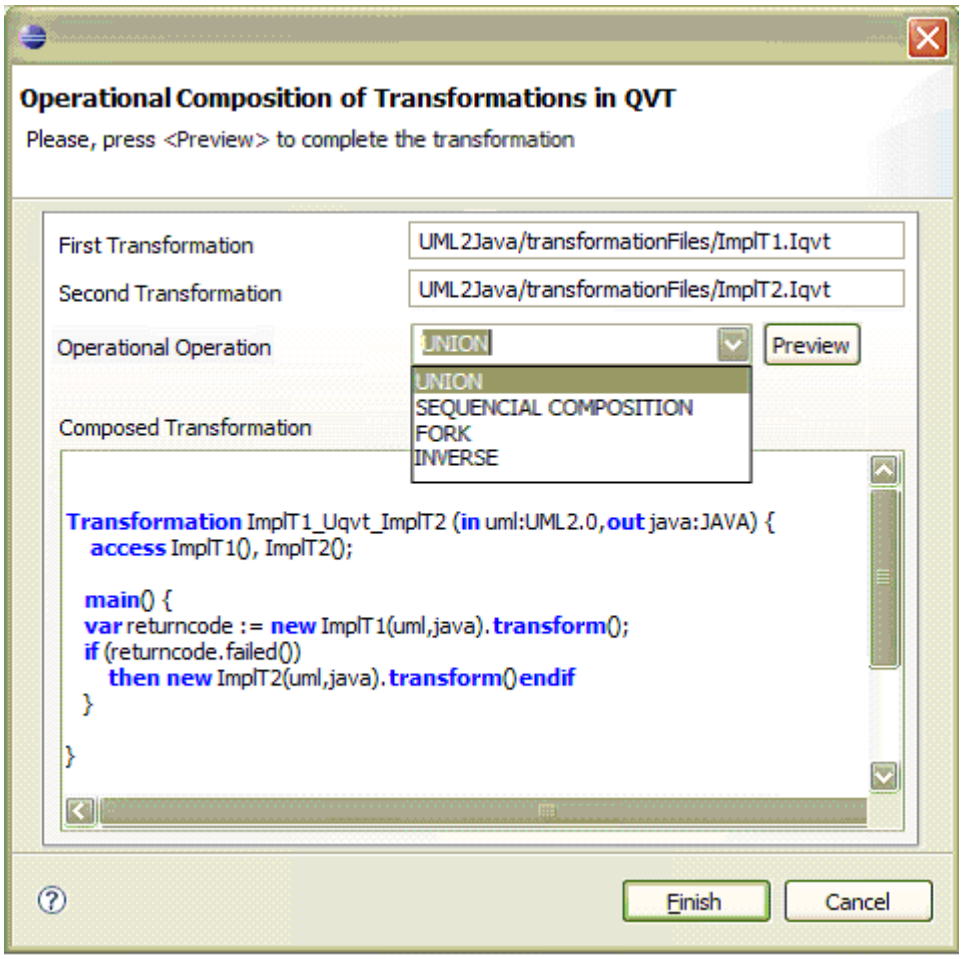

Figura 8-10. Aplicación de la unión para Transformaciones Operacionales

8.3 Conclusión del Capítulo

En este capítulo hemos presentado un prototipo de implementación que permite a los desarrolladores editar y almacenar transformaciones QVT atómicas. Esta herramienta aporta asistencia a las actividades de desarrollo de transformaciones. En base a este prototipo, los desarrolladores podrán construir transformaciones más complejas aplicando las operaciones de composición presentadas en este trabajo. Las transformaciones resultantes serán automáticamente "calculadas" por la herramienta.

Este prototipo constituye un soporte inicial de implementación para las actividades de desarrollo de transformaciones y de composición de las mismas; contando con la base formal para realizar estas actividades, que es el objetivo de nuestro trabajo.

El capítulo 9 presenta y analiza comparativamente, diversos trabajos que guardan relación con el tema de esta tesis.

Capítulo
9

Trabajos Relacionados

En este capítulo presentamos una descripción de algunos de los trabajos sobre composición de modelos y composición de transformaciones que guardan mayor relación con nuestra propuesta:

❖ Mariano Belaunde en [34] describe los mecanismos definidos en el lenguaje QVT Operacional para expresar composición de transformaciones. En particular, discute por un lado las diferentes granularidades de composición y por otro el efecto sobre la interoperabilidad de transformaciones escritas en diferentes lenguajes. En el primer aspecto, se asemeja al Capitulo 5 de nuestro trabajo, que resume los mecanismos de composición que existen en QVT. Belaunde agrega al análisis de QVT una comparación con lenguajes de comandos como *shell* en UNIX, que ofrecen la posibilidad de escribir *loops*, controles *if/then/else*, manipular variables, etc. Observa que para transformaciones externas (*coarse grained*), QVT no inventó nada significativo, ya que utiliza los mismos constructores de programación disponibles para reglas de composición interna (*fine grained*) de reglas y que se encuentran comúnmente en un lenguaje *shell.* Una diferencia con estos lenguajes es que los parámetros que se pasan ahora (para transformaciones) no son simplemente strings sino modelos tipados compatibles con MOF. Lo cual significa que debe haber análisis de tipos sobre modelos para implementar estas composiciones. Finalmente, el autor concluye que gracias al mecanismo caja negra (*Black-box*), QVT permite manejar el problema de la interoperabilidad con transformaciones externas escritas en otros lenguajes diferentes a QVT. Respecto al mecanismo caja negra:

- Dennis Wagelaar en [35] analiza el aspecto de composición black-box de transformaciones, observando la ventaja de poder tratar a las transformaciones como componentes de software (que en general no están escritas en el mismo lenguaje). Afirma que, similarmente a las componentes de software *black-box*, no todas las transformaciones de modelos pueden combinarse. Existen también restricciones en el orden en que las transformaciones tienen que usarse. En el dominio de composición de componentes, han sido usados Lenguajes de Modelado específicos del Dominio (DSMLs) [36] [37] para manejar verificación de composiciones y generación automática de código de la composición; por lo tanto Wagelaar propone aplicar técnicas de DSML para la composición de transformaciones de modelos.

❖ Anneke Kleppe en [30] describe MCC, un ambiente abierto para transformaciones de modelos en el cual los usuarios pueden combinar las

herramientas disponibles que implementan transformaciones y las aplican a modelos en varios lenguajes para producir la salida requerida. Además trata de responder la pregunta de donde y cómo pueden aplicarse exitosamente las transformaciones. Presenta una taxonomía de transformaciones de modelos basada en una propuesta lingüística. Esta taxonomía es necesaria para la formalización de composición de transformaciones que presenta. Las transformaciones son categorizadas de acuerdo a la parte del lenguaje fuente y destino a la que está dirigida. Las herramientas de transformación pueden ser combinadas usando tres operadores combinatorios comúnmente conocidos en la historia de la computación: *secuencia*, *paralelismo* y *selección*.

- Otra propuesta que está estrechamente relacionada al trabajo descrito en [30] es el trabajo realizado por Xavier Blanc et al en [38]. Su proyecto *ModelBus* trata con los mismos problemas, a la manera de OMG's CORBA. Hay sin embargo, algunas diferencias, la más importante es que MCC ofrece un lenguaje de *scripts* para definir nuevos servicios.

- Otro trabajo que guarda similitud con MCC es ToolBus [39]. Una diferencia entre las propuestas ToolBus y MCC es que ToolBus usa una representación de datos común mientras que MCC ofrece mayor flexibilidad y generalidad porque usa representaciones de datos diversas, determinadas por los tipos de modelos disponibles en el ambiente. Además, ToolBus permite comunicación entre procesos, más allá que sólo intercambio de datos, usando mensajes o notas. MCC no ofrece esta posibilidad.

