Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Departamento de Lenguajes y Sistemas Informáticos II

MeTAGeM: Entorno de Desarrollo de Transformaciones de Modelos Dirigido por Modelos

Autora: Verónica Andrea Bollati

Directora de Tesis: Esperanza Marcos Martínez

Co-Directora de Tesis: Belén Vela Sánchez

Móstoles, Diciembre 2010

Universidad
Rey Juan Carlos

La Dra. Dª Esperanza Marcos Martínez, Catedrática de Universidad del Departamento de Lenguajes y Sistemas Informáticos II de la Universidad Rey Juan Carlos de Madrid y la Dra. Dª Belén Vela Sánchez, Profesora de Universidad del Departamento de Lenguajes y Sistemas Informáticos II de la Universidad Rey Juan Carlos de Madrid, directora y codirectora, respectivamente, de la Tesis Doctoral: "MeTAGeM: Entorno de Desarrollo de Transformaciones de Modelos Dirigido por Modelos" realizada por la doctoranda Da. Verónica Andrea Bollati,

HACEN CONSTAR QUE:

Esta tesis doctoral reúne los requisitos para su defensa y aprobación

En Madrid, a 10 de Diciembre de 2010

Fdo.: Esperanza Marcos Martínez Fdo: Belén Vela Sánchez

Cada uno de nosotros es un modelo totalmente nuevo,
parecido a otros modelos pero totalmente diferente.
Anónimo

Resumen

En los ultimos años la Ingeniería Dirigida por Modelos (*Model Driven Engineering*, MDE) ha ido adquiriendo cada vez mayor nivel de madurez. Siguiendo los principios del MDE, en el año 2001, la OMG (*Object Management Group*) propone la Arquitectura Dirigida por Modelos (*Model Driven Architecture*, MDA). Desde entonces, MDE y más concretamente la MDA, se han aplicado con éxito en diferentes contextos, dando lugar a una gran cantidad de metodologías dirigidas por modelos para el desarrollo de software que abarcan todos los campos de la Ingeniería de Software.

Con la llegada de MDE los modelos pasan a ocupar un rol principal guiando el proceso de desarrollo: mediante la definición de modelos precisos que captan todos los requisitos y especificaciones sobre el sistema a construir así como la plataforma donde se implementará. La idea principal es ir generando una serie de modelos que permitan representar el sistema cada vez con menor nivel de abstracción. Así, el nivel de detalle de los modelos obtenidos en las últimas fases del proceso permitirá generar (semi-)automáticamente el código que implementa el sistema.

El eslabón que une cada nuevo paso del proceso (la generación de un nuevo modelo) es una **transformación de modelos**. El propósito principal de las transformaciones de modelos es convertir un modelo (o varios) del sistema, en otro modelo (o varios). Estas transformaciones deberían hacerse de forma (semi)-automática, implementándose mediante la definición de reglas de transformación (*mappings*) entre dichos modelos. Surgen así nuevos lenguajes y herramientas, que facilitan la automatización de la operación de transformación. Estos lenguajes y herramientas difieren en múltiples aspectos tales como: el paradigma (declarativo, imperativo o híbrido); el grado de generalidad (de propósito general o diseñado para dominios específicos); o el nivel de abstracción.

Esta diversidad de tecnologías trae aparejado una serie de problemas: por un lado, el usuario debe ser capaz de seleccionar el lenguaje más adecuado para resolver un problema en particular y si se desea cambiar de lenguaje, debe **aprender a utilizarlo y el tiempo invertido** en este proceso es directamente proporcional a la complejidad del lenguaje. Esto hace que el proceso de implementación de una transformación sea una actividad muy laboriosa. Por otro lado, en general las herramientas de transformación soportan un lenguaje de transformación específico, por lo que existen problemas de **interoperabilidad** entre las mismas.

Por todo ello, se considera conveniente buscar soluciones que permitan facilitar el aprendizaje y uso de los lenguajes de transformaciones, así como mejorar la interoperabilidad entre las herramientas de soporte, de modo que se facilite también la migración de un lenguaje a otro. Dado que estamos en un contexto de MDE, sería lógico tratar de aprovechar las ventajas que la propia ingeniería dirigida por modelos nos proporciona, y aplicar MDE al proceso de definición de las transformaciones.

Además, actualmente, no existe una única propuesta, similar a MOF para la especificación de lenguajes en el ámbito de MDE, que permita unificar los lenguajes de transformación existentes. Desde este punto de vista, la propuesta de entorno de desarrollo que se realiza en esta tesis permitirá realizar la definición de transformaciones de modelos a un alto nivel de abstracción sin tener en cuenta el lenguaje de implementación final de la transformación. Con ello se pretende resolver los dos problemas mencionados anteriormente: a) proporcionar un lenguaje de transformación de alto nivel, independiente de plataforma, y más cercano al usuario y b) permitir la generación (semi-)automática de transformaciones en lenguajes específicos dependientes de plataforma. Estas mejoras facilitarán tanto la tarea de desarrollo del programador de transformaciones, así como la interoperabilidad y migración entre herramientas y lenguajes.

El **entorno de desarrollo de transformaciones de modelos, dirigido por modelos** (MeTAGeM) que se presenta en esta tesis incluye:

- La definición de un proceso metodológico para el desarrollo MDE de transformaciones de modelos.
- La especificación de un meta-modelo de transformaciones de alto nivel que permita modelar las transformaciones a nivel PIM.
- La especificación de un meta-modelo para la aproximación híbrida que permita modelar las transformaciones a nivel PSM.
- La especificación de un meta-modelo de transformaciones que incluya: las transformaciones entre el meta-modelo PIM y el meta-modelo PSM, y las transformaciones entre el meta-modelo PSM y los meta-modelos PDM (ATL y RubyTL).
- La implementación de una herramienta que soporte: a) el modelado de transformaciones de nivel PIM en base al meta-modelo de transformaciones propuesto; b) el modelado de transformaciones de nivel PSM en base al meta-modelo de la aproximación híbrida; c) el modelado de las transformaciones a nivel PDM en base a los meta-modelos del lenguaje de de transformación

ATL y RubyTL; d) un meta-transformador que permita obtener los modelos de las transformaciones conformes al lenguaje de transformación ATL y RubyTL y a partir de estos últimos, el código implementable de la transformación en dichos lenguajes.

Abstract

In recent years, Model Driven Engineering (MDE) has begun to achieve certain levels of maturity. In 2001, following the principles of MDE, the Object Management Group (OMG) proposed the Model Driven Architecture (MDA). Since then, MDE and more specifically MDA have been successfully applied in many different contexts, resulting in a vast amount of model driven methodologies for software development that covers almost every field of software engineering.

With the appearance of the MDE, models occupy a leading role in guiding the development process. The models are now defined to capture accurately all the requirements and specifications of the system to be built, as well as the platform where it will be deployed. The main idea is to generate a set of models to represent the system increasingly lower abstraction levels. Thus, the detail level of the obtained models in latter stages of the process will generate (semi-) automatically the code that implements the system.

A model transformation is the link between each step of the process. The main purpose of models transformation is to convert (one or more) model of the system in another (or several) models. These transformations should be done in (semi)-automatic way, and be implemented by means of transformation rules (mappings) defined between these models. There have appeared new languages and tools arise, that facilitates automatic transformations which differ in many aspects, such as: the paradigm (declarative, imperative or hybrid), the generality degree (general purpose or designed for specific domains), or the abstraction level.

This technology diversity can cause certain problems. First, the developer should be able to select the most suitable language to solve the problem. After that, if they want to change this language they have to learn how to use it, and frequently this learning is more difficult the more complex the language is. Hence, on the one hand, the process of implementing a transformation is a complicate activity and can consume a long time. On the other hand, transformation tools support a specific transformation language, in general; so there are interoperability problems between them.

Therefore, it is recommend found solutions to ease the learning and the use of transformation languages, as well as to improve interoperability between the support tools, and to facilitate the migration from one language to another. Since we are in a MDE context, it would be logical to take advantage of it and apply MDE to the process of defining the transformations.

In addition, currently there is no proposal in the MDE field that unified the existing transformation languages, similar to MOF for the specification of languages.

From this point of view, this thesis proposes a development environment which will allow the definition of transformations models at a high abstraction level, without taking into account the final implementation language of these transformations.

This will allow solving the problems that have already been noted: a) to provide a high-level platform independent transformation language and closer to the user; and b) to enable (semi-) automatic generation of transformations, in platform dependent specific languages.

These improvements will facilitate the development task of transformation programmers, as well as interoperability and migration between languages and tools.

MeTAGeM, the model-driven development environment for models transformations, presented in this thesis includes:

- The definition of a methodological process for MDE development of model transformations
- The specification of a meta-model of high-level transformations that enables modelling the PIM-level transformations
- The specification of a meta-model according to the hybrid approach, which enables modelling the PSM-level transformations
- The specification of a meta-model of the transformations including: the transformations between the PIM meta-model and the PSM meta-model; and transformations between PSM-level meta-model and PDM-level meta-models (ATL and RubyTL)

A tool implementation that supports: a) modelling of PIM-level transformations based on the proposed meta-models of transformations; b) modelling PSM-level transformations based on the hybrid approach meta-models; c) modelling of PDM-level transformations based on meta-models of ATL and RubyTL transformation languages; and d) a meta-transformer to obtain the transformations models in accordance with the ATL and RubyTL transformation languages, and from them, the code that implements the transformation in these languages.

Agradecimientos

En primer lugar quiero agradecer a mi directora Esperanza. Gracias Cuca por la oportunidad que me diste de estar acá, por confiar en mí y por ayudarme a crecer profesional y personalmente. Gracias por acompañarme en todo este proceso, por guiarme con tu experiencia y motivarme día a día. Quiero agradecer también a mi co-directora de tesis, gracias Belén por el apoyo y comprensión que me has dado.

Agradezco también al grupo TARO de la Universidad de la Laguna y al grupo de Bases de Datos de la Universidad Roma Tre, en especial a sus directores el Dr. José Luis Roda y el Dr. Paolo Atzeni, por darme la oportunidad de realizar parte de este trabajo de investigación en dichas universidades. También a los integrantes de ambos grupos con los que compartí tantos buenos momentos y de los que aprendí muchas cosas.

A los Kybelitos, por hacerme sentir parte del grupo desde el primer día. Especialmente a Juancho, por ser mí soporte técnico, por estar a mi lado en los momentos más oscuros y en los más brillantes. Por ayudarme a entender esta idea que al principio me parecía tan lejana. Muchas gracias también a Álvaro y a David, por el soporte en la construcción de MeTAGeM, porque podemos decir que lo conseguimos y sobre todo por haber entendido lo de "soluciones, no problemas".

Gracias a la gente de *Open Canarias*, sobre todo a Antonio, Victor y Orlando, por haberme ayudado a comenzar a *eclipsar* mi mundo y por dejarme "jugar" con QVTo. Gracias Victor, por las interminables conversaciones via msn.

Agradezco también a mis compañeros de patín. Gracias por preocuparse por mis avances y por dejarme hablar de cosas diferentes por lo menos dos horas por semana☺.

Gracias a mi familia, por estar en cada momento de mi vida a pesar de la distancia. Por enseñarme a vivir y a creer en mi misma, y en que uno puede lograr todo lo que se propone y que con esfuerzo y constancia al final todo se consigue.

Muchas gracias a Mariano, por todo, por aguantarme, por estar conmigo, por ser mi *cable a tierra*…no hay nada que pueda escribir que no te haya dicho ya.

Finalemente, agradezco a mis amigos, los nuevos, los viejos, los que están cerca a pesar de la distancia, los que están lejos…a todos. Especialmente a los que terminaron siendo mi familia en España, gracias por estar. Gracias…totales!

Indice

Lista de Figuras

Lista de Tablas

Introducción

En la presente tesis se aborda la definición de una propuesta, en el ámbito de la Ingeniería Dirigida por Modelos (MDE, *Model Driven Enginnering*), que permita definir las transformaciones entre modelos de modo independiente de la plataforma y su generación, de forma (semi-)automática, en un lenguaje de transformación específico.

En la sección 1.1 se presenta la motivación que ha llevado a tomar la decisión de realizar este trabajo. En la sección 1.2 se establece la hipótesis principal y los objetivos directamente derivados de la misma. En la sección 1.3 se describe el contexto en que se desarrolla este trabajo, haciendo referencia a los proyectos de investigación en los que se ha participado y las estancias realizadas por la doctoranda. En la sección 1.4 se resume el método de investigación. Y finalmente, en la sección 1.5 se proporciona una visión general del resto del documento.

1.1 Planteamiento del Problema y Enfoque

El *Object Management Group* (OMG – www.omg.org) es un consorcio internacional que agrupa a unas 800 compañías y produce estándares abiertos para un amplio rango de tecnologías. Uno de los principales objetivos del grupo OMG desde sus inicios en 1989, ha sido ayudar a los usuarios de ordenadores a solventar los problemas de integración e interoperabilidad entre sus sistemas, proporcionando para ello especificaciones abiertas e independientes de fabricantes.

Así, persiguiendo este objetivo, a fines del año 2000, la OMG introduce *Model Driven Architecture* (MDA) [148], una arquitectura para el Desarrollo de Software Dirigido por Modelos (DSDM, [27]). La característica fundamental de MDA es la *definición de modelos formales como elementos de primera clase* para el diseño e implementación de sistemas y la *definición de las transformaciones para establecer relaciones entre modelos y modificar su contenido de forma automática*.

MDA define tres niveles conceptuales, básicos, de modelado en función del nivel de abstracción de los mismos.

Los detalles del negocio y dominio de la aplicación se modelan mediante Modelos Independientes de Computación (CIM, *Computer Independent Model*).

Para representar la funcionalidad y estructura del sistema abstrayéndose de los detalles tecnológicos de la plataforma en la que se implementará, se utilizan los Modelos Independientes de la Plataforma (PIM, *Platform Independent Model*). Los modelos de nivel PIM pueden ser refinados tantas veces como se quiera hasta obtener una descripción del sistema con el nivel de claridad y abstracción deseado.

Por último, para combinar las especificaciones contenidas en un PIM, con los detalles de la plataforma elegida se utilizan los Modelos Específicos de la Plataforma (PSM, *Platform Specific Model*). A partir de los distintos PSM se pueden generar automáticamente distintas implementaciones del mismo sistema.

En la guía de MDA se establece que a nivel PSM se pueden definir diferentes modelos que representan distintos grados de abstracción, pudiéndose agrupar los mismos en modelos que representan elementos generales a todas las plataformas así como modelos que representan elementos dependientes de una plataforma específica. Estos últimos modelos se agrupan en un nuevo nivel conocido como Modelos Dependientes de Plataforma (PDM, *Platform Dependent Models*).

Adicionalmente, podemos considerar el código que implementa el sistema como otro modelo, de más bajo nivel de abstracción, que utiliza una notación textual para su definición.

Una de las mayores aportaciones de MDA reside en la naturaleza de todos estos modelos: son modelos formales, y por lo tanto, entendibles por el ordenador. Así, la característica principal de este nuevo paradigma, al que en adelante nos referiremos como Ingeniería Dirigida por Modelos (MDE, *Model Driven Engineering,* [26]) es el papel que juegan los modelos en el proceso de desarrollo de *software*. De hecho, MDE es un paso natural en la tendencia histórica de la Ingeniería de Software a elevar el nivel de abstracción en la que el software es diseñado y desarrollado. Como ejemplo se puede citar la migración natural desde los lenguajes ensambladores hasta los lenguajes de programación de cuarta generación.

Hasta ahora el uso de modelos estaba relacionado casi exclusivamente con tareas de documentación y, en el mejor de los casos, de diseño, generando un esqueleto del código final (*Rational Rose* es un ejemplo representativo [95]). Por ello, los modelos se descartaban tan pronto como la fase de desarrollo correspondiente terminaba, y no se actualizaban para reflejar los cambios realizados en los modelos posteriores o en el código final.

Con la llegada de MDE, esta situación ha cambiado drásticamente. Los modelos pasan a ocupar un rol principal guiando el proceso de desarrollo,

mediante la definición de modelos precisos, que captan todos los requisitos y especificaciones sobre el sistema a construir así como la plataforma que se usará para implementarlo. La idea principal es ir generando una serie de modelos que permitan representar el sistema, cada vez con menor nivel de abstracción. Así, el nivel de detalle de los modelos obtenidos en las últimas fases del proceso permitirá generar (semi-)automáticamente el código que implementa el sistema.

El eslabón que une cada nuevo paso del proceso (cada generación de un nuevo modelo) es una **transformación de modelos**. El propósito principal de las transformaciones de modelos es convertir un modelo (o varios) del sistema, en otro modelo (o varios). Estas transformaciones deberían hacerse de forma (semi)-automática, implementándose mediante la definición de reglas de transformación (*mappings*) entre los modelos.

Surgen así nuevos lenguajes y herramientas [1, 2, 5, 6, 11, 22, 41, 45, 47, 83, 85, 89, 94, 106, 121], que facilitan la automatización de la transformación. Estos lenguajes y herramientas difieren en múltiples aspectos, tales como: paradigma (declarativo, imperativo o híbrido); grado de generalidad (de propósito general o diseñados para dominios específicos); nivel de abstracción.

Esta diversidad de tecnología trae aparejada una serie de problemas:

1. Por una parte, el usuario debe ser capaz de seleccionar el lenguaje más adecuado para resolver un problema en particular y, dependiendo del problema, puede ser conveniente utilizar uno u otro tipo de lenguaje. Si se desea cambiar de lenguaje, debe aprender a utilizarlo y el tiempo invertido en este proceso es directamente proporcional a la complejidad del lenguaje, lo que hace que el proceso de implementación de una transformación sea una actividad muy laboriosa.
2. En general las herramientas de transformación soportan un lenguaje de transformación específico, por lo que existen problemas de interoperabilidad entre las mismas.

Por todo ello, se considera conveniente buscar soluciones que permitan facilitar el aprendizaje y uso de los lenguajes de transformaciones, así como mejorar la interoperabilidad entre las herramientas de soporte, de modo que se facilite también la migración de un lenguaje a otro. Dado que estamos en un contexto de MDE, parece lógico tratar de aprovechar las ventajas que la propia ingeniería dirigida por modelos nos proporciona, y aplicar MDE al proceso de definición de las transformaciones. Para ello, en esta tesis doctoral, se especifica un entorno de desarrollo en el cual las transformaciones se definan en un lenguaje independiente de plataforma (a nivel PIM), con transformaciones que permitan la

generación (semi-)automática de los modelos de transformación correspondientes a las diferentes aproximaciones de transformación existentes (nivel PSM) y posteriormente los modelos de transformación conformes a los diferentes lenguajes específicos (nivel PDM).

No existe una única propuesta que permita unificar los lenguajes de transformación existentes, similar a MOF (*Meta Object Facility*, [149]), para la especificación de lenguajes en el ámbito de MDA. Desde este punto de vista, la propuesta de entorno de desarrollo que se realiza en esta tesis, permitirá realizar la definición de transformaciones de modelos a un alto nivel de abstracción, sin tener en cuenta el lenguaje de implementación final de la transformación. Con ello, se pretende resolver los dos problemas mencionados anteriormente:

1. Proporcionando un lenguaje de transformación de alto nivel, independiente de plataforma, y más cercano al usuario.
2. Proporcionando la generación (semi-)automática de transformaciones en lenguajes específicos dependientes de plataforma, lo que facilitará tanto la tarea de desarrollo del programador de transformaciones, como la interoperabilidad y migración entre herramientas y lenguajes.

El entorno de desarrollo de transformaciones de modelos, dirigido por modelos denominado MeTAGeM deberá incluir:

- La definición de un proceso metodológico para el desarrollo MDE de transformaciones de modelos.
- La especificación de un meta-modelo de transformaciones de alto nivel que permita modelar las transformaciones a nivel PIM.
- La especificación de meta-modelos conformes a las diferentes aproximaciones de transformación (declarativa, imperativa, híbrida, basada en grafo, etc) que permita modelar las transformaciones a nivel PSM.
- La especificación de un meta-modelo de transformaciones que incluya: las transformaciones entre el meta-modelo PIM y los diferentes meta-modelos PSM, y las transformaciones entre los meta-modelos de nivel PSM y los meta-modelos de nivel PDM.
- La implementación de una herramienta que soporte: a) el modelado de transformaciones de nivel PIM en base al meta-modelo de transformaciones propuesto; b) el modelado de transformaciones de nivel PSM en base a los meta-modelos conformes a cada una de las aproximaciones existentes; c) el modelado de las transformaciones a nivel PDM en base a los meta-modelos de cada uno de los lenguajes de transformación que se soporte; d) un meta-

transformador que permita obtener los modelos de las transformaciones conformes a un lenguaje de transformación en concreto y a partir de éstos últimos el código implementable de la transformación en cada uno de los lenguajes.

En esta tesis, nos centramos a nivel PSM, en la aproximación híbrida, por lo que a nivel PDM se seleccionan los lenguajes ATL y RubyTL que siguen dicha aproximación.

1.2 Hipótesis y Objetivos

La **hipótesis** planteada en esta Tesis Doctoral es que "*es factible la definición de un entorno que, aplicando los principios del MDE, facilite el desarrollo (semi-)automático de transformaciones de modelos mediante la especificación de las mismas en un lenguaje de alto nivel independiente de la plataforma y su posterior transformación a un lenguaje dependiente de la plataforma.*

El **objetivo principal** de este trabajo de investigación, derivado directamente de la hipótesis, es: *"la definición de un entorno que, aplicando los principios del MDE, facilite el desarrollo (semi-)automático de transformaciones de modelos mediante la especificación de las mismas en un lenguaje de alto nivel independiente de la plataforma y su posterior transformación a un lenguaje dependiente de la plataforma. Dicho entorno deberá incluir una guía metodológica y una herramienta de soporte."*

Para la consecución de este objetivo se han planteado los siguientes objetivos parciales:

O1. Estudio de trabajos previos:

- **O1.1.** Análisis y evaluación de trabajos relacionados con el desarrollo de transformaciones de modelos en el ámbito del MDE, incluyendo propuestas metodológicas y herramientas.
- **O1.2.** Análisis y evaluación de lenguajes y herramientas de transformación de modelos.

O2. Especificación de los meta-modelos de los diferentes niveles:

- **O2.1.** Especificación de un meta-modelo para la definición de modelos de transformaciones a nivel PIM.
- **O2.2.** Especificación de meta-modelos para la definición de modelos de transformación a nivel PSM. A nivel PSM se realizará el

modelado de las transformaciones teniendo en cuenta la aproximación seleccionada por el desarrollador, esto es, imperativa, declarativa, híbrida, grafos, etc. En particular para el desarrollo de esta tesis se selecciona la aproximación híbrida.

O2.3. Especificación de los meta-modelos para la definición de modelos de transformación a nivel PDM. En el caso de esta tesis, como lenguajes híbridos a nivel PDM se seleccionan ATL y RubyTL. En el caso de ATL se propone el uso del meta-modelo definido por el propio lenguaje; en el caso de RubyTL, como no tiene el meta-modelo definido, es necesario especificarlo.

O3. Especificación de meta-modelos que permitan automatizar las transformaciones de modelos:

O3.1. Especificación de las transformaciones de modelos de transformaciones de nivel PIM a modelos de nivel PSM.

O3.2. Especificación de las transformaciones de modelos de transformación de nivel PSM a modelos de nivel PDM, es decir, modelos dependientes de un lenguaje de transformación determinado: ATL y RubyTL.

O4. Construcción de la meta-herramienta:

O4.1. Especificación de la arquitectura de la meta-herramienta de soporte.

O4.2. Implementación de los meta-modelos definidos (PIM, PSM y PDM).

O4.3. Implementación del meta-editor para el modelado en forma gráfica transformaciones de alto nivel.

O4.4. Implementación del meta-transformador que deberá incluir:

- El conjunto de reglas de transformación de modelos definidos a nivel PIM a modelos definidos a nivel PSM.
- El conjunto de reglas de transformación de modelos definidos a nivel PSM a modelos definidos a nivel PDM.
- El conjunto de reglas de transformación de modelo a texto, que permitirán obtener el código para un lenguaje concreto (ATL y RubyTL).

O4.5. Integración de la funcionalidad proporcionada como resultado de las tareas anteriores. Para ello, se desarrollarán nuevos módulos

que proporcionen una interfaz visual que facilite la ejecución de las transformaciones.

O5. Validación del entorno de desarrollo propuesto.

O5.1. Para ello se desarrollará un meta-caso de estudio que permitirá validar, tanto la propuesta metodológica, como el correcto funcionamiento de la herramienta.

O5.2. Además, se desarrollará un caso de estudio que consistirá en probar las transformaciones de modelos generadas en el meta-caso de estudio.

1.3 Marco de Investigación

En los últimos años, el grupo de investigación Kybele, al cual pertenece la doctoranda, ha venido trabajando en la definición de MIDAS [69, 70, 195, 202] una metodología centrada en la arquitectura para el desarrollo dirigido por modelos de software. En el marco de MIDAS, se han definido distintos métodos como: SOD-M [68] una aproximación metodológica basada en MDA para el desarrollo orientado a servicios de Sistemas de Información (SI), PISA [3] una arquitectura de integración de portales Web basados en servicios web semánticos, y por último, MIDAS MDA Tool (M2DAT) [191] la herramienta MDA que soporta cada uno de los métodos propuestos en MIDAS.

En la Figura 1-1 se puede ver la arquitectura de MIDAS. Esta arquitectura se basa en los principios de MDA [148] y se define sobre una base multidimensional que se propaga a través de varios niveles de abstracción y los diferentes aspectos del desarrollo del sistema. En cada una de las dimensiones o aspectos se definen una serie de modelos y reglas de transformación entre dichos modelos, además se establecen las relaciones existentes entre los mismos:

- **Dimensión vertical.** Esta dimensión esta directamente alineada con la propuesta de MDA, se definen tres niveles de acuerdo al grado de abstracción de los modelos que representan: CIM, PIM y PSM. De esta manera, los conceptos asociados con el dominio del problema se especifican en los modelos CIM, a nivel PIM se define la funcionalidad del SI sin tener en cuenta los detalles de implementación y, por último, la representación del sistema, teniendo en cuenta las características de la plataforma, en la que se van a implementar, se especifican en los modelos PSM. Para pasar de un modelo a otro se definen transformaciones de modelos.

- **Núcleo de la arquitectura de modelos.** En MIDAS la arquitectura guía el proceso de desarrollo [135], por lo que forma parte del núcleo de MIDAS. De hecho, la arquitectura especifica las características que afectan a todos los aspectos del sistema.
- **Capa interior concéntrica.** En esta capa se organizan los modelos de acuerdo a los principales aspectos del desarrollo de un SI, contenido, hipertexto y comportamiento.

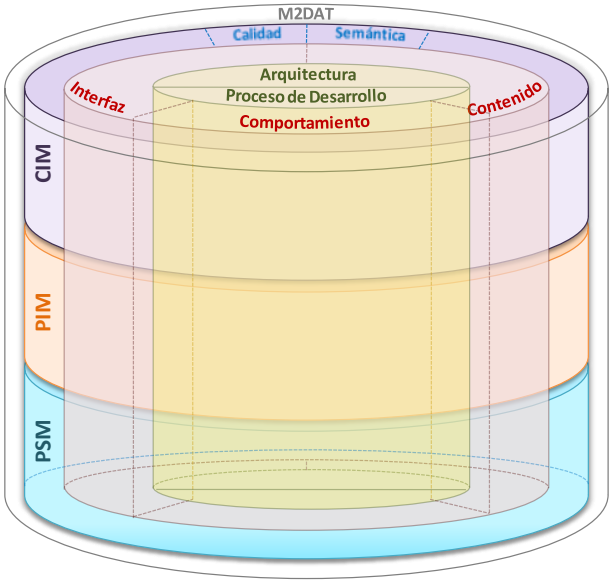

Figura 1-1. Arquitectura de MIDAS

- **Capa externa.** Existen otros aspectos en el desarrollo de los SI, tales como la semántica, la seguridad o la calidad, que son ortogonales a los anteriormente mencionados (contenido, hipertexto, comportamiento). Por ello, se los incluye en otra capa concéntrica del cilindro.
- **Herramienta M2DAT.** Finalmente, la última capa del cilindro hace referencia a la herramienta que soporta MIDAS (M2DAT).

En este contexto, el objetivo de este trabajo de tesis doctoral es la creación de una meta-herramienta que automatice la generación de los módulos de transformación de M2DAT.

1.3.1 Proyectos de Investigación y Estancias

La investigación realizada en esta tesis doctoral se ha llevado a cabo en el Grupo de Investigación Kybele de la Universidad Rey Juan Carlos (URJC). Como puede verse en la Figura 1-2, este trabajo se enmarca dentro de varios proyectos de investigación. Además, parte de la investigación ha sido realizada en dos estancias predoctorales: la primera, una estancia de cuatro meses realizada en la Universidad de La Laguna, en Tenerife (España) y la segunda, una estancia de tres meses realizada en la Universidad Roma Tre, en Roma (Italia). A continuación se explicará brevemente el objetivo de cada uno de los proyectos y los resultados obtenidos en las estancias realizadas.

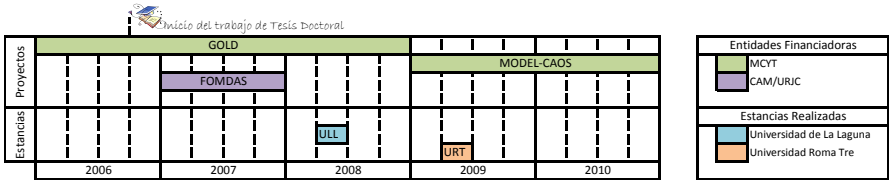

Figura 1-2. Marco de Trabajo de la Tesis Doctoral

1.3.1.1 Proyectos de Investigación

Esta tesis se ha realizado fundamentalmente en el marco de dos proyectos de investigación GOLD y MODEL-CAOS.

GOLD [TIN2005-00010] es un proyecto financiado por el Ministerio de Educación y Ciencia. El principal objetivo de GOLD era, la definición de una plataforma para el desarrollo de Sistemas de Información Web (SIW). La doctoranda ha estado involucrada en actividades relacionadas con proveer una solución técnica, extensible, e interoperable, para construir dicha plataforma. En particular, ha trabajado en la definición e implementación de las transformaciones entre los diferentes modelos propuestos en la plataforma.

Así mismo, la doctoranda ha estado involucrada en el proyecto **FOMDAS** [URJC-CM-2006-CET-0387] co-financiado por la Universidad Rey Juan Carlos y la Comunidad de Madrid: cuyo objetivo era la formalización de los meta-modelos y las transformaciones de modelos especificadas en GOLD.

MODEL-CAOS [TIN2008-03582], financiado por el Ministerio de Ciencia e Innovación, ha comenzado en el año 2009 y tiene una duración de tres años. El objetivo principal de MODEL-CAOS es la especificación de un marco para el desarrollo de manera (semi-)automática de SI, centrándose en la utilización del paradigma de Orientación a Servicios. En el marco de dicho proyecto se propone el desarrollo de una meta-herramienta que permita automatizar las tareas

del DSDM. Este proyecto toma como base los trabajos realizados en los proyectos anteriores, actualizándolos mediante la inclusión de las últimas tendencias en el desarrollo de SI: desarrollo dirigido por modelos, orientación a servicios, etc, poniendo especial énfasis en el aspecto arquitectónico como elemento central que guía el proceso metodológico. En este contexto, la doctoranda está trabajando en el desarrollo de una guía metodológica, que facilite el desarrollo (semi-)automático de transformaciones de modelos.

1.3.1.2 Estancias

Las estancias realizadas por la doctoranda han tenido como objetivo realizar parte del estudio presentado en el estado del arte de esta tesis. A continuación se presenta una breve descripción de cada una de ellas.

Universidad de La Laguna. La estancia se realizó en el grupo de investigación Taro de la Universidad de La Laguna (Tenerife, España) dirigido por el profesor Dr. José Luis Rodas García y tuvo una duración de cuatro meses.

El grupo TARO ha desarrollado un lenguaje de transformación llamado *Atomic Transformation Code* (ATC) [173]. Dicho lenguaje se concibió para formalizar transformaciones descritas en un bajo nivel de abstracción, con el objetivo final de dar soporte a múltiples lenguajes de mayor nivel de abstracción, como el estándar *Query/View/Transformation* (QVT) [158] de la OMG.

En este contexto, la doctoranda ha definido un conjunto de transformaciones, para el modelado de las bases de datos objeto-relacionales, en QVT, utilizando la infraestructura desarrollada por el mismo grupo de investigación para ejecutarlo sobre ATC. Para ello, se ha utilizado el editor del lenguaje QVT *Operational Mappings* (QVTo) [175] desarrollado también por el grupo de investigación en cuestión. La infraestructura de ejecución se encarga de procesar la información introducida en el editor y convertirla a instrucciones compatibles en el entorno ATC, permitiendo de esta manera ejecutar finalmente las transformaciones deseadas. El propósito de esta especificación de transformaciones ha sido la generación automática de modelos en la herramienta M2DAT. Desde el punto de vista del centro receptor, como resultado de la estancia se ha contribuido al refinamiento del soporte al lenguaje QVTo, así como también al refinado de su editor.

Además, se han llevado a cabo actividades relativas a los editores gráficos definidos con GMF (*Graphical Modeling Framework*) [80]. El objetivo era definir una infraestructura para generar, sobre GMF, editores gráficos para DSL (*Domain Specific Languages*, [142]) que reutilizan la notación gráfica UML. Con ello se

buscaba facilitar el uso de la tecnología GMF cuando deseamos generar editores que usan notación UML.

Para la elaboración de dicha infraestructura se han llevado a cabo las siguientes sub-tareas: en primer lugar se definió un DSL para especificar correspondencias entre la notación (sintaxis concreta) que se quiere para el dominio a representar y UML; en segundo lugar se definió un generador que permite transformar modelos, descritos en términos del DSL definido en modelos del domino de GMF (*gmfgraph*, *gmftool* y *gmfmap*). Las transformaciones que constituyen dicho generador se especificaron en QVT.

Universidad Roma Tre. La estancia se realizó en el Grupo de Bases de Datos de la Universidad Roma Tre de Roma (Italia) dirigido por el profesor Dr. D. Paolo Atzeni. Las principales líneas de investigación de dicho grupo giran en torno al campo de las Bases de Datos (BD). El grupo cuenta con un *framework* llamado MIDST (*Model Independent Data and Schema Translation*) [16], que permite realizar el modelado de esquemas de bases de datos de forma independiente de la plataforma de implementación final. MIDST se basa en la observación fundamental de que cualquier modelo de datos existente se puede representar con un conjunto finito de construcciones. MIDST se ha concebido para dar soporte a MODELGEN [17], un operador propuesto también por el grupo de investigación que establece el conjunto de reglas de transformación entre diferentes esquemas de BD de acuerdo a la tecnología seleccionada por el usuario final. El proceso de transformación propuesto por MODELGEN se puede dividir en dos fases: por un lado la elección de la regla apropiada para realizar la transformación y, por otro lado, la ejecución de dicha regla de transformación.

MIDST está desarrollado en Eclipse y las reglas de transformación de MODELGEN se han especificado en *Datalog*. Estas reglas de transformación permiten realizar las transformaciones entre diferentes tecnologías de BD [18, 19], por ejemplo, BD Entidad Relación (ER) a Relacionales o ER a BD Objeto-Relacionales (OR), entre otras.

Durante la estancia se han implementado las reglas de transformación definidas en MODELGEN por el grupo de Bases de Datos de la Universidad Roma Tre, utilizando la herramienta M2DAT-DB definida por el grupo Kybele.

En MODELGEN se propone la implementación de las transformaciones entre diferentes esquemas de BD por medio de la aplicación de operadores. Esto permite definir las transformaciones de manera atómica, de modo que cada transformación pueda aplicarse de forma (semi-)independiente del resto de las transformaciones. El desarrollador debe seleccionar la regla (o el conjunto de

reglas) que desea ejecutar para realizar una transformación. Como se ha dicho anteriormente, MODELGEN permite realizar la transformación entre diferentes plataformas de implementación de BD. En particular, en la estancia, la doctoranda se ha centrado en las transformaciones entre modelos ER y modelos OR.

Para realizar las tareas mencionadas fue necesario implementar el meta-modelo ER en la herramienta M2DAT-DB y desarrollar el editor de modelos correspondiente. Posteriormente, se procedió con la implementación de cada una de las reglas de transformación propuestas por MODELGEN. Para ello se utilizó el lenguaje de transformación ATL [101] y el *plug-in* para Eclipse *ATLFlow* (http://opensource.urszeidler.de/ATLflow/) que permite definir de manera gráfica el proceso de la transformación de modelos, pudiendo de este modo orquestar la ejecución de cada regla de transformación. De este modo, el usuario final puede, por medio de *ATLFlow*, decidir las reglas de transformación que se utilizarán e indicar el orden de ejecución de las mismas. Así, a partir de un mismo modelo de entrada, se pueden obtener diferentes modelos de salida dependiendo de las reglas de transformación que se decidan aplicar.

1.4 Método de Investigación

La diferente naturaleza de las Ingenierías, respecto al resto de las disciplinas empíricas y formales, dificulta la aplicación directa de los métodos de investigación clásicos [109] y en particular, a la investigación en Ingeniería de Software.

El método de investigación que se sigue en esta tesis es la adaptación del propuesto por Marcos y Marcos en [134] para la investigación en Ingeniería del Software. Dicho método se basa en el hipotético-deductivo propuesto por Bunge [51] y se compone de una serie de pasos (ver Figura 1-3) lo suficientemente generales como para ser aplicado a cualquier tipo de investigación.

En el capítulo 2 se introducirán las bases teóricas en las se enmarca esta tesis (cuerpo del conocimiento), a partir de las cuales se ha realizado la definición del problema que se aborda en esta tesis. Para ello, se ha definido la hípotesis, como punto de partida de esta investigación, así como el conjunto de objetivos que nos hemos marcado para alcanzar la solución (Sección 1.2).

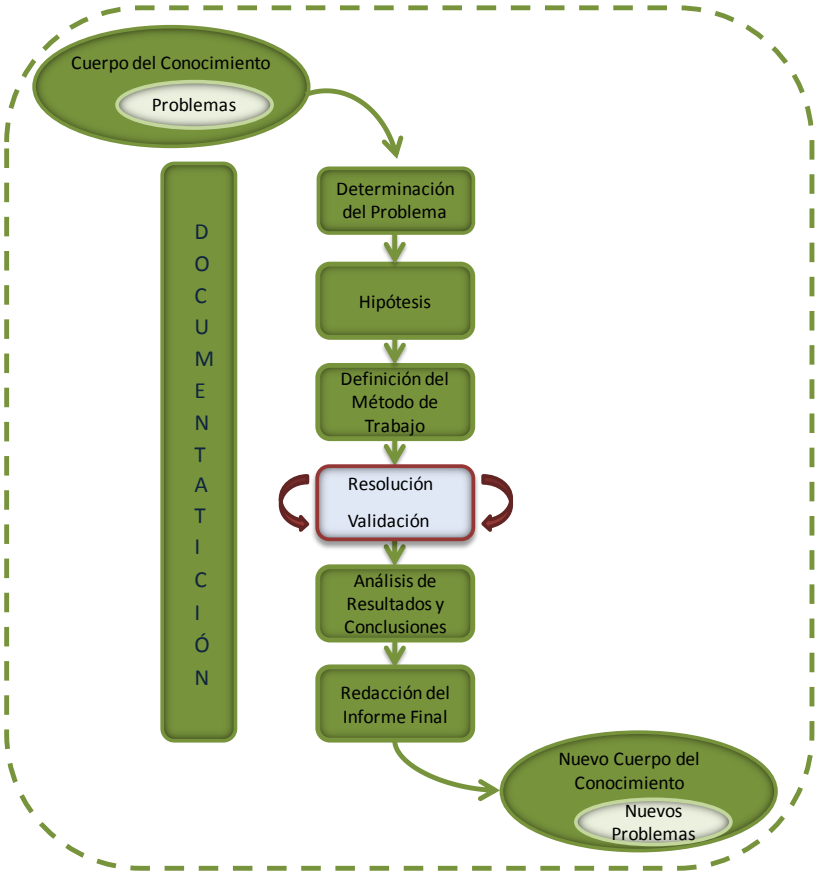

Figura 1-3. Método de Investigación

Como se puede ver en la Figura 1-3 la definición del método de investigación es un paso más del mismo método. Los autores [134] recomiendan definirlo de esta manera, ya que la naturaleza de cada investigación tiene sus propias características y, por lo tanto, no sería conveniente aplicar un único método universal de investigación.

A continuación, se resume el método específico usado en esta tesis para las etapas de documentación y resolución y validación.

1.4.1 Etapa de Documentación

Para el desarrollo de esta etapa se ha utilizado el método de revisiones sistemáticas propuesto en [111]. Este método sirve para identificar, evaluar e interpretar toda la información relativa a un tema de investigación en particular, de

un modo sistemático y replicable. Surge de la investigación en el campo de la medicina, por lo que, según sus autores [111], se ha convertido en una metodología confiable, rigurosa y auditable.

La aplicación de las revisiones sistemáticas en el ámbito de la Ingeniería de Software permite dar un valor científico a la revisión de la literatura que se hace, definir una estrategia de búsqueda de la literatura a evaluar y obtener finalmente hipótesis a favor o en contra de dicha literatura. Para el desarrollo de este trabajo de investigación se ha tomado como referencia la adaptación del método de revisiones sistemáticas para Ingeniería de Software presentado en [40]. En este trabajo se propone una nueva aproximación, en la cual el proceso de las revisiones sistemáticas está compuesto por cuatro grandes fases: *planificación*, *ejecución*, *análisis de los resultados* y *resguardo de los resultados* obtenidos (Figura 1-4).

Figura 1-4. Proceso de Método de Revisiones Sistemáticas

En la fase de **planificación** se debe establecer claramente el objetivo de la investigación y definir el protocolo de revisión a utilizar. Es decir, definir un protocolo para cada objeto a ser investigado, estableciendo el método que será utilizado a lo largo de la realización de la revisión. Además se deben identificar los criterios de inclusión y exclusión que se seguirán para determinar las fuentes de investigación y los estudios (o documentos) a seleccionar.

En la fase de **ejecución** se lleva a cabo lo planificado en la etapa anterior, por lo que en primer lugar, se debe determinar el conjunto de estudios a evaluar. Estos estudios se seleccionan a través de la evaluación, para cada uno de ellos, de los criterios de inclusión y exclusión determinados anteriormente.

En la fase de **análisis de resultados** se debe sintetizar y evaluar la información extraída de cada estudio.

Por último, se debe mencionar que la fase de **resguardo de resultados** se realiza durante todo el proceso de la revisión sistemática, ya que a medida que se ejecutan cada una de las fases, el resultado de las mismas debe ser almacenado.

Como se puede ver en la Figura 1-4, existen puntos de verificación a lo largo del proceso. El primer punto garantiza que la planificación realizada es la

adecuada; para ello, se debe evaluar el protocolo definido y, si hubiera algún problema o incongruencia, se debería volver a la fase anterior, la planificación. El segundo punto de verificación debe realizarse una vez acabada la fase de ejecución; de la misma manera que en el punto anterior, si hubiera algún error en los resultados de esta etapa se debería volver a realizar la ejecución.

Para realizar la revisión sistemática, se han definido una serie de plantillas, basadas en la aproximación presentada anteriormente, con tareas a realizar para cada una de las fases establecidas; en el anexo B se pueden ver dichas plantillas. En el capítulo 2 se presentaran con mayor detalle las revisiones sistemáticas para realizar el estado del arte de esta tesis.

1.4.2 Etapa de Resolución y Validación

El método de resolución y validación seguido en esta tesis es la adaptación, propuesta en [191], de dos métodos conocidos en el campo de la Ingeniería del Software: el método en cascada tradicional [171] y el Proceso Unificado de *Rational* [43], tomando como base la definición de etapas consecutivas del primero y el proceso iterativo del segundo. La elección de estos métodos se basa en la similitud que existe entre la naturaleza del problema a resolver y los problemas que surgen en el desarrollo de software. Existen ciertos problemas de investigación en Ingeniería del Software (como el que se presenta en esta tesis) que tienen en sí mismo una naturaleza ingenieril, ya que se trata de la construcción de nuevos objetos [133]; en el caso que nos ocupa, se trata de construir un entorno para el desarrollo (semi-)automático de transformaciones. Dicho entorno estará compuesto por una metodología y una herramienta. Un método de desarrollo de software da las pautas para la construcción de nuevos objetos (de software); por ello, los métodos de desarrollo de software nos pueden servir de base para la resolución de los problemas de investigación en Ingeniería del Software con carácter ingenieril [133].

Como se puede ver en la Figura 1-5, las diferentes acitividades que se llevan a cabo en la etapa de **resolución** y **validación** están agrupadas en tres grandes bloques: el bloque correspondiente a la especificación de la **metodología**, el bloque correspondiente al desarrollo de la **herramienta** y el bloque correspondiente a las **pruebas** realizadas.

