FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Composición de transformaciones en MDE

AUTORES: Juan Manuel Cappi, Sebastián Ortega

DIRECTOR: Dra. Claudia Pons

CODIRECTOR: Dra. Roxana Giandini

CARRERA: Licenciatura en Informática

Resumen

La Ingeniería de Software Conducida por Modelos (MDE) se ha convertido en un nuevo paradigma 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 estos. MDE promete una mejora de la productividad y de la calidad del software generado debido a que reduce el salto semántico entre el dominio del problema y de su solución, al mismo tiempo que genera una reducción en 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 esta tesis se analizan operaciones algebraicas para la composición de transformaciones de modelos en el lenguaje QVT. También se realiza un estudio sobre las herramientas actuales de ejecución de transformaciones de modelos y soluciones de composición de transformaciones de modelos. Se presenta la Herramienta Calculadora de Composiciones capaz de componer definiciones de transformaciones escritas en cualquiera de los dos aspectos del lenguaje QVT: declarativo y operacional. La conclusión del trabajo reflejado en el desarrollo de un caso práctico de composición de modelos.

Líneas de Investigación

Las siguientes son las lineas de investigaciones en las cuales se encuadra el presente trabajo:

- Ingeniería de Software
- Desarrollo de Software Dirigido por Modelos
- Transformaciones de Modelos
- Herramientas de soporte a MDD

Trabajos Realizados

El trabajo se basa en el lenguaje standard QVT y en el desarrollo formal de operaciones algebraicas de composición de transformaciones de modelos. A partir de ahí, hemos realizado:

- Un análisis desde el punto de vista práctico del lenguaje y de las operaciones algebraicas.
- Un análisis del estado de arte de las herramientas de transformación de modelos y de las soluciones de composiciones de transformación de modelos.
- El desarrollo de una herramienta de software que implementa las operaciones algebraicas.
- El desarrollo de un caso práctico del uso de la herramienta para componer transformaciones.

Conclusiones

Las definiciones de las operaciones algebraicas de composición entre transformaciones brindan un importante avance en la Ingeniería de Software Dirigida por Modelos. Cada una de estas operaciones tiene un sustento matemático que le otorga validez formal. La implementación de una herramienta que permite componer estas transformaciones de manera automática brinda una importante mejora en el nivel de productividad del desarrollo de software. La Herramienta Calculadora de Composiciones es la materialización de esta teoría, y al mismo tiempo la contribución a la comunidad de desarrollo software conducida por modelos.

Trabajos Futuros

Entre los trabajos futuros se encuentra

- La incorporación de nuevos operadores en la Herramienta Calculadora de Composiciones.
- El planteo de mejoras desde el punto de vista práctico de las definiciones de las operaciones algebraicas para mejorar su usabilidad.
- Incorporar mecanismos de interoperatividad con otras herramientas de transformación de composiciones como Validadores de Código QVT, herramientas de ejecución de transformaciones, etc.
- Realizar estudios de derivación del código para mostrar que la herramienta cumple su función con una correcta correspondencia con la Teoría de Problemas.

Fecha de la presentación: Noviembre 2008

Juan

Agradecimientos:

Con este trabajo se cierra una etapa que para muchos, incluyéndome, quizá se haya dilatado demasiado. Por eso quiero agradecer especialmente a todos aquellos que me ayudaron y me dieron el último empujoncito que hacía falta. A Sebas, amigo entrañable y compañero en esta travesía, que tiró del carro duro y parejo a lo largo de todo el año. Sin sus pilas y su entrega, no estaría escribiendo esto ahora. A Claudia, incondicional como pocas personas, que siempre estuvo ahí desde el primer momento dándonos una mano, guiándonos, motivándonos y sacándonos del pantano cuando hizo falta. A Andrés, amigo incondicional si los hay, que me acompañó y me bancó en las buenas y en las malas, en este último trecho y durante toda la carrera. A Fede, Juli, Guille, Seba, Slay, Caro, Ama; los amigos que me regaló la facultad y me acompañaron durantes estos 10 increíbles años, los mejores de mi vida.

Finalmente a los más importantes, a las personas que me dieron el empujoncito para arrancar. A mis padres, de quienes estoy orgulloso y agradecido, mi guía y modelo, quienes me dieron la oportunidad de ser quien soy. A mis abuelos que no se cansaron de esperarme, mis hermanos, mis tíos, toda mi familia. Gracias.

.

Sebastian

Agradecimientos:

A mi familia, que es todo mi sustento.
A mi hermano, que me da todo su apoyo y confianza de manera incondicional.
A mi mamá, que es la mejor mamá del mundo y no me alcanza la vida para devolverle todo lo que hace por mí.
A Juli, que tiene el corazón más grande que alguna vez haya conocido y que me enseñó a creer en que la perseverancia supera cualquier obstáculo.
A Claudia, que en todo momento desplegó una generosidad tan grande que ni en mis mejores sueños podría haberla imaginado y con quien ya no me quedan palabras de agradecimiento para reconocer su dedicación y su apoyo.
A Juan, mi amigo del alma, compañero de la vida, compañero de estudio y compañero de trabajo, con quien compartimos sueños y atravesamos juntos esta etapa, que me hace sentir inmensamente orgulloso por brindarme su increíble amistad.
Y a mi papá, que lo extraño mucho, pero que me sigue dando fuerzas desde el cielo.

Índice General

Índice de Figuras

Capítulo 1

Introducción

Este capítulo describe el contexto en el que se enmarca esta tesis; 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 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 el 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, se enumeran 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 a medida que la fase de codificación progresa. 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 *ExtremeProgramming* (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 enfoque 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, cosa que sucede generalmente después de que se entrega el 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 solo 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, servicios de WEB).
- Son la solución para algunos problemas reales (por ejemplo, XML para el problema de intercambio entre plataformas heterogeneas, o Java para el problema de la portabilidad).
- La empresa que desarrolla la herramienta deja de soportar 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 haciendo foco 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 y 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 versiones o incluso épocas diferentes. Por ejemplo, un sistema WEB necesita usar una base de datos para almacenar información.

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 el concepto “Cliente” definido en el PIM, sabemos qué clase Java genera y qué 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 gran 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

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

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 fundamentales.

Una solución a este problema, a nivel de código, es la facilidad de generar la documentación directamente desde el código fuente, asegurándose de que esté 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 solo en el PIM, el cual a su vez tiene un nivel más abstracto que el del código. El PIM es utilizado entonces para generar PSMs, los cuales posteriormente serán utlizados 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 efectuados 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 la utilización de un conjunto de estándares tales como Meta Object Facility (MOF) [8], UML [9], JMI [10] o XMI [11][2]. Su objetivo es el de 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 mejor preparados para hacer frente 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).

[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**.

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], cuya estandarización fue alcanzada en Abril del año 2008.

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, aunque 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 define 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 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].

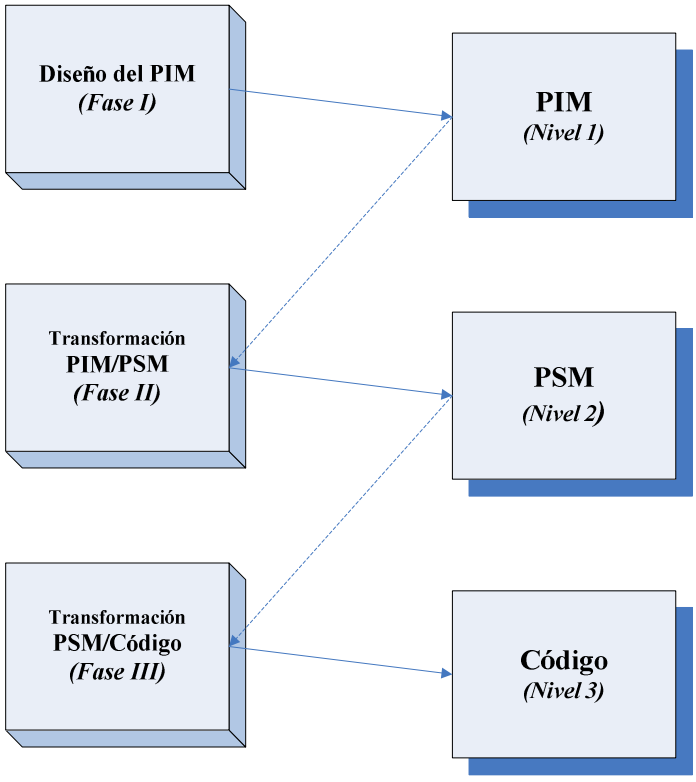

Figura 1-1. Pasos y niveles en el desarrollo MDA

1.2 Nuestra Propuesta: Una herramienta que implementa Composiciones Algebraicas entre Transformaciones de Modelos

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.

Los trabajos en los cuales nos basamos fueron escritos por Giandini y Pons en [16, 17, 18, 19], plantean una especificación adaptada (simplificada) del lenguaje QVT que permite a efectuar operaciones de composición entre transformaciones. Tanto la especificación del lenguaje adaptado como la especificación de las composiciones algebraicas definidas, fueron hasta el momento abordadas principalmente desde el aspecto teórico.

Nuestra propuesta es continuar el trabajo haciendo un estudio práctico de las composiciones de transformaciones de modelos concluyendo con la elaboración de una herramienta que automatiza el proceso de composición.

1.2.1 Objetivo de la Tesis

El objetivo de este trabajo es la implementación de una herramienta de software que permita la composición algebraica de transformaciones de modelos construidos en el contexto de un proceso de desarrollo de software dirigido por modelos [1, 4, 5] utilizando el estándar QVT [15]. El fundamento teórico de nuestro trabajo fue definido por Giandini y Pons en [16] y [17].

Esto se logra mediante el alcance de los siguientes sub-objetivos:

- El estudio de las operaciones algebraicas definidas en Giandini y Pons [16] y [17] entre transformaciones de modelos desde una visión teórico-práctica
- El estudio del estado de arte de las herramientas que implementan transformaciones de modelos y cumplen con la especificación standard de QVT [15]
- El desarrollo de una herramienta que implementa las operaciones algebraicas entre transformaciones de modelos.

1.2.2 Aportes de la Tesis

En este trabajo presentaremos conclusiones sobre las operaciones algebraicas en base a la implementación de la Herramienta Calculadora de Composiciones. Esta herramienta contará con analizadores sintácticos y analizadores semánticos que nos permitirá explotar todo el poder del lenguaje QVT [15].

Como aporte principal, ayudaremos a:

- Validar la definición de las operaciones algebraicas.
- Permitir la composición automática de transformaciones escritas de manera modular.

1.3 Organización del Trabajo

Este trabajo se divide en dos grandes partes:

- La primera de ellas tiene como objetivo situar a nuestro trabajo en un marco teórico-conceptual, refiriéndonos y citando a los trabajos anteriores en los cuales se basó el nuestro. Los capitulos del 2 al 5 hacen repaso de los lenguajes sobre los que trabajaremos, la teoria de problemas y las definiciones de las operaciones de composición según el estudio teórico de Giandini y Pons en [16, 17].
- La segunda parte tiene como objetivo abordar los aspectos prácticos del diseño y construcción de la Herramienta Calculadora de Composiciones de Transformaciones de Modelos. Los capítulos del 6 al 8 van desde el estudio de la situación actual de las herramientas existentes para transformaciones de modelos, la descripción de nuestra herramienta y un ejemplo de un caso de uso real. El capitulo 9 incluye las conclusiones finales de la tesis.

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.

En el capítulo 3 repasamos 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 4 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 5 introducimos la formalización de las operaciones de composición de transformaciones de modelos -a nivel declarativo y a nivel operacional-. Dicha formalización está definida en base al 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.

En el capítulo 6 presentamos el estado actual de las soluciones disponibles para ejecutar transformaciones de modelos y de las soluciones disponibles para efectuar composiciones de modelos y transformaciones..

En el capítulo 7 presentamos a la herramienta Calculadora de Composiciones: el software que automatiza el proceso de composición de transformaciones.

En el capítulo 8 presentamos un ejemplo real de dos composiciones que se componen usando nuestra herramienta implementada, mientras que el capítulo 9 presenta las conclusiones finales y trabajo futuro.

Capítulo 2

Conceptos Básicos Sobre 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 Modelo Relacional.

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).

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

En el **nivel M1** (el modelo del sistema), los 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.

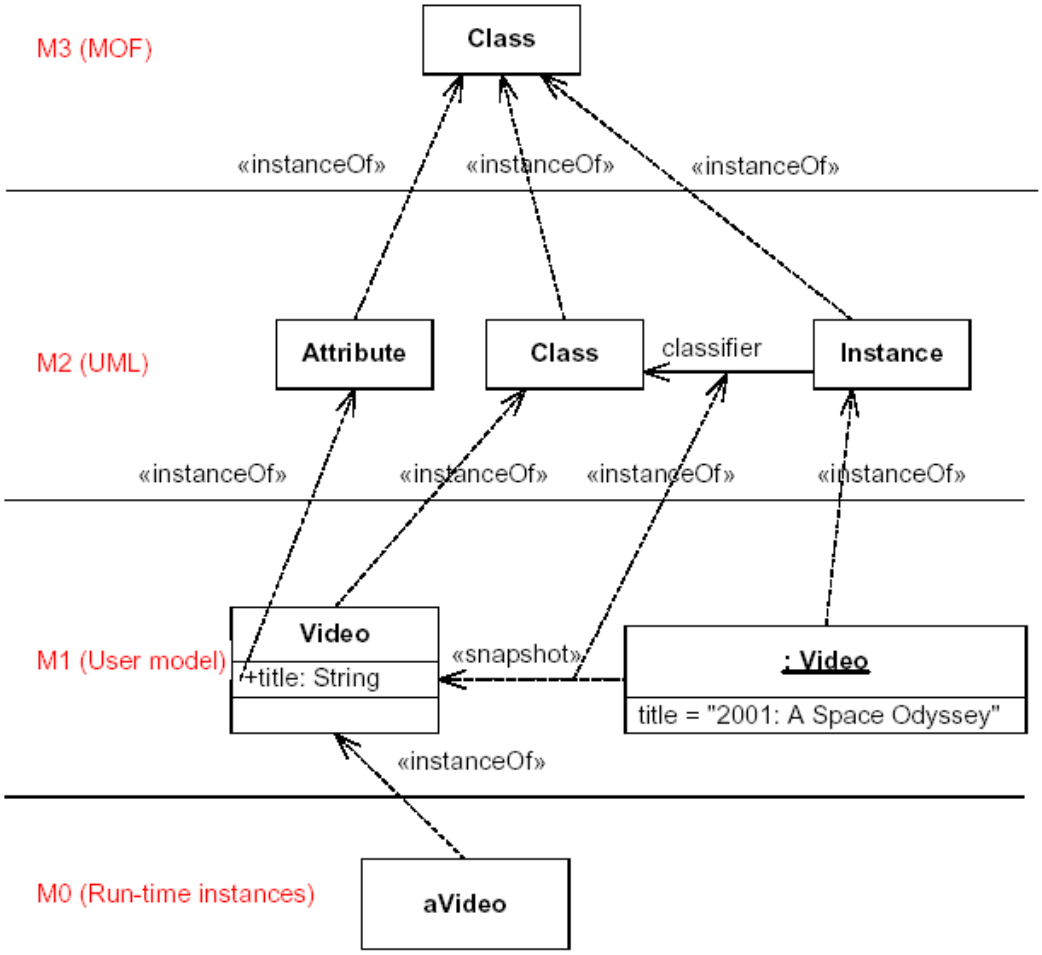

Figura 2-1. Ejemplo de la Arquitectura 4 capas de Modelado

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 se eligió para la definición del lenguaje que introducimos 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 se podrá extender a MOF, por ser una especificación cerrada. Analicemos entonces, otras especificaciones, estrechamente vinculadas a MOF, que define la OMG: una de estas especificaciones es la Infraestructura 2.0 [24] 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.

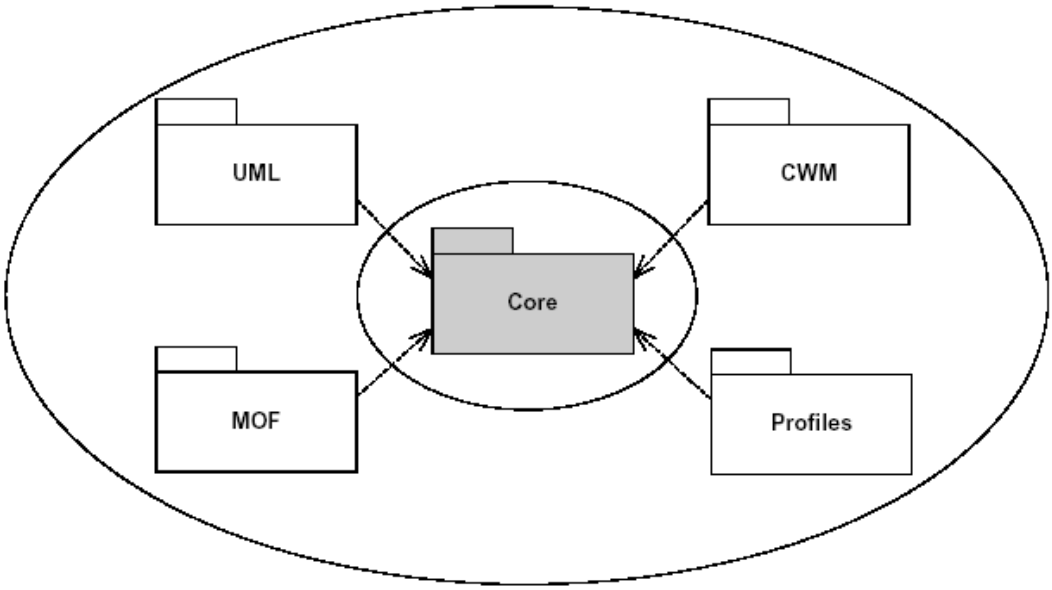

Figura 2-2. El rol del Paquete Core de la Infraestructura

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*.