- El conjunto de herramientas para transformaciones UMLAUT [40] es construido con la misma intención que MCC: proveer al diseñador de modelos de una libertad de elección en cuanto a combinaciones de transformaciones a ser ejecutadas. Una diferencia es que UMLAUT está limitado a transformar sólo modelos UML mientras que MCC nos habilita a manejar modelos escritos en varios lenguajes. Además, aunque UMLAUT proporciona una biblioteca de transformaciones y una arquitectura extensible, la composición de transformaciones en UMLAUT es interna, no externa como en MCC.

❖ Por otra parte, Bézivin y sus colegas en [41] analizan un conjunto de requerimientos básicos para lenguajes de composición de modelos y herramientas de soporte. Basan su presentación en el estudio de tres *frameworks* de composición de modelos: el Atlas Model Weaver [42], el Glue Generator Tool (GGT) [43] [44] y el Epsilon Merging Language (EML) [45] [46]. A partir de estos frameworks los autores derivan un conjunto común de definiciones para composición de modelos, presentando antes un conjunto de requerimientos para la solución de composición. Por ejemplo, la definición que ellos proveen para el término de transformación de modelos - "una transformación de modelos es una operación que toma un conjunto de modelos como entrada, ejecuta un conjunto de reglas sobre los elementos del modelo(s) y produce un conjunto de modelos como

salida" - pone en evidencia el punto de vista operacional de este análisis. Concluyen que existe una necesidad de unificación y conceptualización en el campo de la composición de modelos ya que las tres soluciones analizadas toman diferentes formas y propuestas; remarcan que como se discute en [47], el lenguaje QVT, la propuesta de OMG para transformaciones de modelos, sólo trata parcialmente las cuestiones de composición.

❖ Jon Oldevik en [48] investiga cómo el punto de vista de contar sólo con transformaciones atómicas puede ser enriquecido con mecanismos para composición de transformaciones, para soportar necesidades del proceso empresarial conducido por modelos. Esto lo hace presentando un framework de modelado para transformaciones compuestas, basadas en una jerarquía de tipos de transformación, algunos de los cuales representan transformaciones atómicas simples, mientras los otros representan transformaciones complejas. El trabajo utiliza técnicas de modelado UML 2.0 y guarda relación con QVT. Relacionado a esta propuesta, en [49] se presenta un framework para composición de transformaciones que define un patrón para transformaciones compuestas y un lenguajes léxico para definición de composiciones, el cual maneja configuración (orden de ejecución) de componentes de transformaciones. Sin embargo, este framework no propone ninguna relación con UML ni QVT. En [50] el patrón de transformaciones es introducido para proveer formas para describir combinaciones de transformaciones reusables con enfoque particular en tratar con incompatibilidades de metamodelos específicos de la aplicación. Esta propuesta usa la notación gráfica UMLX [51], ensamblando con la notación gráfica de QVT.

❖ El tema de reuso (que incluye composición en una de sus formas) de modelos es tratado por Olsen et al. en [52]. El trabajo apunta a que actualmente las transformaciones son escritas desde cero, para cada problema que deben solucionar. Esto consume mucho tiempo y dificulta las tareas. Los autores sostienen que deben estar disponibles librerías de transformaciones de modelos. Describen aspectos de transformaciones reusables y combinables, identificando las formas en que se puede aplicar reuso, presentando tres estudios realizados en el proyecto MODELWARE [53] que tiene como uno de sus objetivos principales, incrementar la productividad del desarrollo de software.

❖ El trabajo de Arda Goknil et al. en [54] considera a las reglas de transformación como partes atómicas de las transformaciones y las operaciones de agregar, borrar, y actualizar como reglas de transformaciones también. Discuten el problema de composición y descomposición de estas operaciones. Marcan la necesidad de que los lenguajes para transformaciones deben soportar operaciones de composición para definiciones de patrones complejos.

❖ Por su parte, el reciente trabajo de Bert Vanhooff et al. [59] analiza a las transformaciones de modelos, considerando que pueden descomponerse en una secuencia de sub-transformaciones, es decir una cadena de transformaciones. Sin embargo consideran difícil, contando con las tecnologías actuales, poder reusar y componer sub-transformaciones sin estar muy familiarizado con sus detalles de implementación. También se complica el hecho de elegir, a la hora de combinar

transformaciones construidas con diferentes tecnologías, la tecnología más adecuada para cada sub-transformación. Los autores proponen una metodología basada en modelos para reuso y composición de sub-transformaciones con una visión independiente de la tecnología. Implementan una herramienta llamada UniTI, las cual provee un editor de cadenas de transformaciones.
El trabajo es interesante, aunque sólo considera una única forma de composición de transformaciones: la composición secuencial.

❖ Finalmente, el trabajo de Stevens [60], explora algunas cuestiones que considera fundamentales respecto a que en el lenguaje QVT declarativo, las especificaciones de transformaciones deben corresponder a transformaciones bidireccionales entre modelos. Analiza los requerimientos fundamentales que deben cumplir las herramientas para soportar tales transformaciones, discutiendo algunas cuestiones semánticas a las que se llega y que las transformaciones deberían cumplir, como por ejemplo, se afirma que es necesario que la inversa de una transformación siempre exista, y esto en la práctica, operacionalmente, no puede asegurarse. Analiza también la composición de relaciones en QVT y la composición secuencial de transformaciones, justificando que la combinación puede resultar en transformaciones no biyectivas que igualmente deben tener su inversa. Finalmente, propone un framework y un conjunto de lemas que presentan una definición de “transformación coherente”.

9.1 Conclusión del Capítulo

Excepto el trabajo de Stevens, que estudia las transformaciones de modelos considerando únicamente su forma declarativa, en su generalidad las propuestas existentes están enfocadas en los aspectos operacionales de la maquinaria de composición, ofreciendo soluciones interesantes y útiles para una amplia variedad de necesidades prácticas.
Sin embargo, las propuestas actualmente no cubren el espectro completo de composición. Este es la diferencia principal respecto a nuestro trabajo: está enfocado a proporcionar un fundamento holístico para el problema de composición de transformaciones de modelos que abarque la dimensión declarativa así como la operacional.

Capítulo
10

Conclusiones

A lo largo de esta década la *Ingeniería de Software Conducida por Modelos* (MDE) se ha convertido en un nuevo paradigma de software que propone mejorar la construcción de software a través de un proceso guiado por modelos y soportado por potentes herramientas que generan código a partir de modelos. MDE promete una mejora de la productividad y de la calidad del software generado debido a que se reduce el salto semántico entre el dominio del problema y de la solución; reduciendo también los tiempos de desarrollo. La transformación entre modelos constituye el motor de MDE y de esta manera los modelos pasan de ser entidades meramente contemplativas a ser entidades productivas.

En este trabajo de tesis hemos propuesto una formalización para lenguajes de transformación de modelos, a través de los siguientes logros:

- Hemos definido el lenguaje SQVT para expresar transformación de modelos cuyo metamodelo inicial está inspirado en el lenguaje QVT. En la construcción del lenguaje utilizamos especificaciones ya existentes en OMG; por un lado, mediante la definición de estereotipos, extendemos la Infraestructura 2.0, una especificación más abstracta que UML e independiente de él y, por otro lado utilizamos el lenguaje OCL para expresar patrones de la transformación, evitando la creación de nuevos elementos para modelarlos.
  El lenguaje SQVT fue pensado para ser minimal en el conjunto de lenguajes que permiten expresar relaciones y *queries* de transformación de modelos. Mantener simplicidad y reducir el tiempo de entrenamiento del usuario son las ventajas principales de esta propuesta minimal.