Dentro del bloque correspondiente a la metodología, la primera actividad que se realiza es la de la **especificación del proceso** (MeTAGeM) para la construcción de modelos de transformación a un alto nivel de abstracción. La definición de este proceso se realiza teniendo en cuenta: a) los resultados

obtenidos de los estudios de las aproximaciones metodológicas existentes, así como de los lenguajes y herramientas de transformación de modelos realizados; b) nuestra propia experiencia en la definición de transformaciones de modelos en diferentes dominios [4, 42, 71, 72, 73, 191, 193, 194, 196, 197, 198].

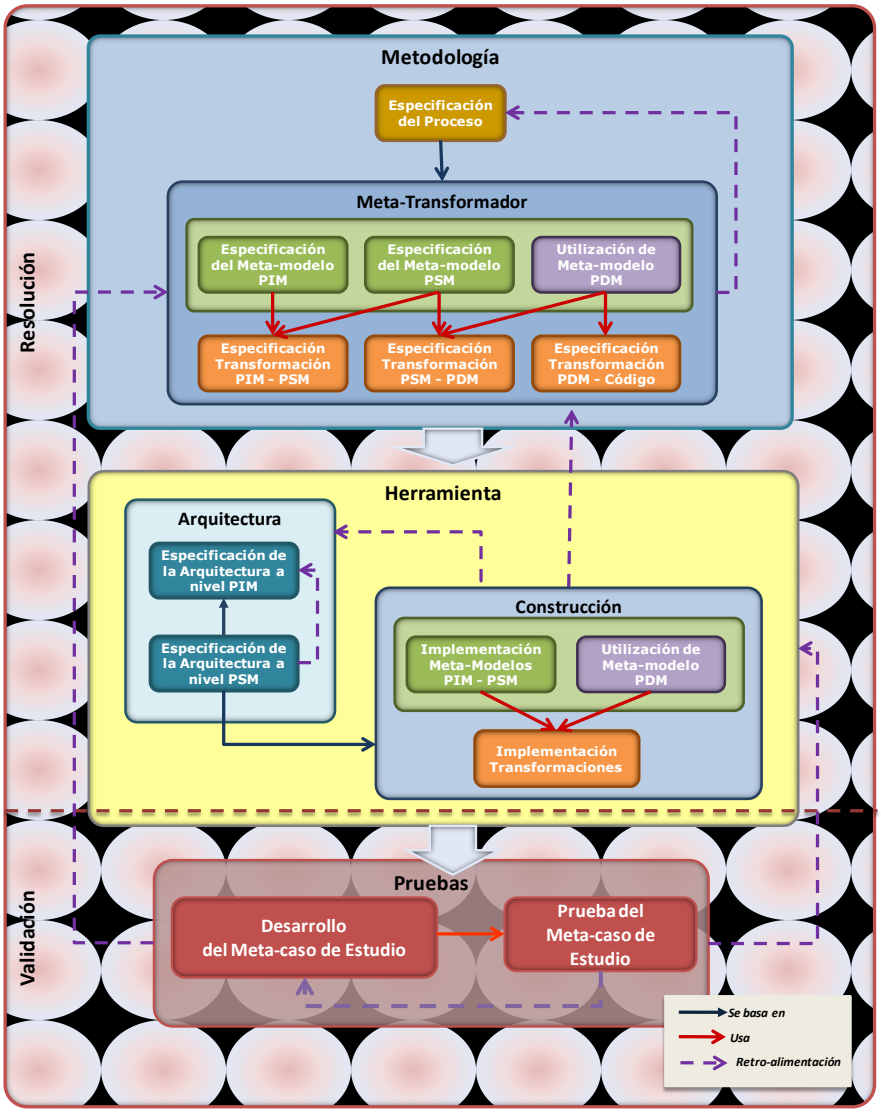

Figura 1-5. Fase de Resolución y Validación del Método de Investigación

Una vez realizada la especificación del proceso se realiza la especificación del **meta-transformador**. Dicho meta-transformador estará formado por dos

componentes: por un lado, el **conjunto de meta-modelos** que permitirá realizar el modelado de las transformaciones de modelos en los diferentes niveles propuestos. Y, por otro lado, el **conjunto de reglas de transformación** que permitirá realizar la transformación entre los modelos conformes a dichos meta-modelos.

De acuerdo al proceso definido, se deberán especificar los meta-modelos correspondientes a cada uno de los niveles de MeTAGeM (PIM, PSM y PDM). A nivel PIM se especificará un meta-modelo en un alto nivel de abstracción. A nivel PSM se especificará el meta-modelo de la aproximación híbrida. Por último, a nivel PDM, se utilizarán los meta-modelos de cada uno de los lenguajes de transformación existentes, en el caso de esta tesis se utilizará el meta-modelo del lenguaje de transformación ATL y se especificará el meta-modelo del lenguaje de transformación RubyTL.

A partir de la definición de los meta-modelos se realiza la especificación de las transformaciones entre elementos de dichos meta-modelos. Así como resultado de esta sub-etapa se obtienen: a) las transformaciones entre modelos definidos a nivel PIM y modelos definidos a nivel PSM; b) las transformaciones entre modelos definidos a nivel PSM y modelos definidos a nivel PDM, y por último, c) las transformaciones, de modelo a texto, que permiten obtener el código implementable de la transformación a partir de los modelos definidos a nivel PDM.

Dentro del bloque correspondiente a la herramienta se realiza, por un lado, la especificación e implementación de la **arquitectura** que soporte el proceso propuesto y, por otro lado, la **construcción** de los diferentes meta-modelos y las reglas de transformación definidas en el bloque de la metodología.

Al estar en el ámbito de MDE, lo más lógico parece ser aplicar MDE, a la definición y construcción de cada uno de los elementos definidos en el proceso metodológico. Y teniendo en cuenta, además, las lecciones aprendidas en la construcción de M2DAT [191], se decide realizar la especificación de la **arquitectura** separando, la definición a un nivel conceptual (nivel PIM), de los detalles técnicos de su implementación a bajo nivel (nivel PSM).

En la definición de la arquitectura a nivel PIM se especifican los módulos o componentes que formarán parte de la meta-herramienta. Posteriormente, en la definición de la arquitectura a nivel PSM se determinarán las tecnologías que serán utilizadas para implementar cada uno de dichos módulos o componentes. Además, se define un punto de retro-alimentación que permite realizar modificaciones en la definición de la arquitectura de nivel PIM.

En la actividad de **construcción**, se realiza la implementación de la herramienta MeTAGeM, siguiendo la especificación del diseño de la arquitectura, la especificación de los meta-modelos y las reglas de transformación realizadas en las actividades anteriores. De esta manera, a partir de los meta-modelos especificados previamente se desarrollarán editores que permitan manipular modelos conformes a dichos meta-modelos.

Por último, durante la fase de **pruebas**, se desarrollará un meta-caso de estudio que permitirán validar el funcionamiento del entorno de desarrollo, es decir, tanto de la metodología definida como de la herramienta que la implementa. Después, las transformaciones obtenidas a partir del meta-caso de estudio serán posteriormente validadas con modelos conformes a los meta-modelos sobre los que se ha implementado dicha transformación.

Cabe mencionar, que al definir el proceso como iterativo, cada una de las fases sirve como punto de control de las fases anteriores, pudiendo volver a las mismas en caso de ser necesario. En los siguientes capítulos se mostrará con detalle cada uno de los elementos definidos.

1.5 Estructura de la Tesis

El resto de los capítulos de esta tesis, se organizan de la siguiente manera:

- El **Capítulo 2** proporciona una visión completa sobre el estado del arte. Para esto, en la sección 2.1 se realiza una revisión de las propuestas existentes para el desarrollo dirigido por modelos de transformaciones de modelos. Mientras que en la sección 2.2 se realiza una revisión detallada de los lenguajes de transformación de modelos existentes y de las herramientas que los implementan.
- El **Capítulo 3** presenta la solución propuesta en esta tesis: el entorno de desarrollo de transformaciones de modelos, MeTAGeM, que, aplicando los principios del MDE, facilita el desarrollo (semi-)automático de transformaciones de modelos. Para esto en la sección 3.1 se define la metodología compuesta por: un proceso de desarrollo de transformaciones dirigido por modelos, diferentes meta-modelos que dan soporte al proceso y la especificación del conjunto de reglas de transformación que permitan el modelado de las transformaciones de modelos de forma (semi-) automática. En la sección 3.2 se presenta la herramienta MeTAGeM que da soporte a la metodología propuesta. Dicha herramienta está formada por: una arquitectura, que define cada uno de los componentes de la herramienta; un conjunto de

editores, que permiten el modelado de las transformaciones de modelos conformes a los meta-modelos definidos por la metodología y un meta-transformador, que permite realizar las transformaciones entre los modelos definidos en los diferentes niveles de abstracción y la posterior generación de código.

- En el **Capítulo 4** se presenta la validación del entorno MeTAGeM. Por lo que, en la sección 4.1 se muestra el desarrollo de un meta-caso de estudio que consiste en la implementación de las transformaciones de modelos entre el meta-modelo de UML y el meta-modelo para el modelado de bases de datos Objeto-Relacionales. Posteriormente, en la sección 4.2 se muestra cómo las transformaciones implementadas se utilizan para transformar modelos específicos conformes a ambos meta-modelos.
- Por último, en el **Capítulo 5** se concluye con un resumen de las principales contribuciones de esta tesis. Para ello, se proporciona un análisis de los resultados obtenidos y las publicaciones realizadas, tanto en foros nacionales como internacionales. Además, se plantean una serie de cuestiones que sirven para futuras investigaciones y se proponen las directrices a seguir para llevar a cabo dicha investigación.
- Además, en el **apéndice A** se presenta el resumen extendido de la tesis en inglés. En el **apéndice B** se detalla brevemente las características del método de Revisiones Sistemáticas utilizado en la etapa de documentación de esta tesis, se presentan las plantillas utilizadas para llevar a cabo la revisión sistemática y se muestran los resultados de aplicar las revisiones sistemáticas en esta tesis. El **apéndice C** presenta el estudio realizado de dos propuestas publicadas en el año 2010 relevantes para el tema de investigación de esta tesis. En el **apéndice D** se muestra la implementación realizada de las reglas de transformaciones entre los diferentes niveles de abstracción de MeTAGeM. El **apéndice E** presenta el manual de usuario de MeTAGeM. En el **apéndice F** se presenta el caso de estudio completo desarrollado como validación de la propuesta realizada en esta tesis. Y, por último, en el **apéndice G** se presenta la especificación e implementación de los meta-modelos propuestos en los diferentes niveles de abstracción de MeTAGeM.

Estado del Arte

En este capítulo se presentan las revisiones sistemáticas que contribuyen al estado del arte de esta tesis.

En primer lugar, se realiza una revisión de las propuestas existentes para el desarrollo dirigido por modelos de transformaciones de modelos. La mayoría de las propuestas existentes abordan parcialmente este problema, ya que, como veremos en las conclusiones de esta sección, no existe ninguna propuesta que partiendo de una especificación de alto nivel sea capaz de generar transformaciones de modelos operativas (Sección 2.1).

En segundo lugar, se realiza una revisión de los lenguajes de transformación existentes, así como de las herramientas que los soportan, en el ámbito de MDE (Sección 2.2).

2.1 Propuestas para el Desarrollo MDE de Transformaciones

En esta sección se presentan las propuestas existentes para el desarrollo dirigido por modelos de transformaciones de modelos. En primer lugar se presenta el conjunto de características a evaluar en cada una de dichas propuestas. En segundo lugar, se presenta el proceso de revisión sistemática que se ha seguido para el estudio de las mismas. Por último, se muestran los resultados obtenidos tras la evaluación realizada.

2.1.1 Características a Evaluar

Para la revisión de las aproximaciones existentes que proponen el uso de MDE en el desarrollo de las transformaciones se han identificado un conjunto de características a ser evaluadas en cada una de ellas. A continuación se describen, brevemente, dichas características:

- **Soporte a Nivel PIM.** Se refiere a si la propuesta soporta la especificación de modelos de transformaciones de forma independiente de plataforma.
- **Soporte a Nivel PSM.** Es decir, si la propuesta soporta la especificación de modelos de transformación específicos de plataforma.
- **Soporte de Transformaciones de PIM a PSM.** Si la propuesta define *mappings* que permiten la transformación entre los modelos de transformación definidos a nivel PIM y los definidos a nivel PSM.

- **Soporte de Transformaciones de PSM a PSM.** Existen situaciones en las que es conveniente definir *mappings* entre modelos de un mismo nivel que permiten realizar refinamientos de los modelos, por ejemplo, convertir modelos conformes a una plataforma en modelos conformes a otra plataforma. Para ello se evalua si las propuestas analizadas soportan transformaciones de modelos de nivel PSM a modelos de nivel PSM.
- **Soporte al Modelado Gráfico de las transformaciones.** Uno de los principales inconvenientes, desde el punto de vista del desarrollador final, a la hora de especificar transformaciones de modelos es el hecho de que en la mayoría de los lenguajes de transformación las transformaciones de modelos se especifican de manera textual. Por ello, se considera conveniente evaluar si las propuestas metodológicas permiten especificar las transformaciones de modelos de forma gráfica, lo que ayuda a reducir la complejidad a la hora de implementar las transformaciones.
- **Soporte a la Generación de Código.** El último paso del proceso de modelado de transformaciones es obtener el código que implementa la transformación en un lenguaje de transformación de modelos específico. Por ello, es necesario determinar si las propuestas evaluadas soportan la generación de código.
- **Soporte a la Validación de Modelos.** El nuevo rol que tienen los modelos en MDE influye en la importancia de tener mecanismos que permitan realizar la validación de los mismos [61]. Esto es debido a que cualquier error en un modelo se transmite a través de los distintos modelos generados y al código que implementa dicho modelo. Los mecanismos de validación permiten detectar errores e inconsistencias en las primeras etapas de desarrollo y contribuyen a aumentar la calidad de los modelos construidos, y, en consecuencia el código generado a partir de ellos. Por ello, otra característica que deben cumplir las propuestas de MDE en transformaciones de modelos es la de soportar la validación de los modelos.

Las características definidas se evaluarán, para cada una de las aproximaciones existentes, de acuerdo a dos puntos de vistas: metodológico y técnico. Desde el punto de vista metodológico se evaluará el grado de cumplimiento de la característica. Desde el punto de vista técnico, se evaluará si existe alguna herramienta qué soporte dicha característica y en que grado lo hace.

En la Tabla 2-1 se muestran, a modo de resumen, las características a evaluar y el conjunto de valores posibles para cada una.

Tabla 2-1. Resumen de Características a Evaluar

Característica	Valor
Soporte a Nivel PIM: 0 - 3	0 – No contempla la especificación de modelos de transformación a nivel PIM 1 – Plantea la necesidad de realizar el modelado de transformaciones a nivel PIM 2 – Propone el uso de algún meta-modelo ya existente para el modelado de transformaciones a nivel PIM 3 – Especifica su propio meta-modelo para el modelado de transformaciones a nivel PIM
Soporte a Nivel PSM: 0 - 3	0 – No contempla la especificación de modelos de transformación a nivel PSM 1 – Plantea la necesidad de realizar el modelado de las transformaciones a nivel PSM 2 – Propone el uso de algún meta-modelo ya existente para el modelado de transformaciones a nivel PSM 3 – Especifica su propio meta-modelo para el modelado de transformaciones a nivel PSM
Soporte de Transformaciones de PIM – PSM: 0 – 2	0 – No contempla la especificación de transformaciones de PIM a PSM 1 – Plantea la necesidad de realizar el modelado de las transformaciones entre modelos PIM y PSM 2 – Especifica un conjunto de *mappings* entre modelos PIM y PSM
Soporte de Transformaciones de PSM – PSM: 0 – 2	0 – No contempla la especificación de transformaciones a nivel PSM 1 – Plantea la necesidad de realizar el modelado de las transformaciones entre modelos del nivel PSM 2 – Especifica un conjunto de *mappings* entre modelos del nivel PSM
Soporte al Modelado gráfico de las transformaciones: 0 - 3	0 – No contempla modelado gráfico de transformaciones 1 – Plantea la necesidad de modelado gráfico de transformaciones 2 – Propone el uso de herramientas ya existentes para el modelado gráfico de transformaciones 3 – Especifica su propio editor para el modelado gráfico de transformaciones
Soporte a la Generación de Código: 0 - 2	0 – No contempla la generación de código 1 – Plantea la necesidad de realizar generación de código 2 – Define *mappings*, o algún mecanismo, que permiten obtener el código a partir de los modelos
Soporte a la Validación de Modelos: 0- 3	0 – No contempla validación de modelos 1 – Plantea la necesidad de validación de los modelos 2 – Propone el uso de mecanismos de validación ya existentes 3 – Especifica sus propios mecanismos de validación

A continuación, se resumen los trabajos encontrados en la literatura. En la sección 2.1.3 se presenta una tabla comparativa de dichos trabajos, así como las principales conclusiones extraídas del análisis realizado.

2.1.2 Revisión Sistemática

En esta sección se presenta, de manera resumida, la revisión sistemática llevada a cabo para realizar esta sección del estado del arte; para consultar la revisión completa ver anexo B.

Etapa de planificación de la revisión:

Objetivo (*Question*): el principal objetivo de esta revisión sistemática es realizar un estudio de las aproximaciones existentes que propongan el uso de MDE para el desarrollo de las transformaciones de modelos y, si las hubiera, de las herramientas que las implementan.

Palabras Claves: las palabras clave que se utilizan para realizar la búsqueda de las aproximaciones son: *transformations*, *meta-transformations*, *high order transformations*, *models*, *tools*, MDA y MDE.

El resultado esperado tras la realización de esta revisión sistemática es obtener el conjunto de las aproximaciones que propongan el uso de MDE para el desarrollo de las transformaciones de modelos. Dichas aproximaciones se analizarán y estudiarán para determinar el alcance y grado de aplicación de las mismas.

Selección de fuentes de búsqueda: en la Tabla 2-2 se muestran las diferentes cadenas de búsqueda que se han utilizado a lo largo de la revisión sistemática, estas cadenas de búsqueda se obtienen realizando una combinación de las palabras claves identificadas.

Tabla 2-2. Cadenas Básicas de Búsqueda

Cadenas básicas de búsqueda	
1	Meta-transformation AND High Order Transformation
2	MDA AND Meta-transformation
3	MDE AND Meta-transformation
4	Meta-transformation AND Tool
5	Transformation Model

Las fuentes iniciales seleccionadas para la realización de las búsquedas son:

- Science@Direct

- IEEE Digital Library
- SpringerLink
- ACM Digital Library
- Journal of Computer Science
- Actas de congresos y workshops en el ámbito de MDE

Etapa de ejecución de la revisión

Selección de Estudios: en el momento de la ejecución de la revisión sistemática, se han ido obteniendo una serie de documentos sobre los que se han aplicado criterios de inclusión y exclusión, para determinar cuales deberían ser incluidos en el proceso de análisis.

El criterio de inclusión que se ha utilizado es realizar un análisis del título, el resumen y las palabras clave de los documentos obtenidos en la búsqueda. De esta manera, se determina si los documentos pertenecen al tema de la revisión sistemática y se los incluye como estudios primarios de la misma. A partir del análisis de cada uno de los primeros estudios obtenidos surjen nuevos documentos que deben ser incluidos en la revisión sistemática. El criterio de exclusión esta dado por el hecho de que los documentos no cumplan con el criterio de inclusión. En la Tabla 2-3 se presentan los resultados obtenidos luego de realizar la búsqueda en cada una de las fuentes.

Tabla 2-3. Distribución de Estudios Seleccionados por Fuente

Fuentes	**Estudios**				**%**
	Encontrados	**No Repetidos**	**Relevantes**	**Primarios**	
IEEE Digital Library	258	62	10	6	20
Science Direct	151	70	20	10	33
SpringerLink	29	10	5	5	17
Google Academic	1526	34	10	6	20
ACM Digital Librery	20	7	3	3	10
Total	**1984**	**183**	**48**	**30**	**100**

Etapa de análisis de resultados

Resultados obtenidos: a continuación, se muestran los resultados obtenidos en cada uno de los documentos evaluados. Es importante mencionar que entre la

literatura consultada se han encontrado algunos autores que proponen aproximaciones de meta-transformaciones, pero, en la mayoría de los casos, no están soportadas por ninguna herramienta.

2.1.2.1 Atzeni, P., Cappellari, P. y Bernstein, P.

En [16] los autores presentan MIDST (*Model Independent Data and Schema Translation*), un *framework* para el modelado de esquemas de bases de datos de forma independiente de la plataforma de implementación final.

MIDST se basa en la observación fundamental de que cualquier modelo de datos existente se puede representar con un conjunto finito de constructores. MIDST da soporte a MODELGEN [17], un operador que establece el conjunto de reglas de transformación entre diferentes esquemas de Bases de Datos (BD) de acuerdo a la tecnología seleccionada por el desarrollador final. El proceso de transformación propuesto por MODELGEN se puede dividir en dos fases: por un lado, la elección de la regla apropiada para realizar la transformación y, por otro lado, la ejecución de dicha regla de transformación.

El desarrollo de MIDST está basado en el concepto de supermodelo (*supermodel*), que es un modelo que tiene todos los contructores que permiten representar cada uno de los meta-constructores de un sistema. De esta manera, cada modelo del sistema es una especialización del supermodelo y cada esquema de cualquier modelo también es un esquema del supermodelo. Esto facilita la definición de las transformaciones entre un esquema de modelos y otro, ya que las mismas se definen en términos de meta-constructores, es decir a nivel de supermodelo. Las reglas de transformación se implementan utilizando el lenguaje *Datalog* por medio de la aplicación de operadores. Esto permite definir las transformaciones de manera atómica, de modo que cada transformación pueda aplicarse de forma (semi-) independiente del resto de las transformaciones. El usuario debe seleccionar la regla (o el conjunto de reglas) que desea ejecutar para realizar una transformación.

MIDST está desarrollado en Eclipse y las reglas de transformación de MODELGEN se han especificado en *Datalog*. Actualmente, permite realizar las transformaciones entre diferentes tecnologías de BD, por ejemplo BD Entidad Relación (ER) a Relacionales o ER a BD Objeto-Relacionales (OR), entre otras. Tiene una interfaz donde el usuario puede seleccionar, o bien conjuntos de reglas de transformación a aplicar, o bien, cada una de las reglas de forma independiente.

En cuanto a las caracteríticas evaluadas:

- ***Soporte a Nivel PIM.*** No brinda soporte para el modelado de las transformaciones a nivel PIM. Las transformaciones son definidas directamente utilizando el lenguaje *Datalog*.
- ***Soporte a Nivel PSM.*** Al estar definidas las transformaciones a nivel de supermodelo como transformaciones atómicas, en el momento de realizar una transformación entre dos esquemas en particular, el desarrollador debe seleccionar el conjunto de reglas a ejecutar. De esta manera, dependiendo de la plataforma de los esquemas, se ejecutan unas u otras reglas, lo cual puede entenderse como una especie de modelado de transformaciones a nivel PSM. Pero, si nos ajustamos a la definición de la característica evaluada, no existe un meta-modelo definido que permita realizar el modelado de las transformaciones a nivel PSM.
- ***Soporte de Transformaciones de PIM a PSM.*** Al no soportar el modelado de las transformaciones a nivel PIM, tampoco soporta el modelado de transformaciones entre los niveles PIM y PSM.
- ***Soporte de Transformaciones de PSM a PSM***. No permite la definición de transformaciones entre los modelos de transformación definidos a nivel PSM.
- ***Soporte al Modelado Gráfico***. No tiene sorpote gráfico para el modelado de las transformaciones, si bien tiene una interfaz en la cual se pueden seleccionar un conjunto básico de reglas de transformación ya definidos para ser ejecutados. Sin embargo, si el usuario desea implementar reglas de transformación adicionales las debe implementar usando Datalog.
- ***Soporte a la Generación de Código***. No soporta la generación del código de la transformación en forma automática.
- ***Soporte a la Validación de Modelos***. No brinda mecanismos de validación de los modelos generados.

A modo de resumen, se puede decir que si bien la propuesta presentada propone un mecanismo para implementar las transformaciones de modelos en el ámbito de las bases de datos, no contempla la posibilidad de realizar la especificación de las transformaciones como modelos de transformación. En otras palabras, no está pensada para realizar el modelado de transformaciones.

2.1.2.2 Bezivin, J., Farcet, N., Jezequel, J. M., Langlois, B. y Mollet, D.

La propuesta presentada en [39] consiste básicamente en considerar las transformaciones como modelos de primer orden. Para realizar el modelado de las mismas se propone la definición de dos niveles de abstracción, correspondiéndose con los niveles PIM y PSM propuestos por MDA. De esta manera, se propone la

especificación de transformaciones independientes de plataforma (*Platform Independent Transformation*, PIT) para luego, a partir de las PIT, generar transformaciones específicas de plataforma ((*Platform Specific Transformation*, PST). En este contexto se entiende como plataforma la herramienta que permite realizar la especificación, diseño y ejecución de las transformaciones.

Esta propuesta puede ser aplicada para desarrollar las transformaciones de modelos de manera ortogonal a MDA, es decir, para desarrollar transformaciones de modelos entre modelos de nivel PIM, entre modelos de nivel PIM y PSM y entre modelos de nivel PSM.

Siguiendo con esta línea de investigación, Bézivin, Büttner, Gogolla, Jouault, Kurtev, Lindow en [34] proponen la idea de realizar el modelado de las transformaciones de modelo, esto es, definir modelos de transformación conformes a meta-modelos de transformación, indicando que estos meta-modelos de transformación deben ser genéricos, es decir, definidos a nivel PIT. A partir de estos modelos de transformación se podrían obtener diferentes transformaciones de modelos, dependiendo de la plataforma de implementación utilizada. En este trabajo los autores presentan además algunas de las ventajas que se obtienen al desarrollar las transformaciones de modelos como modelos de transformación, como, por ejemplo: la uniformidad del lenguaje a utilizar para definir las transformaciones; la definición de transformaciones a alto nivel, es decir, realizar transformaciones de transformaciones; o la facilidad para realizar la validación de los modelos de transformación, entre otras.

En cuanto a las caracteríticas evaluadas:

- ***Soporte a Nivel PIM.*** Los autores proponen el modelado de las transformaciones a nivel PIT (nivel PIM de MDA), donde los modelos de transformaciones se representan por medio de una biblioteca genérica de transformaciones más simples y primitivas. Estos modelos deberán ser refinados hasta el punto en que pueden ser utilizados como modelos origen para generar transformaciones a nivel PST (nivel PSM de MDA). Para implementar los modelos de transformación a nivel PIT los autores proponen el uso de UML, ya que debido a sus mecanismos de estructuración estática (paquetes, clases y métodos), los autores consideran que UML tiene el poder de modelar transformaciones independientemente de la complejidad de las mismas.
- ***Soporte a Nivel PSM.*** Para el modelado de las transformaciones a nivel PST (nivel PSM de MDA), los autores proponen el uso de la sintaxis definida en cada una de las plataformas que se desee implementar las transformaciones.

- ***Soporte de Transformaciones de PIM a PSM.*** En el trabajo se propone el uso de transformaciones entre los modelos a nivel PIT y los modelos a nivel PST. El proceso definido en esta propuesta se compone de dos pasos: en primer lugar se definen los modelos de transformación de manera independiente de plataforma; a continuación, dichos modelos se transforman a modelos específicos de plataforma. Es importante mencionar que a lo largo de los trabajos no se especifican las transformaciones entre los modelos a nivel PIT y los modelos a nivel PST, ni se menciona cómo se llevaría a cabo la implementación de las mismas, tan solo se menciona la necesidad de dichas transformaciones.
- ***Soporte de Transformaciones de PSM a PSM***. La propuesta no contempla la especificación de transformaciones entre modelos a nivel PST.
- ***Soporte al Modelado Gráfico***. No se menciona la necesidad de realizar el modelado gráfico de las transformaciones de modelos.
- ***Soporte a la Generación de Código***. La propuesta contempla la necesidad de obtener el código que implementa dicha transformación en el lenguaje seleccionado por el desarrollador, aunque no se especifica cómo se obtendrá este código ni cómo se debería realizar la implementación de la misma.
- ***Soporte a la Validación de Modelos***. Se indica la necesidad de realizar validación de los modelos participantes en las transformaciones y se propone el uso de UML y *OCL Tools* para implementar dichas validaciones, pero no se especifica cómo se debería llevar a cabo la validación.

En las conclusiones del artículo los autores mencionan que están trabajando en la implementación de un conjunto de herramientas de código fuente abierto que den soporte a las ideas presentadas en el marco del programa de investigación CARROL (*www.carroll.research.org*). Sin embargo, no se han podido obtener dichas herramientas; de hecho, se ha comprobado que el proyecto parece abandonado, ya que una visita a la página del proyecto informa que la última actualización de la misma tiene fecha de 25/08/2005. Por ello, se debe mencionar que en el momento de escribir este documento no se ha podido contrastar esta metodología por medio del uso de una herramienta que la implemente.

2.1.2.3 Didonet Del Fabro, M.

En la tesis doctoral presentada en [79] el autor propone la definición de una solución para la administración de las relaciones genéricas existentes entre los modelos. Esta aproximación, denominada *Model Weaving* (modelo de tejido, modelos de *weaving* a partir de ahora) ha sido definida en el ámbito del MDE con

el propósito de brindar soporte para realizar la representación, el manejo y la utilización de las relaciones entre los elementos pertenecientes a distintos modelos. El alcance de la tesis presentada en [79] es el estudio de los aspectos conceptuales, prácticos y de aplicación en relación con modelos de *weaving*.

Una de las principales aportaciones realizadas en dicha tesis es la del uso de modelos de *weaving* para realizar transformaciones de modelos de manera semi-automática, generando transformaciones de mapeo (*matching transformations*) entre los modelos origen y destino. A modo de resumen, la idea principal es utilizar los modelos de *weaving* para indicar las relaciones existentes entre el modelo origen y el modelo destino, es decir, qué elemento del modelo origen debe ser transformado en qué elemento del modelo destino. Después, mediante la ejecución de una transformación de modelos, se obtiene el código implementable de la transformación en el lenguaje ATL (*ATLAS Transformation Language*, [105]).

Durante el desarrollo de la tesis, el autor se centra en la utilización de los modelos de *weaving* y las transformaciones de modelos en distintos escenarios de interoperabilidad de datos; utilizando los modelos de *weaving* como especificaciones a partir de las cuales se generan las transformaciones de modelos. De esta manera, se brinda una solución a la gestión de relaciones genéricas en el ámbito de la interoperabilidad de datos.

Para validar la aproximación propuesta, el autor ha implementado una herramienta genérica y adaptable denominada *ATLAS Model Weaver* (AMW). La herramienta cuenta con mecanismos de extensión que permiten adaptarla a nuevas situaciones. La interfaz de usuario se genera automáticamente de acuerdo a la extensión del meta-modelo utilizada por el usuario.

En cuanto a las características evaluadas:

- ***Soporte a Nivel PIM.*** El autor no contempla la necesidad del modelado de transformaciones a nivel PIM.
- ***Soporte a Nivel PSM.*** Para realizar el modelado de las transformaciones a nivel PSM el autor propone el uso de la herramienta AMW que permite establecer de una manera sencilla las relaciones existentes entre cada elemento del modelo origen y los elementos del modelo destino. Es importante mencionar que los modelos de transformación definidos en este nivel son conformes a ATL, pero no se contempla el uso de otro lenguaje de transformación en este nivel.

- ***Soporte de Transformaciones de PIM a PSM.*** Al no contemplarse el modelado de transformaciones a nivel PIM, no se contempla la necesidad de transformaciones entre modelos de nivel PIM y modelos de nivel PSM.
- ***Soporte de Transformaciones de PSM a PSM***. Se definen las transformaciones a nivel PSM. Esto es, desde el modelo generado con la herramienta AMW, se definen el conjunto de transformaciones que permite obtener un modelo de transformación conforme al meta-modelo de ATL. Este conjunto de transformaciones se implementan en la herramienta, de tal manera que una vez definidas las relaciones entre los elementos de los diferentes meta-modelos, el usuario sólo debe aplicar la transformación y, de forma automática, se obtiene el modelo de la transformación en ATL.
- ***Soporte al Modelado Gráfico***. La herramienta desarrollada para dar soporte al método presentado por el autor permite la definición de las transformaciones de forma gráfica y de una manera sencilla y amigable para el usuario, simplemente estableciendo un *link* entre los diferentes elementos relacionados.
- ***Soporte a la Generación de Código***. La generación de código está soportada por el uso del extractor implementado con TCS (*Textual Concret Syntax*, [104]) en el lenguaje ATL. De esta manera, una vez obtenido el modelo de la transformación en ATL, el usuario puede obtener el código que implementa dicha transformación, simplemente haciendo uso de la característica brindada por ATL.
- ***Soporte a la Validación de Modelos***. No brinda soporte a la validación de modelos.

Desde nuestro punto de vista, esta propuesta es una de las más interesantes que hemos encontrado, ya que no solo realiza la definición de un método, sino que la validan por medio de la implementación de una herramienta que le da soporte. Sin embargo, presenta dos inconvenientes: el primero de ellos es el hecho de que el autor no ha seguido trabajando en esta línea, por lo que el proyecto, en cuanto a la implementación de transformaciones se refiere, parece estar abandonado. El segundo inconveniente, es el hecho de que las transformaciones de modelos se implementan utilizando ATL, es decir, que la herramienta, y por lo tanto, la solución, están ligadas a la plataforma final de implementación.

2.1.2.4 Küster, J. M., Ryndina, K. y Hauser, R.

En [128] los autores presentan un método sistemático para la construcción de transformaciones de modelos centrado en el diseño de las mismas.

Básicamente, proponen realizar la distinción entre el diseño de las transformaciones de alto nivel, de bajo nivel y, por último, la validación del diseño de las transformaciones.

El método consiste en que el desarrollador especifique reglas de transformación de alto nivel para luego refinarlas hasta obtener el diseño de las reglas de transformación de bajo nivel siguiendo unas guias de diseño establecidas. El método permite el desarrollo sistemático y repetitivo de transformaciones de modelos incrementando la calidad de las mismas.

Si bien en el trabajo se presenta el desarrollo de un caso de estudio aplicando el método definido, no se brinda ninguna información sobre la herramienta que da soporte a dicho método o si está disponible para su uso. A nuestro entender, el método presentado está orientado a la forma en cómo se deberían definir las reglas de transformación, indicando pasos a seguir en el proceso de descubrimiento e implementacion de las mismas. De hecho este punto de vista se puede corroborar en trabajos como [125] y [127] donde se utiliza el método propuesto para el diseño y la implementación de las transformaciones de modelos.

En cuanto a las características evaluadas:

- ***Soporte a Nivel PIM.*** Los autores proponen realizar el diseño de las transformaciones de modelos en distintos niveles, desde nuestro punto de vista el diseño de alto nivel se correspondería con el nivel PIM de la arquitectura MDA. En este nivel, los autores proponen identificar las reglas de transformación de manera informal indicando diferentes casos para cada una de las reglas, estos casos son agrupados en casos soportados y no soportados. Como sintaxis concreta para el modelado de las transformaciones los autores proponen el uso de diagramas correspondientes al ámbito de las transformaciones modeladas.
- ***Soporte a Nivel PSM.*** Cada uno de los casos soportados a alto nivel se especifican como reglas de transformación en el diseño de bajo nivel, lo que se correspondería con el nivel PSM de la arquitectura de MDA y, posteriormente, se expresan en el lenguaje de transformación de modelos seleccionado, lo que se podría corresponder con un nivel más de detalle dentro del nivel PSM. Para el modelado de las transformaciones a nivel PSM los autores proponen el uso de una nueva sintaxis, donde cada regla de transformación se compone de un lado derecho y un lado izquierdo. Ambos lados tienen patrones gráficos que pueden ser emparejados con elementos de los modelos origen y que generan elementos en el modelo destino. La

notación gráfica utilizada en los patrones está basada en los diagramas de objetos de UML. Estos modelos pueden ser refinados tantas veces como el desarrollador considere necesario.

- ***Soporte de Transformaciones de PIM a PSM.*** Los autores definen un proceso iterativo de refinamiento sistemático de los modelos de transformación definidos a alto nivel en modelos de transformación a bajo nivel. Este proceso está definido como un conjunto de guías o instrucciones a seguir para realizar el refinado.
- ***Soporte de Transformaciones de PSM a PSM***. Si bien los autores reconocen la necesidad de realizar el refinado de los modelos de transformación a nivel PSM, no especifican las guias o reglas de transformación que deberían implementarse.
- ***Soporte al Modelado Gráfico***. Los autores reconocen la necesidad del modelado gráfico de las transformaciones. A nivel PIM proponen el uso de la sintaxis utilizada en el contexto de las transformaciones y a nivel PSM proponen el uso de los diagramas de objeto de UML.
- ***Soporte a la Generación de Código***. Como resultado final del proceso de diseño de las transformaciones se espera obtener el código implementable de la transformación, pero no se especifica la forma en la que se realizará este último paso.
- ***Soporte a la Validación de Modelos***. En el trabajo se propone que la validación de los modelos se debe hacer comprobando la validez sintáctica y semántica de los diferentes modelos de transformación de acuerdo al lenguaje seleccionado. Además, se especifica que la validación de los modelos a bajo nivel se realiza en forma parcial al comprobar su correspondencia con los modelo de alto nivel. Con respecto a la validez sintáctica y semántica de los modelos, los autores no especifican el tipo de validación que se realizará.

Siguiendo en la misma línea de investigación, en [127] Küster, Gschwind, y Zimmermann proponen una aproximación para el desarrollo incremental de cadenas de transformaciones de modelos basadas en pruebas automatizadas que permitirá mejorar la calidad de las cadenas de transformación de modelos de forma sistemática, permitiendo agregar nuevas funciones, corregir defectos o cambiar una transformación determinada. La validación de todos los cambios realizados en una cadena de transformación se realiza por medio de las técnicas automatizadas de pruebas, siguiendo el principio establecido de desarrollo basado en pruebas.

La aproximación propuesta sólo abarca las actividades de diseño, implementación y pruebas del proceso de desarrollo de transformaciones de modelos. Para las actividades de especificación de requerimientos, análisis y especificación de las funciones detalladas del proceso de transformación los autores proponen el uso del método presentado en [128].

Como parte de la propuesta han especificado una arquitectura que permite generar casos de pruebas que pueden ser utilizados en el desarrollo incremental de cadenas de transformación de modelos. Además, han validado la técnica propuesta por medio del desarrollo incremental de una cadena de transformación de modelos de gestión de versiones de modelos de procesos.

Como se puede observar, las propuestas realizadas están orientadas a la definición de pasos, o un método, a seguir en el proceso de definición de transformaciones de modelos, persiguiendo en todo momento el objetivo de incrementar la calidad de las transformaciones generadas, pero no se ha encontrado, en el momento de escribir esta tesis, una herramienta que valide el método propuesto.

2.1.2.5 Tratt, L.

En [184], Tratt presenta una aproximación sencilla para el desarrollo de transformaciones de modelos en un alto nivel de abstracción que comprende, únicamente, la etapa de ejecución de la transformación y no el ciclo de vida completo de la transformación como en los anteriores casos presentados.

La aproximación se presenta como una manera de solucionar el problema que existe a la hora de realizar transformaciones entre modelos almacenados en diferentes herramientas, en otras palabras mejorar la interoperabilidad entre las mismas.

Desde el punto de vista del autor, una de las maneras de mejorar la interoperabilidad entre las herramientas, es especificar las transformaciones de modelos en un lenguaje de programación estándar. De hecho, la principal contribución de este trabajo es una nueva aproximación para la definición de transformaciones de modelos, *CONVERGE* [185], un lenguaje de transformación de modelos imperativo que provee mecanismos para asegurar el mantenimiento de la trazabilidad en forma automática. La idea es que sea capaz de proveer de manera flexible, eficiente y práctica de una plataforma para la creación de transformaciones de modelos que facilite la integración de herramientas. La aproximación incluye una serie de pasos, a modo de método, que definen cómo deberían realizarse las transformaciones.

Como parte del trabajo presentado, el autor realiza una evaluación de las diferentes aproximaciones existentes para la ejecución de las transformaciones, aplicando los pasos definidos en la aproximación.

En cuanto a las características evaluadas:

- ***Soporte a Nivel PIM.*** El autor no menciona la necesidad de realizar el modelado de transformaciones a nivel PIM.
- ***Soporte a Nivel PSM.*** Se propone la especificación de las transformaciones a nivel PSM implementándolas directamente en el lenguaje CONVERGE.
- ***Soporte de Transformaciones de PIM a PSM.*** Al no proponer el modelado de las transformaciones a nivel PIM, no se contempla la necesidad de transformaciones entre modelos de nivel PIM y modelos de nivel PSM.
- ***Soporte de Transformaciones de PSM a PSM***. No se menciona la necesidad de de realizar transformaciones entre modelos de transformación de nivel PSM.
- ***Soporte al Modelado Gráfico***. No se menciona la necesidad de soportar el modelado gráfico de las transformaciones.
- ***Generación de Código***. No se propone la generación del código de las transformaciones de forma automática. El desarrolador debe implementar las transformaciones utilizando el lenguaje CONVERGE.
- ***Validación de Modelos***. No se menciona la necesidad de realizar la validación de modelos.

La aproximación presentada por el autor, se centra en justificar la necesidad de un lenguaje estándar para la definición de las transformaciones de modelos. El autor menciona la necesidad de implementar las transformaciones en un alto nivel de abstracción a modo de unificar un mismo lenguaje para todos los contextos. Sin embargo, no se menciona la necesidad del modelado de las transformaciones como modelos en si mismos. Si bien presentan un lenguaje que soporta la aproximación presentada, no deja de ser otro lenguaje de transformación para implementar transformaciones entre modelos y no una herramienta de alto nivel.

2.1.2.6 Vignaga, A.

En el trabajo presentado en [204] el autor propone una metodología para aplicar las técnicas propuestas en MDE al desarrollo y evolución de las transformaciones de modelos con el objetivo de mejorar la calidad de las transformaciones de modelos y la productividad de sus desarrolladores. Como

parte de la propuesta que se presenta, se identifican, de acuerdo al punto de vista del autor, dos problemas: el primero es el hecho de que no exista un paradigma de programación dominante para implementar las transformaciones [65]. De hecho, existen diferentes enfoques, donde la filosofía subyacente difiere sustancialmente, lo que afecta a la manera en la que se definen las transformaciones. El segundo problema detectado por el autor, y con el cual se coincide plenamente en esta tesis, es el hecho de que no existe una categorización unificada de las transformaciones de modelos. Las diferentes propuestas de categorización existentes [65, 100, 141, 180, 184] evalúan características de acuerdo a las necesidades de los autores de dichas propuestas, pero no existe ninguna categorización consensuada.

Si bien la metodología propuesta se centra principalmente en el diseño y la implementación de las transformaciones, abarca todo el ciclo de vida del desarrollo de las transformaciones de modelos. La metodología está definida como un proceso completo expresada mediante un modelo SPEM y propone un ciclo de vida basado en un modelo iterativo e incremental, estructurado en etapas, una para la construcción y una para la evolución.

Siguiendo en la misma línea de investigación en [206], Vignaga realiza la aplicación de una propuesta de especificación de modelos de transformación a un caso de estudio. Finalmente, en este artículo se concluye con un debate sobre la propuesta de modelos de transformación por medio de un serie de ideas basadas en la propia experiencia del autor e identifica algunos usos de la terminología estándar aplicadas al contexto del desarrollo transformaciones de modelos.

Uno de los últimos artículos encontrados del autor sobre este tema es [207], donde se comienza a dislumbrar la implementación de la propuesta metodológica realizada en [204]. Los autores de este trabajo proponen realizar la especificación de las transformaciones de modelos siguiendo una estructura de cuatro niveles de abstracción. El primer nivel, el más bajo, debería contener el código que implementa la transformación, en términos de constructores específicos de plataforma. El segundo nivel debería contener una abstracción de los constructores especificados en el nivel anterior, suprimiendo los constructores específicos de plataforma. En el tercer nivel, se debería especificar las relaciones entre los diferentes elementos. Por último, en el cuarto nivel se debería incluir mecanismos de modularización, especificando lo que debería hacer la transformación, pero con menor detalle que en las etapas anteriores. Sin embargo, en el artículo no se aportan detalles de implementación técnica de esta propuesta, de hecho actualmente, esta línea de trabajo parece haber sido abandonada por su autor, ya que no existe más documentación respecto a la propuesta ni se ha encontrado ninguna herramienta que la implemente.