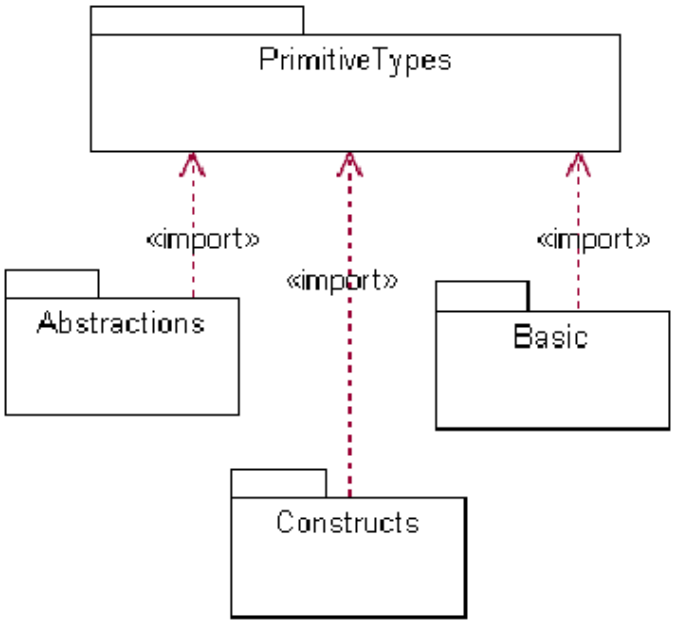

Figura 2-3. Sub-paquetes contenidos en el paquete Core de la Infraestructura

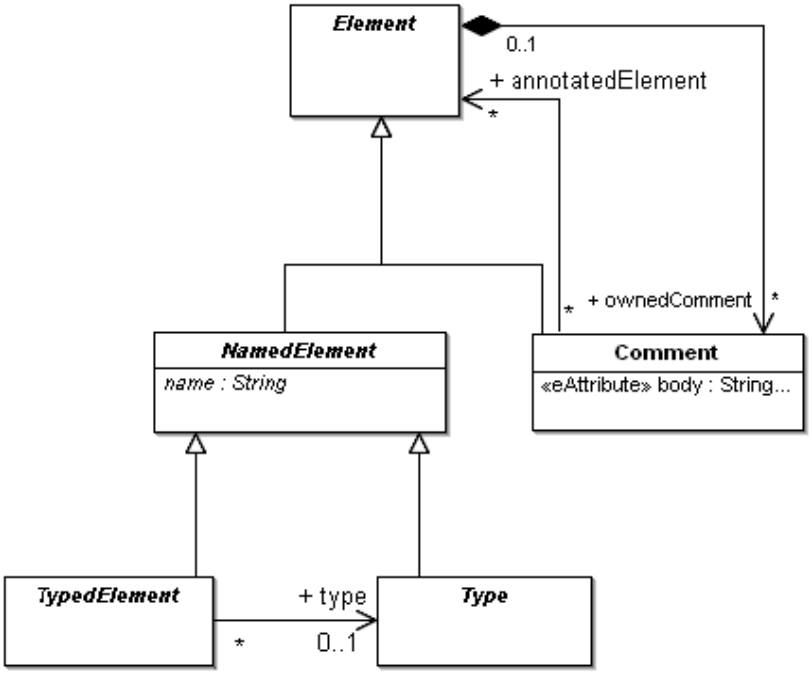

Figura 2-4. Diagrama de Tipos del paquete Basic

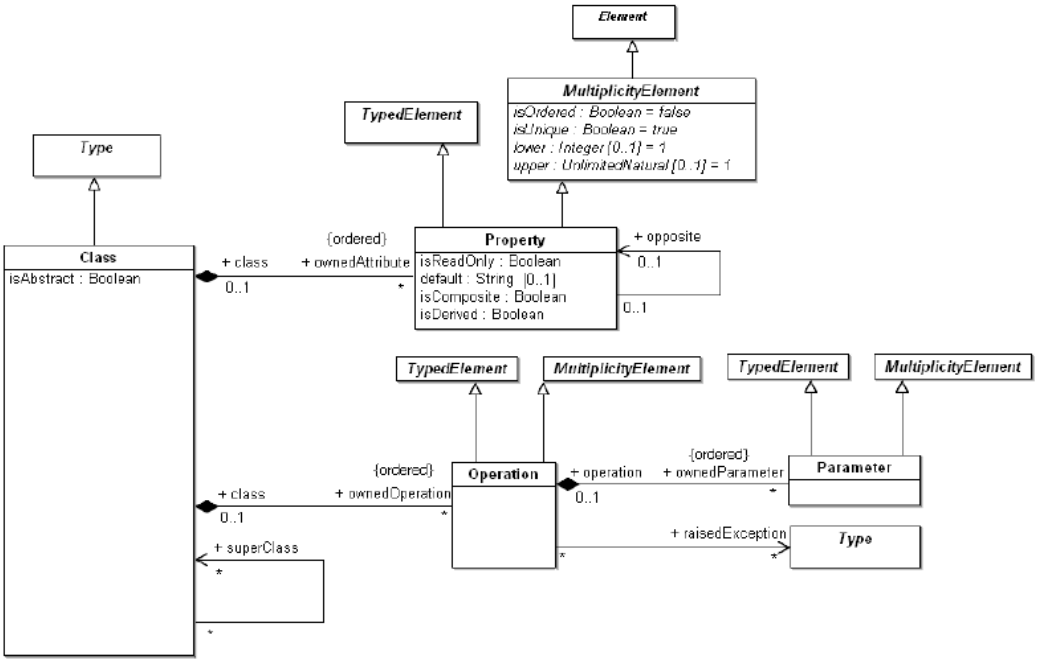

Figura 2-5. Diagrama de Clases del paquete Basic

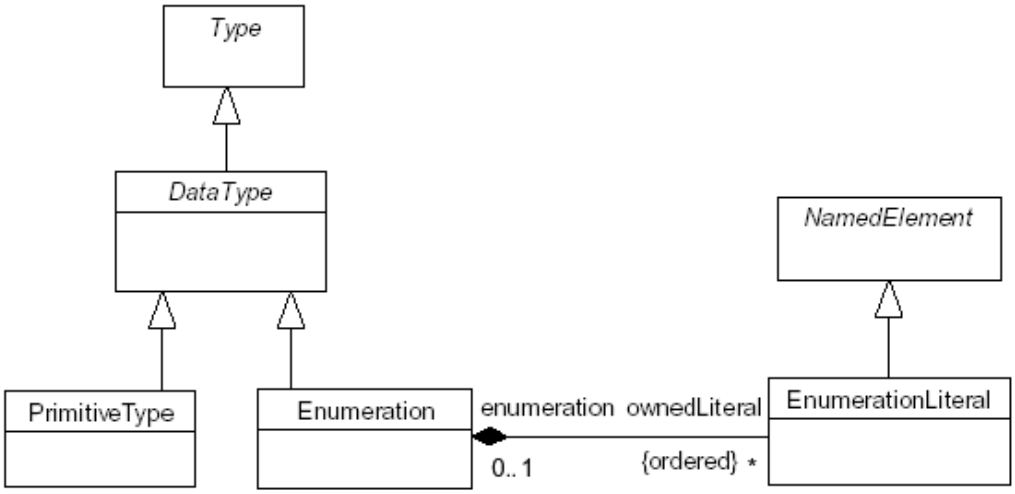

Figura 2-6. Diagrama de Tipos de Datos del paquete Basic

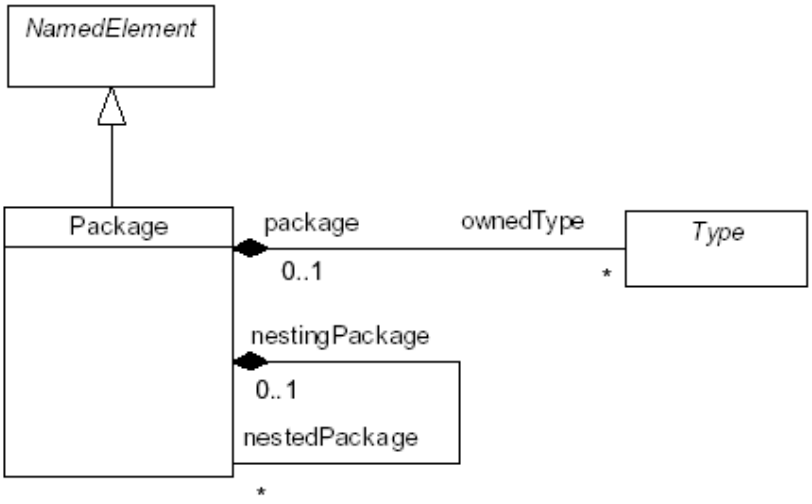

Figura 2-7. Diagrama de Paquetes del paquete Basic

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.

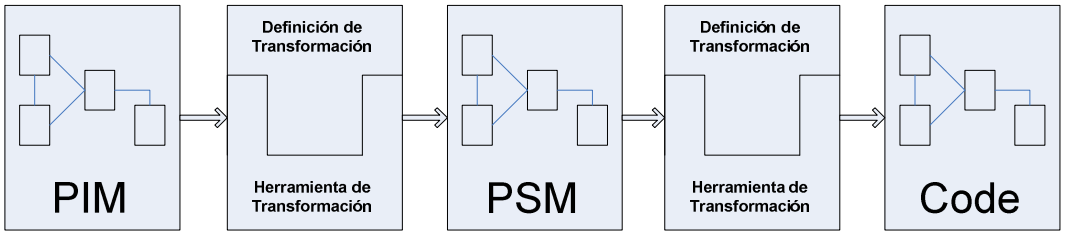

Figura 2-8. Definición de transformación dentro de herramientas

La herramienta utiliza la misma definición de transformación para cada una de ellas para 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 Transformaciones 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 transforma ciones. 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 [20], y todos los paquetes del lenguaje dependen de EMOF (EssentialMOF) [8].

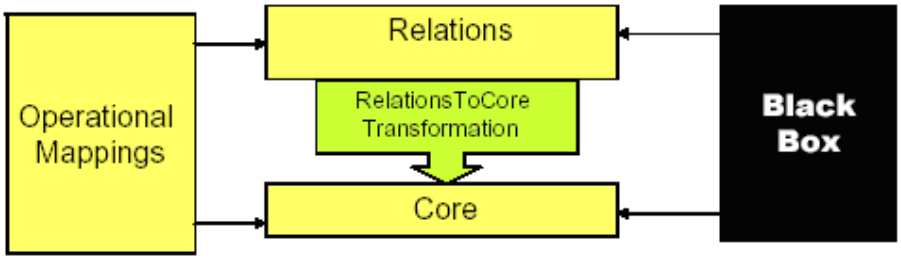

Figura 2-10. Relaciones entre metamodelos QVT

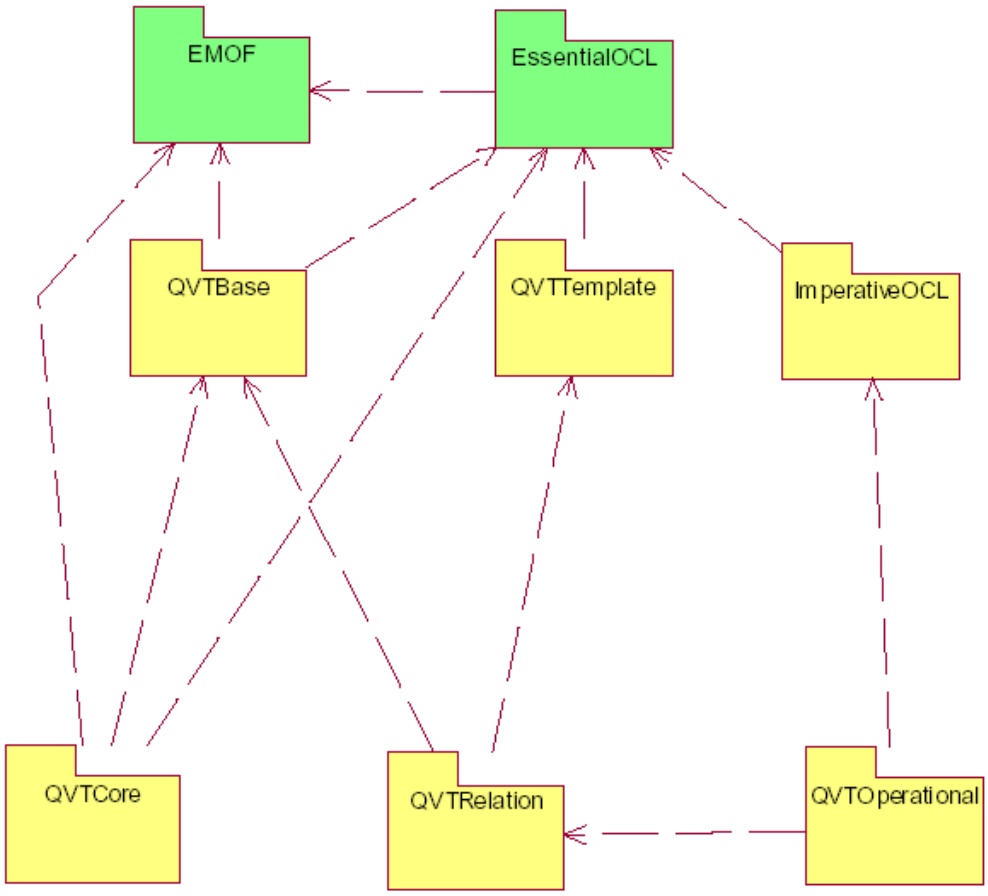

Figura 2-11. Dependencias de paquetes en la especificación QVT

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 transforma ción 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 UMLToRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) { …

En esta declaración llamada "UMToRdbms," 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 forza r el cumplimiento de la relación.

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 se 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:

```
top relation PackageToSchema { /* transforma cada paquete en un
schema*/
   domain uml p:Package {name=pn}
   domain rdbms s:Schema {name=pn}
}
top relation ClassToTable { /* transforma cada clase persistente en
tabla*/
   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 UMLToRdbms (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.

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.

Restricción implícita

Recordemos que cada uno de los patrones puede ser visto como un conjunto de variables y un conjunto de restricciones de elementos del modelo ligadas a esas variables que deben ser satisfechas para calificar como una ligadura válida del patrón (pattern matching). A continuación daremos un ejemplo de una restricción implícita por un patrón dado:

Patrón:

```
c:Class {kind=Persistent, name=cn, attribute=a:Attribute {}}
```

Restricción implícita:

```
c.kind = Persistent and c.name = cn and c.attribute->includes(a)
```

Sintaxis Abstracta del Lenguaje Relations

En esta sección presentamos los principales paquetes y algunas de las metaclases de cada uno de ellos que conforman 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***:

.

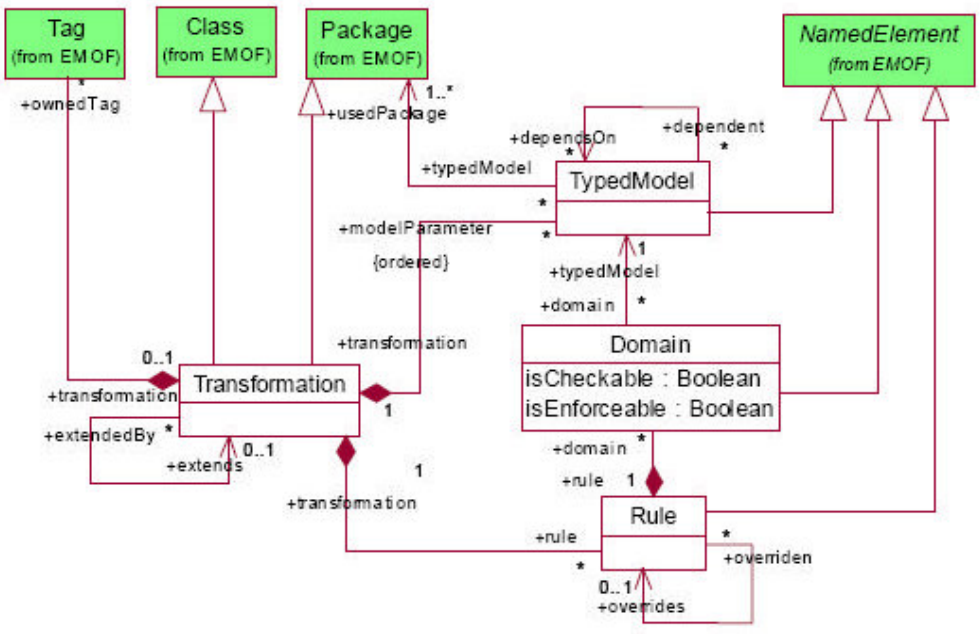

Figura 2-12. Paquete QVT Base – Transformaciones y Reglas.

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.

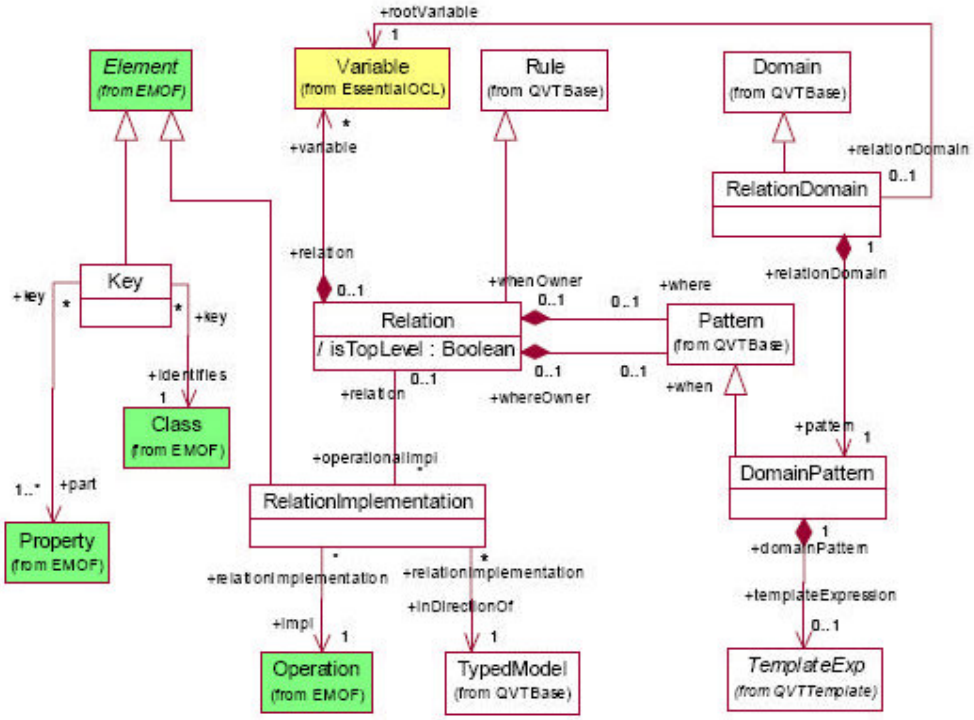

Figura 2-13. Paquete QVT Relations

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* **e**specifica 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.

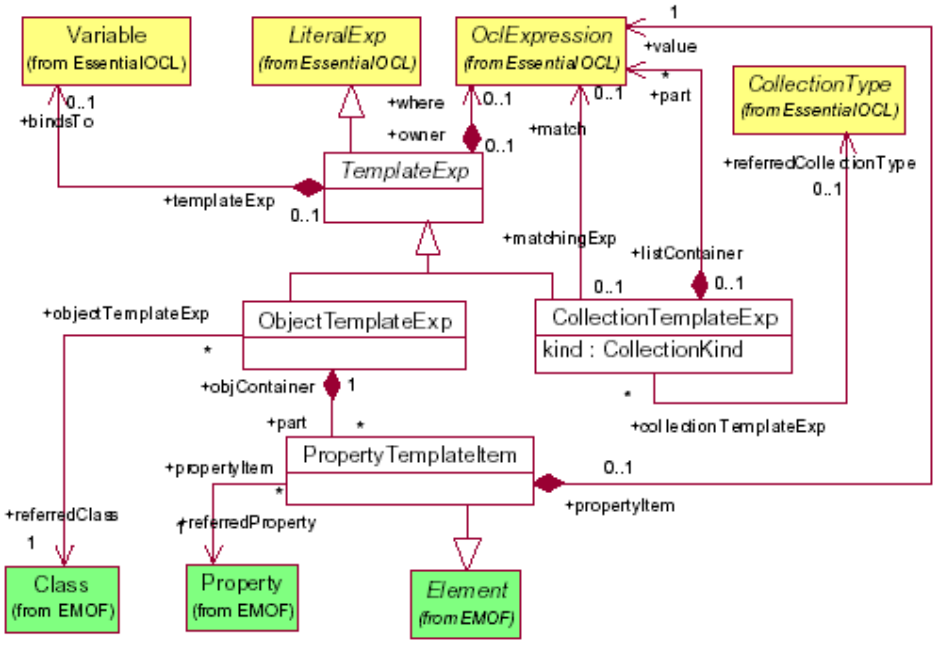

Figura 2-14. Paquete QVT Template

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 correponder 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 implementación 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 las 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 utilizadas para implementar una o más relaciones de una especificación del lenguaje *Relations*. Una transformación escrita usando solamente operaciones *Mapping* es llamada transformación Operacional.

Sintaxis Abstracta del Lenguaje Operational Mappings

En esta sección presentamos los principales paquetes y algunas de las metaclases de cada uno de ellos que conforman 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].

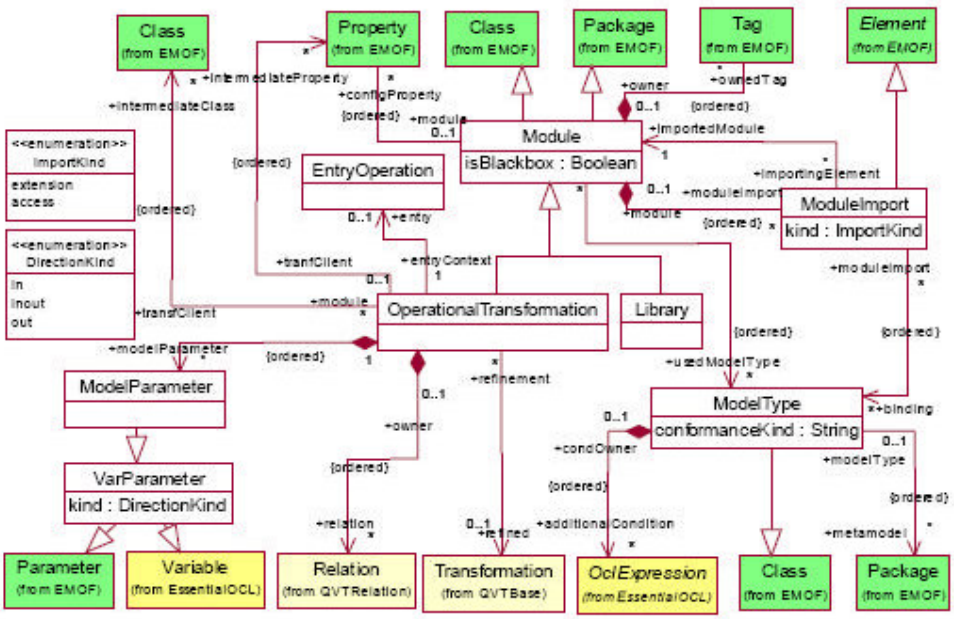

Figura 2-15. Paquete QVT Operational – Transformaciones Operacionales

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 de *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 reutilización 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*.

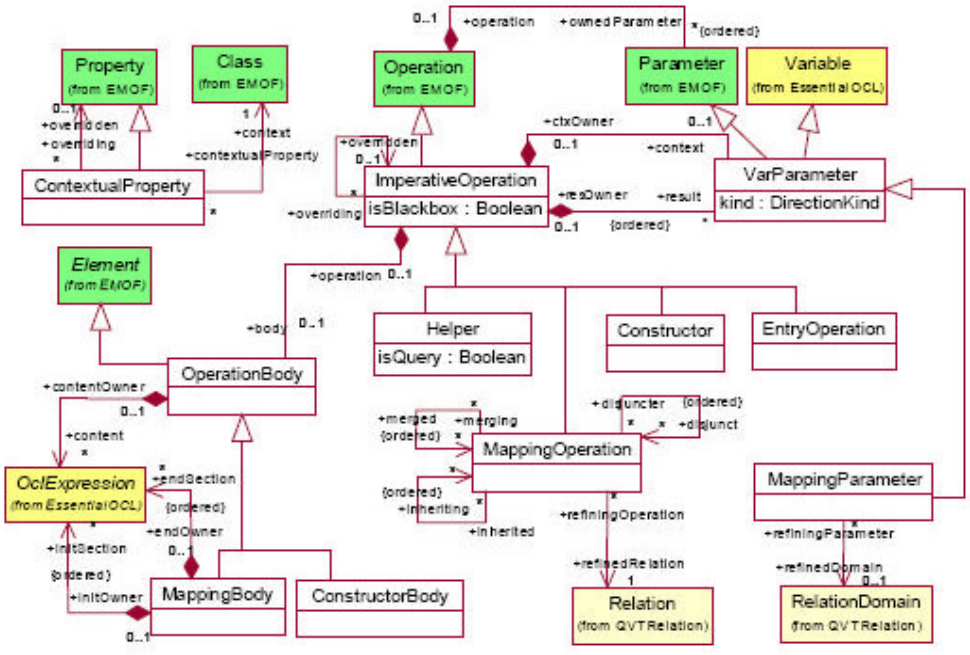

Figura 2-16. Paquete QVT Operational – Operaciones Imperativas

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.

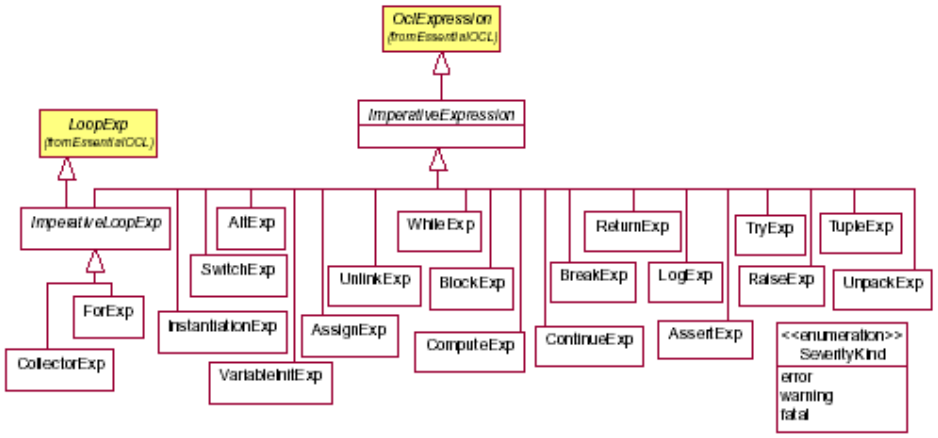

Figura 2-17. Paquete OCL Imperativo

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 UML2Rel 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 UML2Rel y significa que cada vez que UML2Rel se cumple para una clase y una tabla, la relación Attr2Col para sus atributos y columnas respectivamente, se cumple también:

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

top relation Class2Table {
   checkonly domain Uml c: Class {name = n }
   enforce 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}
       }
    }
    enforce 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.

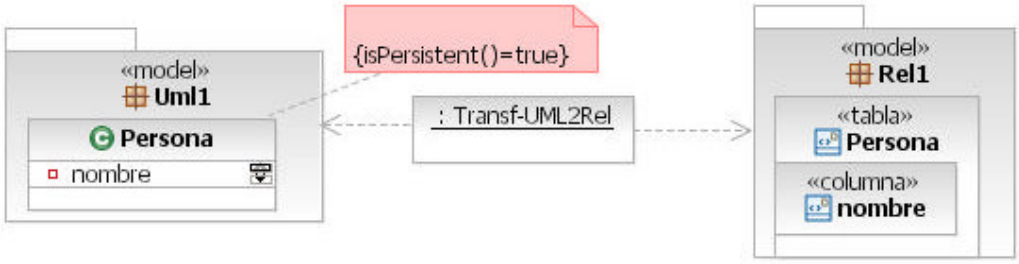

Figura 2-18. Una instanciación gráfica de la Transformación.

2.5 Notación equivalente entre patrones y restricciones OCL en QVT

En el ejemplo de la sección 2.4, se puede observar una de las maneras en que aparecen las especificaciones escritas en QVT. Esto radica en que las relaciones de consistencia entre dominios de una *relation* se especifican mediante expresiones *template* escritas en la definición de cada uno de los dominios. Entonces, para concretar la transformación, se debe poducir un *pattern matching* entre estas expresiones y los elementos del dominio de referencia.

En esencia, el lenguaje QVT Relations (declarativo) especifica dos conjuntos que deben mantener correspondencia entre sus elementos: el conjunto de origen, y el conjunto de destino (o conjunto de los elementos transformados)

Esa correspondencia se puede especificar de dos maneras:

1) Definiendo *template expressions* asociados a los *domains* donde cada elemento de un dominio debe corresponderse con un elemento del otro/s dominio/s. Aquí, cada elemento del dominio de origen que satisfaga el *template expression* debe tener su correspondiente elemento en el conjunto de destino, o codominio, que satisfaga el *template expression* definido para el *domain* de destino.
2) Otra forma es mediante especificaciones restricciones entre elementos de un dominio respecto de elementos del otro dominio mediante el lenguaje OCL en el *when* y el *where* de la *relation*

Una misma relation puede especificar correspondencia entre ambos conjuntos mediante la combinación de los dos mecanismos de especificación de correspondencia. Ambos mecanismos tienen la misma utilidad, pero le brindan dos alternativas al programador del lenguaje QVT para que éste elija el que mejor le resulte para expresar correspondencia entre conjuntos.

En el contexto de las composiciones entre modelos, veremos mas adelante que no está definida la manera de componer dominios cuando éstos están expresados en la forma de *template expressions* pero sí es posible realizar composiciones de relaciones cuando los dominios están expresados en forma de restricciones escritas en OCL. Por esto, entonces, resaltamos la importancia de escribir las transformaciones utilizando la propuesta restricciones en los *when* y *where* para asegurarnos de poder realizar las composiciones.

Ejemplo de equivalencias

En cada dominio se pueden expresar 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, pueden no usarse expresiones *template* ni *pattern matching*.

Las condiciones previas que el atributo debe cumplir, van en la cláusula when, mientras que las de salida se incluyen en la cláusula *where* de la relación escrita en forma de resctricciones en OCL. El ejemplo anterior, escrito con restricciones quedaría:

```
transformation UML2Rdbms (Uml: UML2.0, Rel: RDBMS) {
top relation Class2Table {
   checkonly domain Uml c: Class { }
where                    //El template expression de acá, pasa como
   enforce domain Rel t: Table { }
   when { c.isPersistent = true }
   where {
     c.name = t.name and Attr2Col (c, t);
   }
}
relation Attr2Col {
   checkonly domain Uml c: Class { }
   enforce domain Rel t: Table { }
   when { not a.isMultivalued() and a.type.oclIsKindOf ( DataType ) }
   where  {t.column.type  =  c.attribute.type  and  t.column.name  =
t.attribute.name;}
}
}
```

2.6 Conclusión del Capítulo

En este capítulo hemos introducido a los conceptos básicos sobre transformaciones. Se presentaron los metamodelos de QVT definidos en la especificación: metamodelo del lenguaje declarativo y metamodelo del lenguaje operacional. Hemos visto los paquetes que intervienen. Luego hemos visto la sintaxis y la semántica de cada uno de estos aspectos.

Cabe destacar que existen también otros lenguajes híbridos definidos a partir de los requerimientos especificados por el estándar QVT, como ATL [21], Kent [22] y TefKat [23][3], (algunos de ellos ya cuentan con herramientas CASE probadas), que definen a su vez su propia sintaxis, obligando al usuario a aprender nuevos lenguajes.

En este capítulo hemos visto además las maneras equivalentes de escribir una relation, tanto sean utilizandas expresiones template (patrones) como utilizandas restricciones OCL. Estas características serán aprovechadas para sortear futuros obstáculos en las definiciones de las operaciones algebraicas que nuestra herramienta implementa y que presentaremos en los próximos capítulos.

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

Capítulo 3

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.

3.1 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 utilizada como fundamento de 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 *data 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 e*specificación 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 3-1.

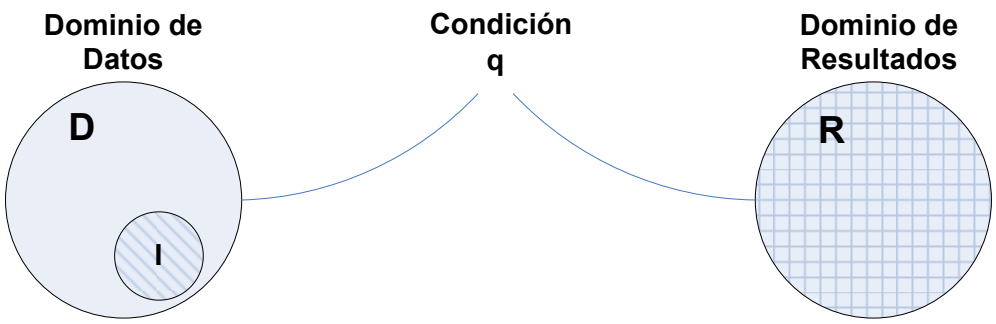

Figura 3-1. Diagrama de un Problema

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 para 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 verifica 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.**

3.2 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 $\mathcal{L}_D$ que tiene su propia sintaxis y una semántica dada por una función **m**. El rol de esta función semántica **m** es permitir dar un significado a las *sentencias* de problemas, asociando cada *sentencia* **Spec,** escrita en el lenguaje $\mathcal{L}_D$, al *problema* **P** = **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 **m** que permite que nosotros definamos 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 $\mathcal{L}_A$ que, además de su sintaxis, tiene semántica dada por una función **u**. El rol de esta función es, como en caso de **m** para problemas, asociar cada (texto de) *programa* **Impl**, escrito en lenguaje $\mathcal{L}_A$, a la *a-función* **d** = **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*.

3.3. 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.

3.3.1 QVT Declarativo vs. QVT imperativo

Recordemos brevemente los conceptos principales de QVT, ya descriptos más detalladamente en las secciones 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 debe 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 $\mathbf{m}$ define la semántica de expresiones declarativas en términos de *problemas*, mientras que la función $\mathbf{u}$ define la semántica de construcciones imperativas en términos de *soluciones.*

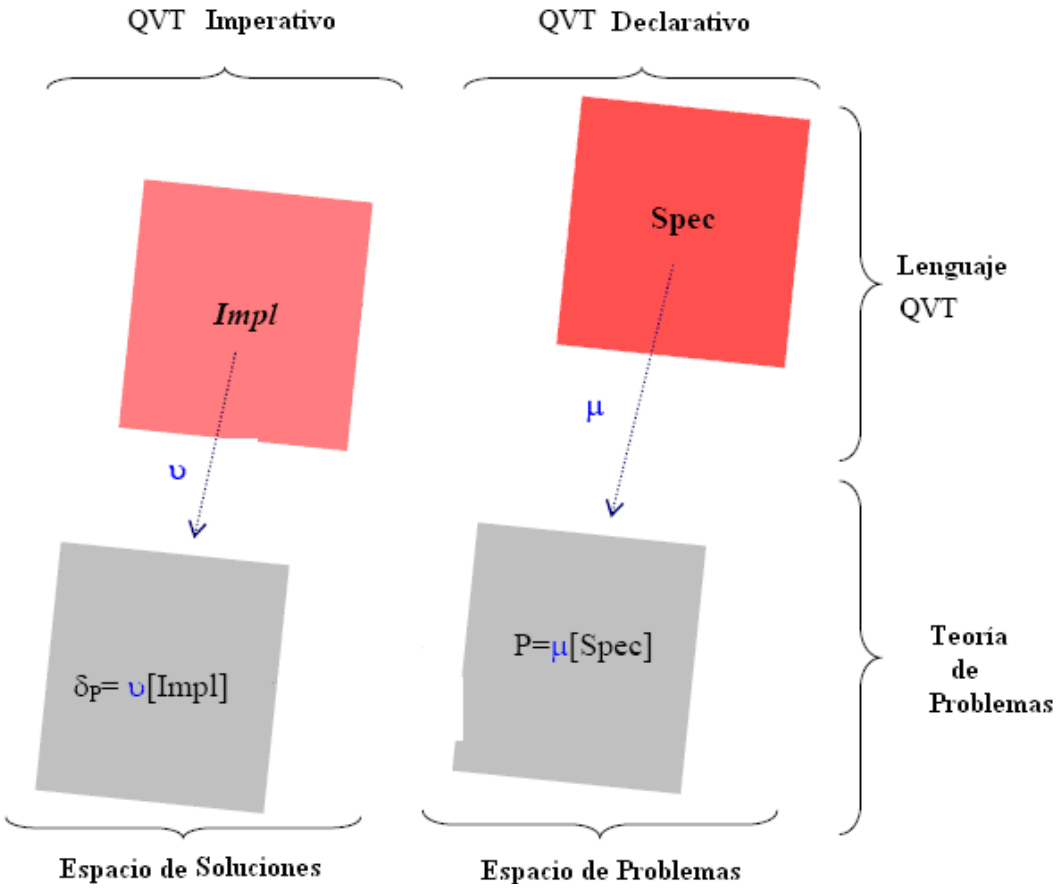

Figura 3-2. Semántica de QVT en términos de la Teoría de Problemas.

3.4 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 una definición semántica precisa de un lenguaje lo hace más sólido y permite probar el cumplimiento de ciertas propiedades.

En este capítulo hemos presentado el formalismo de la teoría de problemas; basándonos en esta teoría introdujimos la semántica (tanto a nivel declarativo como operacional) de los lenguajes para transformación de modelos.

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

Capítulo 4

Conceptos Básicos sobre Composición de Transformaciones

Las transformaciones pueden ser especificadas o implementadas utilizando diferentes herramientas y diferentes lenguajes. Pueden también manipularse como entidades de caja negra. La habilidad de organizar o componer diferentes transformaciones de manera flexible y confiable con el fin de producir el resultado requerido, es uno de los desafíos principales de MDE. En la primera sección de este capítulo se presenta la necesidad de contar con mecanismos de composición entre transformaciones. Seguidamente se introducen mecanismos de composición que incluyen en su definición, la utilización del lenguaje QVT.

4.1 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 cómo 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 4-1 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∪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∪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∪T2;T3 en la Figura 4-1) 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 es necesario implementar el modelo UML 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 4-1. 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: Union(U), Composición Secuencial (;) y Fork (∇)

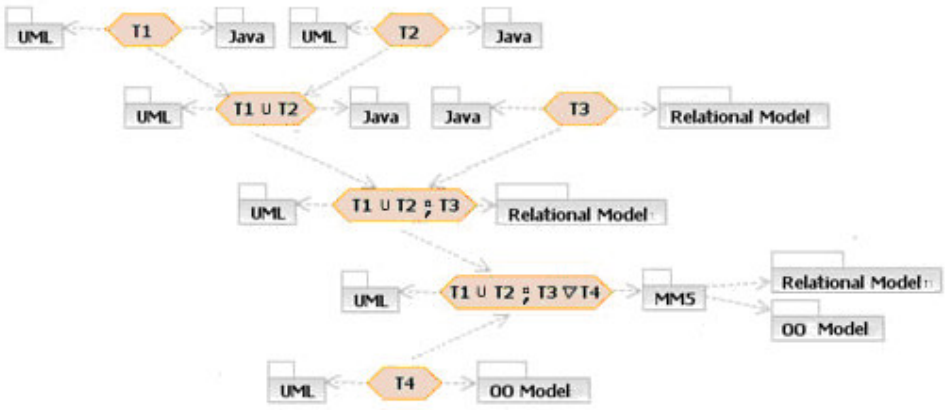

Figura 4-1. Red de transformaciones

4.2 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.2.3.3, permite que las *mappingOperations* puedan ser combinadas mediante llamadas, o utilizando facilidades de reutilización: 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 denominada 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.

4.2.1 Composiciones Internas (*fine-grained*)

El lenguaje proporciona dos facilidades de reutilización al nivel de *mapping operations*: herencia y *merge* de *mappings*. También incluye la disyunción entre *mappings.*

Como ya hemos mencionado, 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 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*.

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;
}
```

4.2.2 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(); // ejecuta el
"cleaning"
   if (not retcode.failed())
      uml.objectsOfType(Package)->map packageToSchema()
   else
      raise "UmlModelTransformationFailed";
}
```

En este ejemplo vemos el uso de los mecanismos de reutilización *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 la definición de transformaciones que utilizan 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 propio conjunto 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
}
```

4.3 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) – la composición de transformaciones de modelos se encuentra desarrollada únicamente por mecanismos de programación en QVT Operacional. En este caso, se pueden definir librerías de transformaciones y librerías de mappings que no son otra cosa que maneras de modularizar las funcionalidades del código fuente de un programa. Es decir, el diseño de la modularización (decidir que cosas se modularizan y que cosas no) no es el resultado de un diseño estratégico de

modularización del lenguaje sino que es el resultado de aplicar el concepto del uso de librerías de cualquier otro lenguaje imperativo para la reorganización del código de un programa.