- Tanto el lenguaje para transformación de modelos QVT como otros basados en él que definen sus propios constructores, carecen de una semántica formalmente definida. Contar con la definición semántica precisa de un lenguaje lo hace más sólido y permite que probemos el cumplimiento de ciertas propiedades y condiciones de corrección. Para tal fin hemos presentado el formalismo de la *teoría de problemas*; basándonos en esta teoría definimos la semántica (tanto a nivel declarativo como operacional) de los lenguajes para transformación de modelos. Consecuentemente pudimos enunciar condiciones de corrección entre ambos niveles de expresión.
  Cabe aclarar que definir la semántica de lenguajes para transformación a través del lenguaje SQVT, no quita generalidad a la propuesta, ya que esta definición semántica puede extenderse en forma trivial para el lenguaje QVT completo.

- A pesar del hecho que QVT ofrece dos perspectivas de modelado – esto nos permite especificar ***que*** hace la transformación (QVT declarativo) y también ***cómo*** se concreta su ejecución (QVT operacional) – muchos de los trabajos actuales sobre transformación de modelos presentan esencialmente naturaleza operacional y ejecutable. Las transformaciones pueden también ser vistas como modelos descriptivos que establecen solamente las propiedades que una transformación tiene que cumplir y omiten detalles de ejecución. En particular, respecto al problema de composición, QVT y otras propuestas se enfocan solamente en los aspectos operacionales de la composición sin considerar su parte descriptiva, ofreciendo una visualización parcial del problema.
  Consecuentemente, en este trabajo hemos descrito cómo la *teoría algebraica de problemas* puede ser aplicada como base para construir un fundamento matemático para el problema de la composición de transformaciones abarcando ambas dimensiones (declarativa y operacional). Específicamente, hemos definimos intuitiva y formalmente las operaciones de unión (no-determinismo), composición secuencial (secuencia) y operación *fork* (paralelismo) que representan los tres operadores combinatorios comúnmente conocidos en la historia de la computación. Además fue definida la operación inversa para transformaciones.
  Definir en forma precisa la composición de transformaciones, nos permitió sincronizar los dos niveles de QVT es decir, es posible combinar automáticamente y de forma sincronizada, tanto especificaciones como implementaciones definidas en QVT.
  Al igual que en la definición de la semántica, SQVT no quita generalidad a la definición formal para las operaciones de composición. En forma directa, la formalización presentada es válida para el lenguaje QVT completo.

- Incluimos un análisis de los casos en que cada una de las operaciones para composición de transformaciones definidas pueden aplicarse en la práctica; cuándo tiene utilidad hacerlo y cuándo no pueden aplicarse.

- Finalmente presentamos un prototipo de implementación mediante el cual los desarrolladores pueden construir transformaciones más complejas aplicando las operaciones de composición presentadas en este trabajo.

10.1 Contribuciones Principales

Las contribuciones principales de este trabajo de tesis a la metodología del desarrollo de software conducido por modelos se enfocan en mejorar la **madurez** de los lenguajes para transformación de modelos. Estas contribuciones pueden resumirse en los siguientes puntos:

- El hecho de contar con la maquinaria de composición en ambos niveles de QVT sincronizada en forma precisa, nos permitirá explotar totalmente el paradigma *dividir-y-conquistar* en el desarrollo de transformaciones de modelos. Es decir, podemos ser capaces de dividir una transformación declarativa en sub-transformaciones cuyas α-soluciones podrían ser combinadas nuevamente (re-combinadas) en una α-solución para la transformación original. Específicamente:

    - Dividir una transformación declarativa significa declarar la transformación dada en términos de operaciones sobre las transformaciones declarativas componentes, mientras que la re-combinación para obtener la α-solución es hecha por medio de las operaciones correspondientes sobre la α-solución de cada componente.

- El lenguaje QVT, en su parte operacional, no proporciona un mecanismo de caja negra limpio para realizar la composición de transformaciones. El desarrollador está obligado a escribir código usando constructores del lenguaje de programación imperativo en vez de aplicar sólo operadores de composición de alto nivel. Por otra parte, QVT no incluye el soporte necesario para combinar transformaciones declarativas. Para vencer estos inconvenientes, hemos especificado formalmente una interpretación en QVT para las operaciones sobre problemas y las operaciones correspondientes sobre soluciones definidas por la *teoría algebraica de problemas*. Esta fundamentación matemática del problema de composición de transformaciones proporciona un claro panorama del mecanismo de composición en ambos niveles de QVT, como así también permite definir precisamente la conexión entre operaciones de un nivel y las operaciones correspondientes en el otro nivel. Esta fundamentación representa una mejora en la especificación de QVT en el sentido que permite la construcción de una maquinaria más simple y más poderosa para composición de transformaciones, que servirá como una base sólida para la definición de lenguajes de transformación de modelos y herramientas.

- Respecto a la expresividad del álgebra de transformaciones propuesta, fue probado en [28] que las teorías de primer orden pueden ser interpretadas como teorías ecuacionales en álgebras *fork*. Además, la especificación algebraica puede ser obtenida algorítmicamente desde la especificación de primer orden usando una correlación computable [28].

  Por consiguiente, una amplia clase de problemas (al menos aquellos que pueden ser descritos en lógicas de primer orden) puede ser especificada en el cálculo de álgebras *fork*. Debido al hecho de que nuestra álgebra de transformaciones es una instancia del álgebra *fork* y que QVT es un lenguaje de primer orden, es trivial probar que cualquier transformación de modelos que puede ser especificada en QVT también puede ser especificada en el álgebra de transformaciones.

10.2 Trabajo Futuro

Numerosas líneas de trabajo a futuro quedan abiertas a partir de esta tesis. Algunas de ellas son:

- **Operaciones de Alto Nivel del Algebra de Transformaciones**

Debido a que las operaciones del álgebra fueron definidas con el objetivo de simplicidad y minimización, podría hacerse difícil para los desarrolladores expresar una composición real usando esas operaciones. De este modo, tenemos que contar con operaciones de más alto nivel que se mantengan más cerca del entendimiento humano. Dichas operaciones de alto nivel, que son definidas en términos de las básicas gracias al poder expresivo del álgebra de transformaciones, ya han sido definidas en el área del Álgebra de Relación [28]. Una de ellas es la operación ⊗, que en nuestra álgebra expresa el producto cartesiano entre transformaciones.

Es trabajo futuro especificar declarativa y operacionalmente estas operaciones de alto nivel en base a las simples ya definidas.

- **Metodologías para Descomposición de Transformaciones Monolíticas**

Nuestro trabajo presenta un álgebra para composición de transformaciones, haciendo más precisa la definición de las operaciones de composición. Al componer desde un repositorio de transformaciones, luego de haber considerado los casos de aplicabilidad analizados en el Capítulo 7, puede suceder que existan otras cuestiones, más relacionadas con las tecnologías usadas para el desarrollo de las transformaciones. Las metodologías y estrategias a seguir para decidir cuales transformaciones combinar y cómo concretar estas combinaciones, teniendo en cuenta ambos niveles de especificación para transformaciones (declarativo y operacional) son cuestiones centrales que deben incluirse como línea de trabajo futuro.

Específicamente consideramos los siguientes puntos:

- Definir estrategias para decidir cómo y por qué dividir o descomponer una transformación monolítica en sub-transformaciones.

- Definir estrategias para seleccionar las sub-transformaciones a combinar que sirvan a cada objetivo en particular.