En cuanto a las características evaluadas:

- ***Soporte a Nivel PIM.*** En los trabajos analizados se propone el modelado de las transformaciones de modelos a un alto nivel de abstracción, que se correspondería con el nivel PIM de MDA. Es importante aclarar que no se menciona la sintaxis que se utilizará para realizar el modelado de las transformaciones en este nivel.
- ***Soporte a Nivel PSM.*** Se propone el modelado de las transformaciones a un nivel intermedio, que se correspondería con el nivel PSM de MDA. De hecho, los autores proponen el modelado en dos niveles de abstracción: el tercer nivel donde se establecen las relaciones entre los diferentes elementos y el segundo nivel donde se especifican dichas relaciones en base a constructores con vías a implementarlos en una plataforma específica. De igual manera que en el punto anterior, no se menciona la sintaxis utilizada para realizar la especificación de estos modelos.
- ***Soporte de Transformaciones de PIM a PSM.*** Si bien los autores reconocen la necesidad de implementar de forma automática las transformaciones entre los modelos de los niveles PIM y PSM, no se especifican las mismas ni se menciona cómo se implementarían.
- ***Soporte de Transformaciones de PSM a PSM***. Se menciona la necesidad de realizar las transformaciones entre modelos del nivel PSM (Tercer y Segundo Nivel) pero no se especifican las transformaciones que deberían implementarse ni se menciona la forma en que se deberían llevar a cabo.
- ***Soporte al Modelado Gráfico***. Se mencionan las ventajas de realizar el modelado gráfico de las transformaciones de modelos, pero no se menciona como se implementarán.
- ***Generación de Código***. El último nivel propuesto por los autores es el del código que implementa la transformación. En este nivel se propone el uso de la sintaxis provista por cada uno de los lenguajes de transformación que se utilicen.
- ***Validación de Modelos***. Se menciona la importancia de realizar la validación de los modelos de transformación generados, pero no se especifica la manera de realizarlo.

La idea presentada por los autores es una de las más completas que se han analizado, lamentablemente esta línea de investigación parece haber sido abandonada por los mismos, ya que al momento de escribir esta tesis no se ha encontrado mas documentación ni se ha podido contrastar la metodología

propuesta por medio de una herramienta. De hecho, uno de los últimos artículos recuperados sobre este tema data del año 2007.

2.1.3 Conclusiones

En la Tabla 2-4 se presenta, a modo de resumen, el análisis realizado en cada una de las aproximaciones.

Como se pueden observar la mayoría de los trabajos evaluados son sólo meras intenciones, a excepción del trabajo presentado por Didonet Del Fabro [79], ninguno valida la propuesta realizada por medio de herramientas que la implemente o casos de éxito completos que la validen. Además se debe mencionar, que la mayoría de las líneas de investigación evaluadas parecen haber sido abandonadas por los autores, o absorbidas por otras líneas de investigación.

A nuestro entender los trabajos más acertados, desde el punto de vista de la propuesta de un método que dé soporte a las meta-transformaciones o, en otras palabras, permitan realizar transformaciones de alto nivel, son los presentados por Bezivin en [39] y Didonet Del Fabro en [79], ya que, ambos trabajos se basan en la idea de aplicar MDE al desarrollo de las transformaciones de modelos, y, en el caso del último trabajo presentan una herramienta que permite validar la aproximación propuesta.

Una de las principales premisas de MDE es la automatización de las tareas para cualquier propuesta de desarrollo de software dirigido por modelos [14, 81]. Como resultado de esto han surgido una serie de herramientas para automatizar dichas tareas relacionadas con MDE. El mayor impacto ha sido en la aparición de herramientas para definir y/o utilizar nuevos lenguajes de modelados. De manera análoga, las transformaciones entre los modelos son una de las principales operaciones de MDE, por lo que han surgido una serie de herramientas o lenguajes que permiten implementar dichas transformaciones de modelos.

Al evaluar los diferentes lenguajes de transformación de modelo a modelo, se puede observar que la mayoría funciona de manera correcta, teniendo en cuenta que los casos de éxito han sido implementados por los propios desarrolladores de los lenguajes. Sin embargo, cuando un desarrollador externo utiliza los lenguajes se encuentra ante dos dilemas: el primero de ellos, la selección del lenguaje a utilizar y, el segundo, una vez seleccionado el lenguaje, el aprendizaje del mismo.

Teniendo en cuenta esto último, y siguiendo las líneas de investigación planteadas en [39] y [79], se ha considerado conveniente desarrollar un entorno (MeTAGeM) que facilite el desarrollo (semi-)automático de transformaciones de

modelos mediante la especificación de las mismas en un lenguaje de alto nivel independiente de la plataforma y su posterior transformación a un lenguaje específico de plataforma. El entorno estará formado por: una metodología, que define la manera en que se deberán desarrollar las transformaciones y una herramienta que valide dicha metodología.

Como se puede observar en la Tabla 2-4 MeTAGeM soporta cada una de las características evaluadas y tiene una herramienta que las valida. En el siguiente capítulo, se mostrará la definición de la metodología y la implementacion de la meta-herramienta que la soporta.

Tabla 2-4. Resumen de las Características Evaluadas en cada una de las Propuestas

Características	**Atzeni et al.**		**Bezivin et al.**		**Didonet Del Fabro**		**Küster et al.**		**Tratt**		**Vignaga**		**MeTAGeM**	
	M	H	M	H	M	H	M	H	M	H	M	H	M	H
Soporte de Nivel PIM	0	x	2	x	0	x	2	x	0	x	1	x	3	√
Soporte de Nivel PSM	3*	√	2	x	3	√	3	x	3	√	1	x	3-2 **	√
Soporte de Transformaciones PIM – PSM	0	x	1	x	0	x	2	x	0	x	1	x	3	√
Soporte de Transformaciones PSM – PSM	0	x	0	x	3	√	1	x	0	x	1	x	3	√
Soporte al Modelado Gráfico	0	x	0	x	3	√	2	x	0	x	1	x	2	√
Soporte a la Generación de Código	0	x	1	x	2	√	1	x	3	√	2	x	2-3***	√
Soporte a la Validación	0	x	2	x	0	x	1	x	0	x	1	x	3	√

* - Se pueden seleccionar diferentes conjuntos de reglas de transformación para transformar los mismos modelos.

** - A nivel PSM, MeTAGeM propone el modelado de las transformaciones siguiendo las diferentes aproximaciones existentes, por otro lado propone una especialización de las transformaciones en un nivel PDM completamente dependiente de plataforma, es decir, seleccionando un lenguaje de transformación en particular. En este último nivel, se propone que, si el lenguaje seleccionado tiene un meta-modelo implementado se utilice dicho meta-modelo, caso contrario, será necesario implementar el meta-modelo del lenguaje.

*** - En cuanto a la generación de código, MeTAGeM propone el uso de los propios generadores de código de cada lenguaje. Si el lenguaje no tuviera un generador de código se debería implementar uno.

2.1.4 Nuevos Trabajos en la Línea

Como se puede observar a lo largo de este capítulo, no existen muchas propuestas donde se apliquen los principios de MDE al desarrollo de las transformaciones. Sin embargo, cuando se estaba finalizando la escritura de esta tesis se encontraron dos trabajos relevantes en cuanto al desarrollo de las transformaciones [93, 126].

Estos trabajos han servido para validar la propuesta que se presenta en esta tesis, ya que en ambos trabajos se manifiesta la importancia de definir un método que permita especificar las transformaciones aplicando los principios de MDE.

Debido al estado avanzado de la tesis no se pudo realizar un estudio exhaustivo de estos trabajos, pero al ser relevantes en el área que se está estudiando se considera conveniente presentarlos, por lo que en el anexo C se muestran las principales características de cada una de las propuestas.

A continuación, se presenta el estudio realizado de los lenguajes de transformación de modelos y de las herramientas que los implementan en el ámbito del MDE.

2.2 Lenguajes de Transformación de Modelos

Trabajar con modelos relacionados entre sí requiere un esfuerzo significativo para llevar a cabo algunas tareas relacionadas con la gestión de los modelos, como el refinamiento, la comprobación de coherencia, la refactorización, etc. De hecho, uno de los principales retos relacionados con el uso de modelos en Ingeniería de Software es automatizar estas tareas. Dentro de dicho ámbito, como se ha visto anteriormente, las transformaciones de modelo han sido aceptadas como la forma de realizar la gestión de los modelos.

En la literatura existente en el ámbito del MDE, se pueden encontrar diferentes definiciones de qué es una transformación de modelos. A modo de ejemplo, se citaran alguna de ellas:

- Según la guía de MDA [144]: "La transformación de modelos es el proceso de convertir un modelo en otro modelo del mismo sistema".
- Kleppe y otros [113], definen una transformación de modelos como "la generación automática de un modelo destino, partiendo de un modelo origen, conformes a la definición de la transformación".

- Tratt [184] realiza la siguiente definición: "Una transformación de modelos es un programa que permite realizar la mutación de un modelo en otro, en otras palabras, algo parecido a un compilador".
- Por último, Sendall & Kozaczynski [180] se refieren a la transformación de modelos "como la automatización de un proceso que tiene uno o más modelos origen y producen uno o más modelos destino como salida siguiendo un conjunto de reglas de transformación".

En el contexto de esta tesis nos hemos basado en la definición proporcionada por Bézivin en [26] que refina la idea propuesta por Lemele en [131], según la cual la transformación de modelos "es un proceso conforme a un meta-modelo". La Figura 2-1 muestra una visión general de todo el proceso propuesto en [26].

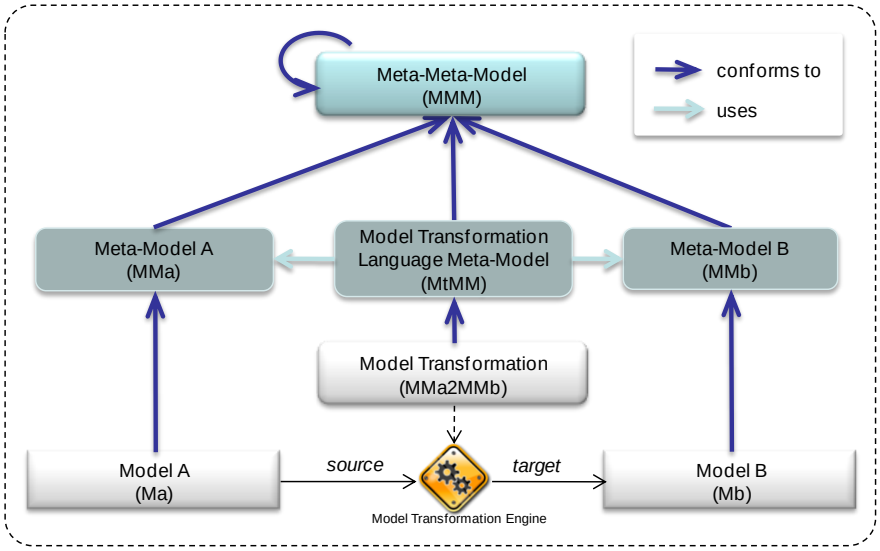

Figura 2-1. Proceso de Transformaciones de Modelos

El principal elemento del proceso es el meta-meta-modelo (MMM), que proporciona un conjunto básico de abstracciones que permiten definir nuevos meta-modelos. De esta manera, se definen los meta-modelos origen y destino, como una instancia del meta-meta-modelo, es decir, conformes a dicho meta-meta-modelo (*conforms to*).

Para realizar el mapeo del modelo Ma (conforme al meta-modelo MMa) al modelo Mb (conforme al meta-modelo MMb) es necesario especificar un conjunto de reglas que codifiquen las relaciones entre los elementos de ambos meta-modelos. Este conjunto de reglas serán recogidas en una transformación de

modelos MMa2MMb definida a nivel de meta-modelo. Dicha transformación será ejecutada por el motor de transformaciones de modelos, permitiendo generar modelos destino a partir de cualquier conjunto de modelos conformes al meta-modelo origen. En otras palabras, la transformación de modelos puede ser ejecutada para cualquier modelo definido de acuerdo con el meta-modelo de origen

Si el conjunto de reglas y restricciones que guían la construcción de la transformación de modelos es recogida en un meta-modelo (MtMM), cualquier transformación de modelos puede ser expresada como un modelo conforme a dicho meta-modelo. Expresar la transformación de modelos como un modelo, es decir, realizar el modelo de la transformación, permite gestionar dichos modelos por medio de otras transformaciones. Esto conlleva varias ventajas asociadas, por ejemplo, cualquier modelo de transformación puede ser utilizado como modelo origen o destino de otra transformación. Para referirse a este tipo especial de transformación de modelos, donde el modelo origen y/o destino es otra transformacion de modelos, se utiliza el término *Higher Order Transformation* (HOT, [189]), es decir, transformación de alto nivel.

Además, la definición de transformación de modelos usando modelos de transformación permite realizar composición de transformaciones de modelos como en cualquier otro tipo de composición de modelos [36, 208]; desarrollando la evolución del meta-modelo y la co-evolución de técnicas de modelado [28, 32, 38, 57]: definiendo cadenas de transformación de modelos [24, 190]; o reutilizando las transformaciones de modelos existentes [178], entre otras cosas.

Si bien el estudio de las técnicas de transformación ha sido tema de investigación durante los últimos 30 años [9, 37, 52, 168, 188], ha estado centrado en las transformaciones de programas (código fuente). Las transformaciones de modelos han despertado poco interés hasta ahora, pero el auge de MDE y las ventajas de MDA han cambiado drásticamente esta situación, ya que las transformaciones de modelos han pasado a tener un rol principal en el DSDM. Una consecuencia de esto ha sido la aparición de numerosos lenguajes y herramientas de transformación de modelos. A pesar de que en los últimos años, se ha avanzado mucho en el campo de investigación de las transformaciones de modelos, aún queda mucho espacio de mejora, como se evidencia en las diferentes definiciones de transformación de modelos citadas anteriormente.

A pesar de que las definiciones dadas son similares entre sí, existen algunos puntos no muy claros, como el concepto de qué es una transformación de modelos en esencia (un proceso de automatización, un programa, una descripción,

un algoritmo, un modelo, etc), o el nivel de automatización que debe proporcionar, o la cardinalidad de los modelos origen y destino, o la bidireccionalidad de las mismas, entre otras cosas.

Si no hay aún un consenso total acerca de lo que es una transformación de modelos, la complejidad asociada a elegir entre la amplia variedad de aproximaciones de transformación de modelos y de propuestas de lenguajes de transformación existentes es considerable.

Esto es uno de los motivos principales por los cuales en esta tesis se aborda el desarrollo de un entorno que permita el modelado de las transformaciones en un alto nivel de abstracción, de tal manera que el desarrollador pueda especificar las transformaciones indicando sólo las relaciones entre los elementos de cada meta-modelo y luego, por medio de transformaciones, obtener el modelo de la transformación siguiendo una aproximación determinada y en un lenguaje de transformación en particular.

Como se ha dicho anteriormente, los lenguajes de transformación de modelos siguen diferentes tipos de enfoques [65, 66]. Para el desarrollo de esta tesis se selecciona el enfoque **híbrido** para el modelado de las transformaciones específicas de plataforma. A continuación, en la sección 2.2.1 se presenta un resumen de las principales aproximaciones y se justifica la elección de la aproximación híbrida.

2.2.1 Aproximaciones de los Lenguajes de Transformación de Modelos

Los lenguajes de transformación de modelos se pueden agrupar en tres grandes conjuntos: lenguajes de propósito general, lenguajes basados en grafos y lenguajes puramente declarativos o imperativos. A continuación se detallan brevemente cada uno de ellos:

1. *Lenguajes de Propósito General*

   En esta categoría se incluyen tres tipos de lenguajes: los basados en la manipulación directa de los modelos [183], los basados en XML [123] y los basados en plantillas [58].

   El principal inconveniente que tienen los lenguajes basados en la **manipulación directa** es que no han sido pensados para realizar manipulación directa de modelos, por lo que, implementar una transformación de modelos resulta en una tarea muy compleja y complicada.

Los lenguajes **basados en XML** funcionan bien para la transformación de documentos expresados en lenguajes de marcado, sin embargo no son muy utilizados para las transformaciones de modelos, ya que no son nada intuitivos ni fáciles de usar.

Por último, los lenguajes **basados en plantillas** se utilizan para realizar generación de código (de hecho, el estándar de la OMG MOF2T es un lenguaje basado en plantillas [151, 152]), pero resulta demasiado rígido para ser aplicado a las transformaciones de modelo a modelo.

2. *Lenguajes basados en grafos*

   Las transformaciones basadas en grafos son, probablemente, las más interesantes desde el punto de vista puramente teórico, debido a que la gramática de grafos está basada en una sólida teoría matemática; por tanto, presentan una serie de propiedades teóricas que permite la formalización de las transformaciones de modelos [62, 82].

   Además, existe una similitud entre los modelos, representados en forma gráfica y los grafos. Un grafo tiene nodos y arcos, mientras que un modelo tiene clases y asociaciones entre esas clases; de esta forma el hecho de que los modelos estén representados en forma de grafos acortan la distancia entre los desarrolladores y los usuarios finales de modelos de transformación de modelos.

   Sin embargo, expresar las transformaciones en términos de reglas de reescritura de grafos aumenta en gran medida la complejidad y el aumento de formalización que proporcionan no compensa la complejidad añadida.

   Asimismo, es importante mencionar que el nivel de adopción de las aproximaciones basadas en grafos es muy bajo en comparación con otros lenguajes para definir transformaciones de modelos. En general, su uso está limitado a los grupos de investigación que proponen (y desarrollan) la aproximación, ya que la utilizan como herramienta para abordar algún problema de Ingeniería de Software en concreto. En nuestra opinión, este comportamiento está directamente relacionado con la complejidad inherente de las transformaciones basadas en grafos, lo que dificulta la adopción de herramientas de este tipo, ya que no solo es necesario que el desarrollador reescriba los *mappings* definidos en sus meta-modelos utilizando las reglas de reescritura de grafos, sino que también debe aprender qué tipo de reglas de reescritura se utilizan en cada lenguaje en particular, ya que cada lenguaje utiliza notaciones diferentes.

3. Lenguajes puramente declarativos o imperativos

Esta subsección se centra principalmente en la comparación de los enfoques **declarativo** vs. **imperativo**. Por un lado, el enfoque declarativo se basa en la definición de las relaciones que deben mantenerse entre los elementos origen y los elementos destino, de tal manera que si, estas relaciones no son satisfechas, el motor crea los elementos necesarios en el modelo destino para que lo sean. Por otro lado, el enfoque imperativo se basa en la creación explícita de elementos destino usando un estilo de programación procedural, añadiendo posteriormente los atributos y las referencias correspondientes.

Los lenguajes declarativos tienen una naturaleza implícita, es decir, tienen mecanismos implícitos que definen patrones de coincidencia entre los elementos de entrada y las reglas de transformación, por lo tanto, no es necesario establecer de forma explícita en el código de la transformación el orden de ejecución de los patrones de coincidencia. Por lo contrario, en los lenguajes imperativos es necesario que el desarrollador realice todas las llamadas a las reglas de forma explícita. De esta manera, una transformación expresada en un lenguaje declarativo suele ser más concisa que la misma transformación expresada en un lenguaje imperativo. Sin embargo, el hecho de que una transformación sea concisa puede dificultar la comprensión de la misma. De hecho, muchas características suelen permanecer ocultas para los desarrolladores no expertos en el momento de especificar una transformación como declarativa, ya que son menos explícitas que en la transformación en imperativo. Todo esto hace que la curva de aprendizaje de los lenguajes declarativos sea más larga.

Una de las mayores ventajas de las aproximaciones puramente declarativas es que cada regla es completamente independiente del resto de las reglas. De esta manera, una vez que el desarrollador domine la técnica de programación declarativa, usar un lenguaje declarativo simplifica enormemente la tarea de codificación de la transformación.

Del mismo modo, las aproximaciones imperativas no mantienen estructuras intermedias (*transient links*) con la relación entre los elementos origen y destino. Lo que complica el soporte a la gestión de trazabilidad.

Las aproximaciones declarativas realizan una separación sintáctica entre los constructores de los elementos origen y destino. Una regla de transformación en una aproximación declarativa consiste en distinguir claramente los patrones origen y los patrones destino, esto ayuda a identificar a qué modelo pertenece cada uno de los elementos en el código de la transformación. Por lo

contrario, en una transformación imperativa se puede encontrar tanto elementos del modelo origen como del modelo destino mezclados en el código.

En cuanto a la forma de programar las reglas, también hay una notable diferencia. En la aproximación declarativa no es necesario planificar el orden de ejecución de las reglas. Es el motor el que se encarga de seleccionar las reglas que se deben ejecutar de acuerdo a los elementos del modelo origen. En contra posición, en las aproximaciones imperativas se debe especificar de manera explícita el orden de ejecución de las reglas, por lo que el desarrollador de las transformaciones debe, al especificar la transformación, definir correctamente qué elemento debe ser transformado en primer lugar y así sucesivamente.

Por último, las aproximaciones imperativas se centran en la creación de los elementos en el modelo destino, sin tener en cuenta las relaciones que se deben mantener entre los elementos del modelo origen y los elementos del modelo destino. Esto dificulta la definición de transformaciones actualizables que faciliten la propagación de cambios, ya que no se cuenta con la información del elemento del modelo origen a partir del cual se ha generado un determinado elemento del modelo destino.

A modo de resumen, el estilo imperativo resulta apropiado para escenarios simples [65]. La transformación de una clase a una tabla de una base de datos relacional es un ejemplo representativo de esto. Cuando se codifica la regla que transforma las clases, es necesario acceder a todos los elementos de la clase, propiedades, métodos y asociaciones, para poder invocar a las reglas que los transforman. Si el meta-modelo es complejo, el hecho de tener tantas llamadas dentro de una misma regla hace que la transformación en sí se vuelva compleja. Si bien, el estilo declarativo es más conveniente para soportar la propagación de cambios y el mantenimiento de la trazabilidad [184]. Sin embargo, no se debe menospreciar el uso del estilo imperativo, ya que por ejemplo, para expresar transformaciones con una gran diferencia estructural entre los meta-modelos origen y destino es necesario utilizar construcciones imperativas, ya que poseen una mayor expresividad. En otras palabras, los lenguajes imperativos facilitan la construcción rápida de modelos; su naturaleza los hace más fácil para los desarrolladores acostumbrados a trabajar con GPLs (*General Purpose Language*), mientras que los lenguajes declarativos ofrecen la manera de vincular semánticamente los elementos de los modelos y facilitan la mantenibilidad.

Para concluir, si bien los lenguajes declarativos facilitan la definición de transformaciones de modelos, en muchos casos, es necesario utilizar algunas construcciones imperativas para evitar la definición de transformaciones muy complejas, por lo que para el desarrollo de esta tesis se considera apropiado seguir un enfoque **híbrido**, donde se combinan el estilo declarativo y el imperativo dándole mayor peso al declarativo.

Antes de comenzar con la revisión sistemática de los lenguajes de transformación de modelos que siguen la aproximación híbrida se considera necesario presentar el estándar propuesto por OMG para la especificación de transformaciones de modelos. A partir de la publicación de dicho estándar han surgido numerosas implementaciones del mismo. En la siguiente subsección se presentará, en primer lugar, un breve resumen de los conceptos principales del estándar y posteriormente, se explicarán algunas de las implementaciones existentes actualmente. Sin embargo, es importante mencionar, que no existe un consenso en común en las diferentes implementaciones de QVT y esto ha sido contraproducente para la adopción definitiva de QVT. Además, como se verá cuando se aborde la evaluación de cada una de las implementaciones, éstas han demostrado que el estándar presenta algunos puntos difusos.

2.2.2 Estándar QVT

El estándar *Query/View/Transformations* (QVT, [158]) es una familia de lenguajes para la definición de transformaciones. Está definido por dos lenguajes a nivel de usuario, *QVT-Operational Mappings* (QVTo) y *QVT-Relations* (QVTr), más un lenguaje de bajo nivel, que puede ser visto como el *byte-code* de QVT, *QVT-Core* (QVTc). En la Figura 2-2 se muestra la arquitectura general de QVT.

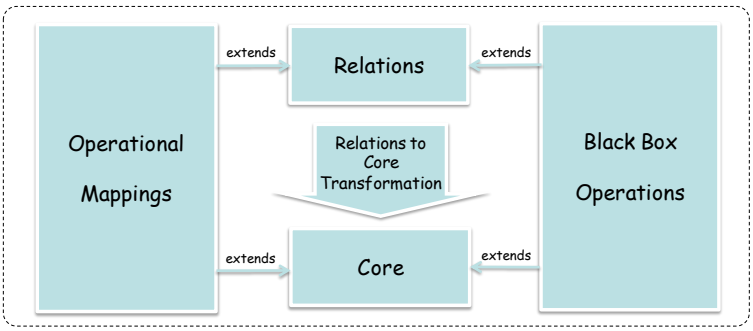

Figura 2-2. Arquitectura de QVT

- *QVT-Core* es un lenguaje relacional (declarativo) que provee un conjunto básico de constructores que permiten definir los patrones de origen y destino y las variables de conexión entre los mismos. *QVT-Core* es la base de los otros dos lenguajes y, en general, no es usado de manera directa por el desarrollador.
- *QVT-Relations* es otro lenguaje declarativo definido en base a *QVT-Core*. Brinda soporte para especificar expresiones complejas y provee con una notación gráfica.
- Por último, *QVT-Operational Mappings* es un lenguaje imperativo que extiende a los otros dos.

Las operaciones de caja-negra (*black-box*) permiten implementar la llamada de cada uno de los lenguajes desde programas externos durante la ejecución.

Hasta que la versión final de la especificación de QVT fue presentada, han surgido numerosos proyectos enfocados en la construcción de un motor de transformación de modelos que cumpla con el QVT RFP [159]. Con la llegada de la especificación final, muchos de estos proyectos fueron abandonados mientras que aparecían nuevos proyectos. Hasta el momento de escritura de esta tesis, ninguno de estos proyectos ha sido capaz de proporcionar una implementación completa, es decir, que incluya los tres lenguajes propuestos en QVT. Sin embargo, se debe mencionar, que existen algunos trabajos que contribuyen a mejorar la especificación.

En las siguientes subsecciones, se mostrarán las implementaciones más relevantes, desde nuestro punto de vista, de los lenguajes que componen QVT.

Implementaciones de *QVT-Relations*. A continuación se presentaran los principales proyectos que se centraron en la implementación del lenguaje *QVT-Relations*.

- **mediniQVT** [96] es un producto comercial desarrollado por la empresa ikv++ e integrado en la plataforma de Eclipse que, hasta el momento, es la implementación más estable y madura de *QVT-Relations*. Existe una versión libre disponible, para fines no comerciales, como licencia pública de Eclipse.

  mediniQVT incluye un editor que permite implementar las transformaciones en forma sencilla, e incluye facilidades como un depurador gráfico, auto-completado de código y resaltado de sintaxis.

Permite implementar transformaciones bidireccionales, aunque presenta algunos inconvenientes cuando se desean implementar transformaciones bidireccionales complejas.

La documentación disponible es insuficiente, de hecho está reducida a la especificación del estándar *QVT-Relations* más algunas guías de cómo usarlo disponible en Eclipse GUI. Los grupos de usuarios disponibles no son muy activos.

- **ModelMorf** [186] es otra implementación comercial de *QVT-Relations* desarrollada por la empresa TRDCC, una subsidiaria de la empresa *TATA Consulting Services*.

  Es un motor de transformaciones poderoso y de fácil uso que permite automatizar la tarea de modelado de sistemas, incluyendo las transformaciones entre modelos. Está integrado en la plataforma de Eclipse, pero, a pesar de esto, no hace uso de EMF, por lo que los modelos y meta-modelos deben ser definidos con su propia herramienta de modelado.

  Actualmente, ModelMorf, no implementa todas las características de *QVT-Relational*. En particular, no soporta la ejecución de transformaciones incrementales, no permite extender las transformaciones y no brinda soporte para la característica de sintaxis gráfica.

  ModelMorf no ofrece ninguna interfaz gráfica para implementar las transformaciones. Las mismas deben ser creadas usando cualquier editor textual y ejecutadas después mediante una llamada con paso de parámetros. Dado que no proporciona una IDE para implementar las transformaciones, los errores son encontrados cuando se ejecuta la misma.

  Su principal inconveniente es que, actualmente, el proyecto parece estar abandonado, la última versión disponible en su sitio Web es la versión 3 publicada en diciembre del 2006 y no hay indicios claros de una próxima publicación. Además la documentación disponible es muy poca y no se ha podido obtener una versión del motor funcionando correctamente.

- **MOMENT-QVT** [46] es un prototipo integrado en el marco de trabajo de MOMENT [45] que provee una implementación parcial de *QVT-Relational* basado en la reescritura de términos de MAUDE [58]. Dado que MOMENT trabaja con modelos EMF, MOMENT-QVT permite definir transformaciones entre modelos EMF.

Proporciona un editor de *QVT-Relations* que brinda facilidades como resaltado de sintaxis y analizador de errores. La transformación de modelo se define utilizando la sintaxis concreta de *QVT-Relations*.

MOMENT-QVT proporciona soporte para la trazabilidad, registrando, en un modelo, la relación entre los elementos del modelo destino y del modelo origen; esto es, qué elemento del modelo destino fue generado a partir de un determinado elemento del modelo origen. Este modelo de trazabilidad se genera de manera automática durante la ejecución de la transformación.

- ***Declarative QVT*: *QVT-Relations* en Eclipse M2M.** El proyecto Eclipse M2M (http://www.eclipse.org/m2m/) es un subproyecto de *Eclipse Modelling Project* (EMP) que provee un marco de trabajo para los lenguajes de transformación de modelo a modelo. Actualmente, existen tres motores de transformación desarrollados dentro del dominio de este proyecto: ATL (que se verá en las próximas secciones), *Procedural QVT* (implementa el lenguaje *QVT-Operational Mappings*) y *Declarative QVT* (implementa el lenguaje *QVT-Relational* y *QVT-Core*).

  Declarative QVT es un subproyecto de Eclipse M2M que tiene por objetivo proporcionar una implementación de *QVT-Relational*. La propuesta del proyecto M2M de Eclipse decía que "una aplicación ejemplar debe estar formada por el lenguaje *QVT-Core*, utilizando EMF como implementación de *MOF Essential* y la implementación de OCL, del subproyecto OCL de Eclipse. El principal resultado de esta parte del proyecto será un motor de ejecución que soporte las transformaciones. El motor ejecutará el lenguaje *QVT-Core* interpretando o compilando cada transformación. A partir de *QVT-Core*, el proyecto M2M de Eclipse, proporcionará una implementación del lenguaje de *QVT-Relations*, basado en el motor *QVT-Core*, EMF y OCL".

  El proyecto *QVT-Relations* (QVTR) ha sido inicialmente desarrollado por *Compuware*. A partir de Julio del 2007 *Compuware* ha cedido el proyecto a la empresa *Obeo* (socio industrial de *AtlanMOD*, uno de los grupos de investigación encargado de ATL).

Implementaciones de *QVT-Operational Mappings*. A continuación se presentarán los principales proyectos que implementan el lenguaje *QVT-Operational Mappings.*

- **SmartQVT** [85] es una implementación de código abierto desarrollada en JAVA de *QVT-Operational Mappings*, construído sobre EMF. Actúa como

un compilador: el código de QVT es compilado a código fuente de JAVA. Esto se logra mediante una arquitectura de dos capas:

El compilador de QVT convierte la sintaxis textual de QVT a la representación correspondiente en términos del metamodelo QVT; esto es, se construye un modelo de transformación de sintaxis abstracta a partir del código QVT.

El compilador de QVT traduce el modelo QVT a un programa en JAVA. Utiliza EMF generando APIs para cada meta-modelo que participe en la transformación.

De esta maneta, el compilador de SmartQVT puede ser utilizado en conexión con otra herramienta capaz de producir modelos QVT conformes al meta-modelos de QVT [158]. Además, se podrían exportar transformaciones en QVT conformes al metamodelos de QVT.

La primera versión de *SmartQVT*, basada en *Python*, no brindaba soporte para la detección de errores, pero las últimas versiones sí lo hacen. Además incorpora la facilidad de resaltado de sintaxis.

Por último, SmartQVT brinda soporte para el mecanismo de resolución tardía (*late resolve*) que se comporta de la misma manera que el *resolveIn* con la diferencia que la operación de resolución es realizada al finalizar la transformación.

- **Borland Together / QVTO** [44] es una suite, comercial, de productos, basada en Eclipse y alineada con el estándar de OMG. Entre otras cosas, tiene editores para UML y BPMN (*Business Process Modeling Notation*), traductor de BPEL4WS (*Business Process Execution Language for Web Services*), permite definir restricciones en OCL y por último, una de las primeras implementaciones de *QVT-Operational Mappings*.

  Además, *Together* permite implementar transformaciones de modelo a texto, usando plantillas.

- ***Procedural QVT*: *QVT-Operational Mappings* en Eclipse M2M.** *Borland* también contribuye activamente con el proyecto de QVTO M2M de Eclipse, cuyo principal objetivo es contribuir con una implementación de código abierto de *QVTo-Operational Mappings*. En particular, *Borland* ha desarrollado un editor de texto, un *parser* y un intérprete para *QVT-Operational Mappings*, que permite generar archivos con extensión ".qvto" con algunas facilidades como soporte para *hyperlinks*.

En realidad, *Together* utiliza QVTO del proyecto M2M, pero incorpora algunos complementos comerciales como el depurador o el auto-completado de código. Aunque al trabajar sólo con la versión de QVTO se pierden las ventajas mencionadas del producto de *Together*, también se obtiene acceso a nuevas características no disponibles en la misma.

- **QVTo de OpenCanarias.** Por último, la empresa española OpenCanarias S.L., ha desarrollado una máquina virtual de código fuente abierta que implementa *QVT-Operational Mappings* [175]. Dicha máquina virtual está basada en ATC (*Atomic Transformation Code*) [83], un lenguaje de transformacion de bajo nivel, desarrollado sobre Eclipse y EMF. El principal objetivo ha sido brindar soporte a QVT, pero desde un enfoque indirecto para evitar el coste asociado a los posibles cambios en la especificación del mismo.

  La idea es inyectar especificaciones de QVTo en modelos QVTo. Dichos modelos son transformados, de manera transparente para el usuario final, a un modelo ATC (que opera a modo de *byte-code*) que es ejecutado en la máquina virtual de ATC. De la misma manera, cualquier otro lenguaje de transformación podría ser también ejecutable en la máquina virtual de ATC.

  Es importante mencionar, que durante el desarrollo de QVTo de OpenCanarias, se ha contribuido a la resolución de algunos *bugs* en otros proyectos de Eclipse, como por ejemplo el de OCL.

  Además, el grupo de desarrollo de OpenCanarias, está trabajando en brindar soporte a *QVT-Core* como un paso intermedio hacia la implementación de *QVT-Relations*. Sin embargo, este proyecto será mucho más difícil, debido a la naturaleza imperativa de ATC, en contraste con la naturaleza declarativa de *QVT-Relations*.

Resumen de las implementaciones de QVT

Tras la evaluación de los distintos motores que implementan QVT se puede mencionar:

Los motores más maduros de QVT, hasta el momento de la escritura de esta tesis, provienen de la industria. Lo que implica, que es necesario tiempo, para obtener un motor de QVT de código fuente abierta.

La mayoría de las implementaciones están desarrolladas sobre Eclipse y utilizan el *framework* de EMF como plataforma para administrar los modelos.

Por último, no está claro cuál es la mejor opción para implementar el motor de transformaciones de QVT: algún tipo de implementación por medio de una

máquina virtual (MOMENT, Declarative QVT o QVTo de OpenCanarias), o una implementación directa (mediniQVT, ModelMorf, SmartQVT). Como es obvio, las implementaciones de QVT necesitan más tiempo para conseguir ser realmente útiles y confiables y poder ser utilizadas en proyectos reales.

Con el objetivo de determinar las principales consideraciones que se deberán tener en cuenta a la hora de dar soporte a la aproximación híbrida se realiza una revisión de los lenguajes que siguen dicha aproximación. Para ello, en la sección 2.2.3 se presentan un conjunto de características a evaluar en cada uno de los lenguajes; posteriormente, en la sección 2.2.4 se presenta el estudio realizado de los diferentes lenguajes de transformación.

2.2.3 Características a Evaluar

En esta sección se presentan el conjunto de características que serán evaluadas en cada uno de los lenguajes de transformación. Estas características han sido seleccionadas en base a la experiencia adquirida en la implementación de transformaciones de modelos.

- **Meta-modelo.** Es necesario determinar si los lenguajes están desarrollados aplicando los principios de MDE. Es decir, si tiene un meta-modelo que lo defina y que permita definir las transformaciones como modelos conformes a dicho meta-modelo.
- **Diferenciación en Tipos de Reglas.** Si el lenguaje permite definir diferentes tipos de reglas para especificar las transformaciones.
- **Cardinalidad de los Elementos de Entrada/Salida de las Reglas.** Es necesario determinar si el lenguaje permite definir reglas de transformación entre múltiples elementos de entrada y múltiples elementos de salida.
- **Funciones Auxiliares.** Si el lenguaje brinda mecanismos para definir funciones auxiliares que pueden ser utilizadas en el momento de especificar las reglas.
- ***Framework*.** Con el objetivo de facilitar la integración de la herramienta, es necesario determinar el *framework* de meta-modelado del lenguaje, de esta manera se determina qué tipos de modelos se pueden procesar con la herramienta, es decir, modelos construidos sobre que plataformas.
- **Documentación.** Otro factor clave para determinar el lenguaje de transformación que soportará MeTAGeM es la documentación disponible del mismo. En realidad, la mayoría de los lenguajes tienen muy poca a ninguna documentación disponible, debido a que el tiempo que los desarrolladores

dedican a la generación de documentación es muy reducido, ya que suelen centrar sus esfuerzos en la mejora y evolución del motor.

- **Nivel de Adopción / Comunidad de Usuarios.** Otro factor determinante, directamente relacionado con el anterior, es el nivel de adopción del lenguaje por los desarrolladores de transformaciones en el ámbito de MDE. Cuanto mayor es la comunidad de usuarios de un lenguaje, mayor es la información disponible sobre él. Así, fuentes de información como casos de estudio, listas de correo o *Wikis* pueden resultar de mayor utilidad que los manuales o tutoriales elaborados por los propios desarrolladores.
- **Mecanismos de Extracción/Inyección de Código.** La idea de aplicar los principios de la MDE al desarrollo de transformaciones implica la necesidad de disponer de:
    - Extractores de código, que generen el código de la transformación en un determinado lenguaje, a partir de un modelo de bajo nivel que represente dicha transformación.
    - Inyectores de modelos, que generen el modelo de la transformación a partir del código que la implementa.

La Tabla 2-5 muestra, para cada una de las características definidas, el conjunto de valores que puede tomar en cada caso.

En el contexto de MDE parece ser evidente la necesidad de definir una manera efectiva para realizar las transformaciones de modelos implícitas en cualquiera de sus procesos. Obviamente, como se ha dicho anteriormente, se podría optar por usar GPLs, como JAVA o C#, en combinación con APIs para EMF y modelos *.Ecore* para codificar la transformación de modelos. Sin embargo, ésta sería una tarea muy tediosa y difícil, cuyo desarrollo y coste de mantenimiento no compensa frente a los beneficios obtenidos.

En respuesta a esta necesidad, durante los últimos años, han surgido una serie de motores de transformación de modelos. En [35, 48] se pueden encontrar un conjunto de los motores más utilizados, que cubren una amplia gama de los enfoques existentes: propuestas basadas en el uso de gramáticas de grafos, como [10, 53, 63, 74]; propuestas centradas en la definición de DSLs (*Domain Specifics Language*s, [142]) para transformaciones de modelos [29, 60, 86, 87, 105, 129, 130, 173, 181]; herramientas CASE propietarias de un lenguaje de transformación de modelos [13, 59, 108, 114]; o motores de transformación de modelos que realizan la traducción de las reglas de transformación a especificaciones algebraicas expresadas en lenguajes formales [45, 132], entre otros.

Tabla 2-5. Características Evaluadas en los Lenguajes de Transformación de Modelos

Característica	Valor
Meta-modelos: 0 – 2	0 – No menciona la necesidad de especificar un meta-modelo que defina el lenguaje de transformación 1 – Se menciona la necesidad de especificar un meta-modelo que defina el lenguaje de transformación pero no lo tiene implementado 2 – Tiene implementado un meta-modelo que define el lenguaje de transformación
Tipos de Regla: 0 – 1	0 – No hace diferencia en tipos de regla 1 – Tiene definido diferentes tipos de regla
Cardinalidad de las Reglas: 1:1 / 1:N / N:1/ N:M/ 0:N	1:1 – Las reglas se definen sobre 1 elemento origen y generan 1 elemento destino 1:N – Se pueden generan múltiples elementos destino a partir de 1 elemento origen N:1 – Se puede generar 1 elemento destino a partir de múltiples elementos origen N:M – Se pueden generar múltiples elementos destino a partir de múltiples elementos origen 0:N – Se pueden generar elementos en el modelo destino de forma espontánea
Funciones Auxiliares: 0 - 1	0 – No brinda soporte para definir funciones auxiliares 1 – Brinda soporte para definir funciones auxiliares
Framework: Nombre del framework	-
DOCUMENTACIÓN: 0 – 2	0 – No existe documentación disponible. 1 – La documentación disponible es incompleta 2 – Existe documentación disponible muy completa y concisa
Nivel de Adopción: 0 – 2	0 – Bajo grado de adopción 1 – Grado de adopción medio 2 – Alto grado de adopción
Mecanismos de Extracción de Código: 0 – 1	0 – No brinda mecanismos de Extracción/Inyección de código 1 – Brinda mecanismos de Extracción/Inyección de código

Para determinar el conjunto de lenguajes a evaluar en esta tesis, se ha llevado a cabo un proceso de revisión sistemática. A continuación se presenta, de manera resumida, el proceso seguido en la misma. Los lenguajes que han sido seleccionados para su estudio, son aquellos que han sido adoptados con mayor

frecuencia y aquellos que, aunque no tan exitosos, presentan un interés especial desde el punto de vista de investigación.

2.2.4 Revisión Sistemática

En esta sección se presenta de manera resumida el proceso de revisión sistemática para el estudio de los lenguajes de transformación de modelos que siguen la aproximación híbrida, para consultar la revisión completa ver anexo B.

Etapa de planificación de la revisión:

Objetivo (*Question*): el principal objetivo de esta revisión sistemática ha sido evaluar los lenguajes de transformación que siguen la aproximación híbrida, y las herramientas que los implementan, existentes en el ámbito del MDE.

Palabras Clave: Una vez definido el objetivo, se realiza la selección de palabras claves que guiarán el proceso de búsqueda de los lenguajes a evaluar. Para realizar esta revisión sistemática se han seleccionado las siguientes palabras claves: *approach*, *hybrid*, *language*, *transformation*, *model*, *tool*, MDA, MDE.

Los resultados esperados al finalizar esta revisión sistemática, serán, entre otros, el de contar con un conjunto de lenguajes y herramientas de transformaciones que puedan ser evaluados aplicando las características definidas anteriormente.

Selección de fuentes de búsqueda: a partir de las palabras claves utilizadas para realizar la búsqueda y haciendo combinaciones de las mismas se han obtenido diferentes cadenas de búsquedas (Tabla 2-6).

Tabla 2-6. Cadenas Básicas de Búsqueda

Cadenas básicas de búsqueda	
1	Hybrid Approaches
2	Hybrid Transfromations Languages
3	MDA AND Hybrid Transformations Languages
4	MDE AND Hybrid Transformations Languages
5	Transformations Tools
6	Model Transformations
7	Model Transformations Languages

Las fuentes y/o motores de búsquedas seleccionadas para aplicar el proceso de búsqueda son las siguientes:

- Science@Direct
- IEEE Digital Library

- SpringerLink
- ACM Digital Library
- Journal of Computer Science
- Actas de congresos y workshops en el ámbito de MDE

Etapa de ejecución de la revisión

Selección de Estudios: como se ha dicho en la sección 1.4.1, para realizar esta tarea se propone un proceso iterativo e incremental. Es iterativo, porque la ejecución de la revisión sistemática (búsqueda, extracción de la información y visualización de los resultados) se realiza para cada una de las fuentes seleccionadas y con cada una de las cadenas de búsqueda determinadas. Es incremental, en el sentido de que el resultado final obtenido, incrementa en cada iteración hasta obtener la versión definitiva.

A partir de la búsqueda realizada se han obtenido numerosos documentos, por lo que ha sido necesario definir criterios de inclusión y de exclusión que han permitido acotar los resultados obtenidos; es decir, determinar los criterios a partir de los cuales un lenguaje será incluido, o no, en el proceso de evaluación. En la Tabla 2-7 se presentan los resultados obtenidos tras realizar la búsqueda en cada una de las fuentes.

Tabla 2-7. Distribución de Estudios Seleccionados por Fuente

Fuentes	**Estudios**				**%**
	Encontrados	**No Repetidos**	**Relevantes**	**Primarios***	
IEEE Digital Library	812	757	50	10	20
Science Direct	20.252	19.262	70	6	12
SpringerLink	50.996	30.562	63	7	14
Google Academia	1.150.000	948.635	77	15	30
ACM Digital Librery	14.185	9.623	25	12	24
Total	**1.236.245**	**1.008.839**	**3.831**	**50**	**100**

El criterio de inclusión definido ha sido el de realizar un análisis del título, el resumen y las palabras clave de los artículos obtenidos en la búsqueda. Esto sirve, en primera instancia, para determinar si los artículos pertenecen al tema que se desea evaluar e incluirlos como estudios primarios para la revisión.