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.

Es deseable entonces poder definir un mecanismo de composición general para transformaciones que permita tanto aplicarse entre transformaciones declarativas como entre transformaciones operacionales.

Para ello repasamos en el capitulo siguiente 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 5

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

En este capitulo estudiamos la formalización de las operaciones de composición de transformaciones de modelos como fueron abordados en Giandini y Pons [16,17] -a nivel declarativo y a nivel operacional-. Dicha formalización fue definida en base el formalismo de la *teoría algebraica de problemas*, por lo que en la primera sección se introducen los conceptos algebráicos que dicho formalismo expresa en términos de problemas y soluciones (estos conceptos básicos de la *teoría de problemas* fueron introducidos en el Capítulo 3). En las siguientes secciones, se aplica esta teoría algebraica a la composición de transformaciones.

5.1 Operaciones sobre Problemas y Operaciones sobre Soluciones

Una estrategia fundamental para el desarrollo de *programas*, y 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 recombinen 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,$\neg$, ;,↵,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:	$R↵ = \{(x,y) \setminus yRx\}$
- Relación identidad:	$1 = \{(x,x) \setminus x \in U\}$
- *fork*:	$R \nabla S=\{(x,y*z) \setminus xRy \wedge xSz\}$[4]

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

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.

5.1.1 Unión de Problemas y Soluciones para la Unión de Problemas

Definición 2. Sean P y Q problemas; el problema P∪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 5-1 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∪Q=Q∪P), asociativa (P∪(Q∪R) = (P∪Q)∪R) e idempotente (P∪P=P). Existe el elemento neutro llamado 0, que es el problema obtenido desde el conjunto vacío: 0=<∅,∅,∅,∅>.

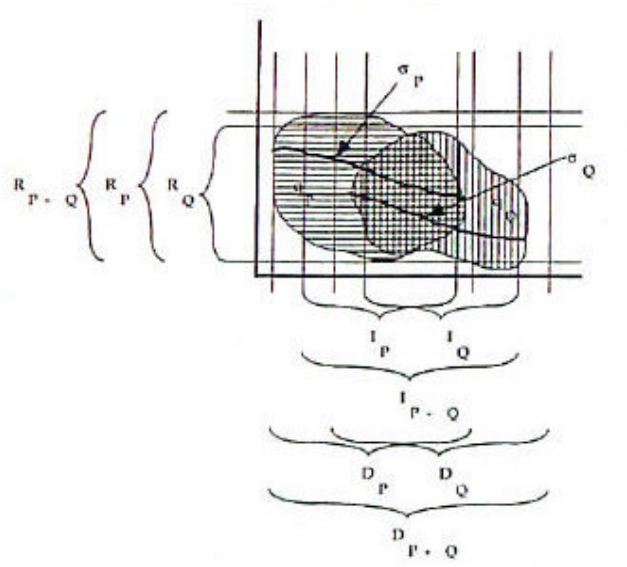

Figura 5-1. Unión de problemas

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 para 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 5-1, 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.

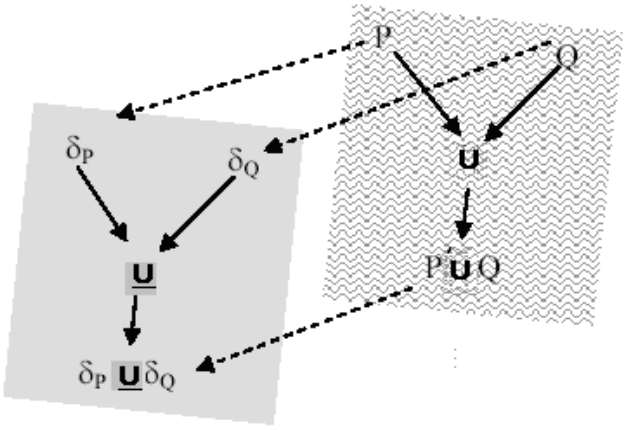

Figura 5-2. Unión de problemas y unión de soluciones

5.1.2 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 5-3 muestra la composición secuencial de los problemas P y Q (para un mejor entendimiento, la representación de Q está rotada 90°).

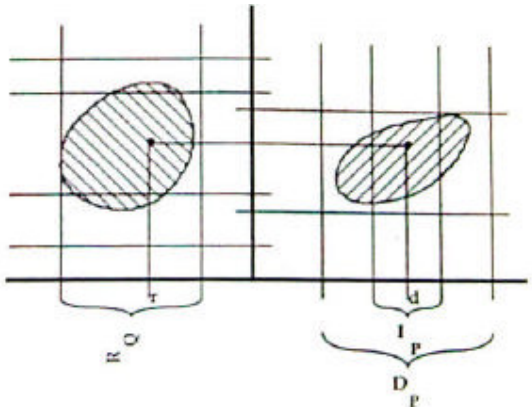

Figura 5-3. Composición secuencial de problemas

a-soluciones para la composición secuencial de problemas

Sean P, Q los problemas en la Figura 5-3 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 δ_P ; δ_Q se define de la siguiente manera:

Definición 5. Sean δ_P y δ_Q funciones; la función δ_P ; δ_Q se define como:
δ_P ; δ_Q (e) = δ_Q (δ_P (e))
Consecuentemente, podemos enunciar el siguiente lema ilustrado en la Figura 5-4:

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.

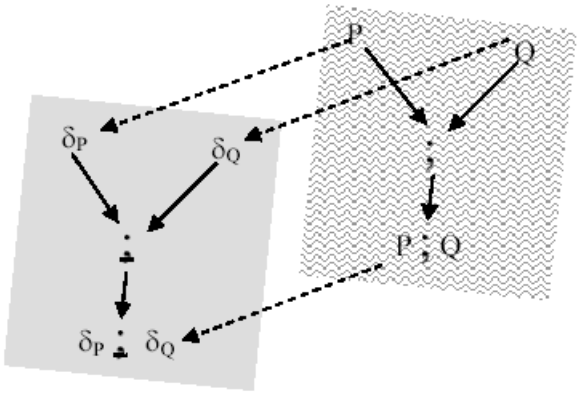

Figura 5-4. Composición secuencial de problemas y soluciones

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

Retomemos la idea presentada en el Capítulo 3, sección 3.2, 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 $\mathcal{L}_D$ 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 $\mathcal{L}_D$ y las operaciones de la teoría algebraica de problemas? A fin de contestar esta pregunta, supongamos que $\mathcal{L}_D$ 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 $\mathcal{L}_D$. 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 3, que las *soluciones* se expresan por medio de *programas*, escritos en un lenguaje algorítmico dado $\mathcal{L}_A$ 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 $\mathcal{L}_A$, a la *a-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 *a-función*.

Entonces, consideremos ahora el léxico del lenguaje $\mathcal{L}_A$. Supongamos que $\mathcal{L}_A$ es un lenguaje de programación habitual, como Java. Es deseable que las operaciones sobre α-soluciones estén expresadas en $\mathcal{L}_A$. 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 $\cup$ puede ser interpretada por medio del comando condicional `if-then-else`.

5.2.1 QVT Declarativo vs. QVT Imperativo

Retomemos ahora los conceptos vertidos en el Capítulo 2, sección 2.3, 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 3-2 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.**

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, $\cup$, ;, etc.), mientras que el lenguaje operacional QVT proveerá los constructores correspondientes en cuanto a las operaciones sobre soluciones (es decir, $\underline{\cup}$, $\underline{;}$, etc.). Por ejemplo, necesitamos una operación $\cup_{QVT}$ para realizar la unión de dos transformaciones declarativas, mientras por otra parte necesitamos una operación $\underline{\cup}_{QVT}$ 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.

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	V_{QVT}	$\underline{V}$	$\underline{V}_{QVT}$

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

5.2.2 Expresando Composición de Problemas en QVT

Como vimos en el Capítulo 4, 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.

5.2.2.1 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) {
top relation PersistClass2ClassWithKey {
checkonly domain Uml c: Class{}
enforce 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) {

top relation Class2Class {
checkonly domain Uml c: Class{}
enforce 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{
checkonly domain Uml a: Property{}
enforce domain Java jm: JavaMethod{}
when { }
where {
   'get' + p.name.firstUpperCase() = jm.name }
}

relation Attr2Setter{
checkonly domain Uml p: Property{}
enforce 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){

top relation PersistClass2ClassWithKey∪QVTClass2Class{
checkonly domain Uml c: Class{}
enforce 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 QVT 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 =
elf.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.

5.2.2.2 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) {

top relation Class2Table {
checkonly domain Java jc: JavaClass{}
enforce 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 {
checkonly domain Java jf: JavaField{}
enforce 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*$\cup_{QVT}$*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 primera 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 primera 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.

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

```
Transformation T1∪QVTT2;QVTT3 (Uml: UML2.0, Rel: RDBMS) {
top relation PersistentClass2ClassWithKey∪QVTClass2Class;QVTClass2Table{
checkonly domain Uml c: Class{}
enforce domain Rel t: Table{}
when {not c1.isPersistent or c1.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")
   AND
   (true IMPLIES 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 QVT 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 la
conjunción del where de la relación receptora y una función de
implicación entre el when y el where de la relación parámetro.
result.where.bodyExpression =
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 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.

5.2.2.3 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, 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) {
top relation Class2OODBMSClass {
checkonly domain Uml c: Class{}
enforce 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 intersección 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)){
top relation PersistentClass2ClassWithKey∪QVTClass2Class;QVT
  Class2Table∇QVTClass2OODBMSClass {
checkonly domain Uml c: Class{}
enforce domain tupleM td: TupleType(t: Table, c2: Class)
when { not c.isPersistent or c.isPersistent AND true }
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")
   AND
```

```
   (true IMPLIES jc.name = t.name and jc.ownedFields -> forAll
   (jf:JavaField | t.column -> exists (co | Attr2Col (jf, co))))
   AND
   (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 QVT 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), es necesario adaptar la clase *Metamodel* de QVT Declarativo cuyo modelado fue presentado en la Figura 2-13 del Capítulo 2. Dicha adaptación puede verse en la Figura 5-5. 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.

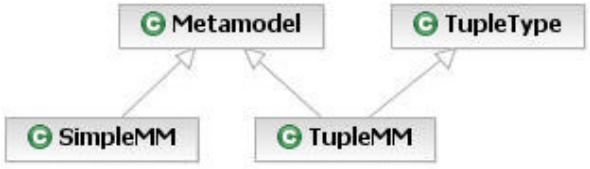

Figura 5-5. Clase *Metamodel* de QVT Declarativo adaptada

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 por la
conjuncion formadas por las cláusulas where de cada relación.
result.where.bodyExpression = (self.where.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 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 especifico en la unión.

5.2.3 Expresando composición de soluciones en QVT

Como vimos en Capítulo 4, 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 *coarsegrained* 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 4.
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.

5.2.3.1 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 $T1 \cup_{QVT} T2$ 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 5.1.1). 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 ImplT1(), ImplT2();
main()
{
   var returncode := (new ImplT1(uml,java)).transform();
   if (returncode.failed())
   then (new ImplT2(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 $\cup_{QVT}$ a tal operador, es decir:

```
Impl = ImplT1 ∪QVT ImplT2
```

Formalización de la unión de transformaciones operacionales

La definición formal de la operación $\cup_{QVT}$ está dada por *pre* y *post* condiciones expresadas en OCL, en el contexto de la metaclase *OperationalTransformation* de QVT 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
```

5.2.3.2 La composición secuencial de transformaciones operacionales

Supongamos ahora que contamos con la transformación operacional **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 4, la especificación de QVT sugiere extender la definición de la transformación **Impl** invocando *in-place* a la transformación **Impl**$_{T3}$,

que se aplica sobre el modelo Java de salida, obtenido al ejecutar la transformación **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 ImplT3
```

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 QVT 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)
```

5.2.3.3 La operación fork de transformaciones operacionales

Supongamos ahora que contamos con la transformación operacional **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 **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 4, la especificación de QVT sugiere invocar *in-place* a las transformaciones **CompositeImpl** y **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 ImplT4 (in uml: UML2.0, out oodbms: OODBMS);
main(){
   var st1 :=(new CompositeImpl(uml, rel)).parallelTransform();
   var st2 :=(new ImplT4 (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 $\underline{\nabla}_{QVT}$ a tal operador, es decir:

```
ParallelImpl = CompositeImpl ∇QVT ImplT4
```

Formalización de la operación *fork* de transformaciones operacionales

La definición formal de la operación $\underline{\nabla}_{QVT}$ está formalmente expresada en OCL, en el contexto de la metaclase *OperationalTransformation* de QVT 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
```

5.2.4 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 5-6:

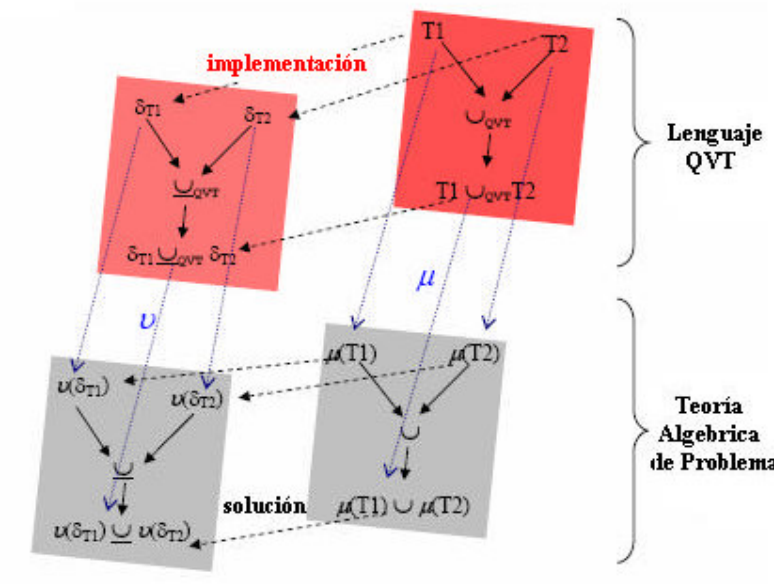

Figura 5-6. Unión de problemas y unión de soluciones en QVT.

5.3 Conclusión del Capítulo

En este capítulo hemos introducido las definiciones de operaciones sobre problemas y operaciones sobre soluciones en la teoría algebraica de problemas tal como fueron definidas en Giandini y Pons [17, 19].

Las operaciones descritas y analizadas en este capítulo, fueron directamente implementadas en la Herramienta Calculadora de Composiciones de Transformaciones

Hasta acá hemos abordado los temas del contexto y los aspectos teóricos de la composición de transformaciones de modelos. En el capítulo siguiente nos avocaremos a la parte implementativa de la herramienta, analizando las tecnologías que intevinieron en el proceso de desarrollo de la misma.

Capítulo 6

Estado actual de las soluciones

La versión 1.0 de la especificación de la OMG fue publicada en Abril del año 2008, por lo que las soluciones en base a la especificación se encuentran en un estado naciente.

Existen diversos enfoques que contribuyen brindando soluciones para implementar transformaciones de modelos: lenguajes, frameworks, engines, compiladores, etc. pero que en su mayoría no son soluciones compatibles con la especificación de QVT. Uno de los proyectos mas representativos en éste campo es ATL [21], que es al mismo tiempo un lenguaje de transformaciones de modelos que trabaja junto con un compilador que traduce el código fuente a sentencias que son interpretadas por una maquina virtual construida para tal fin. ATL fue construido en respuesta a un Request-For-Proposal lanzado por la OMG en etapa previa a la adopción de la especificación de QVT. Actualmente, el lenguaje quedó fuera de la especificación.

De las soluciones referidas, existe solo un pequeño subconjunto de implementaciones que funcionan para dar solución concreta al modelado y ejecución de transformaciones de modelos que soportan la especificación de OMG.

6.1 Implementaciones existentes para transformaciones de Modelos

6.1.1 ATL by INRIA + LINA

ATL (Atlas Transformation Language) [21] consta de un lenguaje de transformación de modelos y un toolkit creados por ATLAS Group (INRIA y LINA) que forma parte del proyecto GMT de Eclipse [13]. En él se plantea un lenguaje híbrido de transformaciones declarativas y operacionales de modelos que si bien es del estilo de QVT [15] no se ajusta a las especificaciones ya que fue contruido en respuesta a un Request-For-Proposal lanzado por la OMG para transformaciones de modelos. Tiene herramientas que realiza una compilación del código fuente a bytecodes para una máquina virtual que implementa comandos específicos para la ejecución de transformaciones [8]

ATL forma parte del framework de gestión de modelos AMMA que se encuentra integrado en Eclipse [36] y EMF [37].

ATL posee un algoritmo de ejecución preciso y determinista. Sin embargo no se apoya sobre ningún método formal, ni proporciona mecanismos para la validación de las transformaciones.

El toolkit es Open Source.

6.1.2 ModelMorf by Tata Consultancy Services

ModelMorf [38] es un engine desarrollado por Tata Consultancy Servicies. Funciona bajo línea de comandos para ejecutar transformaciones de modelos basados en la especificación QVT declarativo. Cumple parcialmente con la es especificación de QVT de la OMG pero la implementación es propietaria.

6.1.3 Medini by ikv++ Technologies

Medini [39] es una herramienta Open Source construida por ikv++ technologies sobre las bases del proyecto OSLO [40]. OSLO (*Open Source Libraries for OCL*) es un proyecto basado en la implementación OCL de la Universidad de Kent que provee las clases del metamodelo de OCL.

Medini es un conjunto de herramientas construidas para el diseño y ejecución de transformaciones declarativas de modelos. Define una estructura de clases que extienden a las librerías de OSLO para modelar transformaciones QVT. Pero además, Medini implementa un motor capaz de ejecutar código QVT declarativo, siendo una especial excepción entre las herramientas de transformación de modelos.

6.1.4 SmartQVT by Telecom France

SmartQVT es un proyecto que nació en el departamento de I+D de France Telecom.

Es un compilador Open Source de transformaciones de modelos escritas en QVT operacional. En esencia, toma como *input* una transformación escrita en lenguaje QVT operacional y obtiene a partir de ella, una clase Java que implementa comportamiento descrito en lenguaje QVT. Esta última, como toda clase Java, puede ser integrada en cualquier aplicación y ejecutada desde cualquier máquina virtual.

Para realizar su trabajo, la herramienta implementa un parser y un compiler. El parser lee código QVT y obtiene a partir de él, su metamodelo. El compiler toma el metamodelo y escribe el texto del código fuente de la clase Java.

Pero este parser tiene una característica que lo hace muy particular:

La instanciación del metamodelo es un proceso que se realiza en dos partes: primero se leen los tokens recibidos por el scanner y se generan las clases necesarias para construir el árbol sintáctico que representa al código leido. Segundo, es necesario usar los elementos y la estructura del mencionado árbol para instanciar los objetos del metamodelo de QVT. Esta última tarea es vista como una transformación de modelos, donde el modelo de entrada es el árbol sintáctico producido por el parser y el modelo de salida es el metamodelo QVT. El equipo de France Telecom escribió entonces una transformación utilizando el lenguaje QVT declarativo que se describe en la especificación. Dicha transformación toma el modelo de un árbol sintáctico generado a partir de un código QVT (que respete la gramática de QVT operacional) y genera un modelo de salida que no es otra cosa que el metamodelo del código representado por dicho árbol sintáctico.

Dicha transformación fue compilada a Java (con versiones anteriores de SmartQVT que utilizaban parsers contruidos en otros lenguajes, como Python) pasando así, a ser parte de la misma herramienta, demostrando de esta manera el poder de expresividad del lenguaje QVT (y de la propia herramienta).

A continuación se muestra un fragmento del código QVT que implementa esta herramienta:

```
########################################################################
--Transformation
########################################################################
mapping TransformationAst::toOperationalTransformation() :
OperationalTransformation {
       name := self.p_transformation_header.p_identifier.value;
       isBlackbox :=
```