- Definir metodologías para combinar las transformaciones seleccionadas según las diferentes situaciones, dado que pueden estar definidas en diferentes tecnologías.

- Identificar y automatizar roles (que inicialmente pueden ser personas con sus capacidades específicas) que se asocien a cada actividad en el desarrollo de transformaciones, considerando las metodologías propuestas. Por ejemplo, los roles que sugerimos inicialmente podrían ser los siguientes:

  - **Especificador de Transformaciones**: un rol capaz de especificar transformaciones declarativas.
  - **Implementador de Transformaciones**: un rol capaz de definir transformaciones operacionales.
  - **Buscador de Transformaciones:** un rol capaz de decidir y seleccionar transformaciones a combinar tanto a nivel declarativo como a nivel operacional.
  - **Ensamblador de Transformaciones**: un rol capaz de ensamblar las transformaciones en ambos niveles lingüísticos.

- **Evaluación del uso del Álgebra para Transformaciones**

Con el fin de evaluar la conveniencia del uso de las operaciones del álgebra, resultará útil seleccionar del repositorio una cierta cantidad de transformaciones monolíticas que puedan dividirse en otras más pequeñas y luego recombinarse mediante la aplicación de los operadores, obteniendo la transformación original.
Luego, pueden considerarse las transformaciones monolíticas y las fragmentadas y evaluarse sobre ellas determinadas características mensurables como pueden ser:

- Tamaño: la cantidad de palabras en la definición de la transformación.

- Complejidad Cognitiva: la cantidad de minutos que una persona entrenada necesita para entender la transformación.

- Grado de Reusabilidad: el porcentaje de la transformación que fue importada desde un repositorio (o sea, que no fue escrita desde cero).

Resultará entonces interesante realizar una comparación entre los resultados obtenidos al evaluar cada transformación monolítica y sus correspondientes transformaciones fragmentadas, sacando conclusiones respecto al efecto que produce la descomposición de transformaciones, por ejemplo, sobre el grado de complejidad cognitiva y sobre la probabilidad de reutilización que pueden permitir transformaciones pequeñas, cohesivas y con bajo acoplamiento (cualidades propiciadas por la descomposición algebraica) por sobre la probabilidad de reutilizar transformaciones monolíticas, no cohesivas.

La concreción de estas mediciones y el hallazgo de otras características a considerar, es tarea futura que aporta un grado de validación a nuestra propuesta.

- **Herramienta de Soporte**

Respecto a aspectos de implementación, quedan por concluir las siguientes tareas:

- Completar el prototipo presentado como extensión de la herramienta ePlatero, haciendo totalmente automatizable y ejecutable esta propuesta con base formal para composición de transformaciones.

- Lograr la integración de la herramienta para transformaciones y composición de transformaciones con las metodologías enunciadas para seleccionar y combinar sub-transformaciones y para descomponer transformaciones monolíticas, en forma automatizada.

- Incluir actividades de evaluación a la herramienta para composición de transformaciones. Dichas actividades fueron enunciadas en el ítem anterior. Realizar en forma automatizada estas mediciones sobre ciertas características que aporta la descomposición de transformaciones, contribuirá a lograr un nivel de validación sobre las actividades para descomposición / composición que se realizarán siguiendo las metodologías sugeridas.

- Integrar nuestra herramienta con herramientas existentes que ejecuten transformaciones. Es decir, nuestro trabajo actualmente abarca los niveles declarativo y operacional para especificación de transformaciones; sin embargo existe un tercer nivel, el nivel de ejecución de transformaciones, que otras herramientas pueden proveerlo y que, integrado a nuestra propuesta, constituirá un ambiente para desarrollo de transformaciones más abarcativo.

- **Interoperabilidad**

Como vimos en la introducción presentada en el Capítulo 1, un problema que la metodología **MDE** debe afrontar es el de la interoperabilidad entre plataformas. Esta cuestión surge en el proceso de transformación de modelos

debido a que pueden generarse, a partir del mismo modelo PIM, varios modelos PSMs, los cuales frecuentemente pueden estar relacionados. Estas relaciones se llaman *bridges* o puentes.

Paralelamente al problema de composición de transformaciones, este tipo de interoperabilidad debería contemplarse en la definición de lenguajes de transformación de modelos. La comunicación puede ser realizada por herramientas para transformación de modelos que generen además de modelos PSMs, *bridges* entre los PSMs que fueron generados en un mismo nivel de transformación desde un mismo PIM y posiblemente correspondan a distintas plataformas. Es una línea de trabajo futuro extender nuestro lenguaje en este sentido.

Glosario de siglas y términos

Este Glosario describe siglas y términos utilizados en el desarrollo de este trabajo. La gran mayoría de estos términos y siglas, cuenta además con la referencia bibliográfica correspondiente, que aparece al utilizarlo por primera vez dentro de la tesis. El contenido que sigue es simplemente, una ayuda rápida para ilustrar algunos conceptos.

Álgebra de Relación

El **Algebra de Relación** es una estructura algebraica definida con la intención de capturar las propiedades matemáticas de relaciones binarias. Es una extensión propia del álgebra Booleana de dos elementos. Matemáticamente, un Álgebra de Relación es un álgebra $A= (P(V), \cup, \cap, \varnothing, V, \neg, ;, \smile, 1)$, tal que $(P(V), \cup, \cap, \varnothing, V, \neg)$ es un álgebra Booleana y $(P(V), ;, 1)$ es un monoide.

AndroMDA

AndroMDA (pronunciado “Andrómeda”) es un framework de código extensible asociado al paradigma MDA (Model Driven Architecture). Los modelos UML son transformados en componentes desplegables para la plataforma elegida (J2EE, Spring, .NET). Al contrario de otros entornos de desarrollo MDA, AndroMDA incluye un conjunto de cartuchos enfocados a los *Kits* de desarrollo actuales. AndroMDA también incluye un *Kit* para desarrollar sus propios cartuchos generadores de código o personalizar los existentes: el cartucho Meta. Utilizándolo, se puede construir un generador propio de código empleando una herramienta de UML. Debido a que su generador de código soporta plataformas actuales, se ha convertido en la principal herramienta de código abierto de MDA para el desarrollo de aplicaciones empresariales.

ArcStyler 5.5

ArcStyler es una de las herramientas MDA comerciales más extendida. Puede generar código a partir de modelos para cualquier plataforma como .NET o J2EE, siendo una herramienta genérica que nos permite transformaciones de modelo a código sin restricciones. También permite transformaciones entre modelos con la nueva herramienta, AIM, que incorpora esta versión. Soporta el lenguaje de modelado UML 1.4 para el diseño, aunque es independiente de cualquier versión UML ya que se apoya en otra herramienta que le proporciona dicha funcionalidad, *MagicDraw*. Se apoya en MOF para definir sus propios modelos y en XMI para almacenarlos, lo que le permite exportar e importar los distintos modelos usados. Además, contiene un repositorio de modelos al cual accede a través de JMI, pero esto no implica que no puedan añadírsele otros repositorios de modelos e incluso

otras convenciones de interfaces de acceso. En general, la arquitectura de la herramienta es bastante flexible.

ASP

Active Server Pages (**ASP**) es una tecnología del lado servidor de Microsoft para páginas web generadas dinámicamente, que ha sido comercializada como un anexo a Internet Information Server (IIS). La tecnología ASP está estrechamente relacionada con el modelo tecnológico de su fabricante. Intenta ser solución para un modelo de programación rápida ya que programar en ASP es como programar en VisualBasic, pero con muchas limitaciones. Lo interesante de este modelo tecnológico es poder utilizar diversos componentes ya desarrollados como algunos controles ActiveX.