Se han definido dos criterios de exclusión: el primero, que el lenguaje debía seguir la aproximación híbrida y, el segundo, que el lenguaje debe tener una herramienta que lo implemente, ya que el objetivo final de esta revisión sistemática es la de analizar los lenguajes de transformación que siguen la aproximación híbrida y las herramientas que los implementan para determinar el conjunto de características que MeTAGeM deberá soportar.

Es importante aclarar que a partir del análisis de los primeros estudios seleccionados se han obtenido nuevos artículos de lenguajes a analizar.

Etapa de análisis de resultados

Resultados obtenidos: a continuación se muestra, para cada uno de los lenguajes seleccionados, ATL, RubyTL y EpsilonTL, los resultados obtenidos tras la evaluación realizada; por último, se presentará una tabla en la que se resumen los criterios evaluados para cada uno de ellos.

2.2.4.1 ATL

ATL (*ATLAS Transformation Language*) [105] es el lenguaje de transformación de modelos desarrollado por el grupo investigación ATLAS (INRIA & LINA) como respuesta a la propuesta de OMG MOF [149]/QVT RFP [158] y forma parte de la plataforma AMMA (*Atlas Model Management Architecture*) [33]. Entre los componentes de AMMA se pueden mencionar: el lenguaje de meta-modelado *Kernel MetaMetaModel* (KM3, [25, 102]) y el lenguaje *Textual Concrete Syntax* (TCS, [104]).

ATL es un lenguaje de transformación de modelos especificado a través de un meta-modelo y su correspondiente sintaxis textual concreta. Está enmarcado en Eclipse y proporciona una IDE que incorpora las facilidades de editores, compiladores, auto-completado de código *(code completion)*, resaltado de sintaxis, registro de meta-modelos, entre otras cosas. Además, está basado en la especificación de OCL [153].

En cuanto a las características evaluadas:

- **Meta-modelo.** Tiene un meta-modelo definido, en el que se establecen el conjunto de los constructores que representan la sintaxis concreta de ATL. La sintaxis abstracta de dicho meta-modelo está implementada utilizando EMF (*Eclipse Modelling Framework*, [49, 50]) y la sintaxis concreta utilizando TCS. Esto permite definir las transformaciones de modelos, como modelos en si mismos, conformes al meta-modelo de ATL.
- **Tipos de Reglas.** Proporciona constructores para definir diferentes tipos de reglas que se corresponden con los diferentes modos de programación

soportados: las reglas *CalledRule*, que son reglas que tienen un comportamiento completamente imperativo; las reglas *MatchedRule*, con un comportamiento completamente declarativo; y las reglas *LazyMatchedRule* que son una especialización de las *MatchedRule*, por lo que también tienen un comportamiento declarativo. Además, las reglas se pueden definir como reglas abstractas, lo que permite reutilizar su funcionalidad.

- **Cardinalidad de las Reglas.** Permite definir reglas de transformación a partir de múltiples patrones de origen (*multiple in patterns*) y generar múltiples patrones de destino (*multiple out pattern*). Además, tiene un tipo especial de regla que permite generar elementos en el modelo destino sin depender de ningún elemento del modelo origen.
- **Funciones Auxiliares.** Permite definir funciones especiales llamadas *Helper*, que pueden ser considerados como los equivalentes de ATL a los métodos de Java. Los *helpers* permiten definir factorización al código ATL y pueden ser invocados desde diferentes puntos de la transformación ATL.
- ***Framework*.** El *framework* de meta-modelado de ATL es Eclipse, por lo que con ATL se pueden definir transformaciones entre meta-modelos basados en EMF.
- **Documentación.** Existe una considerable cantidad de documentación disponible. Proporciona: un manual completo [106]; un conjunto de ejemplos introductorios que cubren los aspectos básicos a la hora de desarrollar transformaciones de modelos con ATL [7]; un conjunto de meta-modelos definidos en varios formatos [20] (en el lenguaje KM3, Ecore, SQL, XMI, DSL Tools, formato XML, etc); una serie de escenarios en los que se ha utilizado ATL de forma exitosa, tanto en la investigación como en contextos industriales; además de un grupo de noticias muy activo que ayuda a resolver cualquier duda que pueda surgir con el desarrollos de las transformaciones. Todos estos recursos están disponibles en el sitio ATL (http://www.eclipse.org/m2m/atl/).
- **Nivel de Adopción.** Actualmente es considerado como el estándar de-facto en el campo de las transformaciones de modelos, ya que el estándar QVT propuesto por la OMG presenta serios inconvenientes. El hecho es que, se pueden encontrar trabajos de investigación que dicen haber utilizado QVT para realizar las transformaciones de modelos, pero en realidad sólo utilizan QVT para formalizar las transformaciones y luego para implementarlas utilizan ATL (ver [115] y [140], por ejemplo).

- **Mecanismos de Extracción/Inyección de Código**. Una de las ventajas que se tiene al implementar la sintaxis concreta de ATL con TCS es el hecho de poder contar con un mecanismo de extractor y de inyector que permite obtener el código que implementa la transformación en el lenguaje ATL a partir del modelo desarrollado y el modelo de la transformación a partir del código de la misma, respectivamente. Por lo que, usando dicho extractor se puede obtener el código implementable de la transformación.

Teniendo todo esto en cuenta, se puede decir que el lenguaje es estable, maduro y es mejorado constantemente. Además, la ausencia de una implementación de QVT de referencia, ha hecho que ATL haya sido ampliamente aceptado como estándar de facto para el desarrollo de transformaciones de modelos. Existen trabajos que intentan alinear ATL y QVT [103], implementar *QVT-Relations* basados en ATL-VM [15, 101] y otros en ese sentido [89].

2.2.4.2 EPSILON TL

Epsilon Transformation Languages (ETL) [121] es un lenguaje de transformación de modelos que forma parte del conjunto de soluciones propuestos en EPSILON (*Extensible Platform for Specification of Integrated Languages for mOdel maNagement*) [118, 120]. Epsilon es un componente del proyecto GMT de Eclipse, que provee una infraestructura para la implementación de lenguajes de administración de modelos uniformes e interoperables entre sí. Permite administrar modelos basados en diferentes meta-modelos y en distintas tecnologías.

El principal componente de Epsilon es *Epsilon Object Language* (EOL) [119], un lenguaje imperativo, basado en OCL, que provee características como, modificación de modelos, acceso a múltiples modelos, constructores de programación básicos, interacción con el usuario y soporte de transacciones entre otras cosas. Aunque EOL puede ser utilizado como un lenguaje de administración de modelos de propósito general, su principal objetivo es ser reutilizado en los diferentes lenguajes específicos de tareas propuestos implementados en Epsilon [118]; el lenguaje para comparación de modelos (*Epsilon Comparison Language*, ECL), el lenguaje de *merging* de modelos (*Epsilon Merging Language*, EML), el lenguaje de validación de modelos (*Epsilon Validation Language*, EVL), el lenguaje de refactorización (*Epsilon Wizard Language*, EWL), el lenguaje de implementación de transformaciones de modelo a texto (*Epsilon Generation Language*, EGL), y, por último, el lenguaje de implementación de transformaciones de modelo a modelo, ETL (*Epsilon Transformation Language*).

Para el desarrollo de esta tesis, interesa evaluar este último, por lo que a continuación se presentará las principales características del mismo.

El principal objetivo de ETL, es contribuir a la capacidad de implementar las transformaciones de modelo a modelo de Epsilon. Al basarse en Epsilon, ETL logra una coherencia sintáctica y semántica, más la interoperabilidad con los otros lenguajes soportados por Epsilon.

En cuanto a las características evaluadas:

- **Meta-modelo.** Tiene especificado un meta-modelo, como una extensión del meta-modelo de EOL, que representa el conjunto de elementos que utiliza, sin embargo, dicho meta-modelo no se encuentra implementado, por lo que no se pueden definir modelos conformes a dichos meta-modelos aplicando los principios de MDE.
- **Tipos de Reglas.** Proporciona constructores para definir diferentes tipos de reglas. Las reglas de tipo abstractas que posibilitan la reutilización de la funcionalidad especificada. Las reglas de tipo principales (*primary*) que son reglas que presentan un comportamiento declarativo y las reglas de tipo *lazy* que deben ser invocadas explícitamente.
- **Cardinalidad de las Reglas.** La cardinalidad de los elementos que participan en una regla es de 1 – N, es decir a partir de un elemento origen se generan uno o varios elementos en el modelo destino.
- **Funciones Auxiliares.** Permite la definición de funciones auxiliares (*Operation*), lo que facilita la modularidad del código que se escribe.
- ***Framework*.** Forma parte del conjunto de soluciones propuestas en Epsilon que es un componente del proyecto GMT de Eclipse, que provee una infraestructura para la implementación de lenguajes de administración de modelos uniformes e interoperables entre si. Permite administrar modelos basados en diferentes meta-modelos y en distintas tecnologías.
- **Documentación**. Como punto negativo, se puede decir que la documentación disponible no es suficiente.
- **Mecanismos de Extracción/Inyección de Código**. No tiene implementados mecanismos de extracción e inyección de código.
- **Nivel de Adopción.** El nivel de adopción de ETL es medio, al ser un lenguaje relativamente joven aún está en proceso de desarrollo, por lo que su grupo de usuarios aún no es muy grande.

ETL es capaz de transformar un número arbitrario de modelos origen en un número arbitrario de modelos destino; adopta un estilo híbrido y cuenta con especificación de la regla declarativa permitiendo definir pre y post condiciones, reglas abstractas, reglas principales y reglas de tipo *lazy*, permite implementar además mecanismos de trazabilidad. Además, como ETL se basa en EOL reutiliza sus características imperativas, permitiendo a los usuarios especificar reglas de transformación más complejas.

2.2.4.3 RubyTL

RubyTL [176, 178] es un lenguaje de transformación de modelos basado en reglas, desarrollado por el grupo de investigación **Tecnología del Software** de la Universidad de Murcia. Ha sido desarrollado como un DSL embebido en el lenguaje de programación dinámico Ruby [172], lo cual influencia en su sintaxis concreta.

RubyTL provee una IDE basada en Eclipse, llamada AGE [177], que incluye un editor Ruby con resaltado de sintaxis. Además, proporciona un mecanismo de extensión que posibilita que un conjunto básico de características pueda ser extendido con nuevas características mediante la creación de *plug-ins* que implementen algunos de los puntos de extensión predefinidos.

Es un lenguaje de transformación híbrido e incluye algunas características importantes como la organización de las reglas en fases [178]. Además, si se especifica en el momento de la configuración de la ejecución de la transformación, permite modificar el modelo de origen.

Según sus autores, RubyTL ha sido creado para satisfacer tres requisitos principales: i) ser un lenguaje híbrido, ya que la expresividad declarativa es poco apropiada para transformaciones complejas que pueden requerir un estilo imperativo; el estilo declarativo debería estar basado en una operación *binding* similar a la proporcionada por ATL, ii) facilitar la experimentación con características del lenguaje, y iii) permitir una rápida implementación. Estos requisitos se han satisfecho a través de dos principales decisiones de diseño: la definición del lenguaje como un DSL embebido en Ruby y la incorporación de un mecanismo de extensión basado en *plug-ins* que permite añadir nuevas características a un núcleo básico.

- **Meta-modelo.** En la documentación disponible del lenguaje se puede encontrar varias especificaciones, diferentes, del meta-modelo de RubyTL. Al ser un lenguaje definido, embebido en el lenguaje Ruby, no tiene implementado su meta-modelo, por lo que el editor proporcionado por la

herramienta no permite definir las transformaciones como modelos de transformaciones.

- **Tipos de Reglas.** Permite especificar diferentes tipos de transformaciones. Las reglas de tipo *TopRule*, que son las reglas principales a partir de la cual se ejecutan el resto de las reglas. Las reglas de tipo *NormalRule*, que son un tipo de regla especial que debe ser invocada de forma explícita para ejecutarse. Por último, las reglas de tipo *CopyRule*, que tiene un comportamiento similar a las *NormalRule*, la diferencia radica en que permite transformar más de una vez el mismo elemento origen.
- **Cardinalidad de las Reglas.** RubyTL permite definir reglas con cardinalidad 1 a N. Es decir, a partir de un elemento del modelo origen se pueden generar 1 o varios elementos del modelo destino.
- **Funciones Auxiliares.** Tiene un tipo especial de regla, llamada *Decorator*, que se utiliza para realizar la implementación de funciones auxiliares.
- ***Framework*.** Provee una IDE basada en Eclipse, llamada AGE, que incluye un editor Ruby con resaltado de sintaxis, plantillas de código y autocompletado de código. Si bien no fue desarrollado para funcionar como parte de EMF, funciona de manera eficiente con modelos *Ecore* importados.
- **Documentación**. Proporciona amplia documentación, como manuales de usuarios, casos de estudios, entre otras cosas.
- **Mecanismos de Extracción/Inyección de Código**. No tiene implementado mecanismos de extracción o inyección de código.
- **Nivel de Adopción.** El nivel de adopción de RubyTL es medio. Si bien existen algunos casos donde se utiliza RubyTL en conjunto con otros lenguajes de transformación [175], no es utilizado por la mayoría de la comunidad de usuarios de MDE.

Para resaltar las principales características de RubyTL, se puede decir que es un lenguaje de transformación de código abierto e híbrido (aunque se recomienda el uso del estilo declarativo), que también brinda soporte para la generación de código a través de DSLs más plantillas de código.

En cuanto a la usabilidad, se puede decir que tiene cierto grado de complejidad, ya que es necesario conocer la sintaxis del lenguaje Ruby. No brinda soporte a la trazabilidad, aunque permite realizar transformaciones bidireccionales.

2.2.5 Conclusiones

En la Tabla 2-8 se presenta la evaluación de cada una de las características en los lenguajes de transformación híbridos analizados. Este análisis permite determinar el conjunto de características que se deberán tener en cuenta a la hora de implementar el meta-modelo para la implementación de transformaciones específicas de plataforma siguiendo la aproximación híbrida.

Como se puede observar los lenguajes soportan la definición de diferentes tipos de reglas y de funciones auxiliares, lo que facilita la codificación de las transformaciones. Si bien cada lenguaje utiliza nombres diferentes para representar los tipos de reglas, todos identifican la misma funcionalidad. A modo de ejemplo, todos permiten definir una regla de tipo abstracta o una regla de tipo principal. En cuanto a la cardinalidad de los elementos en una regla de transformación, solo el lenguaje ATL permite definir reglas entre múltiples elementos del modelo origen y múltiples elementos del modelo destino, el resto de los lenguajes permite generar múltiples elementos en el modelo destino, pero partiendo de un solo elemento del modelo origen.

Tabla 2-8. Lenguajes de Transformación de Modelo a Modelo.

Características	ATL	EpsilonTL	RubyTL
Meta-modelos	2	1	1
Tipos de Reglas	1	1	1
Cardinalidad de Reglas	N-M	1-N	1-N
Funciones Auxiliares	1	1	1
Framework	Eclipse/EMF	Epsilon / Eclipse	Eclipse/RubyTL
Documentación	2	1	2
Mecanismos de Extracción/Inyección de Código	1	0	0
Nivel de Adopción	2	1	1

Si bien todos los lenguajes tienen un meta-modelo especificado que los define sólo el meta-modelo de ATL está implementado, lo que implica que solo en ATL se puedan definir las transformaciones de modelos como modelos en sí mismos. Otro punto a favor de ATL es el hecho de contar con mecanismos de extracción e inyección de código.

En cuanto a la documentación disponible de cada lenguaje, como se puede observar, ATL es el que más documentación ofrece. Esto está relacionado con el hecho de que es el lenguaje más utilizado por los desarrolladores de la comunidad de MDE.

Solución: MeTAGeM un entorno de desarrollo de transformaciones de modelos a alto nivel

Tal como se ha dicho en el capítulo 1, el objetivo principal de esta tesis es la definición de un entorno que, aplicando los principios del MDE, facilite el desarrollo (semi-)automático de transformaciones de modelos mediante la especificación de las mismas en un lenguaje independiente de plataforma y su posterior transformación a un lenguaje específico de plataforma.

El entorno de desarrollo de transformaciones de modelos, dirigido por modelos, (MeTAGeM, *A Meta-Tool for Automatic Generation of transformation Model*) que se define en esta tesis está compuesto por:

- Una **metodología**, que consta de:
    - Un proceso de desarrollo de transformaciones dirigido por modelos.
    - Diferentes meta-modelos que dan soporte al proceso definido, y que permitan realizar el modelado de transformaciones en los diferentes niveles propuestos en el proceso.
    - La especificación de un conjunto de reglas de transformación que permitirán obtener de forma (semi-)automática los modelos conformes a los diferentes meta-modelos propuestos en cada nivel.
- Una **herramienta** que da soporte a la metodología propuesta y que se compone de:
    - Una arquitectura que define cada uno de los componentes que forman la herramienta.
    - Un conjunto de editores que permiten el modelado de las transformaciones de modelos conformes a los meta-modelos definidos por la metodología.
    - Un meta-transformador que permite realizar las transformaciones entre los modelos definidos en los diferentes niveles de abstracción y la posterior generación de código. El meta-transformador consta de diferentes conjuntos de reglas de transformación.

A continuación, se presenta, por un lado, la metodología definida explicando cada uno de sus componenetes, y, por otro lado, la herramienta que da soporte a dicha metodología.

3.1 Metodología

En el capítulo 1 se han detectado una serie de problemas con los que se encuentran los desarrolladores a la hora de implementar transformaciones de modelos:

En primer lugar, uno de los principales problemas es el de la selección del lenguaje y la herramienta. Actualmente, existen muchos lenguajes y herramientas de transformación de modelos (ver Sección 2.2), cada uno con su propio modo de implementación de transformaciones. Esta diversidad de lenguajes y herramientas obliga a invertir en aprendizaje un tiempo directamente proporcional a la complejidad del lenguaje elegido, lo que incrementa el tiempo de desarrollo de la transformación.

MeTAGeM, el entorno para el desarrollo de transformaciones de modelos dirigido por modelos que se define en esta tesis, proporciona mecanismos para especificar la transformación a un alto nivel de abstracción. Para ello permite modelar la transformación sin más que definir las relaciones que deben satisfacerse entre los elementos de los meta-modelos origen y destino, sin tener en cuenta la plataforma final en que se implementará la transformación. Estos modelos de transformación independientes de plataforma se transforman en modelos específicos de plataforma, que representan la transformación utilizando las construcciones propias de la aproximación escogida (declarativa, imperativa, etc.). Finalmente, a partir de estos modelos se generan los modelos dependientes de plataforma, que expresan la transformación en términos del lenguaje de transformación seleccionado, de entre los que siguen la aproximación escogida previamente. Este último modelo es directamente serializable en código que implementa la transformación para el lenguaje seleccionado. De esta manera, utilizando MeTAGeM, el desarrollador sólo debe centrarse en identificar correctamente las relaciones entre los elementos de los meta-modelos que participan en la transformación y no en cuestiones técnicas de implementación, ya que la obtención del código que implementa la transformación se hace de forma (semi-)automática por medio de transformaciones de modelos.

Otro de los problemas más importantes señalados en el capítulo 1 es el de la escasa interoperabilidad entre las diferentes herramientas. La mayoría de ellas están asociadas a un lenguaje de transformación en particular, lo que impide trabajar con modelos generados por otras herramientas, esto es, muchas veces es necesario definir previamente con la herramienta los meta-modelos que se quieren transformar antes de poder comenzar a especificar la transformación. Más aún, la mayoría de las herramientas estudiadas no incluyen mecanismos de importación

y/o exportación de modelos. En nuestra opinión, uno de los mayores inconvenientes para mejorar el nivel de interoperabilidad es el hecho de que son muy pocos los lenguajes o herramientas que aplican los principios de la MDE al desarrollo de las transformaciones. Para constatar esta situación, sirva de ejemplo el hecho de que la mayoría de ellos no cuenta con un meta-modelo que permita definir las transformaciones de modelos como modelos en si mismos.

De acuerdo a los principios de MDE, las transformaciones, como cualquier otro producto software, deberían ser tratadas como modelos, de manera que pudieran validarse, transformarse o someterse a cualquier otro procesamiento que se realice con un modelo. Para poder gestionar las transformaciones como modelos, MeTAGeM define diferentes meta-modelos que permiten modelar las trasnformaciones a diferentes niveles de abstracción. Todos los meta-modelos de MeTAGeM se han definido conformes a MOF, lo que asegura la interoperabilidad con cualquier herramienta del ámbito de la MDE basada en MOF. Así, cualquiera de los modelos elaborados o generados con MeTAGeM podrá ser utilizado y/o gestionado desde cualquier otra herramienta (basada en el uso de modelos) sin más que desarrollar la transformación que permita conectar los meta-modelos de ambas herramientas.

En las siguientes secciones se explicarán cada uno de los componentes de la metodología que se presenta en esta tesis.

3.1.1 Proceso de Desarrollo de Transformaciones

El proceso se realiza de acuerdo a cuatro niveles de abstracción: nivel PIM, PSM, PDM y código. En la Figura 3-1 se muestra el proceso mediante un diagrama de actividad de SPEM2 [154].

En primer lugar se propone comenzar modelando las transformaciones de modelos a un alto nivel de abstracción (modelos a nivel PIM) sin contemplar cuestiones técnicas de implementación final ni la tecnología utilizada para hacerlo. Para ello, se indican las relaciones existentes entre los diferentes elementos de los meta-modelos que intervienen en la transformación.

A partir del modelo especificado a nivel PIM se genera, por medio de transformaciones (semi-)automáticas, el modelo de transformación específico de plataforma (nivel PSM) para una aproximación de transformaciones de modelos concreta (declarativa, imperativa, híbrida, etc) . Este modelo a nivel PSM se podrá refinar manualmente, pudiéndose agregar nuevos elementos, modificar características de elementos existentes o eliminar elementos que no se consideren necesarios.

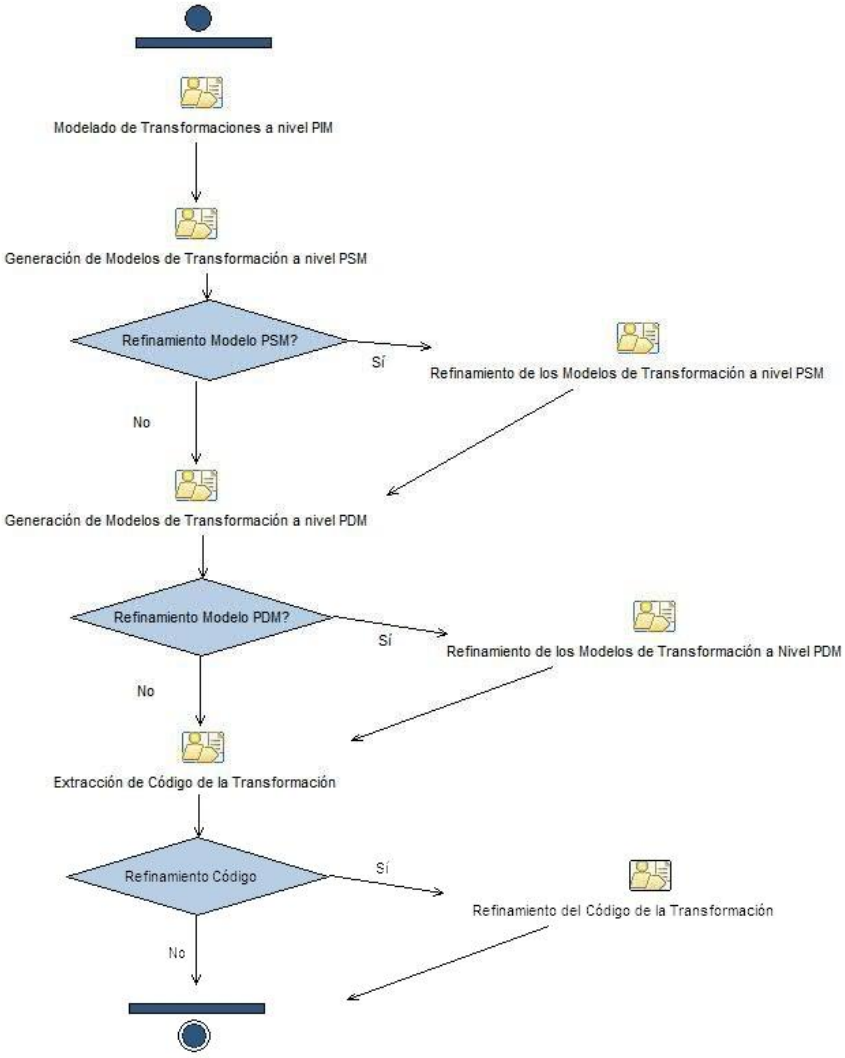

Figura 3-1. Proceso de MeTAGeM

De manera similar, a partir del modelo obtenido a nivel PSM y aplicando nuevamente transformaciones (semi-)automáticas, se genera el modelo de transformación dependiente de plataforma (a nivel PDM). Estos modelos, que también podrán ser refinados, serán conformes al meta-modelo del lenguaje de transformación que se seleccione como plataforma final de implementación de las transformaciones.

La última actividad es la de generación de código de la transformación a partir de los modelos generados a nivel PDM. Esta generación se hará de forma (semi-)automática por medio de transformaciones de modelos a texto. El código generado también puede ser refinado usando el IDE proporcionado por el propio lenguaje de transformación de modelos.

El resultado final del proceso es la obtención del código de la transformación en el lenguaje que se haya seleccionado.

3.1.2 Especificación de Meta-modelos

Para poder realizar el modelado de las transformaciones de modelos en diferentes niveles de abstracción, es necesario definir meta-modelos que permitan especificar modelos conformes a dichos meta-modelos.

En la Figura 3-2 se muestra la relación existente entre los niveles de abstracción definidos en el proceso, los diferentes meta-modelos necesarios para permitir el modelado de las transformaciones en dichos niveles, así como los modelos de las transformaciones conformes a dichos meta-modelos.

Como se puede observar, a ***nivel PIM*** se define un meta-modelo conforme a MOF, que permite modelar las transformaciones de modelos a un alto nivel de abstracción, sin tener en cuenta los detalles técnicos de la implementación final. Este meta-modelo está formado por un conjunto de constructores que permiten modelar las relaciones existentes entre los meta-modelos de origen (*Meta-Modelo-In*) y destino (*Meta-Modelo-Out*), también definidos conformes a MOF. Así, cuando se especifiquen las transformaciones de modelos, se debe definir un modelo de transformación de modelos donde se modelen solo las relaciones existentes entre los elementos de los meta-modelos origen y destino teniendo en cuenta en cada relación, la cardinalidad de los elementos participantes en la misma.

A ***nivel PSM***, se especifican diferentes meta-modelos uno por cada una de las aproximaciones (o paradigmas) de transformación de modelos existentes. En el caso particular de esta tesis, se especifica el meta-modelo de la aproximación híbrida, de la misma manera se pueden especificar los meta-modelos para el resto de las aproximaciones.

A ***nivel PDM*** se representan los modelos de transformación dependientes de plataforma. Estos modelos serán conformes a los meta-modelos de cada uno de los lenguajes de transformación en los que el desarrollador desee implementar la transformación finalmente. Para esta tesis, se seleccionan como lenguajes de

transformación a nivel PDM; ATL y RubyTL. De esta manera, a nivel PDM se obtienen el modelo de transformación representado como un modelo ATL, conforme al meta-modelo de ATL; o como un modelo RubyTL, conforme al meta-modelo de RubyTL.

Por último, a **nivel de código**, se obtiene, a partir de los modelos definidos a nivel PDM, el código que implementa la transformación en el lenguaje de transformación de modelos seleccionado previamente en el nivel anterior. En este caso se obtiene el código ATL ó RubyTL.

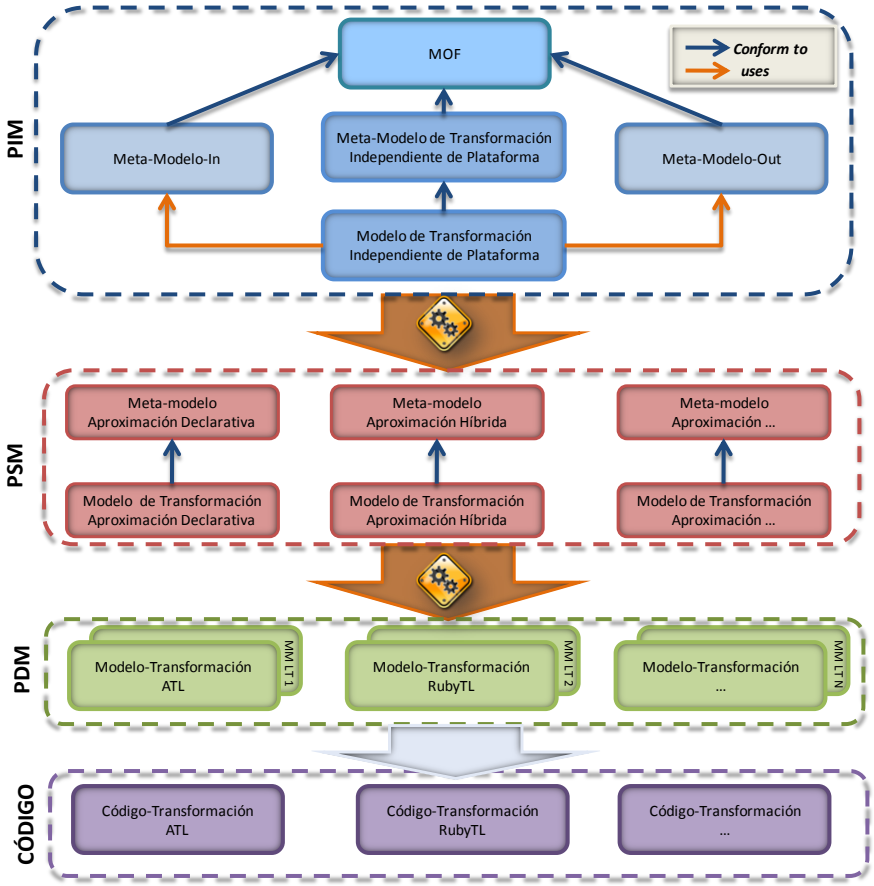

Figura 3-2. Meta-modelos de MeTAGeM

A continuación se muestra la especificación de cada uno de los meta-modelos que permitirán modelar las transformaciones a nivel PIM, a nivel PSM y a nivel PDM.

3.1.2.1 Meta-modelo de Transformación Independiente de Plataforma

Uno de los principales objetivos planteados para el entorno de desarrollo MeTAGeM es que sea lo suficientemente general como para poder ser aplicado a transformaciones de cualquier dominio, y de manera independiente del lenguaje de transformación a utilizar. Con este fin, se han identificado las técnicas o estrategias comunes, como la modularización de las reglas o la concatenación de las mismas, a la mayoría de los dominios de transformaciones de modelos a modelo. Estas técnicas y estrategias se han identificado a partir tanto, del análisis realizado de cada uno de los lenguajes, como de la experiencia adquirida al realizar transformaciones de modelo a modelo en diferentes dominios de aplicación [4, 71, 192, 193, 195, 196, 197, 198].

Como conclusión se puede decir que para que el Meta-modelo de Transformación Independiente de Plataforma (M-TIP) especificado en el nivel PIM de MeTAGeM sea genérico, es necesario identificar:

a) El tipo de elemento involucrado en la transformación.
b) El tipo de relación entre los elementos involucrados en la transformación.

3.1.2.1.1 Tipo de elemento

Un factor importante a la hora de especificar reglas de transformación entre elementos de diferentes meta-modelos es el de identificar el tipo de elemento que se está transformando. Esto es debido a que la forma de definir una regla de transformación difiere, en la mayoría de los lenguajes, de acuerdo al tipo de elemento que se está transformando.

En el ámbito del MDE la mayoría de los DSLs sobre los que se desarrollan los editores que permiten manipular modelos se definen conformes a un meta-meta-modelo común, MOF o, en el peor de los casos se pueden expresar en términos de MOF mediante la definición de una transformación de modelos. Por ello, todos los meta-modelos manejan conceptos como clases, atributos y relaciones, aunque en cada meta-modelo reciban un nombre diferente. Por ejemplo, en UML se habla de *Class*, *Property* y *Association*, mientras que en *Ecore* se habla de *EClassifier*, *EAttribute* y *EReference*, aunque son similares.

Como se ha dicho anteriormente, las reglas de transformación entre elementos de los meta-modelos difieren de acuerdo al tipo de elemento que se esté transformando. Por ejemplo, para transformar un elemento del tipo *Classifier* se deben proporcionar mecanismos que permitan realizar la transformación de los elementos que dependen directamente del elemento *Clasiffier*, como las *Property* o las *Association*. Sin embargo, la transformación de un elemento del tipo

Property o *Association* no desencadena, normalmente, ningún otro tipo de transformación adicional asociada.

3.1.2.1.2 Tipo de Relación

Otro factor importante al definir reglas de transformación es identificar claramente el tipo de relación existente entre los elementos de los diferentes meta-modelos que intervienen en la transformación. La identificación del tipo de relación:

- Facilita al desarrollador la tarea de definición de modelos de transformaciones, ya que sólo debe conocer la relación entre los elementos de los meta-modelos entre los que quiere definir la transformación.
- Ayuda a asegurar el mantenimiento de la trazabilidad entre dichos elementos, esto es, a identificar de qué elemento del meta-modelo origen deriva un determinado elemento del meta-modelo destino.

El hecho de brindar mecanismos que permitan recuperar el elemento del meta-modelo origen a partir de un elemento del meta-modelo destino es un problema al que se le viene dando mucha importancia [21, 113]. Si bien la mayoría de los lenguajes y herramientas existentes se hacen eco de este problema, no todos brindan una solución completa y en algunos casos, los mecanismos implementados no tienen el comportamiento esperado.

Siguiendo la propuesta realizada en [191] se pueden especificar las relaciones existentes entre los elementos de los diferentes meta-modelos que intervienen en la transformación mediante la identificación del número de elementos del meta-modelo origen y el número de elementos del meta-modelo destino, es decir, mediante la cardinalidad de los elementos de cada uno de los meta-modelos que intervienen en la relación. En la Tabla 3-1, se muestran los tipos de relaciones identificadas y, a continuación, se explican cada una de las relaciones.

Tabla 3-1. Tipos de relaciones entre los Elementos de los Meta-modelos

Meta-Modelo Destino / Meta-Modelo Origen	0	1	N
0	-	√	-
1	√	√	√
N	-	√	√

Relación Uno-a-Uno

Este tipo de relación se da cuando existe una correspondencia uno-a-uno entre una meta-clase del meta-modelo origen y una meta-clase del meta-modelo destino (Figura 3-3). Este es uno de los tipos de relación más común a todos los dominios de transformaciones de modelos; como se puede ver, también es el caso más simple. Las relaciones uno a uno entre los elementos facilitan el mantenimiento de la trazabilidad entre los mismos. Aunque se debe mencionar, que este tipo de relación es más común encontrarla entre meta-modelos simples o entre aquellos que son similares semánticamente.

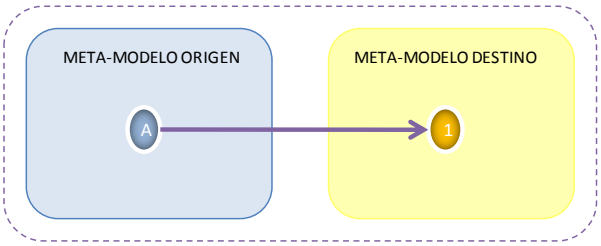

Figura 3-3. Relación Uno-a-Uno

Existe una variación de la relación uno a uno, que se da cuando un elemento del meta-modelo origen puede ser transformado en uno u otro elemento del meta-modelo de destino de forma excluyente. Como se puede ver en la Figura 3-4 el elemento A del meta-modelo origen puede ser transformado en el elemento 1 del meta-modelo destino o en el elemento 1' y estas dos opciones son mutuamente excluyentes. En estos casos lo ideal sería que el desarrollador tome la decisión de a qué elemento se debe transformar; sin embargo, la mayoría de las implementaciones proponen una solución por defecto. Si bien, en esencia, el tipo de relación es la misma que la presentada en la Figura 3-3, es necesario tenerla en cuenta para tener mecanismos que permitan al desarrollador tomar la decisión que desee.

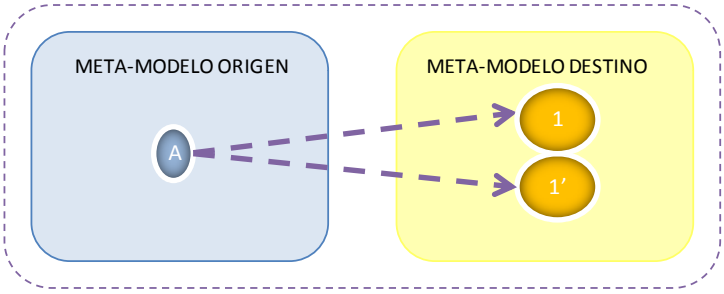

Figura 3-4. Relación Uno-a-Uno con Opciones

Relación Uno-a-N

Esta relación es un poco más compleja que la anterior. Se da cuando una meta-clase del meta-modelo origen se relaciona con N (N o más) meta-clases del meta-modelo destino (Figura 3-5).

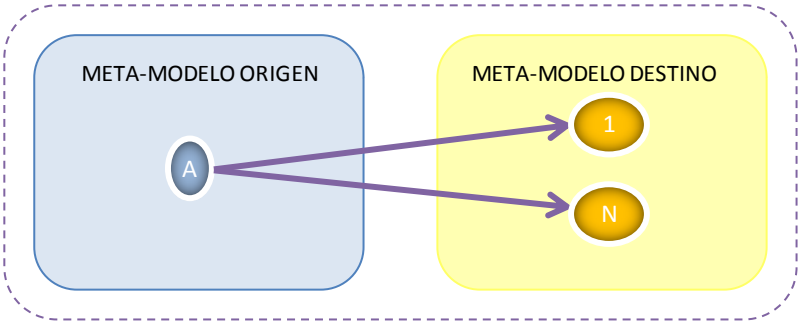

Figura 3-5. Relación Uno-a-N

Para este tipo de relación también existe una variación, que se da cuando una meta-clase del meta-modelo origen se relaciona con varias meta-clases del meta-modelo destino pero, además, existen diferentes conjuntos, excluyentes entre sí, de meta-clases con las que se puede relacionar. Como se puede ver en la Figura 3-6 el elemento A del meta-modelo origen se puede relacionar con los elementos 1 y 2 del meta-modelo destino o con los elementos 1' y 2', de forma mutuamente excluyente.

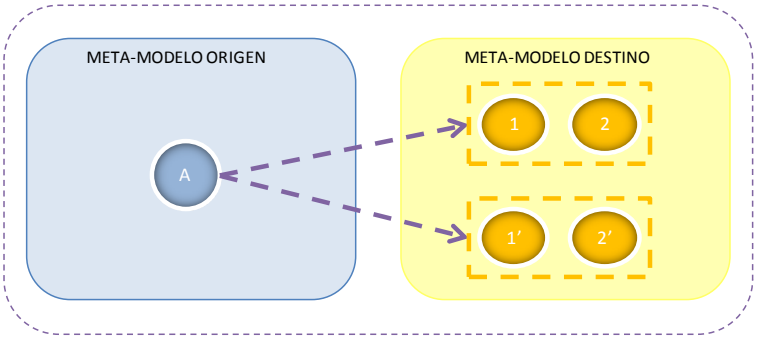

Figura 3-6. Relación Uno-a-N con Opciones

Relación de N-a-Uno

Este tipo de relaciones se da cuando N (dos o más) meta-clases del meta-modelo origen se relacionan con la misma meta-clase del meta-modelo detino (Figura 3-7). Si bien a simple vista este tipo de relación parece más simple que la relación de Uno a N, en realidad su implementación es más complicada a la hora

de soportar la trazabilidad, ya que se deben mantener las trazas desde varios elementos, de manera conjunta, del meta-modelo origen a un elemento del meta-modelo destino.

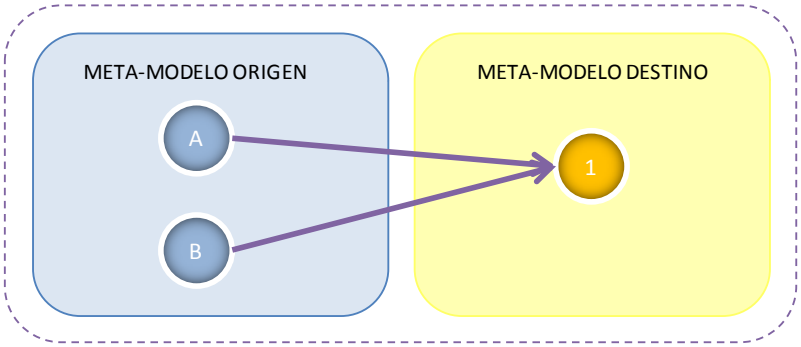

Figura 3-7. Relación N-a-Uno

De la misma manera que en los casos anteriores, también existe una variación de este tipo de relación, que se da cuando en el meta-modelo origen existen varios conjuntos (excluyentes entre sí) de meta-clases que se pueden relacionar con la misma meta-clase del meta-modelo destino. (Figura 3-8)

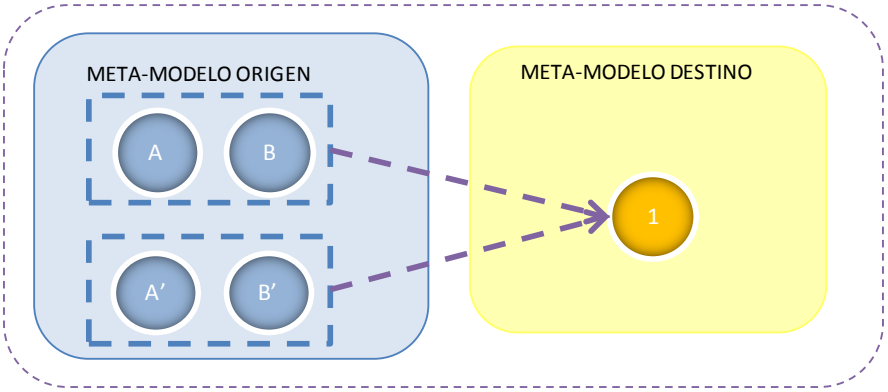

Figura 3-8. Relación N-a-Uno con Opciones

Relación de N-a-M

Los tipos de relaciones N a M son, desde nuestro punto de vista, las más complicadas de implementar (Figura 3-9).

Este tipo de relaciones se puede analizar como una variación del caso anterior, donde se tienen un conjunto de meta-clases en el meta-modelo origen que pueden relacionarse con un conjunto de meta-clases del meta-modelo destino.

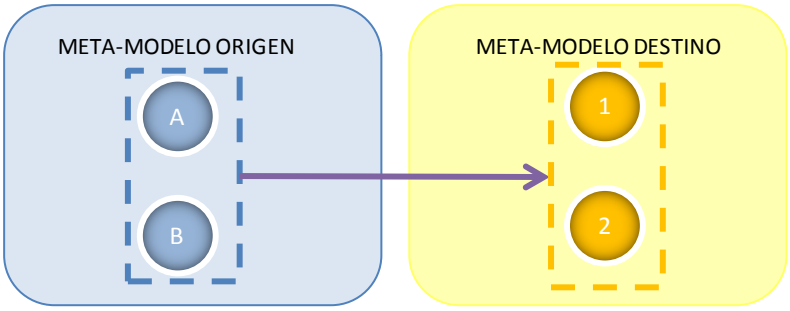

Figura 3-9. Relación N-a-M

En la Figura 3-10 se muestra una variación del tipo de relacion N a M, donde los objetos A y B del meta-modelo origen pueden relacionarse con los objetos 1 y 2 o con los objetos 1' y 2' del meta-modelo destino.

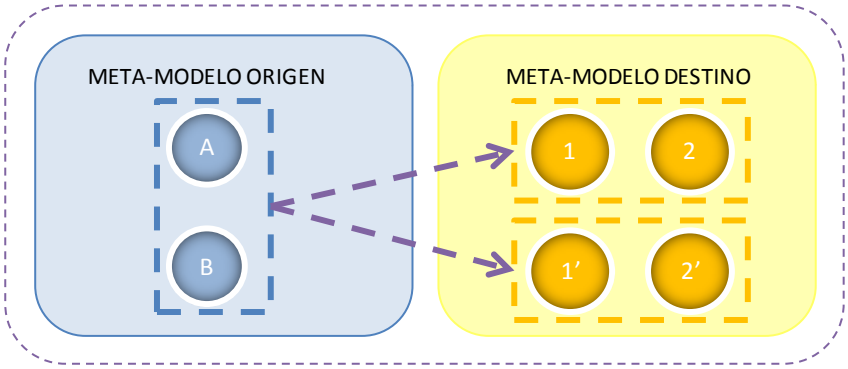

Figura 3-10. Relación N-a-M con Opciones

Otros tipos de Relaciones

En el análisis de los tipos de relaciones se han encontrado dos casos más que, si bien pueden ser vistos como variaciones del tipo Uno-a-Uno presentado anteriormente, es importante tenerlos en cuenta a la hora de definir el meta-modelo a nivel PIM.