```
            self.p_transformation_header.m_qualifier[BlackboxQualifierAst]-
            >size()>0;
        ownedType +=
            self.p_transformation_header.p_transformation_signature.p_simple_sig
            nature.m_param[ParamAst]->map toModelType()->asOrderedSet();
        modelParameter +=
               self.p_transformation_header.p_transformation_signature.p_simple_
               signature.m_param[ParamAst]->map toModelParameter()-
               >asOrderedSet();
        ownedVariable += result.map createThisVariable();
}
```

Figura 6-1. Codigo de SmartQVT hecho en QVT

6.1.5 Together Architecture/QVT by Borland (QVT/Operational)

Together [42] es un producto de Borland que integra la IDE de Java con la herramienta de modelado UML del mismo nombre (tenía sus orígenes en JBuilder). Técnicamente, Together es un conjunto de plug-ins para Eclipse. Permite a los arquitectos de software trabajar focalizados en el diseño UML de manera gráfica; luego la herramienta se encarga de los procesos de análisis del diseño, y de generar un modelo de plataforma específica (PSM) a partir de modelos plataformas independientes (PIM) usando el QVT estándar [43].

Esta herramienta también es propietaria y requiere licencia para su uso.

6.2 Aportes de herramientas existentes a nuestro desarrollo

Las herramientas Medini QVT (para QVT declarativo) y SmartQVT (para QVT operacional) son unas de las más apliamente utilizadas para ejecutar transformaciones entre modelos basados en la especificación de QVT.

Dado que son herramientas de código abierto, fueron la principal fuente de inspiración para el desarrollo del código de la herramienta calculadora de composiciones.

Aunque todavía se encuentran en un estado inmaduro (no tienen implementadas muchas de las especificaciones del lenguaje y algunas otras funcionalidades sufren de errores de programación aún no corregidos), el enfoque en la resolución de los problemas comunes por parte de estas herramientas en las transformaciones de modelos, se encuentran abordado de una manera que facilita la resolucion de nuestro análisis. Es por eso que tomamos cada uno de estos enfoques como referencia para el diseño de la solución.

6.3 Herramientas de Composición

Existen otras herramientas de software que están enfocadas a la composición de modelos, aunque la mayoría está enfocada a reutilización de componentes de los lenguajes. A continuación haremos un repaso de ellas:

6.3.1 Atlas Model Weaver

El AMW (ATLAS Model Weaver) [46] es una herramienta para establecer relaciones (ej., links) entre modelos. Estos links son guardados en un modelo llamado **weaving model**, que es creado conforme al metamodelo de weaving.

Estas relaciones entre modelos permiten definir operaciones de *merge* entre los modelos y metamodelos.

The ATLAS Model Weaver (AMW) está siendo desarrollada por ATLAS group, INRIA.

El prototipo es implementado como un plugin EMF. AMW usa el *EMF reflective API* para, automáticamente, generar un editor standard basado en el metamodelo de weaving.

6.3.2 Glue Generator Tool (GGT)

GGT [47,48,49] es un framework generador de código. Consiste en un conjunto de librerías que funcionan como *drivers* de los componentes de aplicaciones que se quieren componer. Esto sirve para tener acceso a las vistas del código fuente.

El código tomado es procesado y convertido a un código de representación interna del Glue Generator Tool y luego es exportado al lenguaje específico para la maquina en donde se desea ejecutar.

6.3.3 Epsilon Merging Language (EML)

EML [50,51,52] es un lenguaje de Epsilon. Epsilon es una plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc.) sobre Eclipse. Presenta el lenguaje metamodelo independiente Epsilon Object Language que se basa en OCL. Puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. Tres son los lenguajes definidos en la actualidad: Epsilon Comparison Language (ECL), Epsilon Merging Language (EML), Epsilon Transformation Language (ETL), para comparación, composición y transformación de modelos respectivamente.

EML puede ser utilizado para combinar (*merge*) un numero arbitrario de modelos de entrada de potencialmente diversos metamodelos para generar un modelo de salida. Esta operación puede ser vista como una transformación en si misma.

De todas maneras, esta capacidad de combinación de modelos requiere ciertos recaudos: soporte para comparaciones, chequeos de compatibilidad, etc.

Por ejemplo, puede ser utilizado para unificar dos modelos complementarios (aunque potencialmente solapados) que describan diferentes vistas de un mismo sistema.

6.3.4 ModelBus

ModelBus [53,54,55] es un framework de integración orientado a modelos que permite construir herramientas integradas de desarrollo para el proceso de ingeniería de aplicaciones. Uno de los mayores poblemas esta relacionado con el seguimiento y control de todas las referencias que tienen los modelos desde y hacia el exterior. A medida que los modelos crecen se vuelven más difíciles de coordinar su flujo de trabajo. ModelBus aborda esta problemática manejando un un repositorio de modelos internos al framework, que maneja versionamiento, coordina las combinaciones con otros modelos y con otras versiones del mismo modelo y hasta coordina el control de los fragmentos de cada modelo. La limitación principal es que solo mantiene esta coordinación entre los modelos contenidos en el repositorio.

6.3.5 ToolBus

The TOOLBUS [56,57,58] es un coordinador de arquitectura de software para integraciones de componentes escritos en diferentes lenguajes corriendo en diferentes computadoras. Fue creado para mejorar la estructura interna y mantenibilidad de los Meta-Ambientes.

ToolBus permite comunicación entre procesos, más allá que mero intercambio de datos, usando mensajes o notas.

6.3.6 MCC

MCC [59] 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*.

6.3.7 UMLAUT

UMLAUT [60] es un framework para la construccion de herramientas dedicadas a la manipulación de modelos escritos utilizando UML. Fue construido para proveer al diseñador de modelos de una libertad de elección en cuanto a combinaciones de transformaciones a ser ejecutadas.

Esta orientado hacia la produccion de software escalable cuyos componentes y ensamble están descriptos usando UML

EL componente central de UMLAUT es la implementación en Eiffel del metamodelo UML. Esto permite a modelos UML estar representados como una estructura de objetos AST (Abstract Sintax Tree).

UMLAUT tiene la misma intención que MCC, pero UMLAUT está limitado a transformar sólo modelos UML mientras que MCC 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.

6.4 Conclusión del Capítulo

Por un lado hemos analizado el estado actual de las herramientas que ejecutan transformaciones de modelos. Estas herramientas nos permiten analizar el grado de madurez del proceso de análisis de los lenguajes de tranformación de modelos. En

particular, hemos reutilizado parte de la ingeniería de esas herramientas para armar nuestra Herramienta Calculadora de Composiciones.

Seguidamente hemos analizado el estado actual de herramientas y soluciones para composiciones de modelos. La mayoría de estas están enfocadas en capacidades de los lenguajes *corarse-grained* para componer soluciones. Otras están enfocadas al diseño de frameworks para combinar componentes de lenguajes, librerías y/o procesos. Pero la mayoría no están orientadas a lenguajes como QVT. El lenguaje con el que mayor interoperatividad tienen estas herramientas es ATL.

El campo de las soluciones relacionadas con QVT está poco desarrollada. En este caso, se atacamos la solución con un fundamento matemático y se lo hace tanto a nivel declarativo como a nivel operacional con una herramienta que ejecuta operadores de composición algebraica de manera automática.

Capítulo 7

La Herramienta Calculadora de Composiciones

Cada una de las herramientas anteriormente analizadas está enfocada en la tarea de compilar y/o ejecutar transformaciones, pero ninguna de ellas permite componer transformaciones.

La implementación de nuestra herramienta de software permite la composición algebraica de transformaciones de modelos construidos en el contexto de un proceso de desarrollo de software dirigido por modelos [6,7] utilizando el estándar QVT [3].

El fundamento teórico de nuestro trabajo fue definido por Giandini y Pons en [16,17].

Dos herramientas en una

Si bien la misma herramienta implementa las composiciones declarativas y operacionales desde una interface única, en esencia funciona como dos herramientas muy distintas que solo comparten la interface.

QVT Declarativo tiene una sintaxis y una gramática distinta a QVT Operacional. El resultado de las operaciones de composición es radicalmente distinto en cada uno de los casos. Por lo tanto, la Herramienta Calculadora de Composiciones es el resultado del desarrollo de dos herramientas distintas, que trabajan sobre lenguajes distintos, que generan estructuras de datos distintas, y cuyas operaciones implementadas siguen dos lógicas de procesamiento totalmente distintas.

7.1 Aporte de la herramienta a la composición transformaciones

Hasta ahora, la composición de transformaciones fue descripta a partir de derivaciones de modelos teóricos como la teoría de problemas. No existía un software capaz de obtener el resultado de una composición a partir de dos transformaciones definidas independientemente una de la otra.

Con esta herramienta, podremos probar no solo la efectividad práctica de la composición de transformaciones de modelos, sino tambien validar cada una de las operaciones de composición definidas teóricamente.

Las transformaciones compuestas (aquellas que se generan a partir de la composición de las transformaciones originales) son también transformaciones cuya sintaxis y gramática se corresponden con la de las transformaciones originales. Por lo tanto, pueden ser normalmente utilizadas por las herramientas de ejecución de transformaciones.

Además, las transformaciones compuestas pueden ser utilizadas por la Herramienta Calculadora de Composiciones para componerse con nuevas transformaciones y así generar transformaciones de mayor grado de composición.

7.2 Descripcion de la herramienta

La herramienta fue planteada como extensión para ePlatero [34, 35], un plug-in de código abierto para la plataforma de Eclipse[5][36] que corre bajo el *framework* de metamodelado EMF [37].

La definición de las transformaciones escritas en lenguaje QVT es almacenada en archivos de texto con extensión .qvt. En particular, se puede además definir mas de una transformación en lenguaje QVT operacional en un mismo archivo de texto o bien ninguna transformación (solo se definen operaciones auxiliares) resultando así en archivos de tipo Librería QVT, pero que quedan exceptuados del universo de archivos utilizados por la herramienta Calculadora de Composiciones.

Al seleccionar dos archivos de texto que contengan cada uno la definición de una transformación QVT, la herramienta permite procesar las transformaciones contenidas para generar así un nuevo archivo que contiene el código de la definición de una nueva transformación cuyo resultado es equivalente al de la composición de las transformaciones seleccionadas de acuerdo a la definiciones algebraicas desarrolladas en Giandini y Pons [16, 17, 18, 19], presentadas en las secciones anteriores.

Arquitectura Eclipse para el desarrollo de plug-ins

La arquitectura de plug- ins de Eclipse puede verse en la Figura 7-1. 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.

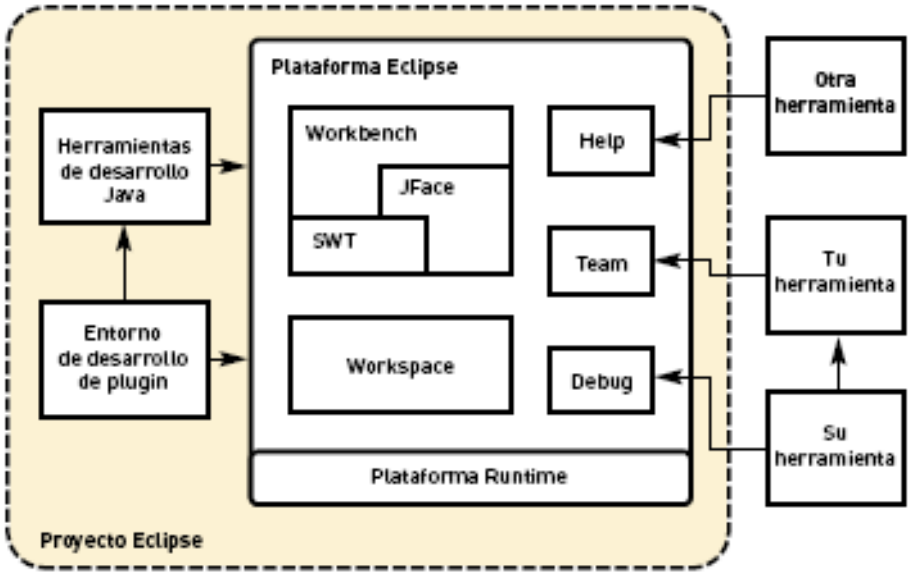

Figura 7-1. Arquitectura de Eclipse, para el desarrollo de plug- ins.

[5] La breve descripción de Eclipse está descripta en el **Glosario de siglas y términos**.

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

7.3 Interface

La inteface ha sido diseñada en base al prototipo de implementación presentado en Giandini y Pons [17].
Dentro de un proyecto de diseño de transformaciones QVT, es posible seleccionar dos archivos de proyecto activo que tengan extensiones .qvt como se muestra en la figura 7-2:

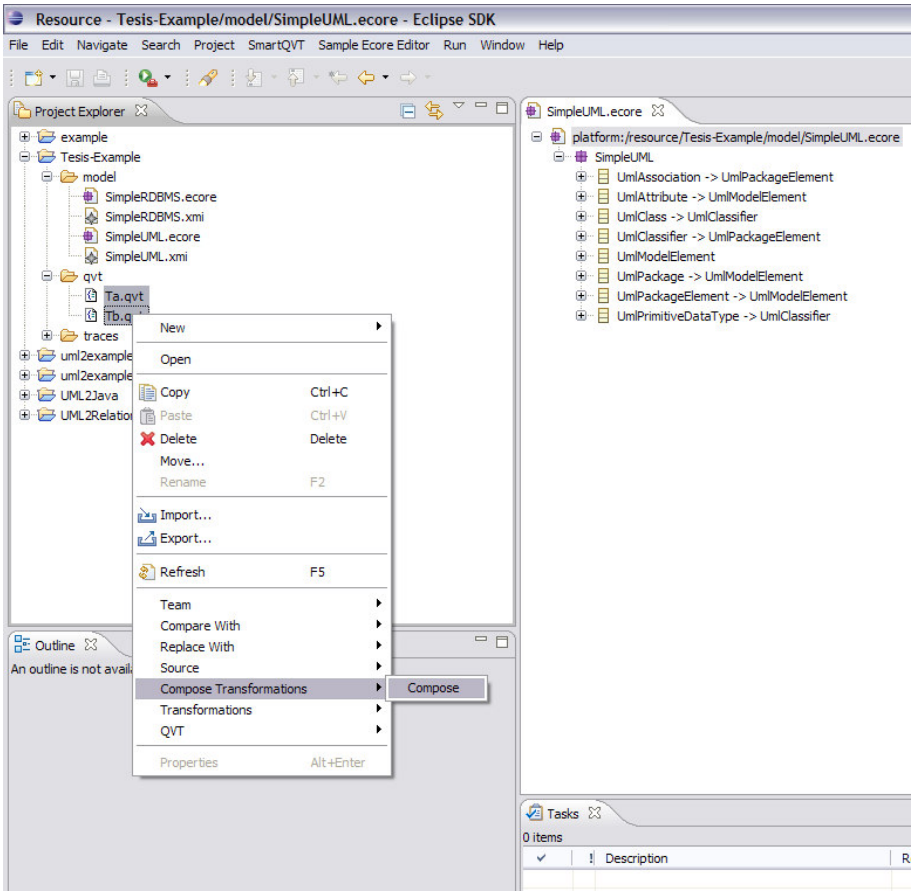

Figura 7-2. Selección de Transformaciones Declarativas

Luego de la selección de archivos, se puede acceder a un menú contextual que permite seleccionar la opcion “Compose Transformation” que será encargada de abrir el diálogo de opciones de la composición.

Este diálogo cuenta con información de los archivos que contienen a las definiciones de las transformaciones a componer y un listado de las operaciones algebraicas disponibles en la herramienta para realizar.

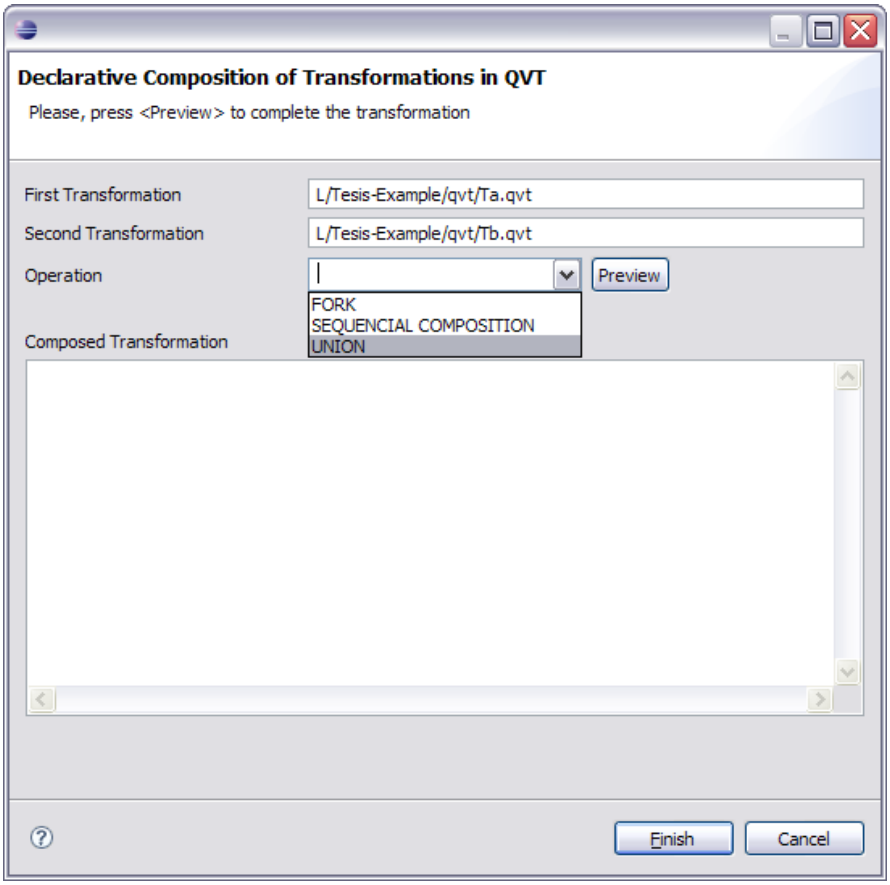

Figura 7-3. Aplicación de la unión para Transformaciones Declarativas

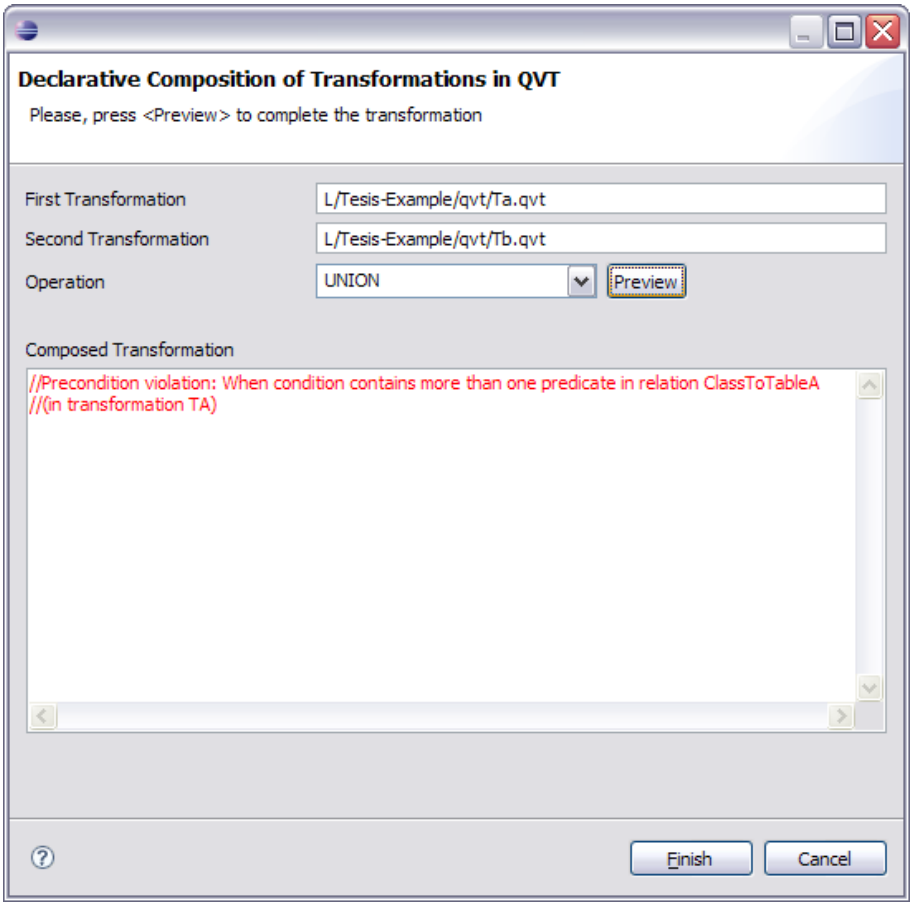

Figura 7-4. Mensaje de error en las transformaciones declarativas

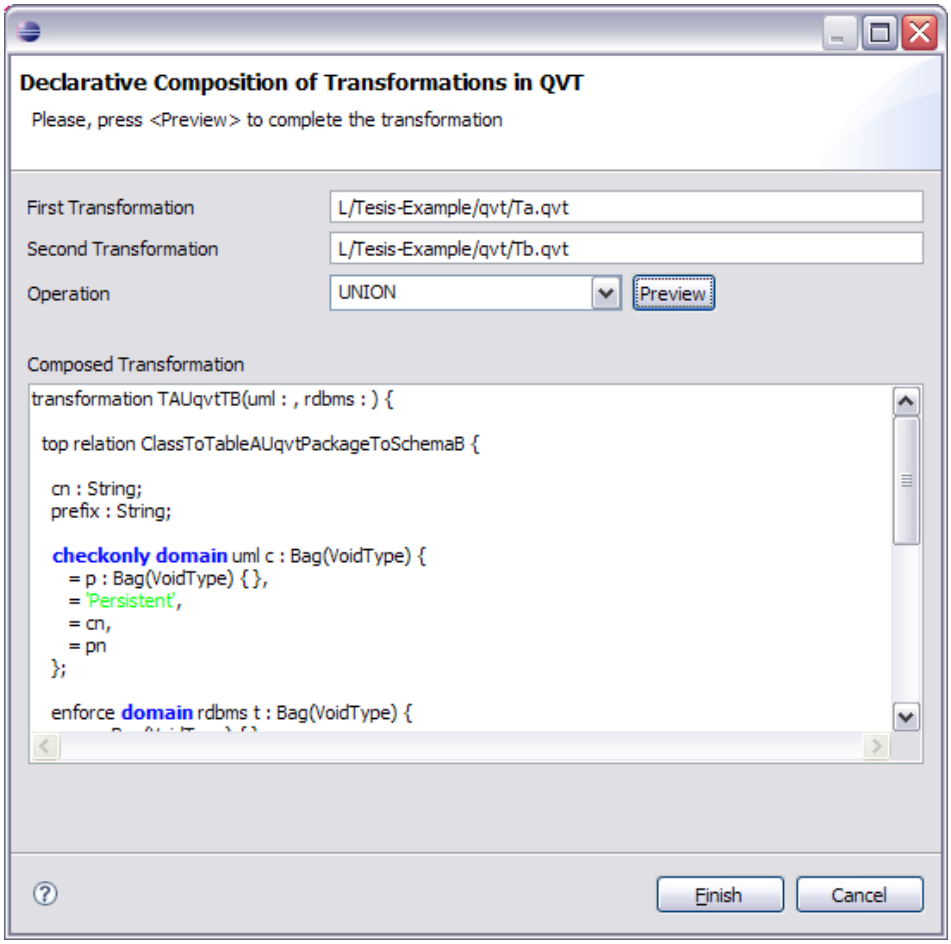

Figura 7-5. Aplicación de la unión para Transformaciones Declarativas

Como se muestra en las figuras, el resultado de una operación de composición de transformaciones es la definición de una nueva transformación a la que llamamos "Transformación compuesta". Pero en esencia, es una secuencia de caracteres que representan a dicha definición. Esa secuencia de caracteres puede ser presentada en la misma ventana de diálogo de la herramienta mostrando el resultado al diseñador de transformaciones o bien puede ser directamente guardado en un archivo de texto

7.4 Proceso de ejecución de las operaciones de composición

Desde el momento en que el usuario selecciona dos archivos a componer y dispara el mecanismo de composición de transformaciones, intervienen en el proceso distintas soluciones para atender a cada una de las etapas

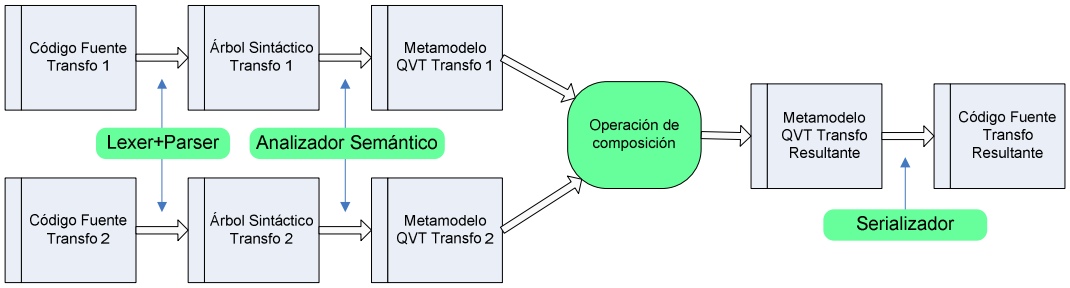

Figura 7-6. Proceso de ejecución de la composición de transformaciones

7.4.1 El analizador léxico: Lexer

El analizador léxico es la herramienta encargada de hacer un scanning del texto que contiene el código fuente de las transformaciones y generar a partir de éste la secuencia de tokens que lo representan.

Para este caso, las herramientas utilizaron el generador de scanner JFlex [44] que está especialmente diseñado para trabajar con el generador de parser CUP que se describe en el siguiente apartado.

7.4.2 El Parser

Los parser utilizados (tanto para las composiciones declarativas como para las composiciones operacionales) están generados por la herramienta CUP (Constructor for Useful Parser) que es una herramienta que permite la generación de parsers LALR (LookAhead Left to Right parser) a partir de una especificación simple [45]. Este tiene el mismo rol que los programas YACC (Yet Another Compiler of Compiler).

Para utilizar esta herramienta, es necesario construir un documento textual que especifique la gramática sobre la cual se quiere que funcione el parser.

Dicho documento se divide en cuatro partes:

- La primera contiene declaraciones para especificar cómo será generado el parser, codigo de inicialización y código para invocar al scanner que retornará cada uno de los próximos tokens a analizar.
- En la segunda parte del documento se declaran los símbolos terminales y no terminales, y la clase que se le asociará a cada uno de ellos.
- En la tercera parte se identifica la precedencia y la asociatividad de los símbolos terminales.
- Y en la última parte se especifica la gramática del lenguaje en EBNF.

Ejemplo de un documento textual para generar un parser de operaciones aritméticas:

```
/* Preliminaries to set up and use the scanner.  */
init with {: scanner.init();              :};
scan with {: return scanner.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;
```

```
/* Non terminals */
non terminal            expr_list, expr_part;
non terminal Integer    expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

/* The grammar */
expr_list ::= expr_list expr_part |
              expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | expr MOD expr
              | MINUS expr %prec UMINUS
            | LPAREN expr RPAREN
              | NUMBER
              ;
```

Figura 7-7. Texto de la gramática tomada por el generador de parser

Estas herramientas son útiles para automatizar el trabajo de construcción de un parser.

Es decir, dado la conjunción de JFlex + CUP junto con el archivo textual con la descripción de la gramática del lenguaje a parsear, nos da como resultado un parser para dicha gramática.

Este proceso de generación de parser se realiza solo una vez. Luego haber generado el parser, este se utiliza en cada ejecución de la herramienta.

7.4.2.1 El Arbol Sintáctico generado

Como resultado de la ejecución del parser descrito en la sección anterior, se obtiene un árbol sintáctico del código fuente analizado.

Cada uno de los nodos del arbol sintáctico de cada lenguaje (declarativo y operacional) requiere una definición de clases especial según sea su estructura interna y la red de objetos con los cuales se vincula dicho nodo. Los proyectos Medini (Declarativo) y SmartQVT (Operacional) proveen los paquetes necesarios con las definiciones de las clases necesarias para la construcción del árbol sintáctico QVT. Pero además, es necesario un conjunto de definiciones de clases para la definición de la estructura del árbol sintáctico de las expresiones OCL (recordemos que dentro de los *when* y *where* de las *relations* es posible incluir expresiones OCL): es decir, por cada declaración de variable, por cada estructura de control, por cada propiedad invocada, etc, existe una rama del árbol sintáctico (objetos relacionados) que representa la estructura definida en el código fuente.

Por ejemplo: para el caso de QVT declarativo, existe una jerarquía de clases definidas por las librerías OSLO (Open Source Library for OCL) que modela cada una de las clases necesarias para armar el árbol sintáctico: TansformationAS, RelationAS, QueryAS, VariableDeclarationAS, etc. Una TransformationAS puede contener varias RelationAS, y varias QueryAS; una RelationAS puede contener varias VariableDeclarationAS.

Para el caso de QVT operacional, SmartQVT propone una jerarquía propia para las clases utilizadas en el árbol sintáctico. Por ejemplo: TransformationDefAst, MappingDefAst, HelperAst, MappingBodyAst, AssignExpAst. Una TransformationDefAst puede contener varias MappingDefAst y varias HelperAst. Un MappingDefAst tiene un MappingBodyAst que tiene un PopulationSectionAst que puede contener varios AssignExpAst.

7.4.3 Instanciación del metamodelo

Una vez creado el árbol sintáctico, el siguiente paso es la instanciación del metamodelo.

Esto se hace recorriendo los nodos del arbol sintáctico para generar los objetos del metamodelo de QVT.

Por ejemplo, el nodo raiz del arbol sintáctico de una transformación declarativa es un objeto de la clase TransformationAS. Este objeto es la raiz de la estructura sintáctica del código fuente de la definición de la transformación analizada. La ocurrencia de un objeto de la clase TransformationAS, indica al generador del metamodelo que es necesario crear un objeto de la clase Transformation del paquete QVT. El objeto de clase Transformation es la raiz de la instancia del metamodelo que representa al código fuente definido.

Como comentamos anteriormente, las estructuras de objetos utilizadas en los casos declarativos y operacionales son muy distintas entre sí. Por lo tanto, la estructura de los nodos del árbol formado por el lenguaje declarativo es distinta a la estructura de los nodos del árbol formado por el lenguaje operacional, por lo que además, el recorrido del árbol se hace de una manera distinta en cada caso, como comentamos a continuación.

7.4.3.1 Instanciación del metamodelo declarativo

Para poder instanciar el metamodelo declarativo, es necesario recorrer toda la estructura del árbol sintáctico e ir instancianciando los objetos del paquete QVT que se describen en el código fuente de la transformación.

No solo hay que instanciar objetos de la clase Relation, sino que entre sus componentes tienen atributos when y where (como vimos anteriormente) que contienen expresiones OCL. OCL tiene su propio metamodelo, por lo que cada expresión OCL debe ser instanciada a partir de su propio metamodelo.

Una definición de transformación simple, con aproximadamente tres *relations*, y tres expresiones OCL, tiene un metamodelo formado por aproximadamente medio centenar de objetos de distintas clases que representan dicho código.

7.4.3.2 Instanciación del metamodelo operacional

Las operaciones algebraicas de composición definidas para el lenguaje operacional, como vimos, tienen la característica de utilizar un mecanismo de composición *coarse-grained*. Esto significa que la composición se logra mediante invocaciones directas a las transformaciones originales sin depender de cual sea el código del cuerpo de las estas transformaciones.
Es por ello que para poder construir las composiciones de transformaciones operacionales no es necesario trabajar con los metamodelos completos. Solo basta

trabajar con los elementos que componen la *signature* de las transformaciones: nombre y parámetros. Al fin y al cabo, estos son los únicos datos que necesita la transformación compuesta para invocar a la transformación original.

7.4.3 Adaptación de los modelos parámetros

Una transformación solo se puede ejecutar si cuenta con las instancias de dos modelos elementales: la instancia del metamodelo de QVT (la transformación) y la instancia del modelo a transformar (ej. una instancia de un modelo UML, JAVA, etc).

Al escribir definiciones de transformaciones, se deben referenciar variables del modelo a transformar: por ejemplo, si una clase de un modelo UML debe ser transformada a tabla solo en el caso de ser una clase marcada como persistente, se debe hacer referencia al nombre y al tipo del atributo en cuestión. El analizador semántico deberá chequear si el atributo referido en la transformación tiene el correspondiente nombre y tipo en el modelo parámetro como muestra la Figura 7-8.

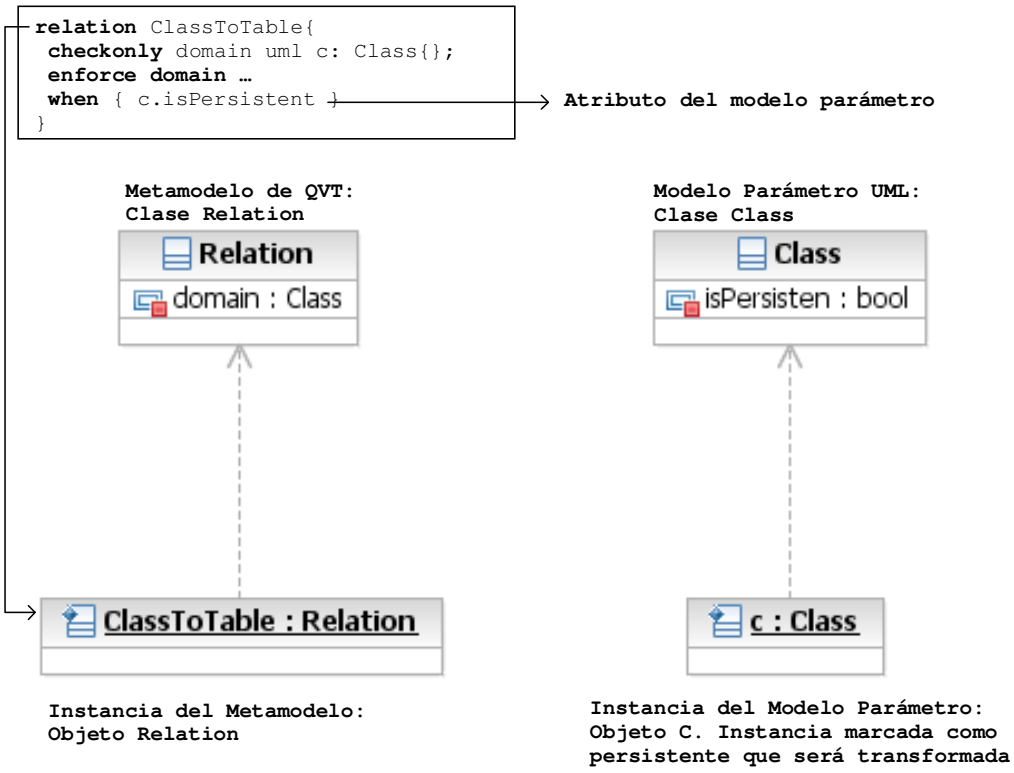

Figura 7-8. Relación entre el Código, metamodelos y modelos parámetros

Como nuestra herramienta funciona operando solo a nivel de objetos del metamodelo del código QVT (sin tener en cuenta a los modelos parámetros que serán transformados), fue necesario realizar adaptaciones en el metamodelo de QVT para poder instanciar metamodelos que pasen los análisis semánticos sin tener en cuenta al análisis de los atributos de los modelos parámetros. Esto se logró redefiniendo la clase ModelParameter.

En la especificación de QVT, la clase ModelParameter tiene atributos simples como “name”, pero además tiene un atributo “type” que tiene una referencia directa al “metamodel” (de clase Package) utilizado como parámetro.

Esas referencias fueron cambiadas en el proceso de instanciación del metamodelo de QVT.

7.4.4 Diseño de las Operaciones

En el caso de QVT Operacional, las transformaciones generadas como resultado de una composición tienen la misma estructura imperativa que no dependen del cuerpo de las transformaciones que se compondrán. Esto se logra debido a que utilizan un mecanismo de reutilización *coarse-grained* capaz de hacer invocaciones a las transformaciones originales.

Recordemos el código de una trasmformación compuesta operacional.

```
transformation Impl(in uml:UML2.0,out java:JAVA)
   access ImplT1(), ImplT2(); //Invocacion a otras transformaciones
main()
{
   var returncode := (new ImplT1(uml,java)).transform();
   if (returncode.failed())
   then (new ImplT2(uml,java)).transform() endif
}
```

Como vemos, el código del cuerpo de la transformación siempre se compone de una definición de variable, seguida de una estructura de control if-then. Lo único que varía es la declaración del *access* a las transformaciones “operando” y la invocación de las transformaciones en el cuerpo.

El resultado de una operación de composición en QVT operacional es entonces un template de código como el mostrado anteriormente en donde se reemplazan los modelos parámetros y las transformaciones invocadas por las que hayan sido seleccionadas por el diseñador de la composición.

7.4.5 Serializacion

La serialización es el proceso por el cual se logra obtener una cadena de caracteres que representa o describe a un objeto y a sus atributos. Un serializador tiene como input una instancia de un objeto y retorna como output una representación textual del mismo.

Como ya se describió con anterioridad, el resultado de una operación de composición de transformaciones es otra transformación. Tal resultado de la composición, es un nuevo objeto de la clase *Transformation* definida en el paquete de QVT. Pero el objetivo de la serialización es que dicha transformación pueda ser expresada por una cadena de caracteres que la represente. En particular, esa cadena es nada menos que el código fuente de la transformación compuesta.

Para implementar la serialización, fue necesario diseñar un algoritmo que permita recorrer los objetos de la estructura de grafo formada por la relación de conocimiento entre instancias.

Esta representación textual (codigo fuente de la transformación compuesta) puede ser utilizada como input de cualquier herramienta de manejo de transformaciones para ser ejecutada junto con los modelos parámetros adecuados; e inclusive, lo que es mas importante aún, puede ser utilizada como input por nuestra propia herramienta para componerla con una nueva transformación.

7.5 Conclusión del capítulo

Con esta herramienta, 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.

Capítulo 8
Ejemplo

8.1 Limitaciones de las herramientas de transformación

Antes de abordar el ejemplo que muestra el funcionamiento de nuestra herramienta, haremos un breve análisis del grado de madurez de las herramientas utilizadas:

Uno de los objetivos iniciales planteados para esta tesis era el de comprobar que las transformaciones compuestas por nuestra herramienta pueden ser ejecutadas por los motores de ejecución de transformaciones que existen en la actualidad (ej. Medini o SmartQVT).

Dado que el desarrollo de las herramientas de ejecución se ecuentra en estado incipiente (producto de la corta vida de la especificación del lenguaje) no es posible ejecutar las transformaciones generadas por nuestra herramienta debido a que los motores de ejecución existentes no implementan aun todas las contrucciones lenguísticas de la especificación, muchas de las cuales son directamente requeridas para la construcción de las transformaciones resultantes.

En cuanto a QVT Declarativo, los motores no implementan aún toda la semántica al momento de la ejecución de transformaciones que contienen expresiones iterativas escritas en OCL en las cláusulas *where* de las *relations*.

Como vimos anteriormente, existen *relations* cuyo efecto son equivalentes:

Caso 1)