Algoritmo British Museum

El algoritmo British Museum es una propuesta general para buscar una solución chequeando todas las posibilidades una por una, comenzando con la menor. El término refiere a una técnica conceptual, no práctica, donde el número de posibilidades es enorme.

ATL

ATL (*ATLAS Transformation Language*) es un lenguaje de transformación de modelos y conjunto de herramientas desarrolladas por el Grupo ATLAS (INRIA & LINA). En el campo de *Model-Driven Engineering* (MDE), ATL provee formas de producir un conjunto de modelos destino, desde un conjunto de modelos fuente. Desarrollado sobre la plataforma Eclipse, el ATL *Integrated Environnement* (IDE) provee un número de herramientas estándar de desarrollo (*syntax highlighting, debugger,* etc.) que facilita el desarrollo de transformaciones en ATL. El proyecto ATL incluye también una librería de transformaciones ATL.

Composición interna (o *fine-grained*) de transformaciones

Mecanismos que permiten que las *mapping Operations* dentro de una transformación operacional, puedan ser combinadas mediante llamadas, o utilizando facilidades de reuso: herencia, *merge* y disyunción. Estos mecanismos son llamados composiciones internas (*fine-grained*) de transformaciones.

Composición externa (o *coarse -grained*) de transformaciones

Los mecanismos que permiten la combinación de transformaciones completas, como entidades de caja negra, son llamados composiciones externas (*coarse - grained*) de transformaciones.

CWM

El *Common Warehouse Model* (CWM) es una especificación que describe el intercambio de metadatos entre almacenamientos de datos, usado en inteligencia empresarial y en la gestión de conocimientos. El Meta Object Facility (MOF) de OMG proporciona la base común para distintos modelos de metadatos. Si dos diferentes metadatos son instancias de MOF, entonces los modelos basados en ellos pueden residir en el mismo repositorio, gracias al intercambio que aporta CWM.

Eclipse

Eclipse es una plataforma de software de código abierto independiente de otras plataformas. Esta plataforma, típicamente ha sido usada para desarrollar entornos integrados de desarrollo (del inglés IDE), como el IDE de Java llamado *Java Development Toolkit* (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse). Sin embargo, también se puede usar para otros tipos de aplicaciones cliente. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Un ejemplo es el recientemente creado Eclipse Modeling Project, cubriendo casi todas las áreas de *Model Driven Engineering.* Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente no lucrativa, que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

EMF

Eclipse Modeling Framework Project (EMF).
El proyecto EMF es un framework de modelado y facilidades de generación de código para la creación de herramientas de instalación y otras aplicaciones basadas en un modelo de datos estructurados. Desde una especificación de modelo descrito en XMI, EMF ofrece herramientas y soporte en tiempo de ejecución para producir un conjunto de clases Java para el modelo, junto con un conjunto de clases adaptadoras que permiten la visualización y la edición basada en comandos, del modelo, y un editor básico.

ePlatero

ePlatero, es un plug-in de código abierto para la plataforma **Eclipse**, desarrollado por nuestro grupo de investigación, que corre sobre el *framework* para metamodelado **EMF**.

Es una herramienta CASE educativa que soporta el proceso de desarrollo de software conducido por modelo utilizando notación gráfica y con fundamento formal. Puede interoperar con herramientas que soporten MDA.

FLASH

Adobe FLASH® (hasta 2005 **Macromedia FLASH®**) o **FLASH®** se refiere tanto al programa de edición multimedia como al reproductor de SWF (Shockwave FLASH) Adobe Flash Player, escrito y distribuido por Adobe, que utiliza gráficos vectoriales e imágenes ráster, sonido, código de programa, flujo de vídeo y audio bidireccional (el flujo de subida sólo está disponible si se usa conjuntamente con Macromedia Flash Communication Server). En sentido estricto, Flash es el entorno y Flash Player es el programa de máquina virtual utilizado para ejecutar los archivos generados con Flash.

HTML

Es el acrónimo inglés de ***H**yper**T**ext **M**arkup **L**anguage*, que se traduce al español como *Lenguaje de Marcas Hipertextuales*. Es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas web. Gracias a Internet y sus navegadores, el HTML se ha convertido en uno de los formatos más populares y fáciles de aprender que existen para la elaboración de documentos para web.

ImperativeExpression

En QVT, metaclase que representa la raíz abstracta de la jerarquía que sirve como base para la definición de todas las expresiones con efectos laterales definidas en la especificación de QVT Operacional. Tales expresiones son AssignExp, WhileExp, IfExp, entre otras. Podemos notar que en contraste con las expresiones OCL puras, libres de efectos laterales, las expresiones imperativas en general, no son funciones.

JFace text

El paquete **org.eclipse.jface.text** y sus sub-paquetes soportan la implementación de editores de texto robustos tales como el editor de texto *workbench* y el editor de Java JDT.

JMI

El **Java Metadata Interface** (JMI) en un estándar para la gestión de los metadatos. La especificación JMI permite la aplicación de una dinámica, independiente de la plataforma para la gestión de infraestructuras de la creación, el almacenamiento, el acceso, el descubrimiento, y el intercambio de metadatos. JMI se basa en la especificación MOF de OMG, un estándar de la industria que apoya la gestión de los metadatos. JMI define el estándar de interfaces Java para modelar estos componentes, y, por tanto, es independiente de la plataforma. JMI permite el descubrimiento, la búsqueda, el acceso y la manipulación de los metadatos, ya sea en tiempo de diseño o en tiempo de ejecución. La semántica de cualquier modelo de sistema puede ser totalmente descubierta y manipulada. JMI prevé también intercambio de meta-metadatos a través de XML utilizando la especificación estándar XML Metadata Interchange (XMI).

JSP

JavaServer Pages (**JSP**) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo. Esta tecnología es un desarrollo de la compañía Sun Microsystems. La Especificación JSP 1.2 fue la primera que ser liberó y en la actualidad está disponible la Especificación JSP 2.1. Las JSPs permiten la utilización de código Java mediante *scripts.* Además es posible utilizar algunas acciones JSP predefinidas mediante etiquetas. Estas etiquetas pueden ser enriquecidas mediante la utilización de Librerías de Etiquetas (*Tag Libraries*) externas e incluso personalizadas.

Kent Model Transformation Language

El lenguaje de transformación de modelos **Kent** es la tercera generación de propuestas para transformación de modelos desarrolladas por la Universidad de Kent. Esta tercera generación de propuestas se diferencia de las anteriores porque incluye conceptos del *Model Driven Development Environment* para manipulación de modelos y transformadores como entidades de primera clase. Por no estar implementado como plug-in Eclipse, este lenguaje no asegura que las transformaciones se realicen entre metamodelos MOF.

MappingOperation

En QVT, metaclase que representa una operación implementando un *mapping* entre uno o más elementos del modelo fuente, en uno o más elementos del modelo destino.
Puede tener sólo signatura o bien ser provisto de la definición de un cuerpo imperativo. En el primer caso, la operación es *black-box*.

Una *mapping operation* siempre refina una relación, donde cada dominio se corresponde con un parámetro del *mapping.*

MDE

Acrónimo inglés de ***Model Driven Engeenering***, en español se traduce como *Ingeniería de Software Conducida por Modelos.*
El paradigma MDE tiene dos ejes principales: - por un lado hace énfasis en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas tecnológicas específicas. Por otro lado, en MDE los modelos son considerados los conductores primarios en todos los aspectos del desarrollo de software. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados **PIM** y modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como **PSM**. Un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

MDD

Otro acrónimo relacionado a MDE es *Model-Driven Development* (MDD), que en español se traduce como *Desarrollo de Software Conducida por Modelos*. Es visto como un sinónimo de MDE, ambos describen la misma metodología de desarrollo de Software.