Relación Uno-a-Cero

Esta relación, que se muestra en la Figura 3-11, se presenta cuando en el meta-modelo origen existe una meta-clase que no tiene una correspondencia directa en el meta-modelo destino.

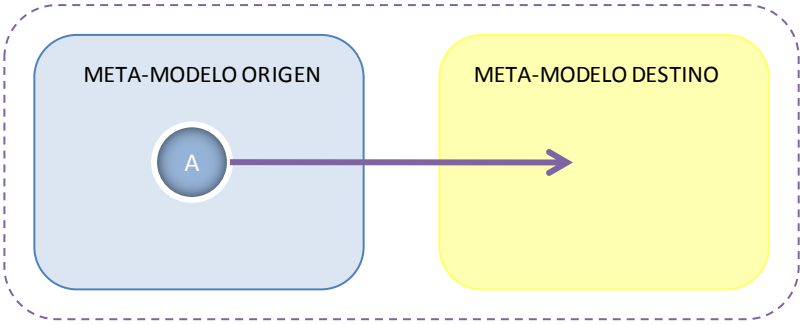

Figura 3-11. Relación Uno-a-Cero

Relación Cero -a-Uno

Este tipo de relación se da cuando en el meta-modelo destino existe una meta-clase que no tiene relación con ninguna meta-clase del meta-modelo origen (Figura 3-12). De la misma manera que el caso anterior, el desarrollador debe decidir qué acciones seguir para permitir la generación de la meta-clase en el meta-modelo destino y no perder la funcionalidad representada por la misma.

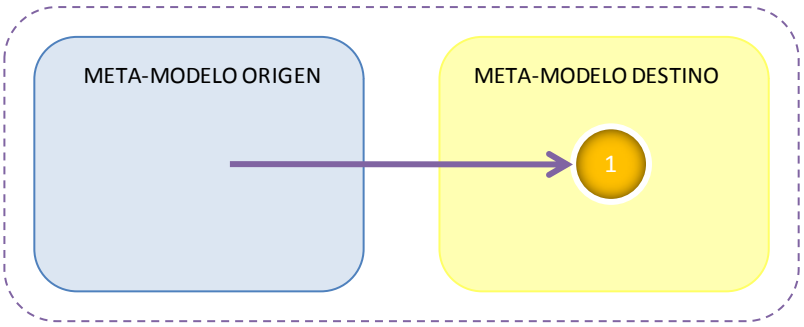

Figura 3-12. Relación Cero-a-Uno

3.1.2.1.3 Especificación de M-TIP

Como se ha mencionado anteriormente, el objetivo del meta-modelo M-TIP es representar las relaciones existentes entre los elementos de los meta-modelos origen y destino de la transformación. A partir de la identificación de los tipos de elementos a transformar y del tipo de relación entre los mismos, se procede a definir el meta-modelo que permitirá modelar transformaciones de modelos a nivel PIM. En la Figura 3-13 se muestra de forma gráfica el meta-modelo definido.

Como se puede observar, el meta-modelo M-TIP proporciona diferentes meta-clases que permiten representar dichas relaciones. Estas meta-clases se

especifican en función de la cardinalidad de los elementos que participan en la relacion. Así, por ejemplo, la meta-clase *OneToOne* permite establer la relación entre un elemento del meta-modelo origen y un elemento del meta-modelo destino; y la meta-clase *OneToMany* permite establecer la relación entre un elemento del meta-modelo origen y varios (dos o más) elementos del meta-modelo destino conjuntamente. Además, se proporcionan mecanismos que permiten realizar la transformación de elementos dependiendo del tipo elemento. Así por ejemplo, para realizar la transformación de elementos que dependen directamente de otro elemento, como es el caso de elementos de tipo *Property* que pueden depender de un elemento de tipo *Classifier*, se establecen referencias entre las meta-clases especificadas, como por ejemplo, la relación de composición entre la meta-clase *OutElement* y la meta-clase *OneToOne*, que permite establecer una nueva relación entre elementos a partir de un elemento ya creado. La especificación completa del meta-modelo de se encuentra en el anexo G.

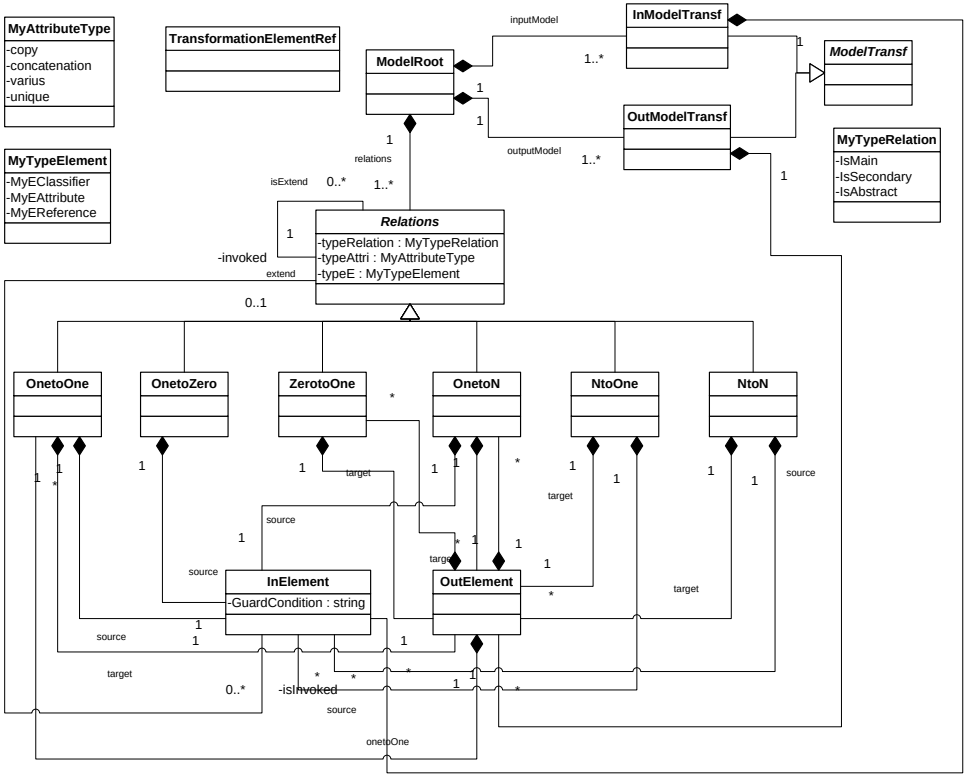

Figura 3-13. Meta-modelo M-TIP

En la sección 3.1.1 se define el proceso de MeTAGeM para el modelado de las transformaciones. En dicho proceso se establece que el modelado de las transformaciones se realiza partiendo de la definición de un modelo de transformación de nivel PIM, y por medio de transformaciones (semi-)automáticas se obtiene el modelo de transformación a nivel PSM, siguiendo la aproximación híbrida. A partir de este último modelo, se obtiene el modelo de la transformación a nivel PDM, conforme al lenguaje seleccionado: ATL o RubyTL, para, finalmente, obtener, el código que implementa la transformación en el lenguaje seleccionado.

Como se dijo anteriormente, a nivel PSM se propone el modelado de las transformaciones siguiendo la aproximación híbrida. Para realizar la especificación del meta-modelo correspondiente a dicho nivel se ha realizado un análisis de los lenguajes de transformación que siguen dicha aproximación. Por lo que, para facilitar la comprensión de las meta-clases que componen el meta-modelo de transformación a nivel PSM se considera conveniente explicar primero los meta-modelos de los lenguajes de transformación seleccionados a nivel PDM, el lenguaje ATL y RubyTL.

3.1.2.2 Meta-modelos de Transformación Dependientes de Plataforma

A nivel PSM se ha decidido implementar el meta-modelo correspondiente a la aproximación híbrida para realizar el modelado de las transformaciones, por lo que a nivel PDM se deben seleccionar lenguajes de transformación que sigan dicha aproximación. Después del análisis realizado de los lenguajes de transformación y de las herramientas existentes (Sección 2.2) se seleccionan dos lenguajes: el lenguaje ATL [105] y el lenguaje RubyTL [176, 178] como lenguajes para el modelado de las transformaciones a nivel PDM.

La selección de estos lenguajes se basa en los siguientes motivos:

- Actualmente, ATL es considerado el estándar de-facto en el campo de las transformaciones de modelos, ya que el estándar QVT propuesto por la OMG presenta serios inconvenientes.
- Para el desarrollo de ATL se han aplicado los principios de MDE, lo que resulta en un entorno completo, que cuenta con la definición de un meta-modelo conforme al cual se definen las transformaciones de modelos, como modelos en sí mismos. Esto facilita enormemente la tarea de modelar las transformaciones a nivel PDM, ya que se utilizará directamente el meta-modelo y los editores de modelos propuestos por ATL. Además, tiene implementado un extractor que permite obtener el código que implementa la transformación en el lenguaje ATL a partir del modelo desarrollado. Por lo

que, usando dicho extractor se puede obtener el código implementable de la transformación.

- El último motivo decisivo se basa en nuestra propia experiencia en el desarrollo de transformaciones de modelos. Durante la evaluación de los diferentes lenguajes de transformación y las herramientas que lo implementan se ha desarrollado al menos un caso de estudio con cada herramienta. De todos los lenguajes evaluados (ATL, QVTo, EpsilonTL, RubyTL, VIATRA, MediniQVT) ATL es el lenguaje que más soporte brinda al usuario a través de la documentación disponible como: manuales de ayuda, casos de exítos o grupos de usuarios. El segundo lenguaje que brinda mayor documentación, es el lenguaje RubyTL.
- El otro lenguaje que se podría haber seleccionado es EpsilonTL [121], que por lo que se ha visto también es bastante maduro, pero al momento de comenzar a escribir esta tesis, el lenguaje estaba aún en fase de definición, ya que entró en vigencia a partir del año 2009, momento en el que esta tesis se encontraba en un estado muy avanzado.

Se debe mencionar, que como futuro trabajo se pretende implementar el modelado de las transformaciones a nivel PDM en los lenguajes de transformación más representativos de las diferentes aproximaciones.

A continuación, se muestran los meta-modelos de los lenguajes ATL y RubyTL.

3.1.2.2.1 ATL

En la Figura 3-14 se muestra de forma gráfica el meta-modelo, simplificado, de ATL. A modo de resumen, a continuación se explican algunos de los constructores que componen el meta-modelo, para mayor detalle consultar [105].

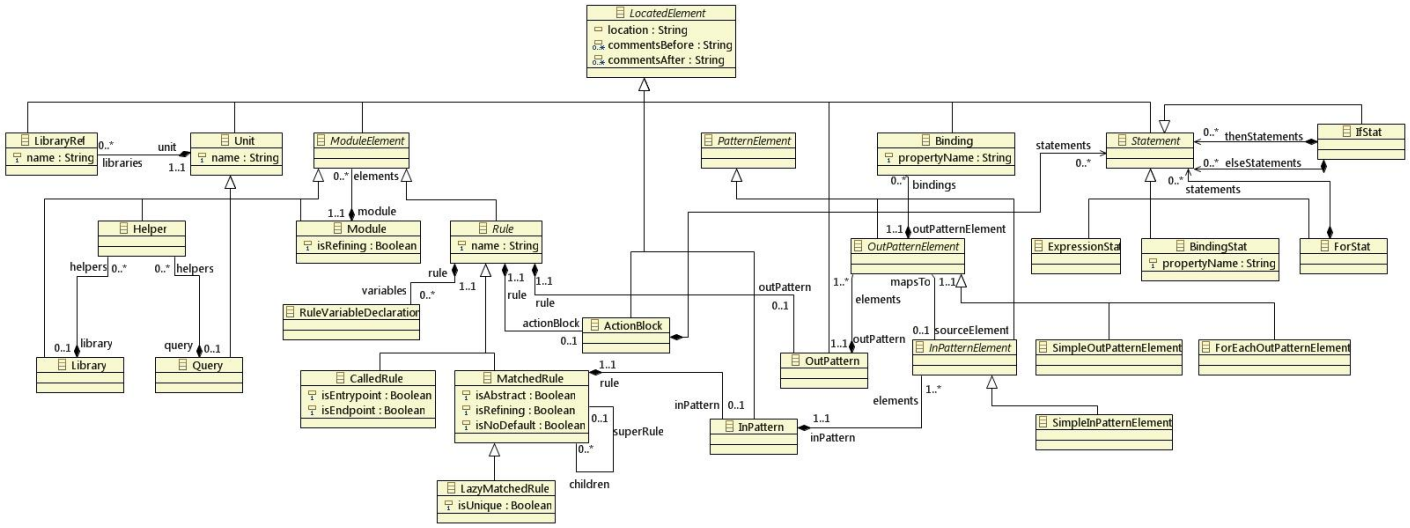

Figura 3-14. Meta-modelo de ATL

En la Figura 3-15 se muestra la meta-clase *Rule* del meta-modelo de ATL y los sub-elementos que dependen de ella. Como se puede ver, en ATL existen diferente tipos de reglas que se corresponden con los diferentes modos de programación soportados: la regla de tipo *CalledRule* (regla imperativa), la de tipo *MatchedRule* (regla declarativa) y la regla de tipo *LazyMatchedRule* que es una especialización de la regla de tipo *MatchedRule*:

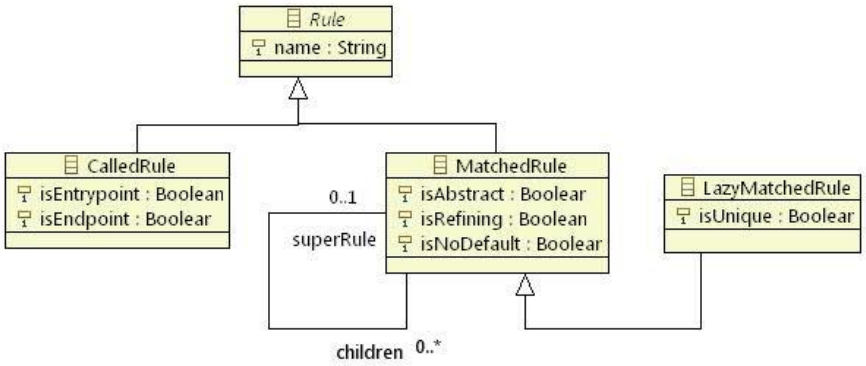

Figura 3-15. Meta-clase *Rule* del Meta-modelo de ATL

- *CalledRule*: este tipo de regla permite introducir constructores imperativos en la definición de reglas de ATL. Las reglas de tipo *CalledRule* deben ser invocadas de forma explícita para ser ejecutadas y pueden recibir parámetros de entrada. Este tipo de regla se invoca en la sección de código imperativo, ya sea de una regla de tipo *MatchedRule* o en otra regla de tipo *CalledRule*. Además, tiene dos propiedades, excluyentes entre sí, de tipo booleanas que permiten identificar el momento de ejecución de la regla: al inicio de la transformación (*isEntrypoint = true*) o al finalizar la transformación (*isEndpoint = true*).
- *MatchedRule*: este tipo de regla constituye el núcleo de la transformación ATL, ya que permite especificar: 1) qué elemento o elementos del modelo destino debe ser generado a partir de un elemento o elementos del modelo origen dado, y 2) la manera en que se debe inicializar la generación de los elementos del modelo destino. En el momento de la ejecución, el motor de ATL evalúa cada uno de los elementos del modelo origen y verifica si en el conjunto de reglas existe alguna cuyo patrón de origen se corresponda con los elementos de modelo origen; si hubiese alguna coincidencia se generan los elementos correspondientes en el modelo destino. El atributo *isAbstract* indica si la regla es abstracta o no.

- *LazyMatchedRule*: este tipo de regla es una especialización del tipo de regla *MatchedRule*. Sirve para definir una regla que se debe ejecutar más de una vez sobre un mismo elemento de origen. Este tipo de regla debe ser invocada de forma explícita por otra regla para ser ejecutada. La propiedad *isUnique* sirve para restringir su ejecución; si tiene el valor verdadero, se ejecutará una sola vez y en las posteriores llamadas devolverá la referencia al elemento generado en la primera ejecución.

Además, existe una meta-clase llamada *Helper*, que se puede ver como los equivalentes en ATL a los métodos de JAVA. Los *Helpers* hacen posible la factorización del código ATL y pueden ser invocados desde diferentes puntos de la transformación ATL.

3.1.2.2.2 RubyTL

Como se ha visto en la sección 2.2.4.3, RubyTL es un lenguaje de transformación embebido en el lenguaje Ruby, por lo que, aunque en la documentación disponible se encuentran algunas especificaciones parciales, no tiene un meta-modelo definido. Por esto, para realizar el modelado de las transformaciones usando RubyTL ha sido necesario realizar la especificación e implementación del mismo. La especificación del meta-modelo RubyTL se realiza a partir del análisis de los artículos disponibles, de los diferentes casos de estudios y del apoyo de los propios autores del lenguaje RubyTL. En la Figura 3-16 se muestra la especificación del meta-modelo. A modo de resumen, a continuación, se explican algunos de los constructores que componen dicho meta-modelo.

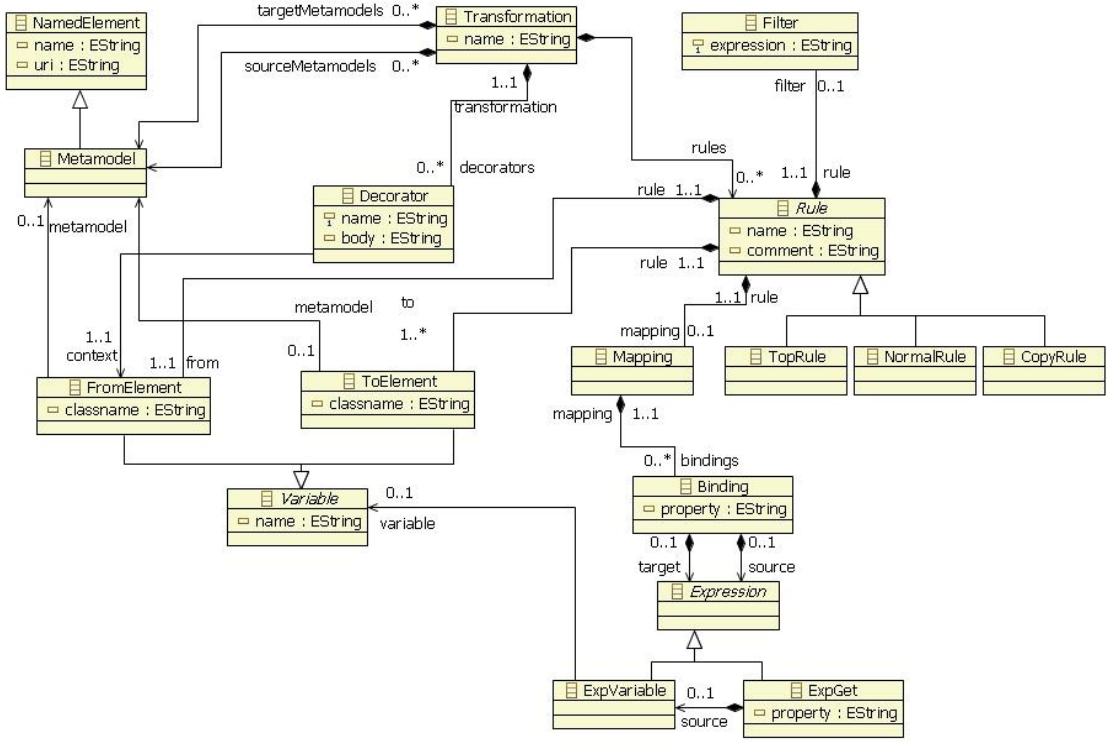

Figura 3-16. Meta-modelo de RubyTL

En la Figura 3-17 se muestra la meta-clase *Rule* del meta-modelo de RubyTL y los subelementos que dependen de ella. Como se puede ver en RubyTL, al igual que en ATL, existen diferentes tipos de reglas: las *TopRule*, las *NormalRule* y las *CopyRule*:

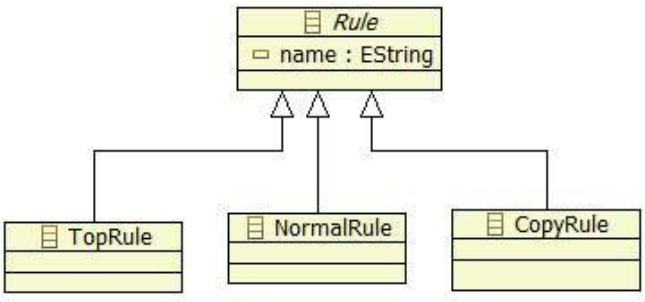

Figura 3-17. Meta-clase *Rule* del Meta-modelo de RubyTL

- *TopRule*: es el principal tipo de regla de RubyTL. Se ejecuta para todas las instancias del elemento origne que figure en el patron de origen de la regla (parte *from*), pero nunca transforma un mismo elemento más de una vez. Haciendo una analogía con ATL se corresponde con la regla de tipo *MatchedRule*. Normalmente existe una regla *top* a partir de la cual se inicia la transformación, mientras que el resto de las reglas son "*no top*". Si hay varias reglas *top*, se ejecutan en el mismo orden que aparecen en la definición de la transformación.
- *NormalRule*: es un tipo de regla especial que debe ser invocada de forma explícita para ejecutarse. Tiene un comportamiento similiar a la regla de tipo *LazyMatchedRule* de ATL con la propiedad *isUnique* en verdadero. La regla de tipo *NormalRule* se ejecuta a partir de la llamada desde otra regla, cuando el filtro de la misma es satisfecho por una instancia del meta-modelo origen. La principal característica diferenciadora de este tipo de regla es que la misma regla nunca transforma dos veces el mismo elemento origen. Esto significa que si la regla es llamada por segunda vez, se devuelve el resultado de la aplicación anterior. Así, este tipo de regla resuelve problemas de recursividad al no permitir transformar más de dos veces el mismo elemento fuente.
- *CopyRule*: el comportamiento es similar a la regla de tipo *NormalRule*, la diferencia radica en que si transforma más de una vez el mismo elemento origen. Esto significa que si se invoca una misma regla de tipo *CopyRule*

varias veces sobre el mismo elemento origen, se crearán varios elementos destino, uno por cada invocación.

En RubyTL también existe una meta-clase llamada *Decorator* que tiene un comportamiento similar a los *Helpers* en ATL y que permiten especificar funciones especiales en RubyTL.

3.1.2.3 Meta-modelo de Transformación Específico de Plataforma

En esta sección se especifica el meta-modelo para el modelado de transformaciones en el nivel PSM siguiendo la aproximación híbrida (Meta-modelo para Lenguajes de Transformación que siguen la aproximación Híbrida, M-LTH) de MeTAGeM. Dicho meta-modelo recoge los constructores necesarios para realizar el modelado de las transformaciones siguiendo la aproximación de transformación de modelos híbrida.

En la sección 2.2.1 se justifica la selección de la aproximación **híbrida** para el modelado de las transformaciones a nivel PSM. Sin embargo, se debe mencionar que, en el futuro se pretende implementar el resto de las aproximaciones replicando el proceso utilizado para implementar la aproximación híbrida.

Como se ha comentado previamente, para la especificación del meta-modelo M-LTH, en la sección 2.2.4, se ha realizado un análisis de los lenguajes de transformación de modelos más representativos que siguen la aproximación híbrida con el objetivo de determinar los elementos comunes a dichos lenguajes que deberían ser contemplados por M-LTH.

Una de las principales conclusiones obtenidas tras del análisis realizado, como se evidencia en la sección anterior, es que los lenguajes de transformación que siguen la aproximación híbrida manejan principalmente dos tipos de elementos, que aunque cada lenguaje lo identifica con un nombre diferente, el funcionamiento es similar en cada uno de ellos:

a) **Reglas:** se utilizan para generar elementos en el modelo destino. Existen diferentes tipos de reglas dependiendo de la forma de ejecución de las mismas: reglas que se ejecutan cuando se produce una correspondencia entre una parte del modelo origen y el patrón de origen de la regla, y reglas que es necesario invocar explícitamente.

b) **Funciones**: se utilizan para obtener algún tipo de valor, ya sea por medio de un cálculo, o recuperando el valor de algún elemento de los meta-modelos que intervienen en la relación.

Tanto las reglas como las funciones se agrupan en un mismo **módulo** donde se identifican además, los meta-modelos que participan en la transformación, esto es, el **meta-modelo origen** y el **meta-modelo destino**. En el contexto de las transformaciones de modelos, se entiende por módulo como el programa en el que se definen las reglas de transformación. Para cada una de las reglas definidas se especifica un patrón origen (que se corresponde con los elementos del meta-modelo origen) y un patrón destino (que se corresponde con los elementos del meta-modelo destino).

En la Figura 3-18 se muestra el meta-modelo para el modelado de las transformaciones de modelos a nivel PSM siguiendo la aproximación híbrida. Como se puede observar, el meta-modelo M-LTH proporciona diferentes meta-clases que permiten representar los elementos identificados en los diferentes lenguajes. Así, por ejemplo, la meta-clase *Rule* que representa las reglas de transformación, y la meta-clase *Operation* que permite definir las funciones. En el anexo G se describe detalladamente cada una de las meta-clases del meta-modelo.

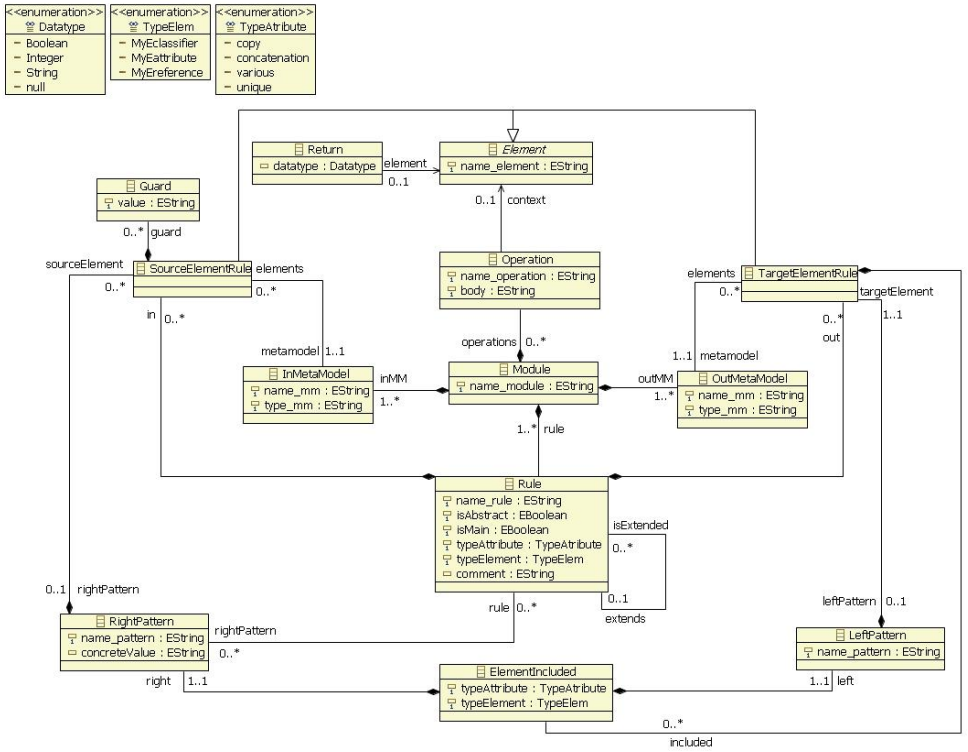

Figura 3-18. Meta-modelo M-LT

A continuación, se presenta la especificación de las reglas de transformación que conforman el meta-transformador.

3.1.3 Meta-Transformador

El meta-transformador está compuesto por el conjunto de reglas que permiten realizar las transformaciones entre los modelos definidos en los diferentes niveles propuestos en MeTAGeM.

En cuanto a la forma de definir las reglas de transformación "puede hacerse utilizando lenguaje natural, algoritmos o modelos de transformaciones" [144]. En esta tesis se sigue la propuesta de [195]:

- Primero, se definen las transformaciones entre modelos utilizando lenguaje natural.
- Después, se estructuran en un conjunto de reglas, expresadas nuevamente en lenguaje natural.
- Posteriormente, estas reglas de transformación se formalizan usando gramáticas de grafos [82].
- Por último, las reglas resultantes se implementan utilizando algún lenguaje de transformación de modelos existente.

El hecho de utilizar grafos [23] para la formalización de las reglas de transformación antes de su implementación facilita la detección de errores e inconsistencias en las primeras fases de desarrollo y ayuda a aumentar la calidad de los modelos construidos, así como el código generado a partir de los mismos. Estas actividades son especialmente importantes en las propuestas de MDE donde los modelos guían todo el proceso de desarrollo, por lo que la formalización de transformaciones simplifica significativamente el desarrollo de herramientas de apoyo a cualquier enfoque dirigido por modelos [146].

Para especificar las transformaciones utilizando gramática de grafos es necesario definir el conjunto de reglas de grafos. Estas reglas siguen la estructura LHS := RHS (*Left Hand Side* := *Right Hand Side*). Ambos lados de la regla, LHS y RHS, se especifican utilizando grafos. En el lado LHS se especifican los elementos del meta-modelo origen que activan la regla, y en el lado RHS se especifican los elementos del meta-modelo destino que se generan a partir de la ejecución de la regla.

En el contexto particular de esta tesis, se sigue la aproximación presentada en [56] para definir las reglas de transformación usando gramática de grafos que se resume a continuación:

- Cada nodo en el lado LHS se identifica por números consecutivos. Estos números hacen posible la identificación de los nodos en el lado RHS.
- Todas las propiedades de los diferentes nodos tienen un valor inicial. En el caso de que el valor inicial sea indefinido, se usa el término "???".
- Para referirse al nodo LHS desde el nodo RHS se usa la expresión "match(x)", donde x es el número que identifica el nodo LHS.
- De la misma manera, para referise a un atributo del nodo LHS se utiliza la notación de punto, por ejemplo "match(x).name".
- Los nodos en el lado RHS también se deben numerar:
    - Si el mismo número aparece en el lado derecho y en el lado izquierdo, el tipo de nodo RHS debe ser el mismo que el nodo LHS.
    - Si el nodo RHS se identifica por un número seguido con un apóstofre (x'), el tipo del nodo debe ser diferente del tipo del nodo LHS respectivo y se deben preservar todas las conexiones con el resto de los elementos.
    - Si existe un número en el lado LHS que no aparece en el lado RHS, se debe eliminar dicho nodo en el lado LHS, así como todas las conexiones en las que participa.
    - Si en el lado RHS aparece un nuevo número que no aparece en el lado LHS se debe añadir un nuevo nodo.

A continuación, se especifican las transformaciones definidas en MeTAGeM siguiendo esta aproximación descrita anteriormente.

3.1.3.1 Especificación de Transformaciones entre M-TIP y M-LTH

Para realizar la especificación de las reglas de transformación es necesario analizar los meta-modelos que intervienen en la transformación, en este caso los meta-modelos definidos a nivel PIM y PSM, para determinar las relaciones existentes entre los elementos de cada uno de ellos.

Siguiendo el proceso mencionado para la definición de las reglas, en primer lugar se definen las reglas usando lenguaje natural y luego se estructuran en un conjunto de reglas de transformación y por último, se formalizan por medio de gramática de grafos. Para facilitar la comprensión al lector, en la Tabla 3-2 se muestran las reglas de transformación definidas en lenguaje natural y a continuación se muestra la formalización de cuatro de las reglas que se consideran más relevantes (sombreadas en la tabla) usando gramáticas de grafos.

Tabla 3-2. Transformaciones de M-TIP a M-LTH

Meta-modelo M-TIP		**Meta-modelo M-LTH**
ModelRoot		*Module*
InModelTransf		*InMetaModel*
OutModelTransf		*OutMetaModel*
Relations	*Sin dependencia de otros elementos*	*Rule*
	Con dependencia de otros elementos	*ElementIncluded*
OneToOne	*Sin dependencia de otros elementos*	- *Rule*: *in* con cardinalidad = 1 *out* con cardinalidad = 1 - *SourceElementRule* - *OutElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* -*LeftPattern*
OneToZero		- *Rule*: *in* con cardinalidad = 1 - *SourceElementRule*
ZeroToOne	*Sin dependencia de otros elementos*	- *Rule* *out* con cardinalidad 1 - *TargetElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* (sin elementos) - *LeftPattern*
OneToMany		- *Rule*: *in* con cardinalidad = 1 *out* con cardinalidad = N - *SourceElementRule* - *N OutElementRule*

Meta-modelo M-TIP		**Meta-modelo M-LTH**
ManyToOne	*Sin dependencia de otros elementos*	- *Rule*: *in* con cardinalidad = N *out* con cardinalidad = 1 - *N SourceElementRule* - *OutElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* (solo 1 *RightPattern* con N *sourceElements* en su interior) -*LeftPattern*
ManyToMany		- *Rule*: *in* con cardinalidad = N *out* con cardinalidad = N - *N SourceElementRule* - *N OutElementRule*
InElement (con *GuardCondition*)		- *SourceElementRule* - *Guard*
InElement (sin *GuardCondition*)		- *SourceElementoRule*
OutElement		- *TargetElementRule*

En la Figura 3-19 se muestra la formalización de la regla que permite la transformación entre elementos del tipo *Relations* del meta-modelo M-TIP (meta-modelo origen) y elementos de tipo *Rule* del meta-modelo M-LTH (meta-modelo destino). Como se puede observar, se completan las propiedades del elemento *Rule* a partir de las propiedades del elemento *Relations*. Así, por ejemplo, la propiedad *isAbstract* de *Rule* se completa con el valor de la propiedad *typeRelation* de *Relation* cuando esta última tiene el valor *isAbstract*. De igual manera, cuando la propiedad *typeRelation* de *Relation* tiene el valor *isMain* se completa la propiedad *isMain* de *Rule*.

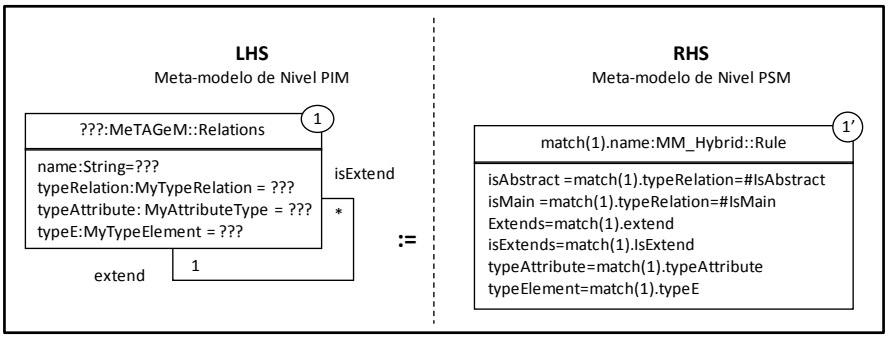

Figura 3-19. Gramática de Grafos: Regla *Relations2Rule*

Es importante mencionar que como la meta-clase *Relations* es abstracta. La regla de transformación también se define como abstracta para posteriormente ser extendida por el resto de las reglas de transformación de los elementos que dependen de *Relations*, como, por ejemplo, la meta-clase *OneToMany* que es una especialización de *Relations.*

Por ejemplo, en la Figura 3-20 se muestra la formalización de la regla que permite la transformación entre elementos de tipo *OneToMany* del meta-modelo M-TIP y elementos de tipo *Rule* del meta-modelo M-LTH. Esta regla extiende a la regla que transforma elementos de tipo *Relations* en elementos de tipo *Rule*.

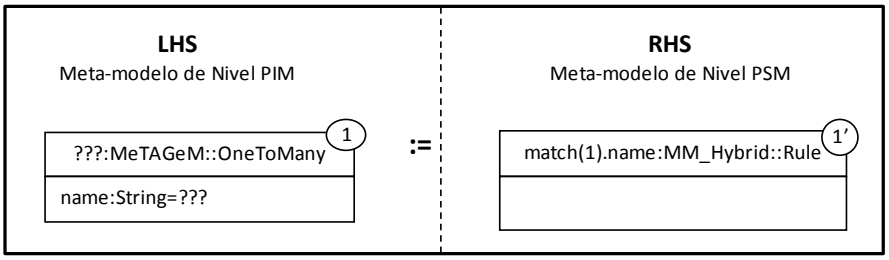

Figura 3-20. Gramática de Grafos: Regla *OneToMany2Rule*

En la Figura 3-21 se muestra la regla de transformación de elementos de tipo *InElement* del meta-modelo M-TIP. Como se puede observar a partir del elemento *InElement* se generan dos elementos en el meta-modelo M-LTH, el elemento *SourceElementRule* y el elemento *Guardcondition*. En el elemento *SourceElementRule* se establecen las propiedades *name* y *metamodel* a partir de las propiedades *name* y *modelRef* del elemento *InElement* respectivamente; y en el elemento *GuardCondition* se establece la propiedad *value* a partir de la propiedad *guardCondition* del elemento *InElement*.

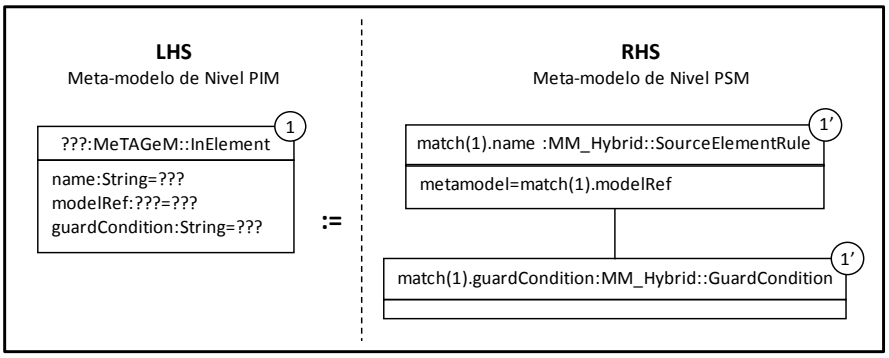

Figura 3-21. Gramática de Grafos: Regla *InElement2SourceElementRule*

En la Figura 3-22 se muestra la regla de transformación de elementos de tipo *OutElement* del meta-modelo M-TIP. Como se puede observar, el elemento *OutElement* esta compuesto por elementos de tipo *Relations*. Por cada elemento *OutElement* se genera un elemento de tipo *TargetElementRule* y por cada elemento de tipo *Relations* que componga el elemento de tipo *OutElement* se genera un elemento de tipo *Rule* usando la regla definida anteriormente.

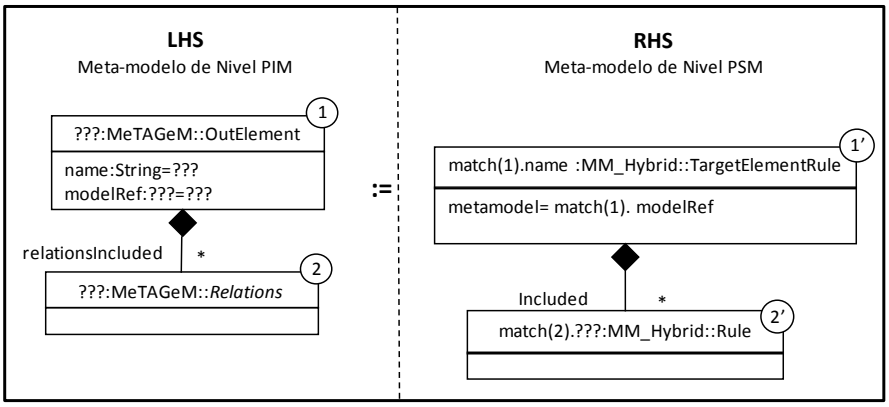

Figura 3-22. Gramática de Grafos: Regla *OutElement2TargetElementRule*

La Figura 3-23 muestra a modo de resumen la transformación entre un elemento de tipo *OneToMany* en un elemento de tipo *Rule* unificando las reglas mostradas anteriormente. Como se puede observar, el elemento *OneToMany* está compuesto de elementos de tipo *InElement* y de elementos de tipo *OutElement*. En el momento de la ejecución de la regla se deben completar las referencias *in* y *out* del elemento *Rule* con los valores de las referencias de los elementos *SourceElementRule* y *TargetElementRule* del meta-modelo destino, en los que se transformaron los elementos *InElement* y *OutElement*, respectivamente.

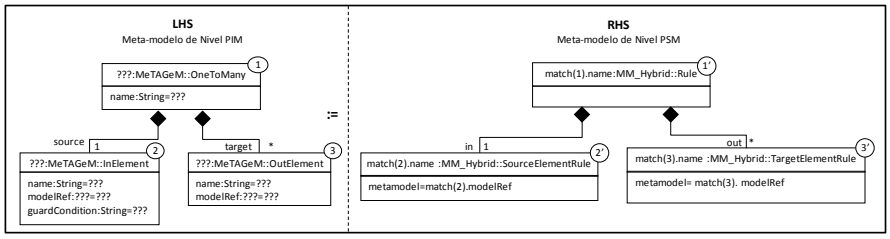

Figura 3-23. Gramática de Grafos: Regla *OneToMany2Rule* completa

3.1.3.2 Especificación de Transformaciones entre M-LTH y PDM

De manera similar que en la sección anterior, se definen a continuación las reglas de transformación entre el meta-modelo M-LTH y los meta-modelos especificados a nivel PDM (meta-modelo de ATL y RubyTL).

3.1.3.2.1 Transformaciones entre M-LTH y meta-modelo de ATL

En la Tabla 3-3 se detallan las reglas de transformación entre el meta-modelo de nivel PSM y el meta-modelo de ATL de nivel PDM, usando lenguaje natural; a continuación, se formalizan tres de las reglas que se consideran más representativas, por medio de gramática de grafos. Estas tres reglas aparecen en gris en la tabla

Tabla 3-3. Transformaciones de M-LTH a Meta-modelo ATL

Meta-modelo M-LTH		**Meta-modelo ATL de Nivel PDM**	
Module		*Module*	
InMetaModel		*OclModel (Input)*	
OutMetaModel		*OclModel (Output)*	
Rule	*isMain = true* y *in > 0*	*Rule*	*Matched Rule*
	isAbstract = true y *isExtended = defined* y *in > 0*		*MatchedRule (superRule= true)*
	isMain = false and in > 0		*LazyMatchedRule*
	(isMain = false and in > 0 and typeAttribute = #unique)		*LazyMatchedRule (unique = true)*
	in = 0		*CalledRule*
SourceElementRule		*SimpleInPatternElement*	
TargetElementRule		*SimpleOutPatternElement*	

Meta-modelo M-LTH	Meta-modelo ATL de Nivel PDM
ElementIncluded *- LeftPattern* *- RightPattern*	*Binding* *- propertyName (Left side)* *- value (Right side)*
Operation *- Return.datatype* *- Boolean* *- Integer* *- String* *- Element*	*Helper* *- Operation.returnType* *- BooleanType* *- IntegerType* *- StringType* *- OclModelElement*

En la Figura 3-24 se muestra la formalización usando gramática de grafos de la regla de transformación entre elementos de tipo *Rule* del meta-modelo M-LTH y elementos de tipo *MatchedRule* del meta-modelo de ATL. Para que esta regla pueda aplicarse se establecen dos condiciones sobre el elemento *Rule*: la primera condición es que la propiedad *isMain* tenga el valor *true*, y la segunda es que la referencia *in* tenga un valor mayor que cero (in>0). Cuando se ejecuta la regla por cada elemento de tipo *Rule* en el modelo origen, que cumpla con las condiciones, se genera un elemento de tipo *MatchedRule* en el modelo destino, y además se completan las propiedades de dicho elemento, como por ejemplo la propiedad *name*.

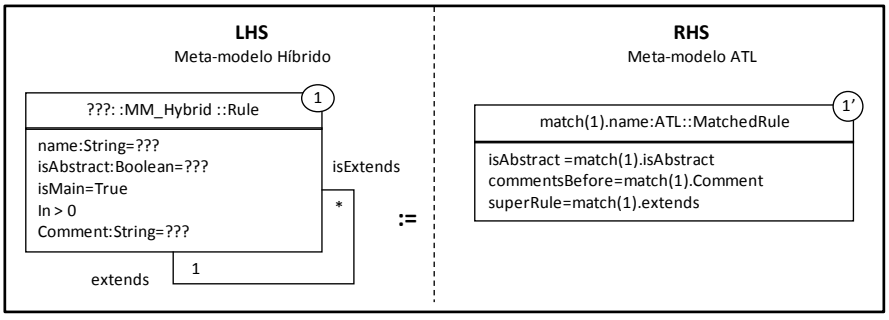

Figura 3-24. Gramática de Grafos: Regla *Rule2MatchedRule*

En la Figura 3-25 se muestra la regla de transformación para los elementos de tipo *SourceElementRule* del meta-modelo M-LTH. Como se puede observar, a partir de éste elemento se generan dos elementos en el modelo destino: el elemento *SimpleInPatternElement* y el elemento *OclModelElement* que define el tipo del elemento *SimpleInPatternElement*.