```
top relation ClassToTable {
    cn : String;
    checkonly domain uml c : Class {name = cn};
    enforce domain rdbms t : Table {name=cn}}
  }
```

Caso 2)

```
top relation ClassToTable {
    checkonly domain uml c : Class {};
    enforce domain rdbms t : Table {};
    where{t.name = c.name ;}
  }
```

El primer caso utiliza *pattern matching* para el chequeo de consistencia entre el conjunto de origen y destino de la transformación. Esto compila y ejecuta correctamente en las herramientas de QVT declarativo en las cuales hicimos los tests.

El segundo caso utiliza una restriccion OCL en la cláusula *where* para establecer la relación de consistencia entre el conjunto de origen y el conjunto de destino. En este caso, el código compila correctamente, pero su ejecución aun no es la esperada según la definición del lenguaje standard QVT. Por tal razón, no siempre es posible hacer una demostración de la ejecución de *relations* que tengan escritas ciertas restricciones OCL en el *when* y *where*.

Por otra parte, hasta el momento la única herramienta de ejecución de transformaciones operacionales es SmartQVT. Si bien la especificación de QVT tiene claramente definidos los operadores de acceso desde una transformación hacia una librería de transformaciones (cláusulas "*import* …"), el motor de SmartQVT aun no permite realizar accesos *coarse-grained* hacia las librerías.

Las cláusulas "*import* …" son la base de las composiciones de transformaciones operacionales desarrolladas por la teoría de composiciones en Giandini y Pons [16, 17].

8.2 Desarrollo del ejemplo

El siguiente es un ejemplo de la utilización de la herramienta para realizar una composición secuencial entre dos transformaciones.

El ejemplo a desarrollar es un caso emblemático de las transformaciones de modelos: la transformación de modelos representados en UML a modelos para Base de Datos relacionales.

Más concretamente aún, mostraremos el caso de un modelo para un sistema de información para alumnos que participan de cursos en un establecimiento educativo. Modelaremos para ello las entidades de Alumno, Sesión de usuario del sistema, Curso, Profesor y Aula.

Para ello, construiremos la estructura de los modelos con los que vamos a trabajar, los modelos en si mismo, y las transformaciones que toman el modelo UML y general el modelo relacional.

Por último, mostraremos el resultado de la ejecución de cada una de las transformaciones escritas de manera separada entre si, y luego, el resultado de la ejecucion de las transformaciones compuestas por la herramienta transformadora de composiciones.

8.2.1 Los metamodelos

En primer lugar, es necesario definir los metamodelos con los que vamos a trabajar. Es decir, la estructura de las instancias de los modelos a transformar:

El metamodelo de origen

Como metamodelo de origen utilizaremos el diseño de una estructura estática de UML: un diagrama de la estructura base de los elementos que vamos a transformar. Aunque lo que vamos a transformar sea la instancia de este metamodelo, la escritura de las transformaciones se realiza en base al metamodelo.

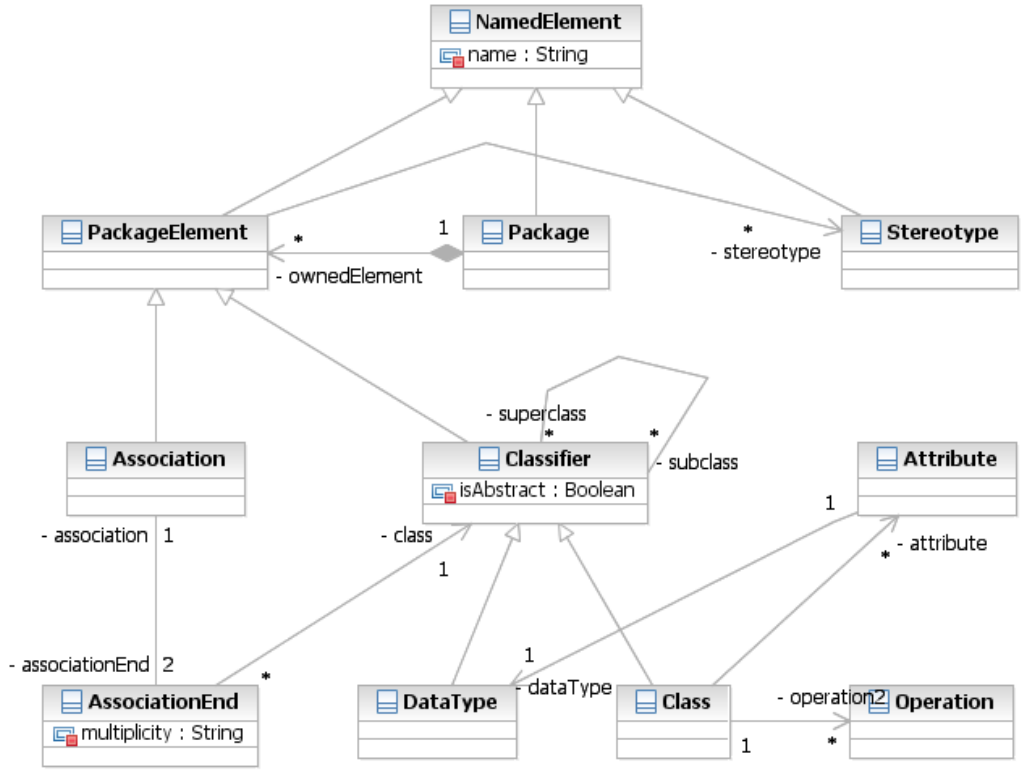

Figura 8-1. Metamodelo de origen del Ejemplo

El metamodelo destino

El modelo de destino generado a partir de las transformaciones, será instancia del metamodelo relacional que a continuación definimos. Por lo tanto, antes de diseñar la transformación es necesario definir el metamodelo que se instanciará.

Este es un metamodelo que representa la estructura de un modelo relacional de tablas.

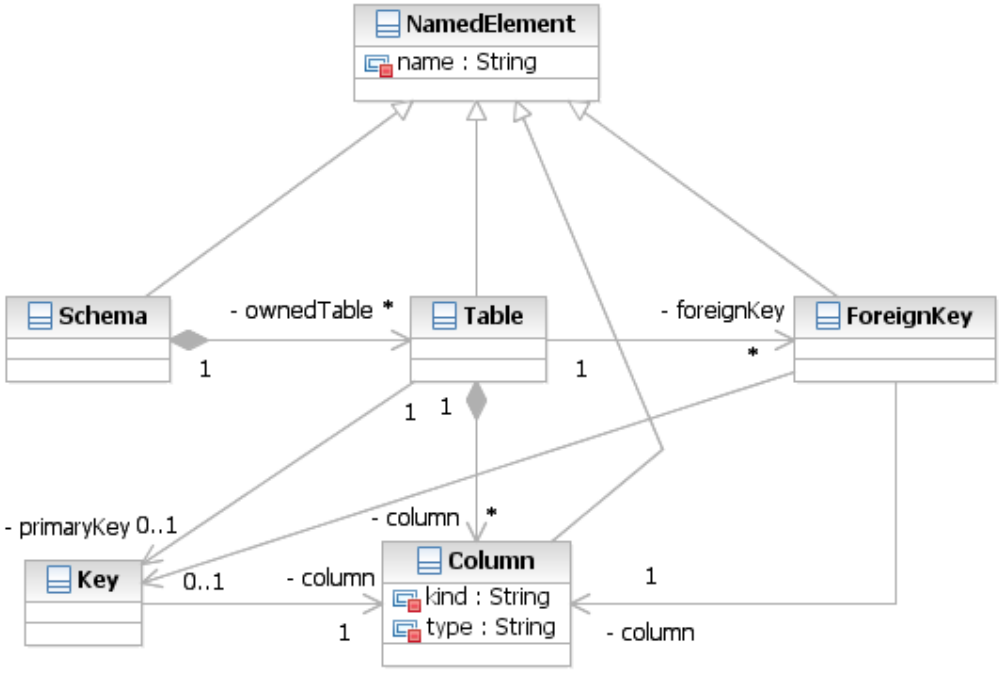

Figura 8-2. Metamodelo de destino del Ejemplo

8.2.2 El modelo de origen

El modelo de origen es lo que efectivamente vamos a transformar.

- Las clases Alumno, Curso, Aula y Profesor son instancias de la metaclase Class.
- Cada variable de instancia de las clases mencionadas, son instancias de la metaclase Attribute.
- Cada relación respresentada entre las clases, son instancias de la metaclase Association.

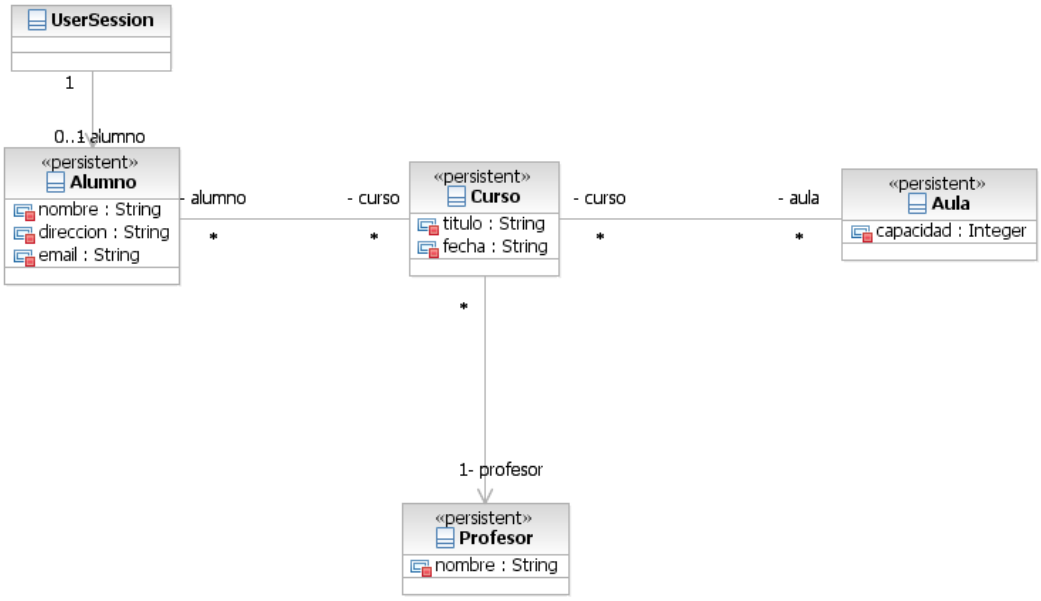

Figura 8-3. Modelo de origen del Ejemplo

8.2.3 Las definiciones de las reglas

A continuación enumeraremos el conjunto de reglas que deben cumplirse entre el modelo UML y el modelo Relacional a fin de poder escribir las transformaciones que las implementen.

1) Por cada Package del modelo UML, se debe generar un Schema en el modelo relacional con el mismo nombre.

2) Por cada instancia de la metaclase Class, no abstracta y con estereotipo "persistente" en el modelo UML, se debe generar una Table en el modelo relacional con el mismo nombre que la clase que se transforma. Cada tabla generada tendrá una clave primaria cuyo nombre será el nombre de la clase que la representa mas el string 'id'.

3) Por cada atributo de clase en el modelo UML, se debe generar una columna de Table en el modelo relacional con el mismo nombre y cuyo tipo se transformará de acuerdo a las siguientes condiciones:

   a. Por cada atributo de tipo String, se genera una columna de tipo VARCHAR[255].

   b. Por cada atributo de tipo Integer, se genera una columna de tipo INTEGER

4) Por cada Association de "muchos a muchos" en el modelo UML, se debe generar una Table en el modelo relacional con las ForeignKey de las tablas que relacional

5) Por cada Association de "uno a muchos" en el modelo UML, se debe crear una ForeignKey en la Table destino, hacia la Table que le corresponde a la clase que la referencia.

8.2.4 Las transformaciones

Para resolver cada uno de estos casos, hemos planteados dos transformaciones independientes entre sí: una de ellas se encargará de transformar las clases con atributos del modelo de origen a tablas con atributos en el modelo de destino. La otra transformación, transformará las asociaciones del modelo de origen en tablas (para los casos de asociaciones "muchos a muchos") o claves foráneas (en caso de asociaciones "uno a muchos") en el modelo de destino.

La primera transformación: ClassToRel

```
/*
* Transformación de clases de un modelo UML a tablas en el modelo
* RDBMS (relacional). Esta transformación resuelve los puntos 1,2 y 3
* de la especificación de reglas.
*/

transformation ClassToRel(uml:UMLModel, rdb:RelModel) {

   /*
   * Transforma un paquete UML a un schema de RDBMS
   */
   top relation PackageToSchema {

      cn : String;

      checkonly domain uml p : Package { name = cn};

      enforce domain rdb s : Schema {};

      where{
         ClassToTable (c, t); //Postcondicion ClassToTable
      }

   }

   /*
   * Cada clase del modelo UML se convierte a Tabla en el modelo
   * RDBMS. Se transforman solo las clases que no sean marcadas como
   * abstractas y que tengan el estereotipo "persistente".
   */
   relation ClassToTable {

      cn : String;

      checkonly domain uml p : Package {
        ownedElement = class : Class{
          stereotype = s : Stereotype {
            name = 'persistent'
          },
          isAbstract = false, //Esto puede llegar a resultar
                              //redundante debido a que la clase fue
                              //marcada previamente como 'persistent'
          name = cn
        }
      };

      enforce domain rdb s : Schema {
        ownedTable = table : Table{
```

```
          name = cn,
          column = newColumn : Column{  //Crea la columna id
            name = cn+'id',
            type = 'INTEGER'
          },
            primaryKey = k : PrimaryKey { //Crea la clave
                column = newColumn : Column{} //Liga la columna creada
                                              //a la clave primaria
          }
        }
      };

      where{
         AttributeToColumn(class,table);
         ClassToPrimaryKey(class, pk);
      }
   }

   /*
   * Cada atributo de clase del modelo UML se convierte a columna en
   * el modelo RDBMS. Los atributos de tipo String se transforman a
   * columnas de tipo VARCHAR[255] y los atributos de tipo Integer se
   * transforman a columnas de tipo INTEGER.
   */
   relation AttributeToColumn {

      aName, attributeType, columnType: String;

      checkonly domain uml class : Class {
        attribute = attr : Attribute {
            name = aName,
            type = attrType : PrimitiveDataType {
               name = attributeType
            }
        }
      };

      enforce domain rdb column : Table {
        column = col : Column{
            name = aName,
            type = columnType //el tipo se asigna en el where.
        }
      };

      where {
        columnType =
            if attributeType = 'String' then 'VARCHAR[255]'else
            if attributeType = 'Integer' then 'INTEGER' endif
            endif;
        }
      }
```

```
  /*
   * Se agrega una el nombre de la clave primaria
  */
  relation ClassToPrimaryKey {

        cn : String;

        checkonly domain uml c : Class { name = cn  };

        enforce domain rdb k : PrimaryKey { name = cn + '_pk' };

      }
  }
};
```

El estado del conjunto de destino luego de la ejecución de la primera transformación

La ejecución de esta transformación sobre el modelo del sistema de alumnos, da origen a las siguientes tablas en el modelo relacional.

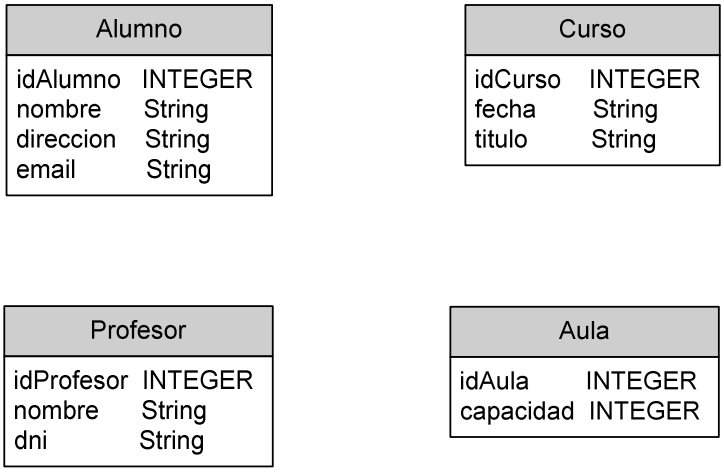

Figura 8-4. Modelo de destino (luego de ejecutar la primer transoformacion)

La segunda transformación: AssoToRel

```
/*
* Transformación de asociaciones "muchos a muchos" del modelo UML a
* tablas en el modelo RDBMS (relacional). En caso de que la asociación
* sea de "uno a muchos" se deberá agregar una clave foránea en la
* clase referenciada. Esta transformación resuelve los puntos 1,4 y 5
* de la especificación de reglas
*/

transformation AssoToRel(uml:UMLModel, rdb:RelModel) {


   key Schema{name};
   key Table{name};
```

```
/*
* Transforma un paquete UML a un schema de RDBMS. Esto es igual a
* la transformación anterior y se realiza debido a que es escrita
* como una transformación independiente de la otra. Esto no genera
* conflicto.
*/

top relation PackageToSchema {

  cn: String;

  checkonly domain uml p : Package { name = cn};

  enforce domain rdb s : Schema { name = cn};

  where{
      AssociationToTable (p, s) and//Postcondicion AssociationToTable
      AssociationToForeignKey(p, s);//Postcondicion
                                          AssociationToForeignKey

  }
}

/*
* Transforma cada relacion "muchos a muchos" de UML en una tabla en
* RDBMS. "associationEnd" es una colección. El template expression se
* hace sobre dos elementos de esa colección.
*/
relation AssociationToTable {

  aName: String;

  checkonly domain uml p : Package {
    ownedElement : association : Association {
      associationEnd = assoEnd1 : AssociationEnd{
            multiplicity = '*',
            class = class1 : Class{
              stereotype = st1 : Stereotype {
                  name = 'persistent'
              }
            }
      },
      associationEnd = assoEnd2 : AssociationEnd{
            multiplicity = '*',
            class = class2 : Class{
              stereotype = st2 : Stereotype {
                  name = 'persistent'
              }
            }
      },
      name = aName
  }
};
enforce domain rdb sch : Schema { //Esto crea solo la tabla y no
                                      //las columnas. Estas últimas
                                      //son creadas en la siguiente
                                      //relation.
  ownedTable = table : Table{
        name = aName
  }
};
```

```
when{
      assoEnd1 <> assoEnd2; //Como los assoEnd son elementos de una
                            //colección y la selección se hace por
                            //template expression, se elimina la
                            //posibilidad de conformar
                            //tuplas[assoEnd1,assoEnd2] donde
                            //assoEnd1=assoEnd2.