MDA

En español, *Arquitectura conducida por modelos.* Han surgido varios enfoques dentro del ámbito de MDE, pero sin duda la iniciativa más conocida y extendida es la MDA, acrónimo de *Model Driven Architecture*, presentada por el consorcio OMG (*Object Management Group)* en noviembre de 2000 con el objetivo de abordar los desafíos de integración de aplicaciones y los continuos cambios tecnológicos. MDA propone el uso de un conjunto de estándares (descriptos en este Glosario) como MOF, UML, JMI o XMI. Su objetivo es separar la especificación de la funcionalidad del sistema de su implementación sobre una plataforma concreta, por lo que se hace una distinción entre modelos PIM y modelos PSM.

Merge de *mappings*

En una transformación, una operación *mapping* puede también declarar una lista de operaciones *mapping* que complementa su ejecución: esto es un *mapping merge* (**mezcla**). En términos de ejecución, la lista ordenada de *mappings* mezclados, es ejecutada en secuencia.

MOF

El **Meta Object Facility (MOF),** es un estándar de OMG para MDD. La página oficial de referencia se puede encontrar en OMG's Meta Object Facility. MOF se originó en el Lenguaje Unificado de Modelado (UML); OMG tenía la necesidad de contar con una arquitectura de Metamodelado para definir el UML. MOF está diseñado como el nivel más abstracto de una arquitectura de cuatro capas o niveles. Proporciona un meta-metamodelo en la capa superior, denominado nivel M3. Este modelo M3 es el lenguaje utilizado por MOF para construir metamodelos, denominados modelos M2. El ejemplo más destacado de un modelo MOF de nivel M2, es el metamodelo UML, es decir el modelo que describe a UML. Estos modelos M2 describen los elementos del nivel M1, y por lo tanto describen modelos M1. Estas serían, por ejemplo, modelos escritos en UML. La última capa es el nivel M0 o capa de datos. Se utiliza para describir el mundo real (instancias de elementos M1).

NET

.NET es un proyecto de Microsoft para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones. Basado en esta plataforma, Microsoft intenta desarrollar una estrategia horizontal que integre todos sus productos, desde el Sistema Operativo hasta las herramientas de mercado. .NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de Sun Microsystems.

OCL

Lenguaje de Restricciones para Objetos (**OCL**, por su sigla en inglés, *Object Constraint Language*) es un lenguaje declarativo para describir reglas que se aplican a metamodelos MOF, y a los modelos UML, desarrollado en IBM y en la actualidad parte del estándar UML. OCL inicialmente era sólo un lenguaje de especificación formal integrado a UML. Sin embargo, OCL puede ser usado con cualquier metamodelo MOF de OMG, incluyendo UML. El *Object Constraint Language* es un lenguaje de texto preciso que permite definir restricciones y consultas sobre expresiones de objetos de cualquier modelo o metamodelo MOF que de otra manera no pueden ser expresadas mediante la notación gráfica. OCL es un componente clave de la nueva propuesta estándar OMG para la

transformación de los modelos, la especificación QVT. Muchos otros lenguajes de transformación de modelos como ATL, también están construidos utilizando OCL.

OMG

El **Object Management Group** u **OMG** (de su sigla en inglés *Grupo de Gestión de Objetos*) es un consorcio dedicado a la gestión y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización no lucrativa que promueve el uso de tecnología orientada a objetos mediante guías y documentos de especificación de estándares. El grupo está formado por compañías y organizaciones de software como lo son: Hewlett-Packard (HP), IBM, Sun Microsystems, Apple Computer.

OODBMS

En una **base de datos orientada a objetos,** la información se representa en forma de objetos tal como se utiliza en la programación orientada a objetos. Cuando la capacidad de la base de datos se combina con capacidades del lenguaje de programación orientado a objetos, el resultado es un objeto del sistema de gestión de base de datos (ODBMS). Un ODBMS hace que objetos de la base de datos aparecen como objetos de lenguajes de programación orientada a objetos. Un OODBMS extiende el lenguaje de programación con persistencia de datos transparente, control de concurrencia, la recuperación de los datos, consultas asociativas y otras capacidades.

OperationalTransformation (QVT Operacional)

En QVT, metaclase que representa la definición de una transformación unidireccional, expresada imperativamente. Tiene una signatura indicando los modelos involucrados en la transformación y define una operación *entry*, llamada **main**, la cual representa el código inicial a ser ejecutado para realizar la transformación. La signatura es obligatoria, pero no así su implementación. Esto permite implementaciones *black-box* (caja negra) definidas fuera de QVT.

OptimalJ

OptimalJ es la herramienta que se puede considerar que mejor adapta la visión MDA, es decir en ella podemos encontrar los niveles bien diferenciados de PIM, PSM y código. Se trata básicamente de un entorno de desarrollo para aplicaciones empresariales que permite generar con rapidez aplicaciones J2EE completas a partir del modelo de alto nivel (PIM). Y precisamente es ahí donde se encuentra su mayor inconveniente, ya que al ser mono-plataforma obliga a sus clientes a tener un cierto dominio en dicha tecnología. Pero también precisamente por dedicarse exclusivamente a ese entorno, consigue adaptarse a los procesos de

desarrollo de J2EE con modelos de forma sorprendente, generándonos a partir de un PIM una estructura de modelos PSM con sus puentes de comunicación que permiten construir aplicaciones web en muy poco tiempo. Además y en la misma línea implementa todo tipo de patrones para dicha plataforma y consigue un PSM y modelo de código de buena calidad.

Pattern matching

En ciencias de la computación, *pattern matching* **(**coincidencia de patrones**)** es el acto de comprobación de la presencia de los componentes de un patrón definido. En contraste con *reconocimiento de patrones*, la muestra está exactamente determinada. Dicho modelo se refiere a las secuencias ya sea convencional o en estructura de árbol. Coincidencia de modelos se utiliza para probar si los elementos tienen una estructura deseada, para encontrar la estructura pertinente, y para sustitución de elementos. La Secuencia (o específicamente cadena de texto) suele describir patrones usando expresiones regulares (es decir retrospectivas) y coincide utilizando algoritmos respectivos. Las secuencias se pueden ver también como árboles de derivación para cada elemento y el resto de la secuencia, o como árboles que se ramifican en forma inmediata.

PIM

Es el acrónimo inglés de ***P****latform* ***I****ndependent* ***M****odel*, que se traduce al español como *Modelo Independiente de la Plataforma*. MDE identifica dos tipos principales de modelos: modelos con alto nivel de abstracción e independientes de cualquier tecnología de implementación, llamados PIM.

PSM

Es el acrónimo inglés de ***P****latform* ***S****pecific* ***M****odel*, que se traduce al español como *Modelo específico de la Plataforma* modelos que especifican el sistema en términos de construcciones de implementación disponibles en alguna tecnología específica, conocidos como PSM. En MDE un PIM es transformado en uno o más PSMs, es decir que para cada plataforma tecnológica específica se genera un PSM específico.

QVT

En MDD, QVT (Query/ View/ Transformation) es un estándar para transformación de modelos definido por el OMG (Object Management Group). La especificación del lenguaje QVT tiene una naturaleza híbrida, declarativa/imperativa, con la parte declarativa dividida en una arquitectura de dos niveles. Esta especificación define tres paquetes principales, uno por cada

lenguaje definido: QVTCore, QVTRelation y QVTOperational. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios.