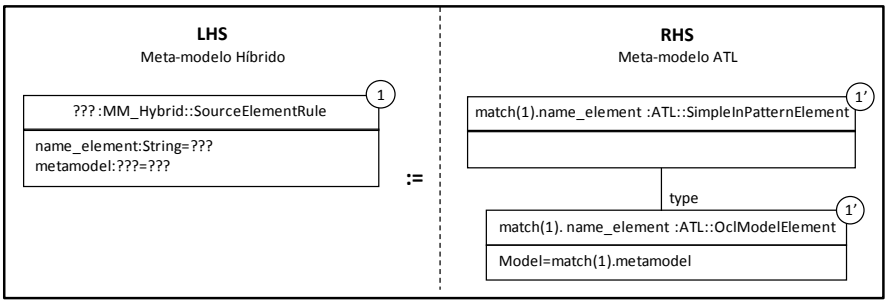

Figura 3-25. Gramática de Grafos: Regla *SourceElementRule2SimpleInPatternElement*

En la Figura 3-26 se muestra la formalización de la regla que permite transformar los elementos de tipo *TargetElementRule*. Como se puede observar, un elemento *TargetElementRule* puede tener incluidos elementos de tipo *Rule*. A partir de un elemento *TargetElementRule* se generan dos elementos: el elemento *SimpleOutPattern* y el elemento *OCLModelElement* que define su tipo; además, por cada elemento *Rule* se genera un elemento de tipo *Binding* que depende del elemento *SimpleOutPattern* generado.

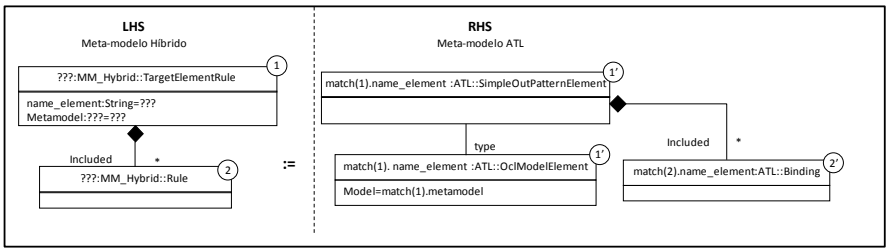

Figura 3-26. Gramática de Grafos: Regla *TargetElementRule2SimpleOutPatternElement*

Por último, en la Figura 3-27 se muestra a modo de resumen la transformación del elemento *Rule* en un elemento *MatchedRule* unificando las reglas mostradas previamente. Como se ha dicho anteriormente, el elemento *Rule* esta compuesto por elementos de tipo *SourceElementRule* y *TargetElementRule*. En el momento de la ejecución de la regla se genera el elemento *MatchedRule*; en dicho elemento deben completarse las referencias *inPattern* y *outPattern* con la referencia a los elementos *SimpleInPatternElement* y *SimpleOutPatternElement* del meta-modelo destino en los que se transformaron los elementos *SourceElementRule* y *TargetElementRule* respectivamente.

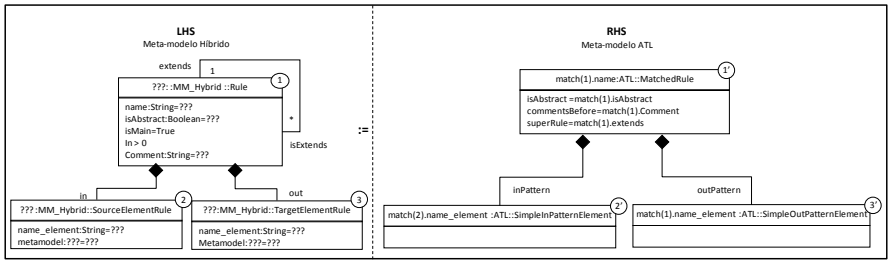

Figura 3-27. Gramática de Grafos: Regla *Rule2MatchedRule* Completa

3.1.3.2.2 Transformaciones entre M-LTH y meta-modelo de RubyTL

En la Tabla 3-4 se detallan las reglas de transformación entre el meta-modelo M-LTH y el meta-modelo de RubyTL de nivel PDM, usando lenguaje natural; a continuación se formalizan tres de ellas (que aparecen en gris en la tabla) por medio de gramática de grafos.

Tabla 3-4. Transformaciones de M-LTH a Meta-modelo de RubyTL

Meta-modelo M-LTH		**Meta-modelo de RubyTL de Nivel PDM**	
Module		*Transformation*	
InMetaModel		*MetaModel(Input)*	
OutMetaModel		*MetaModel (Output)*	
Rule	*isMain = true y in= 1*	*Rule*	*TopRule*
	isMain = true y in> 1		*TopRule (uso del método allObjects en Rule.filter)*
	isMain = false, typeAttribute <>#unique y in= 1		*CopyRule*
	isMain = false, typeAttribute <>#unique y in> 1		*CopyRule (uso del métdo allObjects en Rule.filter)*
	(isMain = false y *typeAttribute = #unique y in = 1)*		*NormalRule*
	(isMain = false y *typeAttribute = #unique y in > 1)*		*NormalRule (uso del métdo allObjects en Rule.filter)*
	in = 0		*Método Estático de Ruby*
SourceElementRule		*FromElement*	
TargetElementRule		*ToElement*	

Meta-modelo M-LTH	Meta-modelo de RubyTL de Nivel PDM
ElementIncluded *- LeftPattern* *- RightPattern*	*Binding* *- ExpGet (Left side)* *- ExpGet (Right side)*
Operation *- Return.datatype*	*Decorator* *- Decorator.body + dataType. toString()*

En la Figura 3-28 se muestra la formalización de la regla que permite la transformación de elementos de tipo *Rule*. Para que esta regla pueda aplicarse se establecen dos condiciones sobre el elemento *Rule*: la primera condición es que la propiedad *isMain* tenga el valor *true*, y la segunda es que la referencia *in* sea igual a 1. Como se puede observar, se generan tres elementos diferentes: el elemento *TopRule* que es la regla propiamente dicha; el elemento *Filter*, que permite establecer las diferentos condiciones por la cual la regla debe ser ejecutada; y el elemento *Mapping*, que agrupa el resto de los elementos que dependen del elemento que se genera.

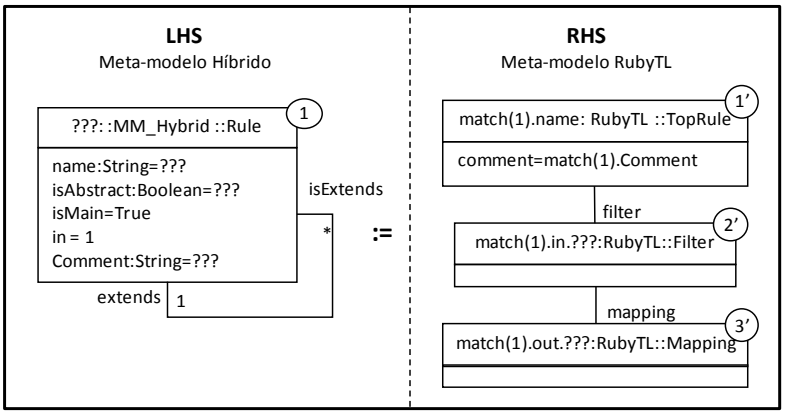

Figura 3-28. Gramática de Grafos: Regla *Rule2TopRule*

En la Figura 3-29 se muestra la regla de transformación entre el elemento *SourceElementRule* del meta-modelo M-LTH y el elemento *FromElement* del meta-modelo de RubyTL. En el elemento *FromElement* se establecen las propiedades *name*, *classname* y *metamodel*.

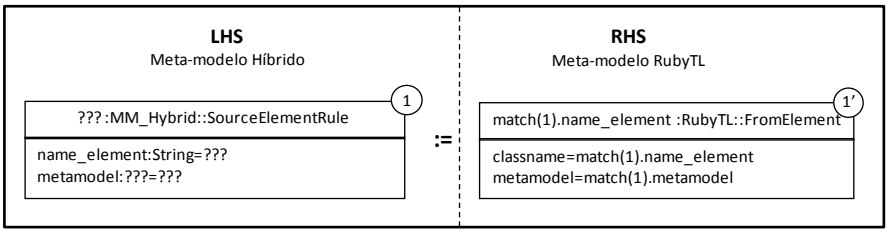

Figura 3-29. Gramática de Grafos: Regla *SourceElementRule2FromElement*

La Figura 3-30 muestra la formalización de la regla de transformación entre el elemento *TargetElementRule* del meta-modelo M-LTH y el elemento *ToElement* del meta-modelo de RubyTL. De igual manera que en el caso anterior, se completan las propiedades *name*, *classname* y *metamodel* del elemento *ToElement*.

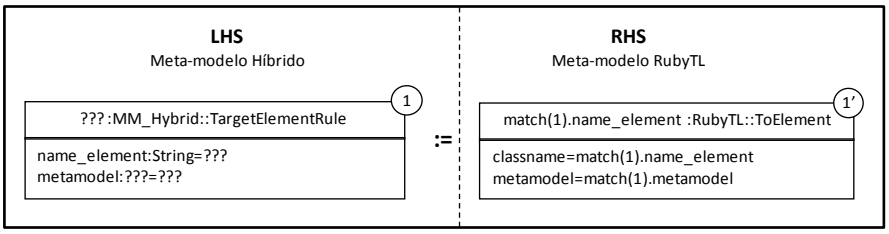

Figura 3-30. Gramática de Grafos: Regla *SourceElementRule2ToElement*

Por último, en la Figura 3-31 se muestra de forma resumida la transformación del elemento *Rule* en un elemento *TopRule*. Como se ha dicho anteriormente, el elemento *Rule* está compuesto por elementos de tipo *SourceElementRule* y *TargetElementRule*. En el momento de la ejecución de la regla se genera el elemento *TopRule*; en dicho elemento deben completarse las referencias *from* y *to* con la referencia a los elementos *FromElement* y *ToElement* del meta-modelo destino, en los que se transformaron los elementos *SourceElementRule* y *TargetElementRule*, respectivamente.

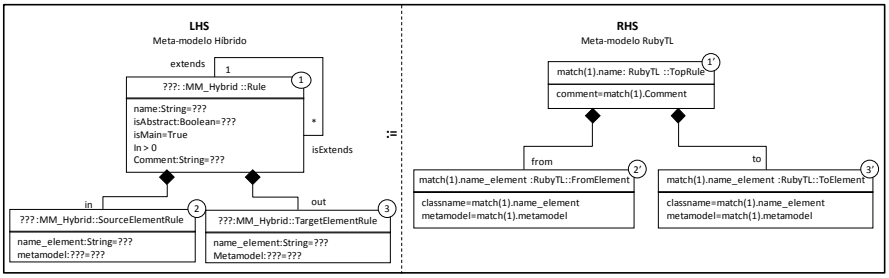

Figura 3-31. Gramática de Grafos: Regla *Rule2TopRule* Completa

3.1.3.3 Especificación de Transformaciones PDM a Código

La generación de código se soporta por medio de la definición de dos conjuntos de reglas de transformación de modelo a texto, una para la generación de código en ATL y otra en RubyTL, implementadas con TCS (*Textual Concrete Syntax*, [104]. Dada la naturaleza de las reglas de transformación de modelos a texto el proceso seguido para la generación de código se muestra en la sección 3.2.6.

3.2 Herramienta

Para dar soporte a la metodología definida se plantea el desarrollo de la herramienta, MeTAGeM, que está compuesta por un conjunto de DSLs [142] que permitan realizar el modelado de las transformaciones de modelos de alto nivel, así como las transformaciones entre los modelos de los diferentes niveles y la posterior generación de código.

Para la definición de la arquitectura de MeTAGeM nos basamos en [191] donde se propone tener en cuenta dos niveles de abstracción separando, por un lado, el diseño conceptual o arquitectura a nivel conceptual (nivel PIM) y por el otro, el diseño técnico (nivel PSM). Además, se propone que el proceso de construcción sea iterativo e incremental, lo que permitirá actualizar las funcionalidades de la herramienta, incorporar mejoras y soportar nuevas aproximaciones y/o lenguajes de transformaciones de forma relativamente fácil.

Siguiendo estas directrices, en la sección 3.2.1 se presenta la definición de la arquitectura a nivel PIM, donde se definen cada uno de los artefactos que formarán parte de la herramienta sin contemplar las cuestiones técnicas de implementación. Posteriormente, en la sección 3.2.2 se presenta la arquitectura a nivel PSM. Dicha arquitectura se obtendrá a partir del refinamiento de la arquitectura conceptual especificando, para cada uno de los artefactos definidos a nivel PIM, las herramientas utilizadas para su implementación.

3.2.1 Arquitectura de MeTAGeM: Nivel PIM

En la Figura 3-32 se muestra la arquitectura conceptual de MeTAGeM. Como se puede observar, se ha definido siguiendo el principio de separación en capas [122, 167], donde cada capa representa una vista en particular de la arquitectura, definiéndose la capa de presentación, la capa lógica y la capa de persistencia. Esta separación en capas permite asegurar el mantenimiento de la

trazabilidad entre los diferentes artefactos, la reutilización de los mismos y el mejor control de su evolución a la hora de incorporar nuevas funcionalidades.

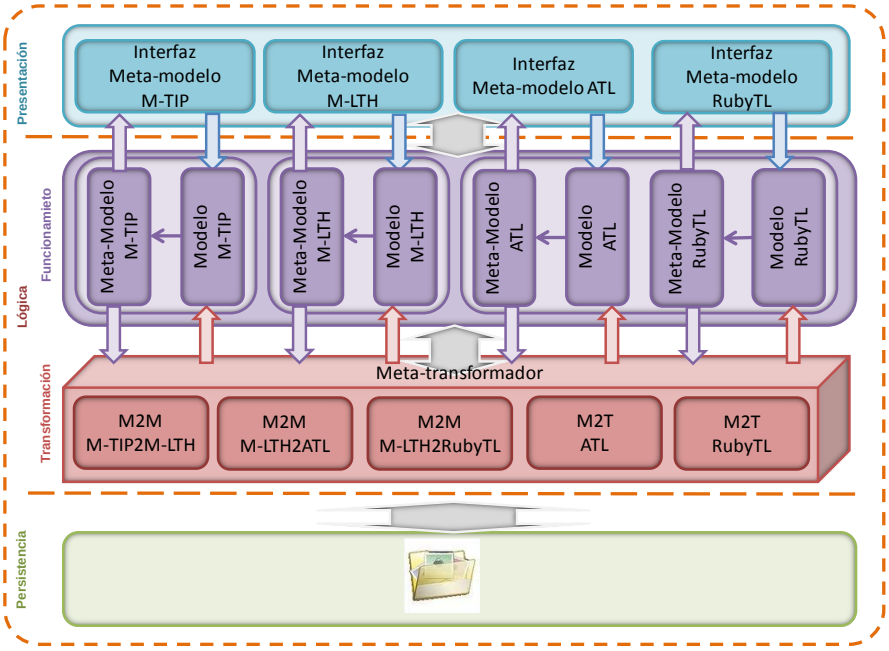

Figura 3-32. Arquitectura Conceptual de MeTAGeM, Nivel PIM

La **capa de presentación** es la capa con la cual el usuario interactúa directamente. Por ello, en esta capa se incluyen los editores gráficos que permiten manipular modelos conformes a los meta-modelos definidos en MeTAGeM.

Como se puede observar en la Figura 3-32, con el objetivo de dar soporte a los diferentes niveles del proceso definido en la sección 3.1, en la capa de presentación se incluye:

- Un editor de modelos conforme al meta-modelo definido a nivel PIM (Interfaz Meta-modelo M-TIP).
- Un editor de modelos conforme al meta-modelo de nivel PSM, siguiendo la aproximación híbrida (Interfaz Meta-modelo M-LTH).
- El editor de modelos proporcionado por el lenguaje de transformación ATL.
- Un editor de modelos conforme al meta-modelo de RubyTL.

En la **capa de lógica** se considera conveniente separar: por un lado, la definición de la sintaxis abstracta de los meta-modelos, y, por otro lado, la

definición de las reglas de transformación necesarias entre los modelos conformes a dichos meta-modelos.

De esta manera, como se puede ver en la Figura 3-32, en el primer nivel se especifica la sintaxis abstracta de los meta-modelos, incluyendo el soporte para realizar manipulación y validación de los modelos con respecto a los meta-modelos.

En el segundo nivel, el meta-transformador, se definen las funciones de transformación que deberá proveer la herramienta. Para esto se definen:

a) El conjunto de reglas de transformación de modelo a modelo entre modelos de nivel PIM a modelos de nivel PSM (M2M M-TIP2M-LTH).
b) Dos conjuntos de reglas de transformación de modelo a modelo entre los modelos a nivel PSM a modelos de nivel PDM conformes al meta-modelo de ATL (M2M M-LTH2ATL) y/o a modelos de nivel PDM conformes al meta-modelo de RubyTL (M2M M-LTH2RubyTL).
c) Dos conjuntos de reglas de transformación de modelo a texto para obtener, a partir de los modelos a nivel PDM el código de la transformación en ATL o en RubyTL.

Además, se establecen puntos de conexión, representados con flechas en la figura, entre los meta-modelos definidos en la capa de lógica y las interfaces de cada meta-modelo definidas en la capa de presentación.

La ejecución de las reglas de transformación permite generar modelos conformes a los meta-modelos correspondientes, por lo que es necesario establecer puntos de conexión entre dichos meta-modelos y las reglas de transformación definidas sobre los mismos.

Por último, la **capa de persistencia** que, de igual manera que en M2DAT [191], se define como un sistema de archivos que incorpora las políticas tradicionales de control de versiones.

3.2.2 Arquitectura de MeTAGeM: Nivel PSM

A partir de la definición de la arquitectura conceptual de MeTAGeM se define el diseño técnico de la misma. La Figura 3-33 muestra las tecnologías utilizadas para implementar los artefactos definidos en los diferentes niveles.

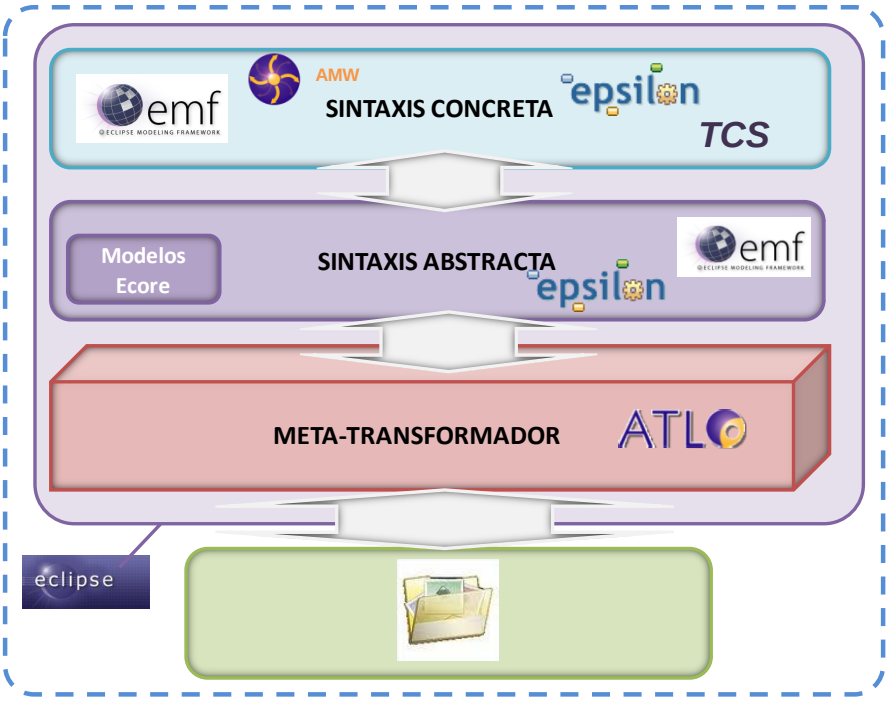

Figura 3-33. Arquitectura de MeTAGeM a Nivel PSM

Siguiendo la propuesta de [191] cada uno de los meta-modelos que conforman a MeTAGeM se implementan por medio de DSL [142], y, de acuerdo con [191], la plataforma seleccionada para la construcción de MeTAGeM es Eclipse y en particular EMF (*Eclipse Modelling Framework*, [49, 50]) que brinda facilidades para la generación de editores a partir de meta-modelos definidos por el usuario. Además, cada componente utilizado para construir MeTAGeM (como ATL, AMW, EVL, TCS, etc) es otro *plug-in* (o conjunto de *plug-ins*) de Eclipse que se ejecuta sobre EMF. Por último, cada uno de los componentes de MeTAGeM se construye como un nuevo *plug-in*, basado en EMF, para Eclipse.

Como se ve en la Figura 3-33 la arquitectura consta de:

- **Sintaxis concreta**, proporcionada por el desarrollo de editores gráficos con formato de árbol para cada uno de los DSLs soportados en MeTAGeM. La sintaxis concreta se corresponde con la capa de presentación de la Figura 3-32 y para su especificación, de acuerdo a lo definido en [191] se utiliza diferentes tecnologías como EMF, AMW (*Atlas Model Weaving*, [77, 79]) y TCS (*Textual Concret Syntaxs*, [104]).

- **Sintaxis abstracta**, representa la especificación de cada DSL y está manejada mediante las facilidades proporcionadas por EMF para la manipulación de modelos, que permite, entre otras cosas, almacenar y recuperar los modelos en el formato XMI [160]. La sintaxis abstracta de cada modelo se utiliza como origen o destino de cada una de las transformaciones definidas.
- **Meta-transformador**, donde se especifican las transformaciones entre los diferentes niveles definidos en MeTAGeM. Para especificar las **transformaciones de modelo a modelo** se utiliza el lenguaje de transformación ATL (Seccion 2.2.4.1). Para especificar las **transformaciones de modelo a texto**, es decir las transformaciones que permiten obtener el código que implementa la transformación en el lenguaje correspondiente se utiliza el lenguaje TCS que brinda, a través de la definición de la sintaxis concreta del meta-modelo del lenguaje, la función de extracción de código a partir de un modelo conforme al meta-modelo de dicho lenguaje.
- Para asegurar la **persistencia** de los modelos se usa un sistema de control de versiones tradicional. En particular, se utiliza el *plug-in Subclipse* de Eclipse, que permite manejar los diferentes componentes creados con MeTAGeM. *Subclipse* es una aplicación de *Subversion* [169] para la plataforma Eclipse.
- Además, se permite realizar la **verificación** sintáctica de los modelos conformes a los DSL definidos. Para esto se utiliza *Epsilon Validation Language* (EVL, [121]), un componente de Epsilon, que permite definir restricciones sobre meta-modelos.

En la siguiente sección se muestra el proceso seguido para el desarrollo de cada uno de los módulos de MeTAGeM.

3.2.3 Proceso de Desarrollo de los Módulos de MeTAGeM

El proceso de desarrollo de MeTAGeM se basa en el proceso de desarrollo propuesto en [191]. En la Figura 3-34 se muestra la adaptación de dicho proceso al desarrollo de MeTAGeM. Como se puede observar, se proponen diferentes pasos. Para cada uno de ellos se representan el/los artefacto/s obtenidos y la/s tecnología/s utilizada para obtenerlos.

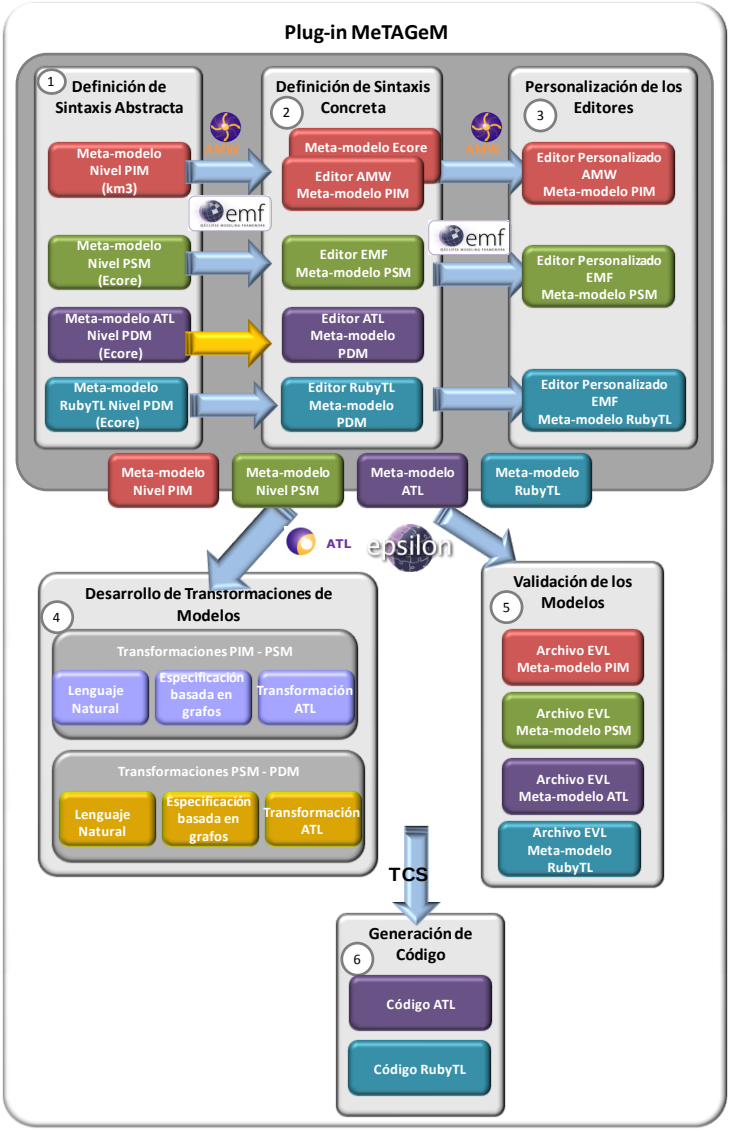

Figura 3-34. Proceso de Desarrollo de MeTAGeM

A continuación se da una breve descripción de cada uno de los pasos:

1. El primer paso es la **definición de la sintaxis abstracta del DSL**. A nivel PIM se realiza la definición de la sintaxis abstracta del DSL utilizando el lenguaje KM3 [25]. A nivel PSM se define la sintaxis abstracta del DSL en términos del meta-modelo *Ecore*, utilizando las facilidades proporcionadas

por EMF. Por último, a nivel PDM se utiliza dos DSL diferentes: por un lado, el DSL proporcionado por el lenguaje de transformación ATL y por otro lado, se define la sintaxis abstracta del DSL para el lenguaje RubyTL en términos del meta-modelo *Ecore*.

2. El siguiente paso es la **definición de la sintaxis concreta del DSL**. Para el DSL a nivel PIM se utiliza el *plug-in* AMW de Eclipse, que a partir de la definición del DSL en KM3, permite realizar de forma automática los editores gráficos en formato de arbol. En el caso del DSL a nivel PSM, se utiliza EMF para generar un editor gráfico de modelos con formato de árbol. A nivel PDM se utiliza, por un lado, el editor propuesto por ATL; y, por el otro lado, se genera el editor de modelos para el lenguaje RubyTL por medio de EMF.
3. Los editores generados hasta este punto permiten editar modelos de los diferentes DSL de manera correcta, pero son demasiados genéricos. Con el fin de obtener editores **personalizados** se introducen algunas características especiales, como por ejemplo, iconos diferentes para representar cada concepto.
4. En el paso de **desarrollo de las transformaciones de modelos** se especifica e implementa cada uno de los conjuntos de transformaciones de modelos que servirán de enlace entre los diferentes DSL. Esto es, las transformaciones de nivel PIM a PSM y las transformaciones de nivel PSM a PDM, para los lenguajes ATL y RubyTL. Para cada conjunto de transformaciones, tal como se ha visto en la sección 3.1.3, se realizan las siguientes actividades:
    - Definir un conjunto de reglas estructuradas en lenguaje natural.
    - Traducir este conjunto de reglas a una especificación basada en grafos con el objetivo de formalizarlas.
    - Por último, traducir la especificación basada en grafos a una transformación en ATL ejecutable.
5. Una vez que se han implementado todas las transformaciones, se comienza a definir las diferentes restricciones para realizar la **validación automática de los modelos** generados. A medida que se codificaban las reglas de transformación y se realizaban pruebas con las mismas surgían situaciones que debían ser consideradas en la validación. Como se verá posteriormente, en el apartado de validación, se han implementado mecanismos para que cuando se invoque una transformación de modelos, se realice la validación de forma automática. Las restricciones se codifican a nivel de meta-modelo y se adjuntan al código de EMF generado para cada uno de los modelos; de esta manera, la validación se realiza sobre los modelos generados a partir de

dichos meta-modelos. Este conjunto de restricciones se almacenan en archivos EVL, uno por cada uno de los meta-modelos definidos.

6. Por ultimo, a partir de los modelos validados se realiza la generación de código, lo que permite obtener el código implementable de la transformación para los lenguajes seleccionados: ATL y/o RubyTL.
7. El conjunto de DSLs desarrollados siguiendo el proceso descrito es básicamente un conjunto de *plug-ins*. Con el fin de facilitar la tarea de distribuir y utilizar dichos *plug-ins*, en Eclipse, se realiza la integración de los mismos en un módulo.

A continuación, se detalla el desarrollo de cada uno de los artefactos obtenidos en cada uno de los pasos del proceso descrito.

3.2.4 Implementación de Meta-modelos

En la sección 3.1.2 se ha realizado la especificación de cada uno de los meta-modelos que forman MeTAGeM. En esta sección se muestra cómo se realiza la implementación de dichos meta-modelos.

3.2.4.1 Meta-modelo de Transformaciones Independiente de Plataforma

En la sección 3.1.2.1 se ha establecido que el modelado de las transformaciones de forma independiente de plataforma se debe realizar simplemente estableciendo las relaciones existentes entre los elementos de los diferentes meta-modelos que participan en la transformación.

El *framework* de meta-modelado *Atlas Model Weaver* (AMW) [79] proporciona un *workbench* genérico y adaptativo que permite manipular las relaciones entre diferentes modelos (*weaving models*). Por ello, se propone la utilización de dicho *framework* para desarrollar el DSL correspondiente al meta-modelo M-TIP.

AMW proporciona un meta-modelo de *weaving* base (*Core Weaving Metamodel*) [79] que contiene un conjunto de clases abstractas para representar información sobre las relaciones entre los elementos de dos modelos; además, proporciona puntos de extensión que permiten obtener nuevos meta-modelos a partir del meta-modelo *Core*. AMW está desarrollado como un *plug-in* para Eclipse basado en EMF [77], lo que asegura la interoperabilidad con el resto de las tecnologías utilizadas para implementar MeTAGeM (Figura 3-35).

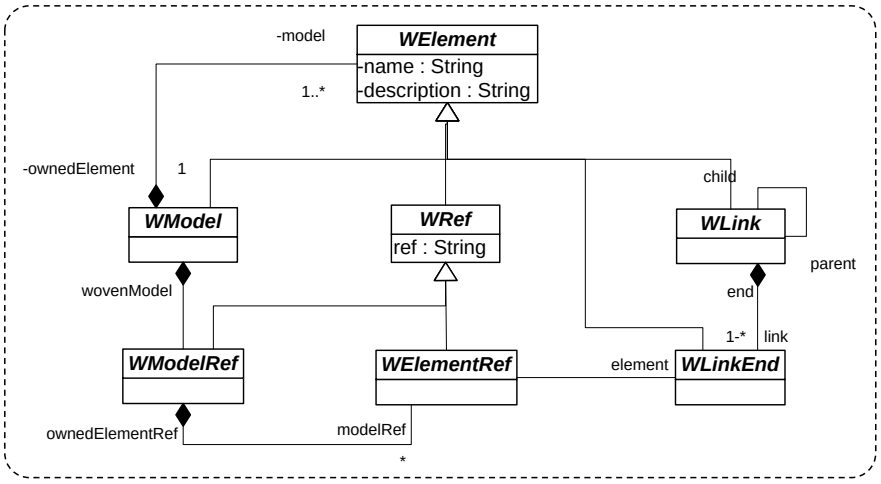

Figura 3-35. ***Core Weaving Metamodel***

A continuación, se explican brevemente cada una de las clases:

- *WElement* es el elemento principal del cual dependen todos los demás elementos. Tiene las propiedades *name* y *description*.
- *WModel* representa el elemento raiz (*roo*t), que contiene todos los elementos del modelo. Está compuesto por los elementos referenciados (elementos *weaving*) y los modelos que participan en la relación (modelos *woven*).
- *WLink* expresa el link entre los elementos de los modelos. Para poder expresar los diferentes tipos de enlace y la semántica de cada uno de ellos, el elemento *WLink* es extendido por diferentes elementos del meta-modelo.
- *WLinkEnd* define los tipos de elementos finales de una relación (*endpoint*). Cada *endpoint* representa una referencia a un elemento del modelo; además, permite crear referencias *n-ary*.
- *WElementRef* representa los elementos asociados con la función *dereferencing*. Esta función toma como parámetro el valor del atributo *ref* y devuelve el elemento vinculado con el mismo. Por razones prácticas, se define como un atributo de tipo cadena. Existe también la función de identificación inversa que toma el elemento vinculado como parámetro y devuelve un identificador único (*ref*).
- *WModel* contiene los elementos *WModelRefs*, que tiene una definición equivalente a las referencias de los elementos *WLinkEnd* y *WElementRef*, pero para los modelos.

Es posible asociar la funcion de *dereferencing* directamente con los *endpoints*. Sin embargo, el uso de los elementos separados a traves de *WElementRef* permite referenciar al mismo elemento del modelo por varios *endpoints*.

Normalmente, las clases del meta-modelo base de AMW se extienden para definer nuevos meta-modelos para contextos específicos.

El meta-modelo M-TIP se define como una extensión del meta-modelo base de AMW. Para realizar la definición de la sintaxis abstracta de M-TIP se utiliza el lenguaje KM3 (*Kernel MetaMetaModel*, [25]) que es el lenguaje utilizado para realizar la extensión del meta-modelo base de AMW. En base a la especificación de M-TIP realizada (Figura 3-13) a continuación se muestra la implementación de las meta-clases más significativas de dicho meta-modelo, la implementación completa se puede consultar en el CD adjunto y en el anexo G:

- *Meta-Clase ModelRoot*: es el elemento raíz del meta-modelo, del cual dependen el resto de los elementos. En la Figura 3-36 se muestra el código correspondiente a la definición de la meta-clase usando KM3. Esta meta-clase se define como una extensión de la clase *WModel* del meta-modelo base de AMW por lo que hereda todas las propiedades del mismo. Además tiene definidos varios elementos de tipo referencia: a) *inputModel*, referencia a la meta-clase *InModelTransf* que representa el/los meta-modelos origen de la relación; b) *outputModel*, referencia a la meta-clase *OutModelTransf* que representa el/los meta-modelos destino de la relación y c) *relations*, referencia a la meta-clase *Relations* que agrupa todos los tipos de relaciones definidas en MeTAGeM.

```
class ModelRoot extends WModel{
  -- @subsets wovenModel
  reference inputModel [1-*] container : InModelTransf;
  -- @subsets wovenModel
  reference outputModel [1-*] container : OutModelTransf;
  -- @subsets ownedElement
  reference relations [1-*] container : Relations;}
```

Figura 3-36. Definición de la Meta-clase *ModelRoot*

- *Meta-clase ModelTransf*: representa los meta-modelos que participan en la transformación, se define como una meta-clase abstracta que hereda de la meta-clase *WModelRef* del meta-modelo base de AMW. Esta meta-clase es instanciada por medio de dos meta-clases: *InModelTransf* y *OutModelTransf*

que representan a los meta-modelos origen y destino, respectivamente (Figura 3-37).

```
-- @welementRefType TransformationElementRef
abstract class ModelTransf extends WModelRef {}

-- @welementRefType TransformationElementRef
class InModelTransf extends ModelTransf{}

-- @welementRefType TransformationElementRef
class OutModelTransf extends ModelTransf{}

-- @wmodelRefType ModelTransf
class TransformationElementRef extends WElementRef {}
```

Figura 3-37. Definición de la Meta-clase *ModelTransf*

- Meta-clases *Relations*: define todos los tipos de relaciones identificadas entre los elementos de los meta-modelos origen y destino (sección 3.1.2.1), esta definida como una meta-clase abstracta que hereda de la meta-clase *WLink* del meta-modelo base de AMW (Figura 3-38). La meta-clase *Relations*, o mejor dicho, las meta-clases que la implementen, pueden ser invocadas desde cero o muchos elementos del tipo *InElement*, esto se indica a través de la referencia *isInvoked* definido como opuesto (*oppositeOf*) a la referencia *invoked* de la meta-clase *InElement*.

```
abstract class Relations extends WLink{
  attribute typeAttri : MyAttributeType;
  attribute typeE: MyTypeElem;
  attribute typeRelation : MyTypeRelation;
  -- @subsets child
  reference isInvoked[0-*]: InElement oppositeOf invoked;
  -- @subsets child
  reference extend [0-1]: Relations oppositeOf isExtend;
  reference isExtend [*]: Relations oppositeOf extend;}
```

Figura 3-38. Definición de la Meta-clase *Relations*

A partir de la especificación de la sintaxis abstracta del DSL de M-TIP AMW permite generar un editor gráfico para la edición de modelos conforme a dicho metamodelo.

El *workbench* de AMW está definido por diferentes puntos de extensión que conectan los componentes de AMW. Para realizar una extensión al meta-

modelo de AMW se utiliza la operación de extensión de meta-modelo. El formato de entrada de los meta-modelos que van a extenderse (el meta-modelo base de *weaving* y el meta-modelo M-TIP) debe ser KM3. La herramienta toma el meta-modelo base de *weaving* y el nuevo meta-modelo con las extensiones correspondientes como entrada y genera un meta-modelo de *weaving* extendido y que es usado por la herramienta para generar modelos de *weaving* conformes al nuevo meta-modelo de *weaving* extendido. A partir de este meta-modelo de *weaving* extendido se obtiene el meta-modelo en formato *Ecore*.

De esta manera, para generar el editor gráfico del DSL conforme al meta-modelo M-TIP se parte de la definición de su sintaxis abstracta realizada en KM3 como una extensión del meta-modelo base de AMW. Para esto se debe generar un nuevo modelo de *weaving* (Figura 3-39) seleccionando como entrada el meta-modelo en formato KM3 con las extensiones realizadas (*Weaver Wizard* 1/3). La herramienta AMW toma automáticamente como entrada el meta-modelo base de *weaving* sobre el que se realiza la extensión.

La herramienta AMW propone dos interfaces para generar el editor, una interfaz por defecto y una interfaz específica, para realizar transformación de modelos. Para el editor que se esta generando en esta tesis, se selecciona esta última (*Weaver Wizard* 2/3). Así se genera, de forma automática, una interfaz acorde al meta-modelo con las extensiones seleccionadas.

El siguiente paso es la selección de los meta-modelos que participan en la transformación, es decir, los meta-modelos origen y destino de la transformación. Nótese que la interfaz de la herramienta toma los nombres de los meta-modelos de acuerdo a lo definido en el meta-modelo con KM3, *InModelTransf* y *OutModelTransf* (*Weaver Wizard* 3/3).

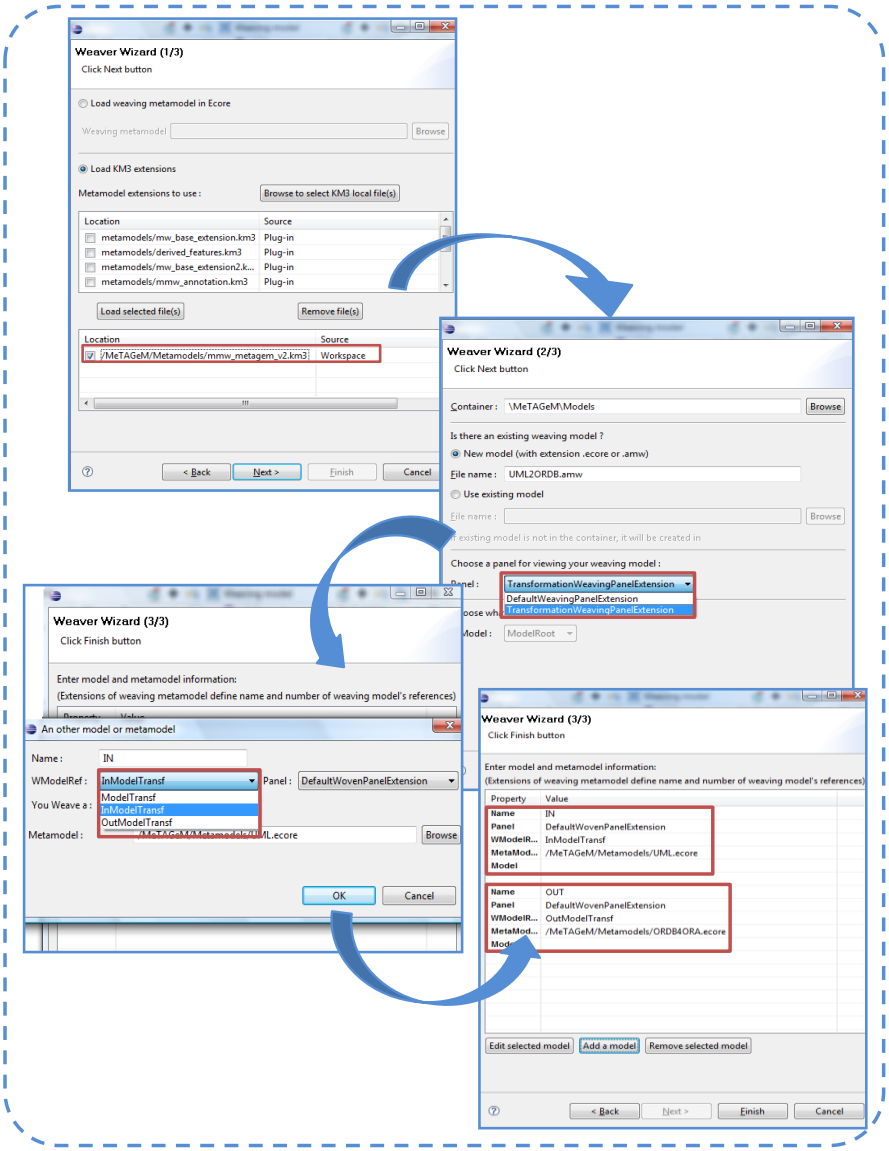

Figura 3-39. *Wizzard* de *Weaver* para la Creación de Extensiones

Una vez seleccionados los meta-modelos origen y destino se genera la interfaz del editor de modelos de M-TIP (Figura 3-40). El editor esta dividido en tres columnas: la columna izquierda muestra el meta-modelo origen de la transformación; la columna derecha muestra el meta-modelo destino de la transformación y por último, la columna central representa al modelo de *weaving*

conforme al meta-modelo de *weaving* extendido. En este modelo el desarrollador deberá indicar las relaciones entre los elementos de ambos meta-modelos (origen – destino). Es importante mencionar que el editor proporciona algunas facilidades de manera automática como el hecho de que al seleccionar un elemento del modelo de *weaving* se seleccionan los elementos que están relacionados en los meta-modelos origen y destino mostrados a la derecha e izquierda del editor, respectivamente.

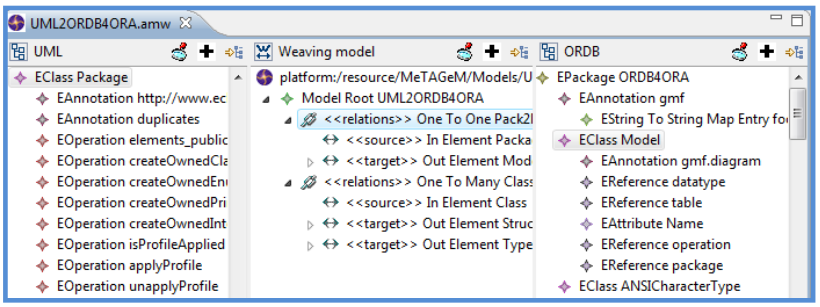

Figura 3-40. Editor del Meta-modelo de Nivel PIM de MeTAGeM

En la Figura 3-41 se muestra el meta-modelo M-TIP en formato *Ecore*. Como se puede observar está compuesto de tres paquetes: el paquete *mwcore*, que representa el meta-modelo base de AMW que es extendido por el meta-modelo M-TIP, el paquete *PrimitiveTypes*, que contiene los tipos primitivos utilizados en *mwcore* y el paquete *mmw_metagem*, que contiene cada una de las meta-clases que extienden el meta-modelo base de *weaving* definidas con KM3.