}
where{
      AssociationToTableColumn(association, table)
}

}

/*
* Agrega una columna y una clave foránea en la tabla por cada relacion
* "muchos" en el dominio UML. Eso solo lo hace para las relaciones
* "muchos a muchos" dado que esa restriccion se encuentra en el
* pattern expression de AssociationToTable
*/
relation AssociationToTableColumn {

  checkonly domain uml association : Association {
        associationEnd = assoEnd : AssociationEnd{
            multiplicity = '*',
            class = referencedClass : Class{}
        }
   };

   enforce domain rdb table : Table {
       column = newColumn : Column{
            name = referencedClass.name + 'id',
            type = 'INTEGER'
       },
       foreignKey = fk : ForeignKey{
            name = referencedClass.name + 'id',
            column = newColumn : Column {},
            refersTo = pKey : PrimaryKey {}
        }
   };
   where{
    ClassToPrimaryKey(class, pKey);
   }
}


/*
* Agrega una foreignKey por cada relación "1..muchos".
*
*/
relation AssociationToForeignKey {
  an, fkn, fcn, multNotN : String;

  checkonly domain uml p : Package {
    ownedElement = asso  : Association{  //Selecciona relaciones
                                         // 1..Muchos
      name = an,
      associationEnd = ae1 : AssociationEnd {
        multiplicity = multNotN,
        class = class1 : Class {
```

```
          stereotype = st1 : Stereotype{
              name = 'persistent'
            }
      }
     },
      associationEnd = ae2 : AssociationEnd {
        multiplicity = '*',
        class = class2 : Class {
            stereotype = st2 : Stereotype{
              name = 'persistent'
            }
      }
   }
  }
};

enforce domain rdb s : Schema{
  ownedTable = tabla : Table{
    name = class2.name, //Se vincula la clase "muchos" con su
                        //repectiva tabla
    column = fc : Column{},
    foreignKey = fk : ForeignKey { //Crea la foreignKey para cada
                                   // relación 1..Muchos
      name = fkn,
      column = fc : Column {
         name = fcn,
         type = 'INTEGER',
      },
      refersTo = pKey : PrimaryKey {}
  }
};

when {
    multNotN <> '*';
}

where {
    fkn = class1.name+'id';
    fcn = fkn;
}
}
```

El estado del conjunto de destino luego de la ejecución de la segunda transformación

El resultado de la aplicación de esta segunda transformación, es la creación de tablas en el caso de las transformaciones "muchos a muchos" y la inserción del las claves foráneas de las tablas relacionadas con relaciones "1 a muchos". El estado graficado no incluye los elementos creados por la ejecución de la primera transformación.

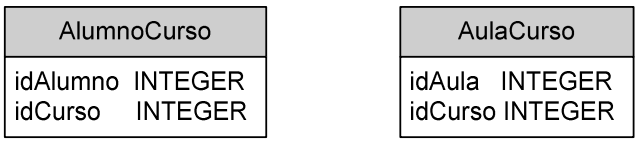

Figura 8-5. Modelo de destino (luego de ejecutar la relation AssociationToTable de la segunda transoformacion)

Luego, en el caso de la tabla Curso, se le agrega la clave foránea idProfesor, producto de la relación "uno a muchos" en el diagrama UML.

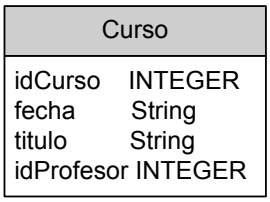

Figura 8-6. Modelo de destino (luego de ejecutar la relation AssociationToForeignKey de la segunda transoformacion)

El modelo de destino completo

La ejecución de las dos transformaciones anteriores sobre el modelo UML del sistema de alumnos, da como resultado el siguiente conjunto de tablas en el modelo Relacional.

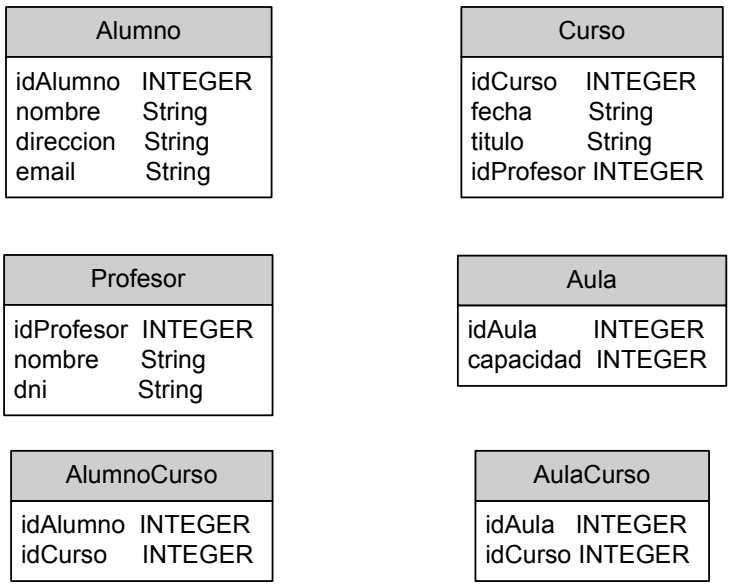

Figura 8-7. Modelo de destino completo (luego de ejecutar ambas transoformaciones)

8.2.5 La utilización de la Herramienta Calculadora de Composiciones

Como mencionamos anteriormente, cada una de las transformaciones fueron escritas guardando independencia entre si. Cada una de estas pudo haber sido escritas hasta por equipos de desarrollos independientes, siempre y cuando se mantengan dentro del mismo metamodelo.

Ahora, y como producto de esta tesis, podemos usar la Herramienta Calculadora de Composiciones para obtener una transformación compuesta producto de la operación de composición denominada "Composición Secuencial" entre las dos transformaciones descritas anteriormente.

8.2.6 El Resultado de la composición

A continuación presentamos el código resultado de aplicar la operación de "Composición Secuencial" a las dos transformaciones anteriormente descritas. Dicha transformación puede utilizarse como Input de las herramientas que ejecutan transformaciones, y su resultado (tal como lo definie la operación algebraica de "Composicion Secuencial") será equivalente a la ejecución de la primera transformación y seguidamente, la ejecución de la segunda transformación… todo como resultado de la ejecución de una sola ejecución de esta transformación compuesta:

```
transformation ClassToRel;QVTAssoToRel(uml:UMLModel, rdb:RelModel) {

   /*
   * Resultado de la unión algebraica de transformaciones.
   */
   top relation PackageToSchema;QVTPackageToSchema {

      checkonly domain uml p : Package {};

      enforce domain rdb s : Schema {};

      when { true }//when de la primera transformación (que es vacio)

      where{
      ClassToTable (c, t) and //where de la primera transformacion
      (true implies AssociationToTable (p, s) and //where de la
            AssociationToForeignKey(p, s))  //segunda transformacion
      }

   }

   relation ClassToTable {…} //de la primera transformación
   relation AttributeToColumn{…} //de la primera transformación
   relation AssociationToTable {…} //de la segunda transformación
   relation AssociationToTableColumn {…}//de la segunda transformación
   relation ClassToPrimaryKey {…}//de la segunda transformación
   relation AssociationToForeignKey {…}//de la segunda transformación

}
```

8.3 Conclusiones del capítulo

A partir de este ejemplo vimos que es posible construir distintas transformaciones modulares que luego pueden ser compuestas reutilizando el código escrito para cada una de estas, obteniendo así transformaciones de más alto nivel.

En este caso en particular hemos diseñado el ejemplo a partir de dos transformaciones que son comúnmente utilizadas en el ámbito del desarrollo de aplicaciones de Arquitecturas Orientadas a Modelos.

Es necesario destacar que cada una de las transformaciones tiene una única *top relation*. Como mínimo, una transformación debe tener al menos una top relations, pero no tiene establecido una cantidad máxima, esto depende del diseño que le de al desarrollador. Como vimos en el capítulo 2, las *top relations* son el punto de partida en la ejecución de

las transformaciones. Solo las *top relations* de cada transformación son *mergeadas* en una única *top relation* en la transformación resultado de la composición. Las *top relations* de cada transformación, son las únicas que deben maximizar el uso de las restricciones OCL para poder ser utilizadas por la Herramienta Calculadora de Composiciones con las operaciones algebraicas definidas. El resto de las relations (las que no son top) puede utilizar cualquiera de los recursos del lenguaje, como los *template expressions*, para de definición de reglas de consistencias entre el conjunto de origen y el conjunto de destino.

En general, los modelos con los que trabajamos en MDD suelen tener una raiz, es decir, un elemento destacado a partir del cual se unen los demas elementos del modelo: en nuestro ejemplo un Package como raiz de UML y Schema como raiz de RDB. Esta característica tiene implicancia de usos y costumbres en el desarrollo del código de las transformaciones: para estos casos, la primera *relation* que se define en una transformación, suele ser una *top relation* que transforma precisamente el elemento raiz del modelo. Estas transformaciones suelen ser las mas simples de escribir. Además ocurre que, cuando existen dos transformaciones distintas que transforman elementos distintos de un mismo modelo, ambos elementos son identificados para transformarse a partir de hacer la transformación del elemento raiz: aún cuando una transformación es escrita para transformar Clases y otra es escrita para transformar Asociaciones, ambas tienen una relation comun que es la transformación de PackageToSchema, que es la transformación del elemento raiz y que fueron escritas de la misma manera.

Capítulo 9

Conclusiones y trabajo futuro

Las definiciones de las operaciones algebraicas de composición entre transformaciones brindan un importante avance en la Ingeniería de Software Dirigida por Modelos. La implementación de un software que permite componer estas transformaciones de manera automática, brinda un importante nivel de productividad en el desarrollo de aplicaciones a partir de la reduccion de la brecha semántica que existe entre el dominio del problema y la solucion, reduciendo además los tiempos de desarrollo.

QVT tiene un completo soporte operacional que permite expresar composiciones de transformaciones operacionales. Aun así, 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

Vale la pena mencionar que las transformaciones pueden ser manipuladas en tres niveles: especificación, implementación y ejecución de transformaciones. El desarrollo de nuestra herramienta aboda los niveles de composición de especificaciones e implementación. La ejecución de nuestra herramienta toma dos especificaciones textuales de transformaciones y logra obtener la especificación textual de la transformación resultado de la composición. La ejecución de las transformaciones parámetros de los operadores de composición y la ejecución de la transformación compuesta, está fuera del alcance de nuestro trabajo.

9.1 QVT Declarativo vs. QVT Operacional

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.

En cuanto a su implementación, QVT Declarativo y QVT Operacional tienen enfoques totalmente distintos entre si.

QVT operacional está planteado como un lenguaje imperativo donde cada relacion es una función o procedimiento con algunas restricciones. Tiene un punto de arranque “main” que es igual al cuerpo del programa principal que existe en la mayoria de los lenguajes imperativos. Cuenta con definición de parámetros formales y con mecanismos de reutilización de librerías.

El flujo de control de una transformación operacional arranca en el cuerpo de la transformación y luego se delega en las invocaciones a los mappings de manera secuencial.

QVT Declarativo tiene un enfoque diferente. Se basa en un conjunto de reglas que deben cumplirse simultáneamente para determinar si existe consistencia entre dos conjuntos (dominio y codominio). Si no se cumpliera la consistencia entre conjuntos, entonces se crean los elementos necesarios (habitualmente en el codominio) para forzar la consistencia… Ese mecanismo de fuerza se lo puede interpretar como una transformación de los elementos del dominio en elementos del codominio según las reglas (*relations*) definidas en la Transformation.

Para este caso, no existe flujo de control de la ejecución. Todas las reglas se tienen que satisfacer simultáneamente. Lo que se produce es un grafo de reglas, donde alguna regla tiene como precondición tanto a predicados como a otras reglas (en el *when*) y como postcondición, tambien predicados y otras reglas de transformación (en el *where*). A veces uno se ve tentado a asociar a las postcondiciones de otras reglas con "invocaciones" a funciones (*relations*), pero no es así. Tampoco hay reglas que puedan intepretarse como que se ejecutan "primero" sino que en realidad, una *top relation* tiene precedencia sobre una *relation* que no es *top*.

En particular, y solo para hacer un análisis de la implementacion, el motor de ejecución de Medini QVT (Declarativo) dispara concurrentemente el chequeo de las *top relations* (son mini-threads chequeadores de consistencia). Utiliza las precondiciones como guardas de sincronizacion (cuando se cumple la el when, se ejecuta el chequeo de la relation), y posteriormente se dispara el chequeo de las postcondiciones.

Tantas son las diferencias entre QVT Declarativo y QVT Operacional, que la construcción de la herramienta demandó la construcción y el involucramiento de dos lenguajes que tenían muy poco en común, excepto su finalidad.

9.2 Trabajo futuro

Las operaciones algebraicas implementadas por nuestra herramienta estan basadas en una maximización de escritura de *relations* utilizando restricciones OCL. Las cláusulas expresadas en las cláusulas *where* son restricciones sobre el conjunto de propiedades que se definen en las *template expression.* Por lo tanto, el poder de la utilización de restricciones definidas en las cláusulas *where* se incrementa solo en la medida en que se utilice como complemento de las *template expression*. Las operaciones algebraicas, tal como estan definidas hasta ahora, no contemplan las situaciones en las que se utiliza *template expressions* para definir las reglas de consistencia entre el conjunto de origen y el conjunto de destino. Esto puede implicar que el programador de las transformaciones no esté libre de utilizar la especificación completa del standard de QVT al momento de escribirlas. Para ello, es necesario lograr avances en las definiciones de las operaciones algebraicas para que éstas contemplen los casos en los cuales las relations son escritas mendiante la utilización de *template expression* en los *domains* para poder así contemplar todo el espectro del lenguaje QVT. La contribución al mejoramiento de estas definiciones a partir del estudio de casos prácticos realizados en esta tesis, es planteada como trabajo futuro.

Por otra parte, Si bien existen diversas operaciones que pueden ser aplicadas para componer transformaciones, hemos elegido solo tres de ellas para mostrar que es posible desarrollar una herramienta de software que implemente las operaciones definidas en los trabajos de Giandini y Pons en [15, 16]. Además, hemos elegido las operaciones de Union, Composicion Secuencial y Fork porque son las más útiles y representativas de las composiciones de transformaciones de modelos. No obstante, y a partir de este desarrollo, es posible incorporar nuevos operadores de composición a la herramienta. Esta herramienta fue diseñada utilizando separación de entidades conceptuales y por lo tanto permite construir nuevas operaciones de manera independiente e incorporarlas a la herramienta de una manera simple. Nuevos operadores como Intersección, operación Inversa, Vacío e Identidad son planteados como trabajo futuro de la herramienta.

De todas maneras, para potenciar plenamente el poder del álgebra es necesario contra con un diseño de estrategia que permita descomponer una transformación en pequeñas partes, las cuales puedan ser re-combinadas por la Herramienta Calculadora de Composiciones para recomponer la transformación original. En general, ejemplos de diseño de estrategias incluyen el análisis de casos, analisis de backtraking y muchos más. El desarrollo de tales estrategias son planteadas como trabajo futuro.

Hasta el momento, los operadores de composición fueron definidos de manera binaria, es decir, están preparados para aceptar dos parámetros. Para poder hacer una composición algebraica entre más de dos transformaciones, es necesario aplicar el operador sobre dos transformaciones; luego, al resultado, que es otra transformación, aplicar un nuevo operador para componerla con otra transformación, y asi sucesivamente.

Para potenciar el uso de los operadores, planteamos como trabajo a futuro la respuesta a la necesidad de contar con operadores que logren componer mas de dos transformaciones al mismo tiempo. Es decir, que desde la herramienta se puedan seleccionar tres o más transformaciones, y luego aplicar alguna de las operaciones de composición entre ellas.

No cualquier transformación puede componerse con la Herramienta Calculadora de Composiciones. Las operaciones algebraicas tienen como precondición que las transformaciones que se pueden componer posean a lo sumo una *top relation*. La especificación del lenguaje QVT obliga a tener definida al menos una relacion *top*, pero no restringe el número de relaciones *top* dentro de una transformación. Por lo tanto, a partir de la especificación del lenguaje y la especificación de las operaciones algebráicas, solo podemos componer aquellas transformaciones que cuentan con una y solo una relación *top*. Ahora bien, es posible reescribir una transformación con varias relaciones *top*, en una equivalente que sea especificada con solo una relación *top*. Este trabajo se puede realizar siguiendo un algoritmo teórico ejecutado mediante la escritura (manual) de un desarrollador, o también podría hacerse de manera automática. Para nuestro caso, se podría ampliar radicalmente la base de trasformaciones en condiciones de ser compuestas, si nuestra herramienta contara con un mecanismo automatizado para cambiar una transformación con varias *top relati*ons a otra con solo una, o bien a partir de redefinir las operaciones algebraicas de composición para poder componer transformaciones que contengan varias *top relations*.

Con respecto a la herramienta, otros de los objetivos a cumplir en el futuro es la integración con otras herramientas de ejecución de transformaciones. En la actualidad, la Herramienta Calculadora de Composiciones permite tomar el texto de la definición de

transformaciones, componerla con otra transformación textual, y luego retornar el código de la transoformación resultado de aplicar la operación de composición entre las transformaciones "operandos". Es deseable lograr una integración completa que logre validar, ejecutar transformaciones, componerlas y ejecutar las transformaciones compuestas desde una misma herramienta. Esto también está planteado como trabajo futuro.

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), ∪, ∩, ∅, V,¬, ;,↵,1), tal que (P(V), ∪, ∩, ∅, V, ¬) 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 desple gables 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 unmodelo 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 lenguajesde 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 especifica 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 plugin Eclipse para el Eclipse Modeling Frame work (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 especificacio nes 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/cgibin/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. Meta Object Facility (MOF) 2.0. Query/View/Transformation Specification (QVT) - Version 1.0. April 2008. http://www. omg.org.

16. Claudia Pons, Roxana Giandini, Gabriela Perez, Gabriel Baum. An Algebraic Approach for Composing Model Transformations in QVT. ATEM-4th International Workshop on Software Language Engineering at the 10th International Conference MoDELS 2007. Nashvillle, TN, US. October 2007.

17. Giandini, R. Un Marco Formal para Transformaciones en la Ingeniería de Software Conducida por Modelos. Tesis de Doctorado. UNLP (2008)

18. Composición de Transformaciones de Modelos en MDD basada en el Álgebra Relacional. Roxana Giandini and Gabriela A. Pérez and Claudia Pons 10º Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes Software, IDEAS 2007. Mayo, 2007. Caracas, Venezuela. ISBN: 978-980-325-323-3. pág: 239-252. (2007)

19. Pons, C., Pérez, G. and Giandini, R The Algebraic Theory of Problems as Holistic Foundation for Model Transformation Composition. To be published.

20. OMG. *The Object Constraint Language Specification* – Version 2.0, for UML 2.0, revised by the OMG, http://www.omg.org, April 2004.

21. 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

22. 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

23. 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

24. *The Unified Modeling Language Infrastructure* version 2.0, OMG Final Adopted Specification. March 2005. http://www.omg.org

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 (1887).

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, N0 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. 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.

35. *ePlatero Home page*. http://sol.info.unlp.edu.ar/eclipse.

36. *Eclipse* http://www.eclipse.org.

37. *Eclipse Modeling Framework* (EMF), http://www.eclipse.org/emf/

38. *ModelMorf Home page*. http://www.tcs-trddc.com/ModelMorf/index.htm

39. *Medini Home page*. http://projects.ikv.de/qvt

40. *OSLO Home page*. http://oslo-project.berlios.de/

41. *SmartQVT Home page*. http://smartqvt.elibel.tm.fr/

42. *Together Home page*. http://www.borland.com/us/products/together/index.html

43.*Together FAQ* http://www.borland.com/resources/en/pdf/products/together/together_faq.pdf

44. *JFlex Manual* http://jflex.de/manual.html

45. *CUP Project* http://www2.cs.tum.edu/projects/cup/

46. *Atlas Model Weaver Project* Web Page. http://www.eclipse.org/gmt/amw/, 2005.

47. Dionisio de Niz and Raj Rajkumar. *Glue Code Generation: Closing the Loophole in Model-based Development* www.cse.wustl.edu/~cdgill/RTAS04/rtas04.pdf

48. 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.

49. 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

50. Epsilon Documentation http://www.eclipse.org/gmt/epsilon/doc/

51. 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.

52. 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.

53. 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.

54. ModelBus homepage: http://www.modelbus.org/modelbus/

55. Xavier Blanc, http://www.eclipse.org/proposals/eclipse-mddi/main_data/ModelBusWhitePaper_MDDI.pdf

56. Bergstra, J.A. and Klint, P.. The discrete time toolbus – a software coordination architectture. *Science of Computer Programming,* 31:205–229, 1998.

57. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.4194

58. Paul Klint *The ToolBus: a Service-oriented Architecture for language processing tools*. http://www.win.tue.nl/ipa/archive/springdays2007/toolbusIPA2007.pdf

59. Kleppe, Anneke. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Spain, June 2006.

60. http://www.irisa.fr/UMLAUT/Architecture/umlaut.html