RDBMS

Un RDBMS es un Sistema Administrador de Bases de Datos Relacionales. RDBMS viene del acrónimo en inglés *Relational Data Base Management System*. Los RDBMS proporcionan el ambiente adecuado para gestionar una base de datos.

Relation (QVT declarativo)

En QVT, metaclase que representa a la unidad básica de especificación del comportamiento de la transformación en el lenguaje Relations. Se define por dos o más dominios que especifican los elementos de modelos a relacionar. Su cláusula **when** especifica las condiciones necesarias para que la relación se establezca, y su cláusula **where e**specifica la condición que debe satisfacerse por los elementos del modelo que están siendo relacionados.

TefKat

Tefkat es un lenguaje de transformación de modelos y también un motor de ejecución de transformaciones. El lenguaje tiene base formal. El motor es un plug-in Eclipse para el Eclipse Modeling Framework (EMF). Tefkat define un *mapping* desde un conjunto de metamodelos fuente a un conjunto de metamodelos destino. Una transformación Tefkat consiste de reglas, patrones y *templates*. Las reglas contienen un término fuente y un término destino. Los patrones son simplemente términos fuente compuestos y con nombre, similarmente, los *templates* son simplemente términos destino, compuestos y con nombre. Estos elementos están basados en programación lógica pura, sin embargo la ausencia de símbolos de función aporta una reducción significativa de la complejidad.

TemplateExp

Una *TemplateExp* en QVT es una metaclase que especifica un patrón que *machea* elementos de modelos definidos en una transformación. Los elementos del modelo pueden ligarse a variables y esta variable puede ser usada en otras partes de la expresión.

Teoría algebraica de problemas

El formalismo de la *teoría algebraica de problemas*, introduce conceptos algebraicos que son expresados en términos de problemas y soluciones. Las operaciones sobre problemas son aquellas del **Algebra Fork**, un álgebra obtenida extendiendo el Algebra de Relación con un nuevo operador llamado ∇ (*fork).*
La teoría algebraica de problemas se aplica, en este trabajo, a la composición de transformaciones.

Teoría de problemas

Teoría basada en las ideas intuitivas desarrolladas por Pólya. En los últimos años, la Teoría de problemas ha sido usada como fundamento para cálculos de derivación de programas, tales como las álgebras Fork. Analiza los conceptos de problema y solución.

Transformation (QVT declarativo)

En QVT, metaclase que define cómo un conjunto de modelos puede ser transformado en otro. Contiene un conjunto de reglas (*rules)* que especifican su comportamiento en ejecución. Se ejecuta sobre un conjunto de modelos con tipo, especificados por un conjunto de parámetros (*typed model)* asociados con la transformación.

UML

Lenguaje Unificado de Modelado (**UML**, por su sigla en inglés, *Unified Modeling Language*) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es el lenguaje estándar oficial, respaldado por el **OMG** (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir la estructura y el comportamiento del sistema, incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

UML, Profile

Un **perfil**, en el Lenguaje Unificado de Modelado, proporciona un mecanismo de extensión genérico para la construcción de modelos UML en dominios particulares. Un perfil se basa en estereotipos adicionales y valores etiquetados que se aplican a elementos, clases, atributos, métodos, asociaciones, etc. Un perfil es una colección de dichas extensiones y las restricciones que, en conjunto, describen algunos de los problemas del modelado particular, y facilitan la

construcción de modelos en el dominio específico. Los perfiles UML adaptan el lenguaje a áreas específicas: el modelado de negocios y otros. Por ejemplo, el Perfil UML para XML está definido por David Carlson en el libro "Modelado de aplicaciones XML con UML" y describe un conjunto de extensiones a los elementos de modelado básicos de UML para el modelado preciso de esquemas XSD. SysUML es un perfil OMG estandarizado de UML para la realización de ingeniería de sistemas.

UML 2.0, Infrastructure

La Infraestructura UML 2.0 es la primera de dos especificaciones complementarias que representan la principal revisión para el UML de OMG. La segunda especificación es la Superestructura, la cual usa la base arquitectural provista por la primera. La Infraestructura UML 2.0 provee los constructores básicos elementales requeridos para definir lenguajes de modelado, en particular para definir UML 2.0, pero también es útil como base para la definición de otros lenguajes. En el caso de UML, se complementa con la Superstructure.

UML 2.0, Superstructure

La Superestructura UML 2.0 es la especificación que complementa a la Infraestructura definiendo los constructores a nivel usuario requeridos por UML 2.0. Las dos especificaciones complementarias constituyen una especificación completa del lenguaje de modelado UML 2.0.

XML

Es el acrónimo inglés de ***eXtensible Markup Language*** («lenguaje de marcas extensible»), es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Permite definir la gramática de lenguajes específicos. Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

XMI

XMI o *XML Metadata Interchange* (XML de Intercambio de Metadatos) es una especificación para el Intercambio de Diagramas. La especificación para el intercambio de diagramas fue escrita para proveer una manera de compartir modelos UML entre diferentes herramientas de modelado. En versiones anteriores

de UML se utilizaba un esquema XML para capturar los elementos utilizados en el diagrama; pero este esquema no decía nada acerca de cómo el modelo debía graficarse. Para solucionar este problema la nueva Especificación para el Intercambio de Diagramas fue desarrollada mediante un nuevo esquema XML que permite construir una representación SVG (Scalable Vector Graphics).

Referencias bibiográficas

1. Favre Jean-Marie, Estublier Jacky, Blay Mireille. *Beyond MDA : Model DrivenEngineering (L'Ingénierie Dirigée par les Modèles : au-délà du MDA).* Edition Hezmes-Lavoisier, ISBN 2-7462-1213-7. (2006).
2. Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston: Addison Wesley, 2000.
3. Cockburn, Alistair. *Agile Software Development*. Boston: Addison-Wesley, 2002.
4. Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
5. Mellor, Stephen J. and Scott, Kendall and Uhl, Axel and Weise Dirk. *MDA Distilled, Principles of Model_Driven Architecture.* Addison-Wesley, 2004.
6. Object Management Group, *MDA Guide*, v1.0.1, omg/03-06-01 (2003).
7. *Object Management Group* (OMG) http://www.omg.org
8. *Meta Object Facility* (MOF) 2.0. OMG Adopted Specification. October, 2003. http://www.omg.org.
9. *The Unified Modeling Language Superstructure.* version 2.0.,OMG Final Adopted Specification. April 2004. http://www.omg.org.
10. *Java Metadata Interface* (JMI), http://java.sun.com/products/jmi/
11. *XML Metadata Interchange* (XMI), v2.1, http://www.omg.org/cgi-bin/doc?formal/2005-09-01 , (full specification)
12. *OptimalJ* (Compuware), 2007, www.compuware.com/products/optimalj
13. *ArcStyler* 5.5 (Interactive Objects), 2006, www.interactive-objects.com/products/arcstyler
14. *AndroMDA*, v3.2, Nov.2006, www.andromda.org/
15. MOF 2.0 *Query/View/Transformations* - OMG Adopted Specification. March 2005. http://www.omg.org.
16. OMG. *The Object Constraint Language Specification* – Version 2.0, for UML 2.0, revised by the OMG, http://www.omg.org, April 2004.
17. Jouault F., Kurtev I. *Transforming Models with ATL*. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

18. Akehurst D., Howells W., McDonald-Maier K. Kent. *Model Transformation Language*. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

19. Lawley M., Steel J. *Practical Declarative Model Transformation with TefKat*. Workshop in Model Transformation in Practice at the MODELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

20. *The Unified Modeling Language Infrastructure* version 2.0, OMG Final Adopted Specification. March 2005. http://www.omg.org

21. Blankenhorn Kai & Jeckle Mario, *A UML Profile for GUI Layout*, NODe 2004, LNCS 3263, pp.110-121, 2004 Springer-Verlag Berlin Heidelberg 2004.