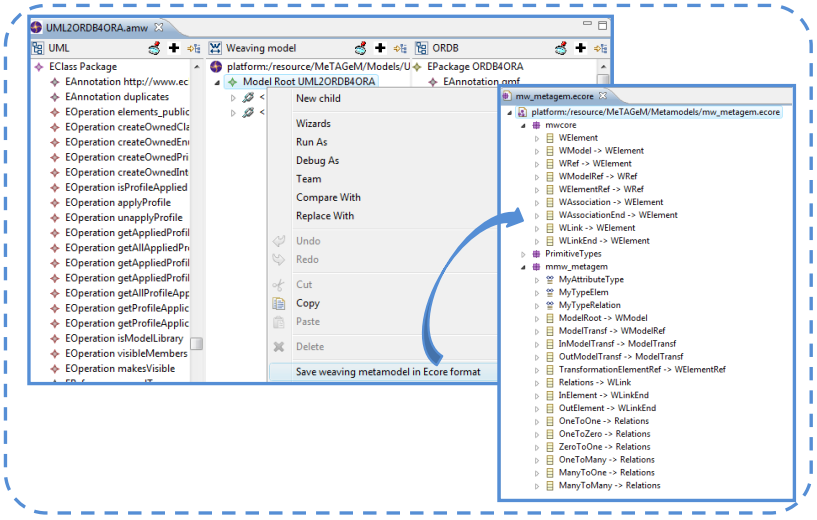

Figura 3-41. Meta-modelo M-TIP en Formato *Ecore*

A partir de la definición del meta-modelo en formato *Ecore* y utilizando las facilidades brindadas por Eclipse y EMF se obtiene el meta-modelo en formato gráfico (Figura 3-42)

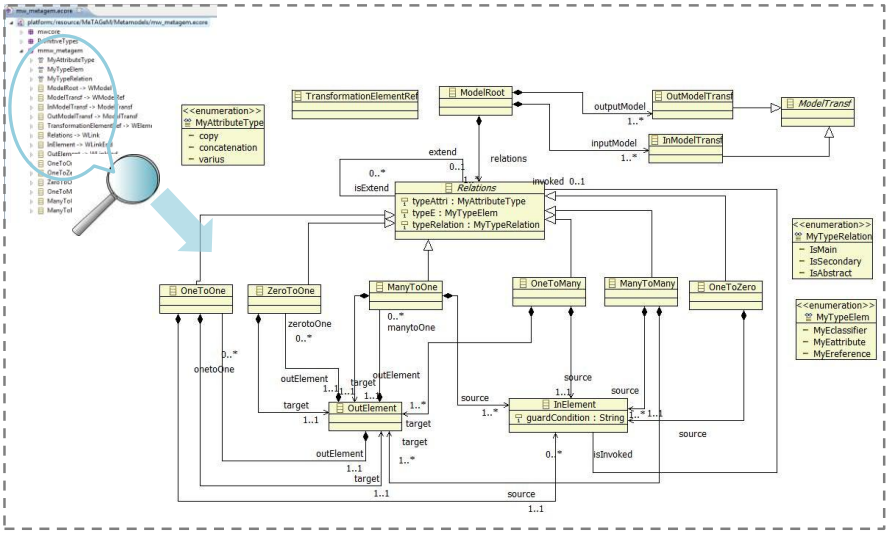

Figura 3-42. Meta-modelo M-TIP en Formato Gráfico

3.2.4.2 Metamodelo de Transformación Específico de Plataforma

El meta-modelo de transformación específico de plataforma M-LTH se implementa en términos de *Ecore*, el meta-meta-lenguaje de EMF. *Ecore* es una implementación simplificada de EMOF (*Essential MOF*, [150]) que genera un archivo con extensión *.ecore* donde se implementan cada una de las meta-clases especificadas en el meta-modelo. Es importante mencionar, que debido a la naturaleza XML subyacente de *Ecore*, cualquier meta-modelo definido a partir de *Ecore* debe incluir un elemento raíz. En M-LTH, el elemento raíz es la meta-clase *Module*, del cual dependen el resto de los elementos.

A partir del archivo *.ecore* que representa el meta-modelo basado en EMF, es decir, la sintaxis abstracta, EMF permite generar un editor, con formato de árbol (*tree-like*), de modelos conforme a dicho meta-modelo es decir, la sintaxis concreta.

El primer paso para la generación del editor es la creación, a partir del modelo *Ecore*, de un modelo EMF (*Genmodel*). El modelo *Ecore* es conocido como modelo núcleo o modelo base, y el modelo *Genmodel* se conoce como modelo generador.

La mayoría de los datos necesarios utilizados por el generador de EMF se almacenan en el modelo base (modelo *.ecore*); por ejemplo, se almacenan las clases generadas, sus nombres, atributos y referencias. Sin embargo, hay más información, como por ejemplo, dónde almacenar el código generado y el prefijo a utilizar para los elementos generados, que no se almacenan en el modelo base. Todos estos datos deben ser almacenados en algún lugar para asegurar su disponibilidad por si hubiera que regenerar el modelo en un futuro. El generador de código EMF utiliza el modelo *Genmodel* para almacenar esta información.

La importancia de todo esto es que a partir de EMF se genera el modelo *Genmodel* que permite generar los editores de modelos. En la Figura 3-43 se muestra gráficamente dicha situación, *Genmodel* es un modelo EMF, y las clases que lo componen tienen referencias a las clases del modelo *Ecore* y proporcionan información adicional sobre las mismas.

La separación del modelo *GenModel* del modelo base tiene la ventaja de que el meta-modelo *Ecore* permanece puro e independiente de cualquier información que sólo es relevante para la generación de código. La desventaja principal es que el modelo generador puede desincronizarce si las referencias al modelo cambian. Para evitar este problema, las clases del modelo generador incluyen métodos que propagan los cambios desde el modelo base al modelo generador. El uso de estos métodos aseguran que los dos archivos se mantengan sincronizados automáticamente.

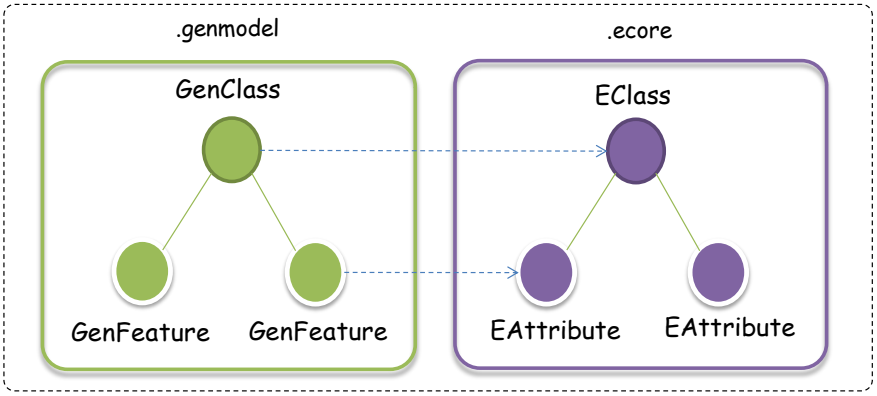

Figura 3-43. Relación entre el Modelo *.genmodel* y el Modelo *.ecore*

A partir del *Genmodel*, EMF genera el código JAVA estructurado en tres grandes paquetes: el código del modelo (*Java model* o *model code*), el codigo de edición (*Java edit* o *edit code*) y el código del editor (*Java editor* o *editor code*).

Además se puede generar el código para los casos de pruebas (*test code*) aunque en realidad rara vez se usa.

En esencia, el *model code* permite el acceso al meta-modelo, crear modelos conformes a dicho meta-modelo, serializarlo y des-serializarlo. El *model code* es utilizado por el *edit code* y el *editor code*, que relaciona las funcionalidades con una interfaz gráfica; es decir, ofrecen un editor simple (en forma de árbol) para el manejo de los modelos conformes al meta-modelo *Ecore* utilizado como punto de partida (Figura 3-44)

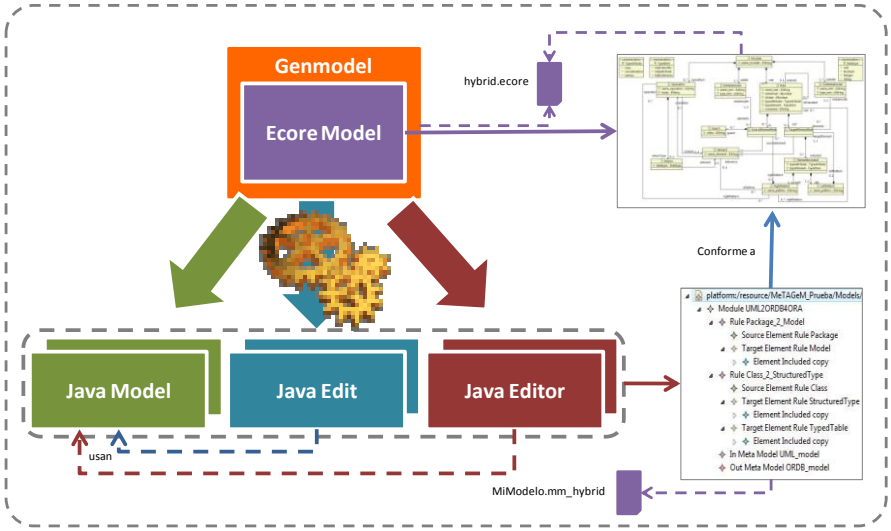

Figura 3-44. Vista General de la Generación de Editores EMF

A modo de resumen, a partir del modelo *.ecore* y del modelo *.genmodel*, donde se recoge la sintaxis abstracta del meta-modelo, se genera el código JAVA que implementa un editor, con formato de árbol, sencillo y potente, que permite modelar modelos conforme al meta-modelo. De esta manera, para el caso que se presenta en esta tesis, se puede editar modelos *.mm_hybrid* conformes al meta-modelo M-LTH definido a nivel PSM de MeTAGeM.

3.2.4.3 Meta-modelos de Transformación Dependientes de Plataforma

Como se comentó en la sección 3.1.2.2, para el modelado de las transformaciones a nivel PDM se propone el uso de ATL y RubyTL. Para el modelado usando el lenguaje ATL se utiliza el *plug-in* de ATL para Eclipse, por lo que no es necesario implementar el DSL correspondiente. Por el contrario, para el modelado de las transformaciones usando RubyTL es necesario implementar el DSL de RubyTL. En esta sección se muestra la implementación de dicho DSL.

La implementación de la sintaxis abstracta del meta-modelo del DSL de RubyTL se realiza en terminos de *Ecore* siguiendo los pasos indicados en la sección 3.2.4.2. Esta sección se centra en mostrar cada uno de los elementos que conforman el meta-modelo.

Al igual que en la definición del meta-modelo M-LTH, es necesario definir el elemento raíz del meta-modelo. En este caso, el elemento raíz es la meta-clase *Transformation* a partir de la cual se definen el resto de los elementos.

Cada meta-clase *Transformation* está compuesta por: elementos del tipo *Metamodel*, a través de las relaciones *SourceMetamodels* (identifica al meta-modelo origen) y *TargetMetamodels* (identifica al meta-modelo destino); elementos de tipo *Decorator*, que permiten especificar funciones que deben ser implementadas; y por elementos de tipo *Rule*, que permiten especificar las reglas de transformación entre los diferentes elementos de los meta-modelos (Figura 3-45).

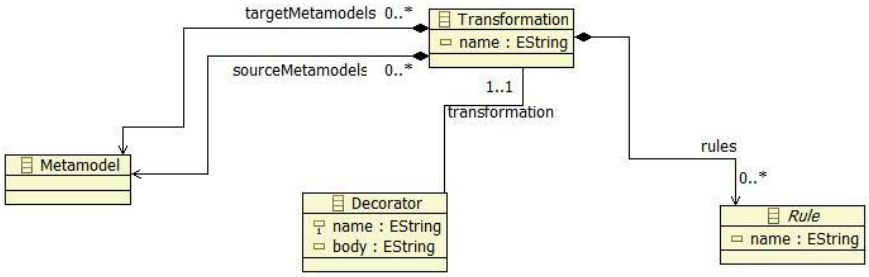

Figura 3-45. Vista Parcial del Meta-modelo de RubyTL – Meta-clase *Transformation*

En la Figura 3-46 se muestra la meta-clase *Rule* definida de forma abstracta. A diferencia que en ATL, en RubyTL una *Rule* se define sobre un único elemento del meta-modelo origen (*FromElement*) y pueden generar más de un elemento del meta-modelo destino (*ToElement*). Además, se pueden establecer condiciones que se deben cumplir para que la regla sea aplicada (*Filter*). Las relaciones entre las propiedades de los elementos que participan en la relación se establecen usando la meta-clase *Mapping* que esta compuesta por elementos de la meta-clase *Binding*.

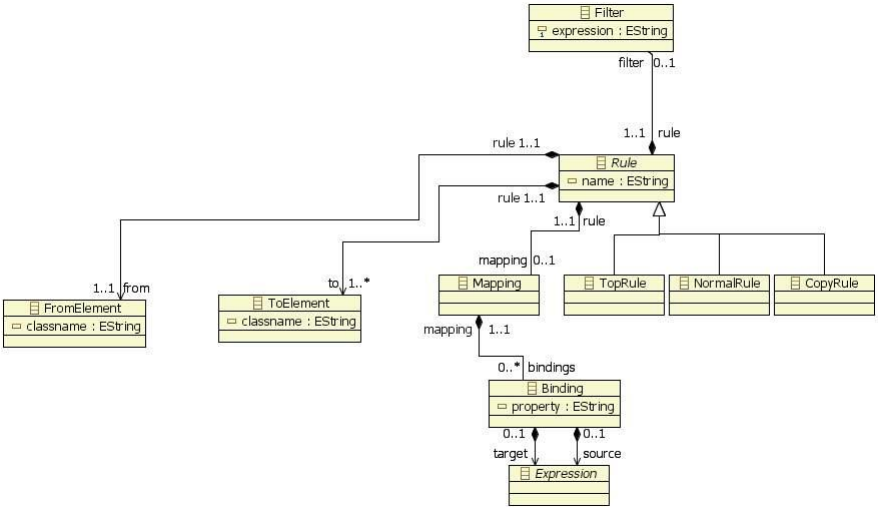

Figura 3-46. Vista Parcial del Meta-modelo de RubyTL – Meta-clase *Rule*

En el CD adjunto se encuentra la implementación completa del DSL que permite realizar el modelado de las transformaciones a nivel PDM conformes a RubyTL.

3.2.4.4 Personalización de los Editores

Como se ha visto en las secciones anteriores el desarrollo de los editores para cada uno de los DSLs propuestos por MeTAGeM se realiza usando EMF, que permite obtener los editores gráficos, en formato de árbol, con la funcionalidad requerida. Sin embargo, estos editores son muy genéricos, esto es, todos tienen la misma apariencia y muestran la misma información de cada elemento en todos los casos. En los editores de cada uno de los DSLs es necesario mostrar información concreta de cada elemento del modelo, para lo que se ha trabajado en la personalización de los mismos.

A continuación, se presenta las personalizaciones realizadas en los editores gráficos de MeTAGeM.

Personalización de Iconos de cada Elemento

El editor generado con EMF para representar cada uno de los elementos de los modelos propone iconos genéricos. Una de las mejoras realizadas es la de personalizar dichos iconos con imágenes representativas de cada elemento. Para esto, en primer lugar se ha seleccionado para cada elemento una imagen adecuada y luego se ha procedido a cambiar la imagen original por la seleccionada.

En la parte izquierda de la Figura 3-47 se muestra el editor genérico propuesto por EMF para el DSL M-LTH. En la parte derecha se muestra el mismo editor con los iconos personalizados (*Rule*, *SourceElement*, *TargetElement*, etc).

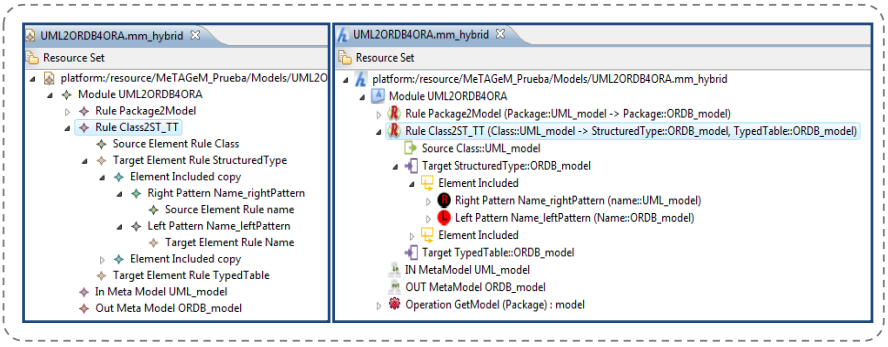

Figura 3-47. Personalización de Iconos

De la misma manera se ha realizado la personalización de los iconos del editor de modelos conforme a RubyTL y de los lanzadores de las transformaciones.

Información Mostrada en cada Elemento

Otra de las mejoras realizadas a los editores es el de la información que se muestra de cada elemento de los modelos. En la Figura 3-48 se muestra un ejemplo muy simple para explicar mejor el problema detectado y la solución realizada.

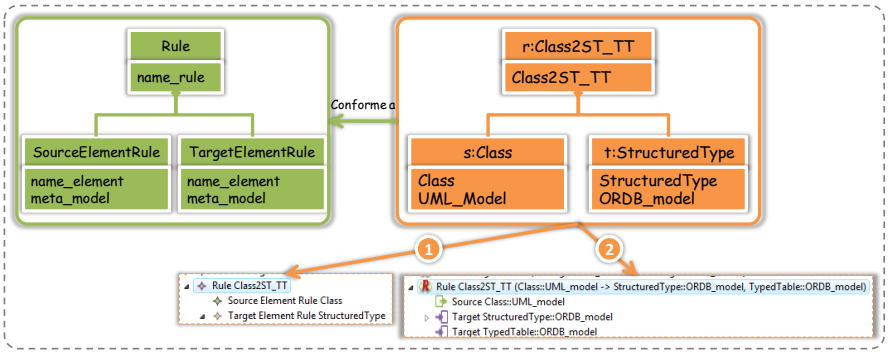

Figura 3-48. Información Mostrada en los Nodos

En el lado izquierdo de la figura se muestra un extracto del meta-modelo M-LTH y en el lado derecho se muestra un ejemplo de instanciación. El elemento de tipo *Rule Class2ST_TT* tiene un elemento de tipo *SourceElementRule* llamado *Class* y un elemento de tipo *TargetElementRule* llamado *StructuredType*. En la

parte inferior de la figura se muestra cómo se representa esta regla en el editor generado por EMF (1) y en el editor personalizado de MeTAGeM (2). En este último caso, se muestra, además del nombre de la regla, el valor de los elementos *SourceElementRule* y *TargetElementRule*, indicando su nombre y el meta-modelo al cual pertenecen.

Para que se muestre el elemento *Rule* y los elementos *SourceElementRule* y *TargetElementRule*, así como el meta-modelo al que pertenecen, es necesario modificar el código JAVA generado por EMF. En particular, se debe modificar el método *getText()* de la clase *RuleItemProvider*, añadiéndole la funcionalidad requerida de esta manera se muestra mejor y más completa información en términos de usabilidad.

Se ha procedido de la misma manera con el resto de los elementos de los DSLs de MeTAGeM.

Selección Automática del Elemento Raíz del Modelo

En las secciones anteriores se menciona que todos los modelos basados en EMF deben tener un elemento raiz (*root element*). En el momento en que se crea un nuevo modelo, conforme a un meta-modelo, el *wizard* proporcionado por EMF pregunta al usuario cúal es la meta-clase que será instanciada como elemento raiz. Cuando el meta-modelo tiene muchos elementos, encontrar la meta-clase correcta puede ser complicado. Para evitar la necesidad de dicha selección, se ha modificado el código EMF generado. En particular, en cada editor se modifica el método *createControl()* de la clase *MM_HybridModelWizardInitialObjectCreationPage*. De esta manera, en el momento de la creación de un nuevo modelo, el *wizard* identifica automáticamente el elemento raíz. En la Figura 3-49 se muestra el *wizard* original (1) y el *wizard* modificado de MeTAGeM (2).

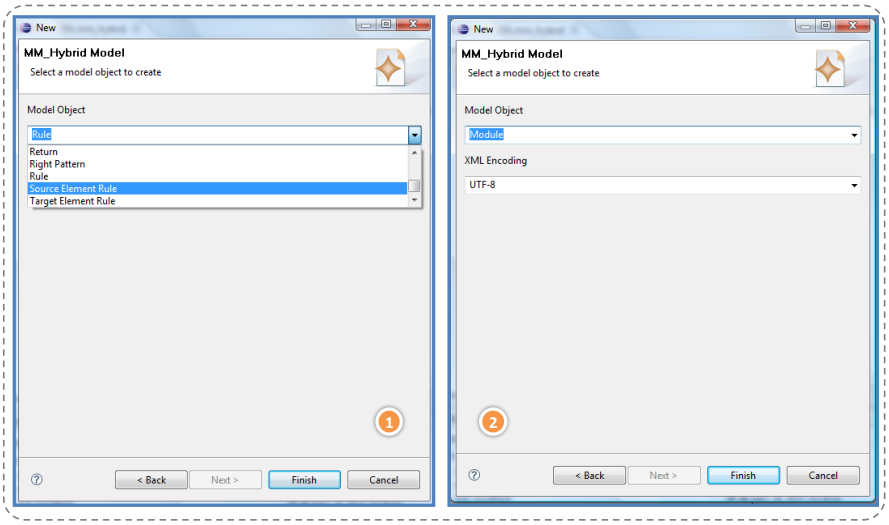

Figura 3-49. Selección Automática del Elemento Raiz

3.2.5 Implementación de Transformaciones

Hasta el momento se ha mostrado cómo a través de las facilidades de meta-modelado brindadas por EMF se definen nuevos DSL que representan los meta-modelos definidos en cada uno de los niveles de MeTAGeM. Para esto, se ha definido la sintaxis abstracta de cada DSL, usando KM3 y EMF y la sintaxis concreta usando AMW y EMF para realizar la generación de editores.

El próximo paso para la integración de estos nuevos DSL es el desarrollo de las transformaciones de modelos que permitirá conectar los modelos definidos a partir de los DSL con el resto de modelos que propone MeTAGeM. Para esto, partiendo de la especificación formal de las transformaciones usando gramáticas de grafos realizada en la sección 3.1.3, en esta sección se realiza la implementación de las mismas, usando para ello el lenguaje ATL.

ATL proporciona una manera de generar modelos destino a partir de un conjunto de modelos origen. Está desarrollado sobre la plataforma de Eclipse; su IDE (*IntegrateD Environment*) comprende una serie de facilidades estándares (resaltado de sintaxis, depurador, editor, etc) que facilita el desarrollo de las transformaciones de modelos. ATL está basado principalmente en el estándar de OCL y soporta la aproximación declarativa y la imperativa conjuntamente, es decir la aproximación híbrida, aunque sus autores recomiendan el uso de la aproximación declarativa.

Las transformaciones en ATL se implementan definiendo un conjunto de reglas: cada regla especifica un patrón origen y un patrón destino, ambos a nivel de meta-modelo. Cuando la transformación ATL se ejecuta, el motor de ATL establece las relaciones entre los patrones origen y cada uno de los elementos del modelo origen. Por cada relación se instancia el patrón destino en elementos del modelo destino.

En contraste con la mayoría de los lenguajes existentes, ATL permite la herencia de reglas y la programación implícita y explícita de las mismas. La programación implícita se soporta por medio de las construcciones imperativas. Cuando se comienza la ejecución de la transformación, el algoritmo comienza ejecutando las reglas definidas como puntos de entrada (*entry point*) pudiendo llamar a otras reglas a partir de éstas. Terminada esta primera fase, el motor verifica automáticamente las coincidencias entre los patrones origen y los elementos del modelo origen y ejecuta las reglas correspondientes. Por último, se ejecutan las reglas definidas como puntos de salida (*exit point*). La programación explícita se soporta por medio de la llamada a reglas desde el bloque imperativo de otras reglas. Estas reglas se transforman en instrucciones para la máquina virtual de ATL, que ejecuta la transformación. Este comportamiento es similar a JAVA y su máquina virtual.

A continuación, se muestra la implementación realizada de cada uno de lo módulos correspondientes a las reglas de transformación de modelo a modelo.

3.2.5.1 Transformaciones entre M-TIP y M-LTH

Partiendo de la definición realizada en la Tabla 3-2 y de la formalización de las reglas usando gramática de grafos, en esta sección se muestra la implementación de las mismas usando el lenguaje ATL. El meta-modelo M-TIP es una extensión del meta-modelo base de AMW, y permite realizar modelos indicando las relaciones entre elementos de los meta-modelos que participan en la transformación. Para la definición de las reglas de transformación entre los modelos M-TIP a modelos M-LTH, se debe tomar como entrada el meta-modelo de AMW y los meta-modelos que intervienen en la relación.

Como se puede ver en la Figura 3-50, estos meta-modelos están definidos conforme a MOF. El modelo generado a partir de esta transformación será conforme a M-LTH (MM_Hybrid en el código).

```
--@atlcompiler atl2006
module MeTAGeM2Hybrid;
create OUT : MM_Hybrid from IN : AMW, left : MOF, right : MOF;
```

Figura 3-50. Cabecera Módulo ATL - MeTAGeM 2Hybrid

Una vez definida la cabecera del módulo que contiene las reglas de transformación se comienza con la implementación de cada una de las reglas. En primer lugar se realiza la implementación de la regla que transforma los elementos de tipo *ModelRoot* a elementos de tipo *Module* (Figura 3-51), completándose la propiedad *name* y las referencias a los elementos *inMetaModel*, *outMetaModel* y *Rule*. De acuerdo al meta-modelo MM_Hybrid especificado, un elemento del tipo *Module* está compuesto por elementos de tipo *inMetaModel*, *outMetaModel*, *Rule* y *Operation*.

```
rule Module {
   from
      amw : AMW!ModelRoot
   to
      hybrid : MM_Hybrid!Module (
                name_module <- amw.name,
                inMM <- amw.inputModel,
                outMM <- amw.outputModel,
                "rule" <- amw.relations)
}
```

Figura 3-51. Regla de Transformación ATL – *Module*

Los elementos de tipo *Operation* no tienen su correspondencia a nivel PIM, ya que el meta-modelo de nivel PIM sólo recoge las relaciones entre los elementos de los meta-modelos participantes. El desarrollador podrá agregar, de forma manual, elementos del tipo *Operation* al modelo generado.

La Figura 3-52 muestra la definición de las reglas que transforman los elementos de tipo *InModelTransf* y *OutModelTransf* en los elementos *InMetaModel* y *OutMetaModel*, respectivamente. Estos elementos representan a los meta-modelos origen y destino que participan en la transformación final. Para cada uno de los elementos generados se especifica el nombre del meta-modelo (*name_mm*) y el tipo del meta-modelo (*type_mm*).

```
rule inModel{
   from
      in_MM_amw: AMW!InModelTransf
   to
      in_MM_hybrid: MM_Hybrid!InMetaModel(
             name_mm <- in_MM_amw.name+'_model',
             type_mm <- in_MM_amw.name)
}

rule outModel{
   from
      out_MM_amw: AMW!OutModelTransf
   to
      out_MM_hybrid: MM_Hybrid!OutMetaModel(
             name_mm <- out_MM_amw.name+'_model',
             type_mm <-  out_MM_amw.name)
}
```

Figura 3-52. Regla de Transformación ATL – *inModel* y *outModel*

En el momento de realizar la transformación de los elementos de tipo *Relations* se ha considerado conveniente realizar una regla abstracta *Relations2Rule* (Figura 3-53) que agrupara las propiedades de los diferentes tipos de relaciones.

```
abstract rule Relations2Rule{
   from
      relation:AMW!Relations(relation.isNotIncluded())
   to
      r_hybrid:MM_Hybrid!Rule(
            name_rule <- relation.getRuleName(),
            isAbstract <- relation.typeRelation=#IsAbstract,
            isMain <- relation.typeRelation=#IsMain,
            "extends" <- relation.extend,
            isExtended <- relation.isExtend,
            typeAttribute <- relation.typeAttri,
            typeElement <- relation.typeE)
}
```

Figura 3-53. Regla de Transformación ATL – *Relations2Rule*

Para mejorar la modularidad de las reglas se han definido algunas funciones auxiliares llamadas *helpers* como, por ejemplo, el *helper getRuleName* (Figura 3-54) que permite recuperar el nombre de la regla que se está generando. Dicho nombre se genera a partir de la concatenación de los nombres de los elementos de tipo *InElement* y *OutElement* que participan en la relación. En caso de que el desarrollador no haya especificado dichos nombres en el modelo a nivel

PIM, por medio de otro *helper*, *getInOutPatternName*, se genera un nombre por defecto. De esta manera se asegura que todas las reglas tengan un nombre, lo que evita errores en la generación de los siguientes modelos.

```
helper context AMW!Relations def : getRuleName () : String =
      if self.name.oclIsUndefined() then
            self.getInOutPatternName()
      else
            if self.name = '' then
                  self.getInOutPatternName()
            else
                  self.name
            endif
      endif;

helper context AMW!Relations def : getInOutPatternName () : String =
      if ((self.oclIsTypeOf(AMW!ZeroToOne))  or  (self.oclIsTypeOf(AMW!OneToZero))) then
            'R' + thisModule.countRules.toString()
      else
      if not self.source.asSequence()->first().oclIsUndefined()
            and not self.target.asSequence()->first().oclIsUndefined() then
            self.source.asSequence()->first().name + '_2_'+ self.target.asSequence()->first().name
      else
            'R' + thisModule.countRules.toString()
      endif
      endif;
```

Figura 3-54. *Helper* ATL – *getRuleName*

En la Figura 3-55 se muestra la regla que permite transformar elementos del tipo *ManyToMany* del meta-modelo de nivel PIM a elementos del tipo *Rule* del meta-modelo de nivel PSM. Como se puede observar, se extiende la regla *Relations2Rule* por medio de la palabra reservada *extends* y se agrega la generación de las propiedades *in* y *out* asignándoles los valores de las propiedades *source* (*InElement*) y *target* (*OutElement*) del elemento *ManyToMany*.

```
rule ManyToMany2rule extends Relations2Rule{
   from
      relation: AMW!ManyToMany
   to
      r_hybrid: MM_Hybrid!Rule(
            "in" <- relation.source.asSequence(),
            out <- relation.target.asSequence())
      do {
            thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura 3-55. Regla de Transformación ATL – *ManyToMany2Rule*

El resto de los tipos de relaciones, *OneToOne*, *OneToZero*, *ZeroToOne*, *OneToMany* y *ManyToOne*, se codifican de manera similar, variando para cada uno de los casos la generación de las propiedades *in* y *out*. En el Anexo D se muestra la codificación de todas las reglas.

3.2.5.2 Transformaciones entre M-LTH y PDM

En esta sección se muestra la implementación de las reglas de transformación definidas entre M-LTH y los meta-modelos de transformación dependientes de plataforma: ATL y RubtyTL.

3.2.5.2.1 Transformación M-LTH a ATL

Partiendo de la Tabla 3-3 y la formalización de las reglas usando gramática de grafos, se implementa las mismas usando el lenguaje ATL. En la Figura 3-56 se muestra la cabecera del módulo ATL en el que se definen las reglas de transformación. Como se puede ver, se define como meta-modelo origen (*IN*) el meta-modelo M-LTH (MM_Hybrid en el código) y como meta-modelo destino (*OUT*) el meta-modelo ATL.

```
module Hybrid2ATL;
create OUT : ATL from IN : MM_Hybrid;
```

Figura 3-56. Regla de Transformación ATL – *Hybrid2ATL*

La Figura 3-57 muestra la regla que implementa la transformación entre los elementos del tipo *Module* del meta-modelo MM_Hybrid y los elementos de tipo *Module* del meta-modelo ATL. En el elemento de tipo *Module* generado se completan las propiedades *isRefining* y *name* y las referencias: *inModels*, que indica los meta-modelos origen que participan en la transformación; *outModels*, que indica los meta-modelos destino de la relación; y *elements*, que representa a las reglas de transformación que forman parte del módulo.

```
rule Module {
   from
      mm_hybrid : MM_Hybrid!Module
   to
      atl : ATL!Module (
            isRefining <- false,
            name <- mm_hybrid.name_module,
            inModels <- mm_hybrid.inMM,
            outModels <- mm_hybrid.outMM,
            elements <- mm_hybrid."rule",
            commentsBefore <- Set {'-- @atlcompiler atl2006'})
}
```

Figura 3-57. Regla de Transformación ATL – *Module* - ATL

Para la transformación de los elementos de tipo meta-modelo origen (*InMetaModel*) y de tipo meta-modelo destino (*OutMetaModel*) se definen dos

reglas que generan el correspondiente elemento de tipo *OCLModel* en el modelo destino conforme al meta-modelo ATL (Figura 3-58).

```
rule inMM{
   from
      inMM_hybrid : MM_Hybrid!InMetaModel
   to
      inMM_ATL : ATL!OclModel(
          name <- inMM_hybrid.name_mm,
          metamodel <- ametamodelinMM),

      ametamodelinMM : ATL!OclModel (
          name <- inMM_hybrid.type_mm)
}

rule outMM{
   from
      outMM_hybrid : MM_Hybrid!OutMetaModel
   to
      outMM_ATL : ATL!OclModel(
           name <- outMM_hybrid.name_mm,
           metamodel <- ametamodeloutMM),

      ametamodeloutMM : ATL!OclModel(
           name <- outMM_hybrid.type_mm)
}
```

Figura 3-58. Regla de Transformación ATL – *inMM* y *OutMM* - ATL

La meta-clase *Rule* en ATL es abstracta, y se especializa en tres meta-clases diferentes: *MatchedRule*, *LazyMatchedRule* y *CalledRule*. La meta-clase *MatchedRule* sirve para especificar explícitamente qué elemento del meta-modelo origen se transforma en qué elemento del meta-modelo destino; este tipo de regla se ejecuta siempre que el motor de ATL encuentre una coincidencia entre el patron origen de la regla y una parte del modelo origne. La meta-clase *LazyMatchedRule* tiene un comportamiento similar, aunque la diferencia radica en que su ejecución debe realizarce explícitamente. Por último, la meta-clase *CalledRule* representan las reglas utilizadas para generar elementos en el modelo destino de manera imperativa, es decir, no existe un elemento del modelo origen a partir del cual se genera el elemento del modelo destino.

Para la transformación de los elementos de tipo *Rule* del meta-modelo M_LTH, a sus correspondientes elementos de tipo *Rule* en ATL es necesario definir tres tipos de reglas diferentes, una por cada tipo de regla en ATL. Además, es necesario determinar la condición por la cual un elemento de tipo *Rule* en M-

LTH se transforma a un elemento de tipo *MatchedRule*, *LazyMatchedRule* o *CalledRule* en ATL. Como se puede observar en la Figura 3-59, donde se transforma el elemento *Rule* a un elemento *MatchedRule*, en la condición de guarda se evalúa el valor de la propiedad *isMain* del elemento. Además, se especifica una función especial, el *helper getSizeIP* que verifica la cantidad de elementos del tipo *SourceElementRule* que dependen del elemento *Rule*.

De esta manera, si el valor de la propiedad *isMain* es *true* y el valor que retorna el *helper getSizeIP* es mayor que cero, es decir la regla tiene elementos de tipo *SourceElementRule*, entonces el elemento *Rule* se transforma a un elemento de tipo *MatchedRule*. Si el valor de la propiedad *isMain* es *false* el elemento se transforma a un elemeto de tipo *LazyMatchedRule*. Por último, si el valor de la propiedad *isMain* es *true*, pero el valor que retorna el *helper getSizeIP* es igual a cero, es decir, no existen elementos del tipo *SourceElementRule* que dependan del elemento *Rule*, entonces, el elemento *Rule*, se transforma a un elemento de tipo *CalledRule*. En la Figura 3-59 se muestra la regla que transforma el elemento *Rule* al elemento *MatchedRule*, los otros dos casos son similares a este. En el anexo D se muestra la implementación completa de todas las reglas.

```
rule createRule2MatchedRule{
   from
      mm_hybrid_rule : MM_Hybrid!Rule
      (mm_hybrid_rule.isMain=true and mm_hybrid_rule.getSizeIP()>0)
   to
      atl : ATL!MatchedRule (
            name <- mm_hybrid_rule.name_rule,
            isAbstract <- mm_hybrid_rule.isAbstract,
            isRefining <- false,
            isNoDefault <- false,
            superRule <- mm_hybrid_rule."extends",
            inPattern <- inPattern,
            outPattern <- outPattern,
            commentsBefore <- Set {'-- Comments -> This is a MatchedRule:' +
                                    mm_hybrid_rule.name_rule}),

      inPattern : ATL!InPattern (
            elements <- mm_hybrid_rule."in".asSequence()),
      outPattern : ATL!OutPattern(
            elements <- mm_hybrid_rule.out.asSequence())
}
```

Figura 3-59. Regla de Transformación ATL – *createRule2MatchedRule* - ATL

3.2.5.2.2 Transformación M-LTH a RubyTL

Partiendo de la Tabla 3-4 y la formalización de las reglas usando gramática de grafos, se implementan las mismas usando el lenguaje ATL. En la Figura 3-60 se muestra la cabecera del módulo ATL, en el que se definen las reglas de transformación. Como se puede ver, se define como meta-modelo origen (*IN*) al

meta-modelo M-LTH (MM_Hybrid en el código) y como meta-modelo destino (*OUT*) al meta-modelo RubyTL.

```
module Hybrid2RubyTL;
create OUT : RubyTL from IN : MM_Hybrid;
```

Figura 3-60. Regla de Transformación ATL – *Hybrid2RubyTL*

La Figura 3-61 muestra la regla que implementa la transformación entre los elementos del tipo *Module* del meta-modelo M-LTH y los elementos de tipo *Transformation* del meta-modelo RubyTL. En el elemento de tipo *Transformations* se completan la propiedad *name* y las referencias: *sourceMetamodels*, que indica el meta-modelo origen que participa en la transformación; *targetMetamodels*, que indica el meta-modelo destino de la relación; *rules*, que representa las reglas de transformación que forman parte del módulo; y *decorators*, que representa las funciones que pueden definirse.

```
rule Module {
   from
      mm_hybrid : MM_Hybrid!Module
   to
      rubytl : RubyTL!Transformation (
        name <- mm_hybrid.name_module,
        sourceMetamodels <- mm_hybrid.inMM,
        targetMetamodels <- mm_hybrid.outMM,
        rules <- mm_hybrid."rule",
        decorators <- mm_hybrid.operations)
}
```

Figura 3-61. Regla de Transformación ATL – *Module* - RubyTL

Para la transformación de los elementos de tipo meta-modelo origen (*InMetaModel*) y meta-modelo destino (*OutMetaModel*), se definen dos reglas que generan los correspondientes elementos de tipo *Metamodel* en el modelo destino conforme al meta-modelo RubyTL (Figura 3-62).

```
rule inMM{
   from
      inMM_hybrid : MM_Hybrid!InMetaModel
   to
      inMM_rubytl : RubyTL!Metamodel(
            name <- inMM_hybrid.name_mm)
}

rule outMM{
   from
      outMM_hybrid : MM_Hybrid!OutMetaModel
   to
      outMM_rubytl : RubyTL!Metamodel(
            name <- outMM_hybrid.name_mm)
}
```

Figura 3-62. Regla de Transformación ATL – *inMM* y *OutMM* - RubyTL

La meta-clase *Rule* en RubyTL es de tipo abstracta, y se especializa en tres meta-clases diferentes: *TopRule*, *NormalRule* y *CopyRule*.

Para la transformación de los elementos de tipo *Rule* del meta-modelo M-LTH a sus correspondientes elementos de tipo *Rule* en RubyTL, es necesario definir tres tipos de reglas diferentes, una por cada tipo de regla. Además, es necesario determinar la condición por la cual un elemento de tipo *Rule* en M-LTH se transforma a un elemento de tipo *TopRule*, *NormalRule* o *CopyRule* en RubyTL. Como se puede observar en la Figura 3-63, donde se transforma el elemento *Rule* a un elemento *TopRule*, en la condición de guarda se evalúa el valor de la propiedad *isMain* del elemento. Además, se especifica una función especial, *helper getSizeIP*, que verifica la cantidad de elementos del tipo *SourceElementRule* que dependen del elemento *Rule*.

De esta manera, si el valor de la propiedad *isMain* es *true* y el valor que retorna el *helper getSizeIP* es igual a uno, es decir, la regla tiene un elemento de tipo *SourceElementRule*, entonces el elemento *Rule* se transforma a un elemento de tipo *TopRule*. Si el valor de la propiedad *isMain* es *false* y la propiedad *typeAttribute* es igual a *isUnique*, el elemento se transforma a un elemento de tipo *NormalRule*. Por último, si el valor de la propiedad *isMain* es *true*, pero el valor *typeAttribute* es distinto a *isUnique*, entonces se transforma a un elemento de tipo *CopyRule*. En la Figura 3-63 se muestra la regla que transforma el elemento *Rule* al elemento *TopRule*, los otros dos casos son similares a este. En el anexo D se muestra la implementación completa de todas las reglas.

```
rule createRule2TopRule{
   from
      mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()=1
      and mm_hybrid_rule.isMain=true)
   to
      rubytl : RubyTL!TopRule (
            name <- mm_hybrid_rule.name_rule,
            "from" <- mm_hybrid_rule."in".asSequence().first(),
            "to" <- mm_hybrid_rule.out.asSequence(),
            comment <- mm_hybrid_rule.getComment(),
            mapping <- amapping),

      amapping : RubyTL!Mapping (
            bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura 3-63. Regla de Transformación ATL – *createRule2TopRule* - RubyTL

3.2.6 Generación de Código

Como se ha comentado en la sección 3.1.3.3 para obtener el código que implementa la transformación a partir del modelo de transformación en ATL y/o RubyTL se utiliza TCS. En el caso de ATL se utiliza el inyector/extractor incluido en el *plug-in* de ATL. En el caso de RubyTL es necesario realizar la implementación con TCS. A continuación se detallan ambas situaciones.

3.2.6.1.1 Generación de Código en ATL

El *framework* de ATL tiene una estructura modular, basada en la plataforma de Eclipse. En la Figura 3-64 se muestra de forma gráfica el conjunto de módulos básicos de ATL que se encargan de las operaciones que se llevan a cabo durante la ejecución de la transformación.

El primer módulo es el *parser* de ATL (*injector*) que toma como entrada un archivo con extensión *.atl* y genera un modelo conforme al meta-modelo de ATL como salida. Además, genera un modelo donde se almacenan todos los errores detectados en el código (*problem model*) y que luego es utilizado por el editor de ATL para mostrar los errores correspondientes en el archivo ATL.

El segundo módulo es el compilador de ATL, que es el encargado de transformar el modelo ATL a *bytecode* (el código de la transformación) representado en formato ASM interpretable por la máquina virtual de ATL (*ATL Virtual Machine*, ATL VM, [15]). De tal manera que, a partir de un modelo de transformación ATL, conforme al meta-modelo de ATL, se puede obtener el código de la transformación en el lenguaje ATL. Este módulo está implementado usando el compilador orientado a DSL ACG (*ATL Code Generation*) [15].

El último módulo es ATL VM, que permite ejecutar la transformación compilada sobre un modelo origen específico y generar un modelo destino determinado.

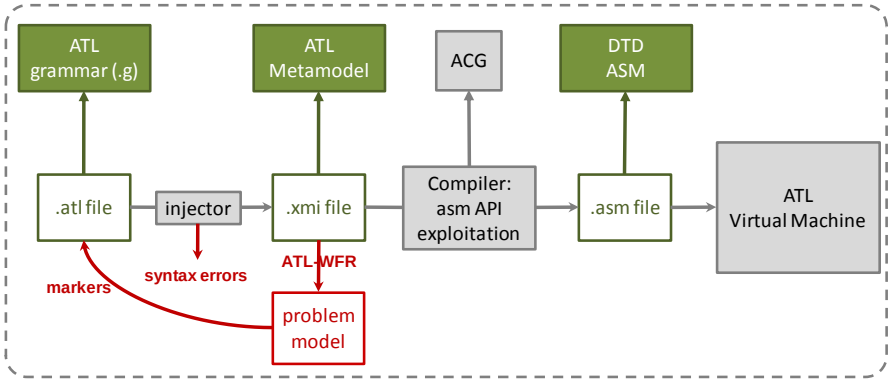

Figura 3-64. Framework de ATL

En la Figura 3-65 se muestra como a partir del modelo ATL de la transformación, el archivo *UML2ORDB4ORA-atl.ecore* en nuestro ejemplo, se genera el código ATL que implementa dicha transformación en un archivo con extensión *atl*, en la figura el archivo *UML2ORDB4ORA-atl.atl*.