22. Fuentes, L.,Troya, J.M., Vallecillo, A. *Using UML Profiles for Documenting Web-based Application Frameworks.* Annals of Software Engineering, Vol. 13, pp.249-264, June 2002.

23. Grassi,V., Mirandola, R., and Sabetta, A. *A UML Profile to Model Mobile Systems.* Seventh International Conference on UML Modeling Languages and Applications, UML 2004. Lisboa, Portugal.

24. Ziadi,T. Héloüet, L., and Jézéquel, J.M. *Towards a UML Profile for Software Product Lines*, PFE 2003, LNCS 3014, pp. 129–139, 2004 Springer-Verlag Berlin Heidelberg 2004.

25. Haeberer, A.M. and Baum, G. and Veloso, P.A.S. *On an Algebraic Theory of Problems and Software Development.* Pontificia Universidad Catolica. Research Report MCC 2/87. Rio de Janeiro (1987).

26. Pólya, G., *How to Solve it: a new aspect of the mathematical method*, Princeton University Press, Princeton, 1945 (2nd. ed..1956, repr. 1971)

27. Veloso, P.A.S. *Outline of a mathematical theory of general problems.* Philosophia Naturalis; vol. 2/4 No. 1, pp. 354-365. (1984)

28. Frias, M. and Veloso, P.A.S and Baum, G. *Fork Algebras: past, present and future*. Journal on Relational Methods in Computer Science. Vol.1, 2004, pp.181-216.

29. Czarnecki K. and Helsen S.. *Feature-based survey of model transformation approaches.* IBM System Journal, Vol. 45, No 3, 2006.

30. Kleppe, Anneke. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006.

31. Bézivin, J., Büttner, F., Gogolla, M., Jouault,F., Kurtev, I., Lindow, A. *Model Transformations? Transformation Models!*. MODELS 2006 Int. Conf

proceedings. Lecture Notes in Computer Science. ISSN 0302-9743. volume 4199 © Springer-Verlag. (2006)

32. Maddux, Roger D. *Relation Algebras*, vol.150 in Studies in Logic and the Foundations of Mathematics. Elsevier Science. (2006).

33. Lano, K., *Catalogue of Model Transformation*. 2006, www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf

34. Belaunde, Mariano. *Transformation composition in QVT*. First European Workshop on Composition of Model Transformations at ECMDA 2006.

35. Wagelaar, Dennis. *Blackbox Composition of Model Transformations using Domain-Specific Modelling Languages*. First European Workshop on Composition of Model Transformations - CMT at ECMDA 2006

36. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. *Composing Domain-Specific Design Environments.* IEEE Computer 34 (2001) 44–51

37. Tolvanen, J.P., Rossi, M. *MetaEdit+: defining and using domain-specific modeling languages and code generators.* In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003), Anaheim, CA, USA, ACM Press (2003) 92–93

38. Blanc, X., Gervais, M., Lamari, M. and Sriplakich, P.. *Towards an integrated transformation environment (ITE) for model driven development (MDD).* In Proceedings of the 8th World Multi- Conference on Systemics, Cybernetics and Informatics (SCI'2004), USA, July 2004.

39. Bergstra, J.A. and Klint, P.. The discrete time toolbus – a software coordination architectture. *Science of Computer Programming*, 31:205–229, 1998.

40. Jézéquel, Jean-Marc and Ho, Wai-Ming and Le Guennec, Alain and Pennaneach, Franccois. *UMLAUT: an extendible UML transformation framework*. In Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999.

41. Bézivin, J. Bouzitouna, S. Del Fabro, M. D. Gervais, M. P. Jouault1, F. Kolovos, D. Kurtev I. et Paige R. F. *A Canonical Scheme for Model Composition*. Proceedings of the European Conference on Model Driven Architecture - (ECMDA-FA'06), LNCS, Springer-Verlag, June 2006.

42. *Atlas Model Weaver Project* Web Page. http://www.eclipse.org/gmt/amw/, 2005.

43. Bouzitouna, M. P. Gervais and X. Blanc, *Model Reuse in MDA*, Proceedings of the International Conference on Software Engineering Research and Practice (SERP'05), Las Vegas, USA, June 2005.

44. Bouzitouna and M. P. Gervais, *Composition rules for PIM reuse*, Proceedings of the Second European Workshop on Model Driven Architecture with

Emphasis on Methodologies and Transformations (EWMDA'04), Canterbury, UK, September 2004, pp36-43

45. Kolovos, Dimitrios S., Paige, Richard F. and Polack Fiona A. C.. *The Epsilon Object Language (EOL).* In Proceedings of Model Driven Architecture Foundations and Applications: 2nd European Conference, ECMDA-FA, vol 4066 of LNCS, pages 128– 142, Spain, June 2006.
46. Kolovos Dimitrios, Paige Richard and Polack Fiona. *Merging Models with the Epsilon Merging Language (EML).* In Proceedings of MoDELS 2006 Conference, Session Model Integration. Genova, Italy, October 2006.
47. Jouault, F. and Kurtev, I.. *On the Architectural Alignment of ATL and QVT*. Proceedings of Symposium on Applied Computing (SAC 06), ACM Press, April 2006.
48. Oldevik, J. *Transformation Composition Modeling Framework*. DAIS 2005. Lecture Notes in Computer Science 3543, pp. 108-114. (2005).
49. Marvie, Rafael. *A Transformation Composition Framework for Model Driven Engineering*, LIFL 2004 n10, November 2004
50. Willins, Harris. *The Side Transformation Pattern*, at the Software Evolution through Transformations (SETra), 2004.
51. Willink. Edward. *UMLX : A graphical transformation language for MDA*, GMT Consortium, www.eclipse.org/gmt, 4 September 2003
52. Olsen, Gøran K.; Aagedal, Jan; Oldevik, Jon. *Aspects of Reusable Model Transformations.* First European Workshop on Composition of Model Transformations - CMT at ECMDA 2006
53. MODELWARE, *D1.6 Definition of Reusable Transformations, 2006* *http://www.modelwareist.org*.
54. Goknil, Arda; N. Yasemin Topaloglu. *Composing Transformation Operations Based on Complex Source Pattern Definitions.* First European Workshop on Composition of Model Transformations - CMT at ECMDA 2006
55. Pons C., Giandini R., Pérez G., Pesce P., Becker V., Longinotti J., Cengia J.,Correa N. and Labaronnie P. *Precise Assistant for the Modeling Process in an Environment with Refinement Orientation.* In "UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers". Lecture Notes in Computer Science number 3297. Springer, Oct., 2004.
56. *ePlatero Home page*. http://sol.info.unlp.edu.ar/eclipse.
57. *Eclipse* http://www.eclipse.org.
58. *Eclipse Modeling Framework* (EMF), http://www.eclipse.org/emf/

59. Vanhooff, Bert; Dhouha, Ayed et al. *UniTI: A Unified Transformation Infrastructure*. In Proceedings of MoDELS 2007 Conference, Session Model Transformation. Nashville, USA. Lecture Notes in Computer Science number 4735. Springer, October 2007.

60. Stevens, Perdita. *Bidirectional Model Transformations in QVT*. In Proceedings of MoDELS 2007 Conference, Session Model Transformation. Nashville, USA. Lecture Notes in Computer Science number 4735. Springer, October 2007.