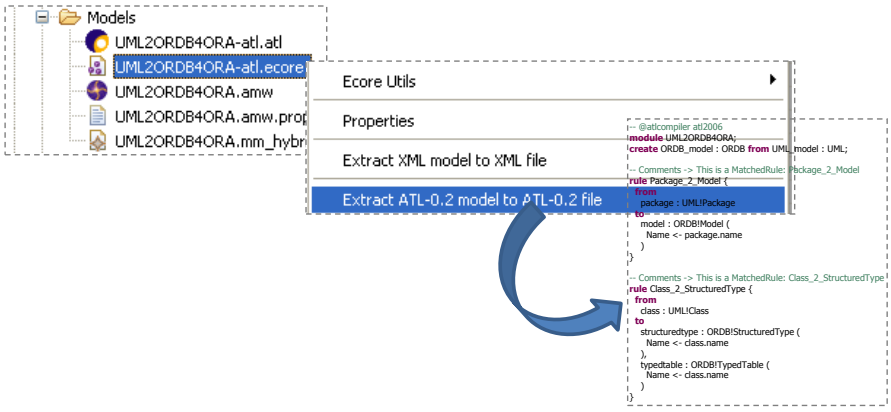

Figura 3-65. Generación de Código ATL

3.2.6.1.2 Generación de Código RubyTL

Para la generación de código a partir de los modelos definidos a nivel PDM conformes a RubyTL, se utiliza TCS, un componente del proyecto Eclipse GMT que permite definir la sintaxis concreta de un DSL.

A partir de la definición de la sintaxis concreta de dicho DSL, TCS permite, por un lado, obtener mediante un mecanismo de extracción el código que implementa la transformación a partir del modelo generado a nivel PDM, y, por otro lado, a partir del código se puede obtener, mediante un mecanismo de inyección, el modelo de nivel PDM.

Como se comentó en las secciones anteriores, para implementar las transformaciones con RubyTL en el nivel PDM de MeTAGeM ha sido necesario implementar un DSL que permita la manipulación de modelos. Para realizar la generación de código a partir de dichos modelos, es necesario realizar la especificación de la sintaxis concreta del DSL de RubyTL usando TCS. A continuación se detalla el proceso seguido.

Para comenzar con la definición de la sintaxis concreta del meta-modelo del DSL, en primer lugar, se debe crear un nuevo proyecto de tipo *TCS Language Project* al que se le debe asignar un nombre, que en el caso que se muestra, se corresponde con el nombre del lenguaje y una extensión de los ficheros de código que se generarán (.rb). Automáticamente se genera un proyecto con una estructura de carpetas como la que se ve en la Figura 3-66.

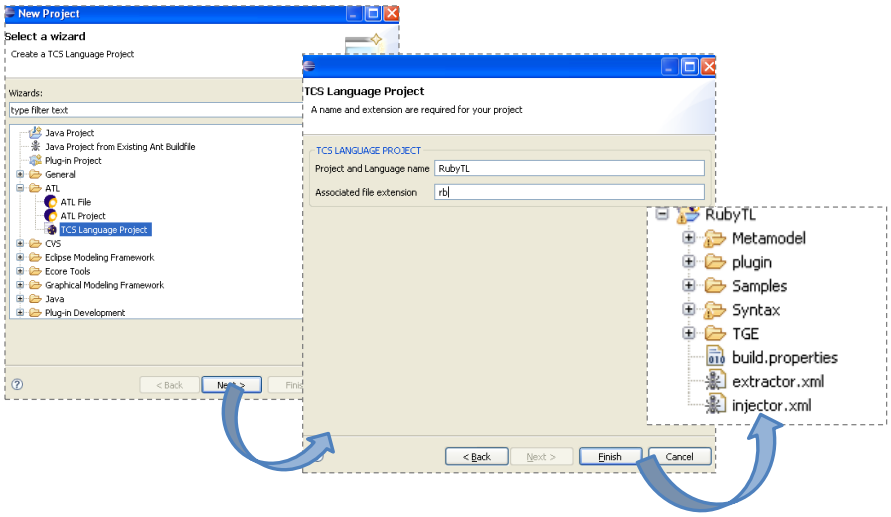

Figura 3-66. Proyecto TCS

La carpeta *Metamodel* contiene el meta-modelo del DSL definido en formato *Ecore* y KM3; además, se incluye un fichero *.ann* que permite incluir anotaciones sobre el meta-modelo en KM3. Al crear el proyecto, TCS genera un meta-modelo por defecto que puede ser editado por el usuario o sustituido por otro

meta-modelo (conservando el nombre de los ficheros con extensión *.ecore* y *.km3*).

La carpeta *plugin* contiene el *plug-in* (fichero con extensión *jar*) que permite instalar el proyecto TCS que se está desarrollando.

La carpeta *Samples* contiene un ejemplo de un fichero de código especificado mediante el DSL definido.

La carpeta *Syntax* contiene: el fichero con extensión *.tcs* en el que se define la sintaxis concreta del DSL; el fichero con extensión *.g* en el que se define la gramática del lenguaje en términos de ANTLR; y un fichero *RubyTL-parser.jar* que contiene las clases *Lexer* y *Parser* que permiten realizar las transformaciones entre los modelos y los ficheros de código del lenguaje.

La carpeta *TGE* contiene dos ficheros con extensión *.xmi*: *RubyTL-Editor.xmi* y *RubyTL-Outline.xmi*; que permiten definir el aspecto de las vistas del editor de texto y del *Outline*.

Una vez creado el proyecto se debe especificar; por un lado, el DSL usando KM3; y por otro lado, la sintaxis concreta del meta-modelo por medio de TCS. La Figura 3-67 muestra un extracto de código del archivo *RubyTL.tcs* donde se muestra la definición de la sintaxis concreta de la meta-clase *Transformation* y la meta-clase *Metamodel*.

RubyTL.tcs

```
        value = "Double.valueOf(%token%)";
-- END Primitive templates

-- BEGIN Class templates
-- Specifies representation of classes.
-- This is the main section to work on.

    template Transformation main context
        :   "transformation" name{as = stringSymbol}   <newline>
            "input" sourceMetamodels <newline>
            "output" targetMetamodels
            (isDefined(decorators)?<newline><newline>)
            decorators{separator=<newline><newline>}
            (isDefined(rules)?<newline><newline>)
            rules {separator=<newline><newline>}
        ;

    template Metamodel addToContext
        :   name{as = stringSymbol}(isDefined(uri)?" => "uri{as = stringSymbol})
        ;
```

Figura 3-67. Definición Sintaxis Concreta RubyTL

Cada vez que se actualiza el fichero con extensión *.km3*, o el fichero con extensión *.tcs*, el proyecto se reconstruye de forma automática generando, de acuerdo a los cambios que se realizaron, el resto de componentes del proyecto

(meta-modelo en formato Ecore, la gramática, el *plug-in*, etc.). Cada vez que se modifica el fichero *.tcs* es necesario generar nuevamente el modelo TCS (Figura 3-68).

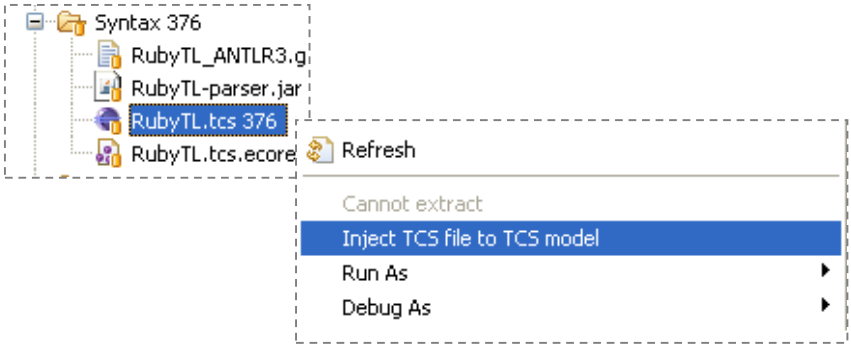

Figura 3-68. Generación de Modelo TCS

Es importante mencionar, que el *plug-in* que se genera automáticamente no funciona correctamente. Para poder utilizar los mecanismos de extensión e inyección proporcionados por TCS, es necesario crear un nuevo proyecto de Eclipse de tipo *Plug-in Project* (o emplear uno ya creado, en el caso que se presenta, el propio *plug-in* de MeTAGeM) y añadir la siguiente extensión *org.eclipse.gmt.tcs.metadata.language* en el fichero *plugin.xml* (Figura 3-69).

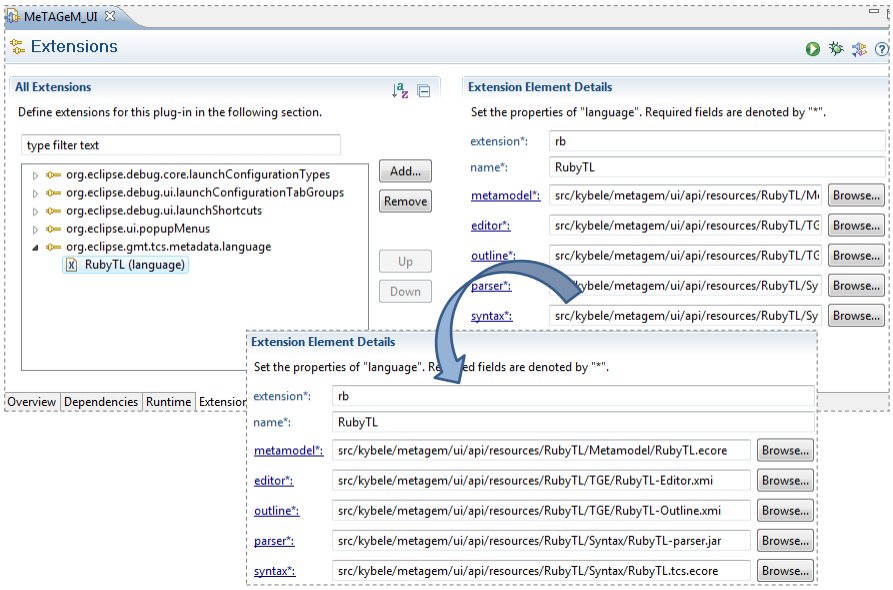

Figura 3-69. Extensión del *Plug-in* de MeTAGeM

A modo de resumen, siguiendo los pasos detallados anteriormente se define la sintaxis concreta del meta-modelo del DSL utilizando TCS. A partir de esta definición de la sintaxis concreta se crea un *plug-in* para Eclipse que permite generar ficheros de código a partir de modelos conformes al meta-modelo de RubyTL y viceversa. Para esto, TCS proporciona dos controles (lanzadores a partir de este momento) que realizan la ejecución de los mecanismos de extracción y de inyección. Así, en el caso del lenguaje RubyTL se puede realizar la extracción de código a partir de un modelo cuya extensión sea *.rb.xmi* o *.rb.ecore* y la inyección si se trata de un fichero con extensión *.rb*. Es importante mencionar que, si bien TCS proporciona de forma automática los dos lanzadores, para la herramienta MeTAGeM, se han creado lanzadores propios con la intención de trabajar con modelos cuya extensión es *.rubytl* (Figura 3-70). Esta decisión está motivada por el empleo de editores personalizados para este tipo de modelos.

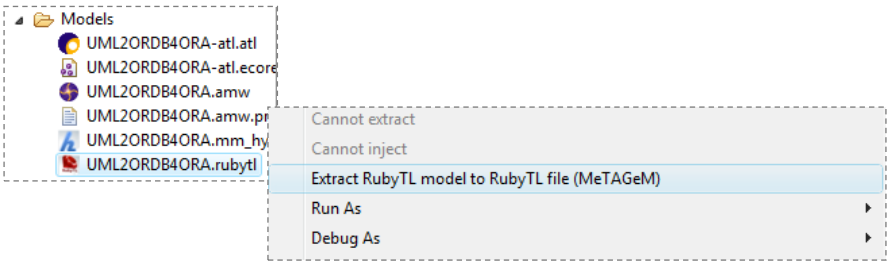

Figura 3-70. Mecanismo de Extracción de RubyTL

3.2.7 Verificación Automática de Modelos

Por último, en esta sección se explica cómo se completa la especificación de los DSLs de MeTAGeM por medio de la definición de un conjunto de restricciones adicionales que cada modelo generado deberá cumplir para ser un modelo válido. Para realizar la verificación automática de los modelos en MeTAGeM se usa EVL un lenguaje proporcionado por Epsilon.

Epsilon [120] (*Extensible Platform for Specification of Integrated Languages for mOdel Management*) es un componente de Eclipse que soporta algunas de las tareas relacionadas con el desarrollo dirigido por modelos. Para esto, tiene un conjunto de lenguajes especializados en las diferentes tareas como comparación de modelos o *merging*.

Epsilon Validation Language [118] (EVL) es uno de esos lenguajes, que permite especificar y evaluar restricciones sobre los modelos. Para esto, se deben definir las restricciones, que deben ser verificadas, a nivel de meta-modelo en un

archivo o módulo EVL; luego, estas restricciones son evaluadas (bajo demanda) en cada uno de los modelos conformes a dicho meta-modelo.

EVL usa una sintaxis similar a OCL. Cada especificación de una validación en EVL se estructura en invariantes (*Invariants*), donde cada invariante se aplica sólo a los objetos cuyo tipo haya sido especificado en el contexto de la invariante. De esta manera, se puede ver EVL como un OCL con anotaciones que proporciona facilidades adicionales:

- **Guardas** que restringen el contexto de cada invariante.
- ***Fixes*** que permite la interacción con el usuario. Un *fix* muestra un mensaje con un valor alternativo para solucionar el problema cuando una invariante no se cumple completamente. Los *fix* se implementan usando EOL (*Epsilon Object Language*, [118]), también basado en OCL, que permite navegar y modificar los modelos. De esta manera, el mecanismo de validación que se implementa en los diferentes DSL, incorpora facilidades para actualizar los modelos, solucionando así los errores detectados.
- Existen dos sub-tipos diferentes de invariantes (***Constraint*** y ***Critique***) que permiten la separación entre los errores, que hacen inválidos a los modelos, y las advertencias, que decrementan la calidad de los modelos.

En la Figura 3-71 se muestra un ejemplo simple de cómo trabaja EVL: una invariante de tipo *Critique* que controla que los elementos de tipo *module* no tengan la propiedad *name* vacía.

```
context ModelTransf{
   -- Model name cannot be empty
   constraint notEmptyModelName{
   check : self.name.isDefined()
      message : getMessageNotEmptyName(self.type().name.asString())
      fix {
      title : getTitleNotEmptyName(self.type().name.asString())
      do {
            self.name := getInputNotEmptyName(self.type().name.asString());}
      }
}
```

Figura 3-71. Ejemplo EVL

Cabe mencionar que se han implementado verificaciones para cada uno de los DSLs definidos en MeTAGeM.

3.2.8 Integración de los Módulos en MeTAGeM

Una de las características más importantes a la hora de desarrollar herramientas en el ámbito del MDE es la *usabilidad*. El uso de Eclipse, como entorno de desarrollo, ayuda a proveer una interfaz común, fácilmente extensible y adaptable a las necesidades específicas.

En MeTAGeM, una de las principales funcionalidades de la interfaz es proporcionar una manera sencilla y amigable de invocar las diferentes transformaciones de modelos definidas en cada nivel. En esta sección se presenta cómo se han utilizado las facilidades brindadas por Eclipse para desarrollar los controles (lanzadores) que permitan la ejecución de las transformaciones de modelos.

El objetivo es demostrar como, a través del uso de los componentes existentes y la documentación disponible de Eclipse, es factible desarrollar dichos lanzadores. Para ello, esta sección se centrará en presentar los resultados obtenidos en cada caso, sin profundizar en el código implementando para desarrollarlos. En el CD que se adjunta en esta tesis se encuentra el código completo de la herramienta.

3.2.8.1 Desarrollo e Integración de los *Plug-ins*

Básicamente, la integración de la funcionalidad provista por un nuevo módulo de MeTAGeM reside en el desarrollo de un *plug-in* integrado. Cada *plug-in* implementa lanzadores (*launchers*) para cada una de las transformaciones de modelos provistas por el módulo y las facilidades necesarias para que el usuario invoque dichos lanzadores. Cada *plug-in* depende de los diferentes *plug-ins* que implementan los DSLs de MeTAGeM y de los *plug-ins* provistos por EMP [90].

La Figura 3-72 muestra las dependencias entre las transformaciones implementadas en MeTAGeM y los diferentes *plug-ins* que componen o son usados por los módulos.

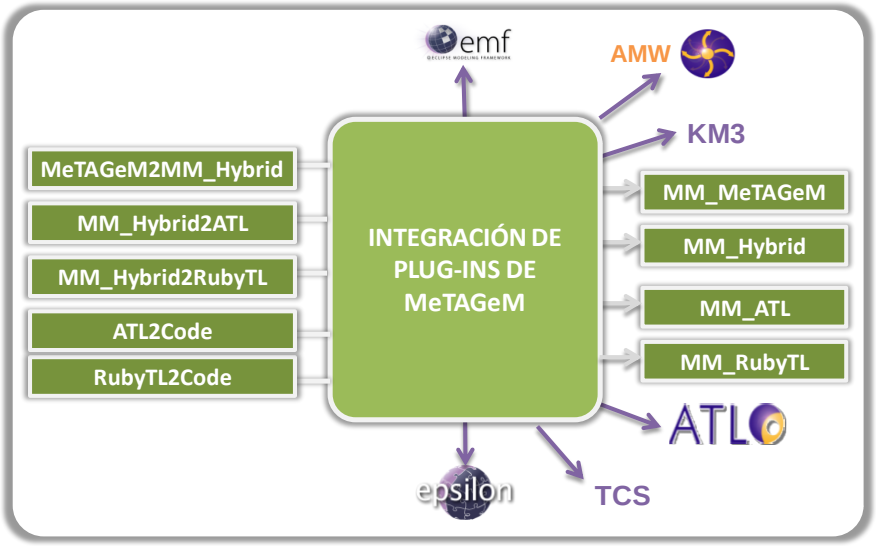

Figura 3-72. Dependencias entre los *Plug-ins* de MeTAGeM y las Transformaciones

El *plug-in* de MeTAGeM tiene implementado tres módulos de transformaciones de modelo a modelo: el módulo correspondiente a las transformaciones entre el nivel PIM y el nivel PSM (MeTAGeM2MM_Hybrid), y los módulos correspondientes a las transformaciones entre el nivel PSM y el nivel PDM, tanto para el lenguaje ATL (MM_Hybrid2ATL) como para el lenguaje RubyTL (MM_Hybrid2RubyTL). A medida que MeTAGeM soporte otras aproximaciones a nivel PSM u otros lenguajes de transformación a nivel PDM se incrementará la cantidad de módulos de transformación de modelo a modelo. En cuanto a la generación de código, como se dijo en la sección 3.2.6, se utiliza el extractor proporcionado por TCS; que en el caso de ATL ya se encuentra incorporado en el *plug-in*, y que en el caso de RubyTL ha sido necesario implementar.

Estos *plug-ins* dependen tanto, de los diferentes componentes provistos por EMP como EMF, ATL, KM3, AMW, Epsilon y TCS, como de los *plug-ins* que implementan los DSLs en MeTAGeM.

A continuación, se describe cómo se han integrado los diferentes DSLs en el *plug-in* de MeTAGeM: por un lado, se presenta la parte que soporta la configuración del lanzamiento de las transformaciones de modelos y, por otro lado, la parte que proporciona la interfaz de usuario para realizar el lanzamiento de las transformaciones.

3.2.8.2 Configuración del Lanzamiento de las Transformaciones.

La Figura 3-73 muestra la cabecera del módulo de la transformación ATL MeTAGeM2Hybrid. Como se puede observar, para realizar el lanzamiento de la transformación es necesario indicar los modelos origen y destino, conformes a sus correspondientes meta-modelos origen y destino.

```
--@atlcompiler atl2006
module MeTAGeM2Hybrid;
create OUT : MM_Hybrid from IN : AMW, left : MOF, right : MOF;
```

Figura 3-73. Cabecera del Módulo MeTAGeM 2Hybrid

De acuerdo con la cabecera del módulo, a partir de un modelo conforme al meta-modelo de AMW y dos modelos, *Left* y *Right*, conformes al meta-modelo MOF, se genera un modelo destino conforme al meta-modelo M_LTH. En tiempo de ejecución, las correspondencias entre las variables que representan los meta-modelos y modelos y los elementos reales se realizan utilizando los asistentes proporcionados por el IDE de ATL. De esta manera, el usuario puede crear un lanzador que permita ejecutar la transformación, recuperarla y utilizarla en cualquier momento. Lo que se pretende es eliminar la necesidad de tener que hacer este tipo de configuraciones de ejecución en cada momento o, por lo menos, simplificarla, con el fin de facilitar la tarea y proporcionar una interfaz de uso más amigable de MeTAGeM.

En primer lugar, se debe programar el lanzador de la transformación, esto es, configurar la ejecución de la transformación. Una vez que se crea la configuración del lanzamiento de la transformación, la misma, puede ser invocada desde el código que controla la interfaz del usuario.

El principal elemento a la hora de programar el lanzador de las transformaciones en MeTAGeM es la clase *Transformations*. Una de las principales responsabilidades de la clase *Transformations* es la de obtener los diferentes módulos de las transformaciones que soporta MeTAGeM. Como ésta es una tarea costosa en términos de memoria y tiempo de procesamiento, la clase *Transformations* sigue el patrón *singleton* para evitar duplicar la carga de los meta-modelos que intervienen en la transformación. De esta manera, el proceso de carga de las transformaciones se realiza la primera vez que se utiliza el *plug-in* y las transformaciones permanecen disponibles hasta que se cierre Eclipse. La Figura 3-74 muestra el contructor *Trasnformations* de MeTAGeM.

```
private Transformations() {

      modelHandler = (AtlEMFModelHandler) AtlModelHandler.getDefault(AtlModelHandler.AMH_EMF);

      //logger.setUseParentHandlers(false);
      //logger.setLevel(Level.OFF);

      METAGEM2HYBRID_TransfoResource = Transformations.class.getResource
                                       ("resources/MeTAGeM2Hybrid.asm");
      HYBRID2ATL_TransfoResource = Transformations.class.getResource
                                       ("resources/Hybrid2ATL.asm");
}
```

Figura 3-74. Extracto del Constructor *Transformations* de MeTAGeM: Obtención de Meta-modelos

Además, la clase *Transformations* contiene un método para lanzar cada una de las transformaciones de modelo a modelos incluidas en el módulo.

A modo de ejemplo, se muestra el código para lanzar la transformación MeTAGeM2Hybrid. Todo el código que implementa MeTAGeM, y por lo tanto la integración del *plug-in*, se encuentra en el CD adjunto.

Para poder ejecutar la transformación ATL MeTAGeM2Hybrid, la clase *Transformations* implementa el método *metagem2hybrid* (Figura 3-75) que recibe cuatro parámetros diferentes que se corresponden con los modelos origen (*inFilePath*, *leftFilePath* y *rightFilePath*) y destino (*outFilePath*) que necesita la transformación.

```
public void metagem2hybrid(String inFilePath, String leftFilePath,
            String rightFilePath,String outFilePath) throws Exception{

            Map<String, Object> models = new HashMap<String, Object>();
            initMetagem2HybridMetamodels(models);

...}
```

Figura 3-75. Método del Lanzador *metagem2hybrid*

Para inicializar las variables de los meta-modelos que se usarán durante la transformación, se define una tabla *hash* que recoge la información proporcionada por el encabezado de la transformación ATL. La información de dicha tabla se completa a través de la invocación del método *initMetagem2HybridMetamodels* (Figura 3-76).

```
private void initMetagem2HybridMetamodels(Map<String, Object> models) {

      metagemMetamodel = (ASMEMFModel) modelHandler.loadModel(
            "METAGEM", modelHandler.getMof(),
            this.getClass().getResourceAsStream("resources/mw_metagem.ecore"));
      hybridMetamodel = (ASMEMFModel) modelHandler.loadModel(
            "MM_Hybrid", modelHandler.getMof(),
            this.getClass().getResourceAsStream("resources/MM_Hybrid.ecore"));
      MOFMetamodel = (ASMEMFModel) modelHandler.loadModel(
            "MOF", modelHandler.getMof(),
            this.getClass().getResourceAsStream("resources/CMOF.ecore"));

      models.put("AMW", metagemMetamodel);
      models.put("MM_Hybrid", hybridMetamodel);
      models.put("MOF", MOFMetamodel);
}
```

Figura 3-76. Método *initMetagem2HybridMetamodels*

Para poder completar la información en la tabla, en el método *initMetagem2HybridMetamodels*, se usa la API proporcionada por ATL para el manejo de modelos. En particular, se usa el método *loadModel* que permite recuperar cada uno de los meta-modelos proporcionados en la variable *path*. De esta manera, las claves de la tabla *hash* se corresponden con los nombres de las variables usadas en la cabecera de ATL (“AMW”, “MM_Hybrid”, “MOF”), mientras que los valores se corresponden con los respectivos meta-modelos.

El siguiente paso es recuperar los modelos que se manejan en la transformación, que también se guardan en una tabla *hash* llamada *models*. En la Figura 3-77 se muestra el código correspondiente.

```
...
// get/create models
ASMEMFModel metagemInputModel = (ASMEMFModel) modelHandler.loadModel
      ("IN", metagemMetamodel, URI.createFileURI(inFilePath));
      models.put("IN", metagemInputModel);

ASMEMFModel leftInputModel = (ASMEMFModel) modelHandler.loadModel
      ("left", MOFMetamodel, URI.createFileURI(leftFilePath));
      models.put("left", leftInputModel);

ASMEMFModel rightInputModel = (ASMEMFModel) modelHandler.loadModel
      ("right", MOFMetamodel, URI.createFileURI(rightFilePath));
      models.put("right", rightInputModel);

ASMEMFModel hybridOutputModel = (ASMEMFModel) modelHandler.newModel
      ("OUT", URI.createFileURI(outFilePath).toFileString(), hybridMetamodel);
      models.put("OUT", hybridOutputModel);
...
```

Figura 3-77. Carga de Modelos para la Ejecución de la Transformación ATL

Por ultimo, una vez que todos los modelos y meta-modelos han sido recuperados, se ejecuta la transformación usando los métodos provistos por la API de ATL (Figura 3-78).

```
...
// launch
AtlLauncher.getDefault().launch(this.METAGEM2HYBRID_TransfoResource,
      Collections.EMPTY_MAP, models,
      Collections.EMPTY_MAP, Collections.EMPTY_LIST,Collections.EMPTY_MAP);

modelHandler.saveModel(hybridOutputModel, outFilePath, false);
dispose(models);
...
```

Figura 3-78. Programación del Lanzador de la Transformación ATL

Como resultado se genera el modelo destino, que se almacena en la variable *outFilePath*.

3.2.8.3 Soporte Gráfico para el Lanzamiento de las Transformaciones

Una vez que la programación del lanzamiento de las transformaciones desde el código funciona correctamente, se considera conveniente desarrollar una interfaz a través de la cual se pueda invocar la ejecución de las transformaciones. Para esto, Eclipse provee algunos tipos de lanzadores genéricos que pueden ser extendidos de acuerdo a necesidades específicas. Estos lanzadores genéricos se basan en dos conceptos principales: **configuración de lanzamientos** (*LaunchConfigurations* más conocida como *Run Configurations*) y **configuración de tipos de lanzamientos** (*LaunchConfigurationTypes*).

El nivel más simple es ***LaunchConfigurationType*** que es como un molde que definen como deberían ser los lanzadores (conocidos como *cookie cutres*). Los ***LaunchConfigurations*** (conocidos como *cookies*) son lanzadores realizados con dichos moldes. Cuando el desarrollador de *plug-in* decide crear un lanzador, lo que está haciendo es crear un tipo específico de *cookie cutters* que permitirá a los usuarios utilizar tantas *cookies* como necesiten. En términos más técnicos, un *LaunchConfigurationType* (en adelante, tipo de configuración) es una entidad que sabe cómo poner en marcha determinados tipos de configuraciones de lanzamiento y que determina los parámetros necesarios en cada caso. Los *LaunchConfigurations* (en adelante, configuración) son entidades que contienen toda la información necesaria para realizar la ejecución de un lanzador específico.

A modo de ejemplo, en la Figura 3-79 se muestra como el menú contextual sobre cualquier archivo de Eclipse brinda acceso a los diferentes tipos de configuración disponibles en ese tipo de archivo.

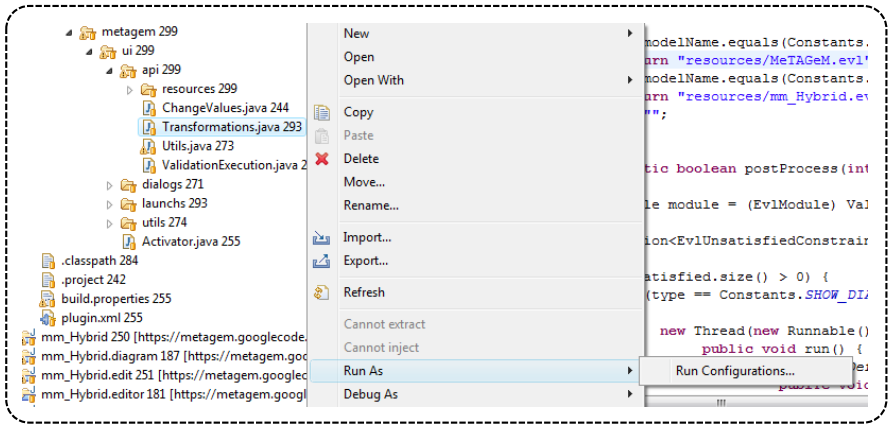

Figura 3-79. Extracto de Eclipse: *Run Configurations*

Con el objetivo de brindar el soporte gráfico al lanzamiento de las transformaciones de modelos en MeTAGeM se utiliza las APIs definidas en Eclipse y los mecanismos para construir dicho soporte gráfico. Para ello se incluyen tres nuevos *Launch ConfigurationTypes*, uno para cada conjunto de transformaciones de modelo a modelo soportada en MeTAGeM

A continuación, se muestra las configuraciones realizadas para MeTAGeM. Para mayor información, en el CD adjunto se encuentra todo el código de MeTAGeM.

3.2.8.4 Tipos de Configuraciones para las Transformaciones de Modelo a Modelo de MeTAGeM

En MeTAGeM se incluyen tres tipos de configuraciones diferentes (Figura 3-80), una por cada tipo de transformación de modelo a modelo soportada.

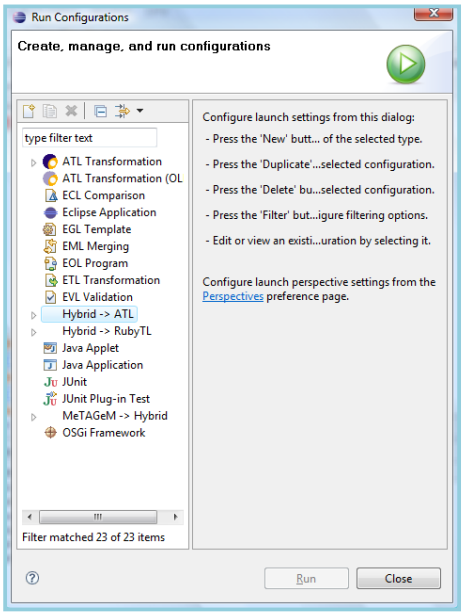

Figura 3-80. Tipos de Configuraciones en MeTAGeM

Una vez que se crea un nuevo tipo de configuración se puede utilizar en cualquier vista de Eclipse. Por ejemplo, en la Figura 3-81 se muestra cómo el menú contextual para los modelos del nivel PIM de MeTAGeM, modelos con extensión *.amw*, incluyen la posibilidad de ejecutar el nuevo tipo de extensión creada (MeTAGeM→Hybrid)

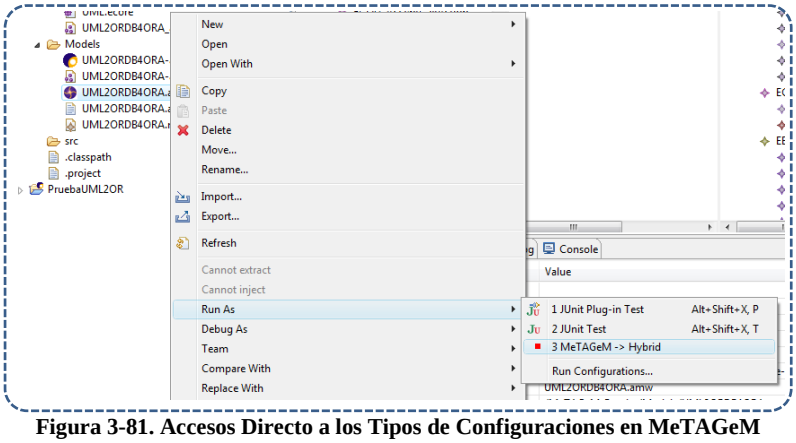

Figura 3-81. Accesos Directo a los Tipos de Configuraciones en MeTAGeM

El usuario puede acceder a la edición de la nueva configuración simplemente seleccionando la opción de *Run Configurations* (Figura 3-82).

De esta manera, se accede al *wizard* de edición y se pueden configurar los parámetros necesarios para ejecutar la transformación (modelo origen, modelo destino, etc.).

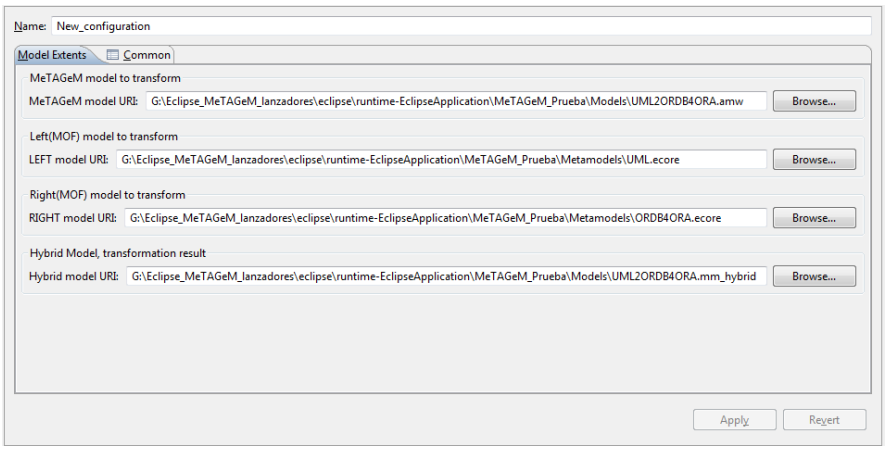

Figura 3-82. *Wizard* para la Configuración del Lanzador de MeTAGeM2Hybrid

Cada una de las configuraciones creadas se almacena con el nombre provisto en el campo *Name*. De esta manera, el usuario puede invocar la misma configuración de transformación cada vez que lo necesite, sin necesidad de volver a configurarla. Si el usuario define más de un tipo de configuración para ejecutar la transformación MeTAGeM→Hybrid, cada vez que desee ejecutar la transformación deberá seleccionar que configuración desea (Figura 3-83).

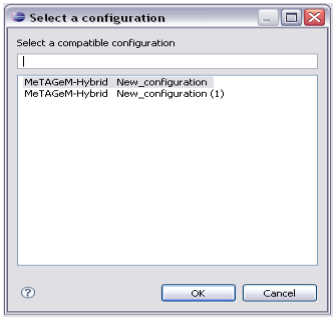

Figura 3-83. Selección del Tipo de Configuración para MeTAGeM2Hybrid

Validación

En el capítulo 3 se ha presentado la definición del entorno de desarrollo MeTAGeM a través de la especificación de la metodología y de la herramienta que lo compone. MeTAGeM permite aplicar los principios del MDE al desarrollo de las transformaciones de modelos, es decir, definir transformaciones de modelos como modelos en sí mismos, sin tener en cuenta los detalles de la implementación final, para posteriormente, por medio de transformaciones (semi-)automáticas obtener el código de cada transformación.

En este capítulo se presenta, en primer lugar, el desarrollo de un meta-caso de estudio que valida el funcionamiento de la herramienta. Dicho meta-caso de estudio consiste en la generación automática de las transformaciones de modelos, propuestas en la herramienta M2DAT-DB, usando MeTAGeM. Posteriormente, se mostrará cómo las transformaciones generadas se utilizan para un caso de estudio concreto, es decir con un modelo específico.

4.1 Meta-Caso de Estudio

Para validar el funcionamiento de la herramienta MeTAGeM se ha desarrollado como meta-caso de estudio la implementación de un subconjunto de las reglas de transformación de uno de los módulos propuestos en M2DAT-DB (sección 4.1.1), en particular, el módulo que permite el modelado Bases de Datos y, más específicamente, las reglas que permiten el modelado de Bases de Datos Objeto-Relacionales, para el producto *Oracle10g*, en M2DAT [191].

Esta selección se debe al hecho de que las transformaciones de M2DAT-DB, ya se encuentran implementadas y se ha validado su funcionamiento. Al realizar el desarrollo de las mismas con MeTAGeM es posible comparar los resultados obtenidos. Esta comparación se realizará desde dos puntos de vista: en primer lugar, desde el punto de vista sintáctico, es decir, comparando los dos archivos de transformaciones para verificar las similitudes y las diferencias; y en segundo lugar desde el punto de vista del funcionamiento, esto es, comprobando que la ejecución de la transformación con un modelo origen en particular genera el mismo (o similar) modelo destino en ambos casos.

4.1.1 M2DAT-DB

M2DAT-DB [191] es una herramienta para el desarrollo dirigido por modelos de esquemas de Bases de Datos modernas. Soporta el desarrollo de una base de datos, desde la definición de los modelos a nivel PIM hasta la obtención del código de implementación final. En particular M2DAT-DB soporta la generación de: esquemas de Bases de Datos Objeto-Relacionales para el producto Oracle [164, 165, 194] y para el estándar SQL:2003 [98]; y esquemas XML [8, 200, 201, 209, 210, 210], a partir de un modelo conceptual de datos definido a nivel PIM y representado por medio de un diagrama de clases de UML [99, 157].

En la Figura 4-1 se muestra, de manera simplificada, la arquitectura de M2DAT-DB. En adelante, nos centraremos en las funcionalidades brindadas por M2DAT-DB y no en la implementación completa de la herramienta. Para mayor información sobre la misma se puede consultar [191].

Como se puede ver en la Figura 4-1 el modelado del esquema de la BD a nivel PIM se realiza por medio de un modelo conceptual de datos representado por un diagrama de clases de UML.

Para el modelado a nivel PSM, M2DAT-DB propone dos modelos lógicos: el modelo OR y el modelo XML. Para ello define tres DSLs diferentes: uno para el modelado OR siguiendo el estándar SQL:2003 [98], otro para el modelado OR para el producto específico Oracle 10g [165] y el último, para el modelado de esquemas XML [209, 210] siguiendo el estándar.

Para transformar los modelos definidos a nivel PIM a modelos definidos a nivel PSM se definen tres conjuntos de transformaciones de modelo a modelo diferentes, uno por cada DSL propuesto a nivel PSM:

- Del diagrama de clases UML al modelo OR para Oracle (**UML2ORA**).
- Del diagrama de clases UML al modelo OR para el estándar SQL:2003 (**UML2SQL2003**).
- Del diagrama de clases UML al modelo de esquema XML (**UML2XML**)

Por último, se definen el conjunto de transformaciones de modelo a texto que permite obtener para cada caso el código implementable de la base de datos.

- Del modelo OR para Oracle se obtiene el código SQL para Oracle.
- Del modelo OR para el estándar SQL:2003 se obtiene el código SQL para el estándar SQL:2003.
- Del modelo de esquema XML se obtiene el código que implementa el esquena XML.

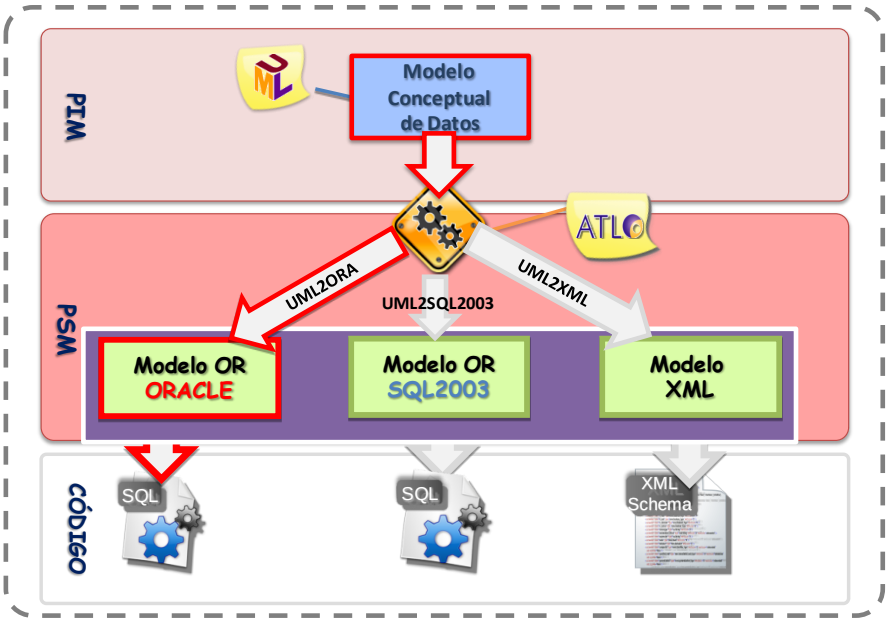

Figura 4-1. Proceso de Desarrollo Dirigido por Modelo de Esquemas de BD en M2DAT-DB

Para validar el funcionamiento de MeTAGeM se han implementado las transformaciones UML2ORA de la Figura 4-1, es decir, entre los modelos de nivel PIM, representados mediante un diagrama de clases de UML [156], y los modelos de nivel PSM, representados mediante un modelo OR para el producto específico de Oracle10g.

Como se puede observar en la Figura 4-2, la implementación de las reglas UML2ORA consiste en: definir un modelo de transformación a nivel PIM, tomando como meta-modelo origen el meta-modelo de UML y como meta-modelo destino el meta-modelo OR. A partir de este modelo de transformación a nivel PIM y aplicando una serie de transformaciones se obtiene el modelo de la transformación a nivel PSM que sigue la aproximación híbrida. De la misma manera, a partir del modelo de transformación de nivel PSM y aplicando nuevamente una serie de transformaciones se obtiene el modelo de transformación a nivel PDM conforme a un lenguaje de transformación en particular, en este caso se selecciona el lenguaje ATL. Por último, por medio de los mecanismos de extracción brindados por ATL se obtiene el código que implementa la transformación en el lenguaje seleccionado.

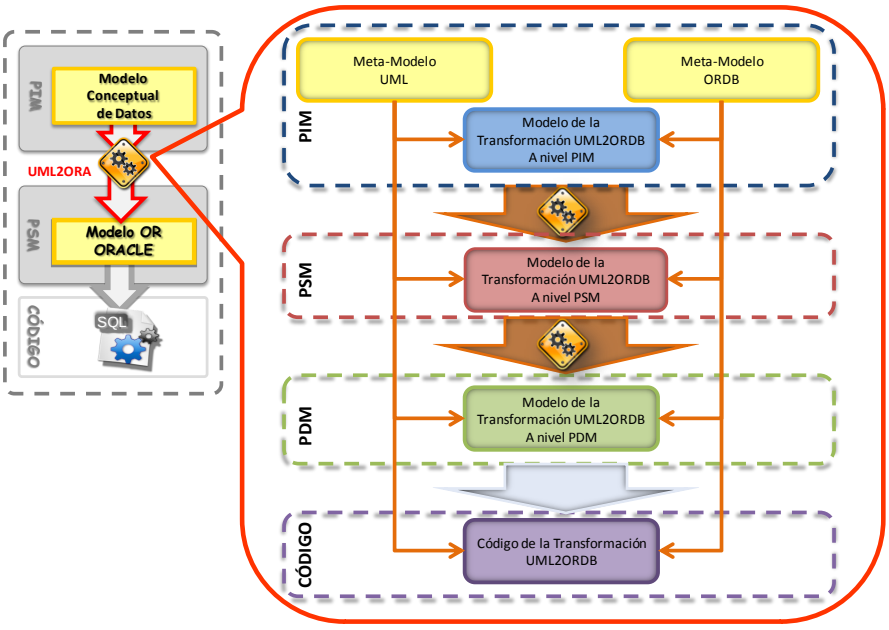

Figura 4-2. Implementación de las Reglas UML2ORDB en MeTAGeM

Dado que el meta-modelo de UML es de sobra conocido [155], a continuación se presentan, en primer lugar, el meta-modelo ORDB para el producto Oracle10g y posteriormente, el conjunto de reglas de transformación definidas entre los elementos del meta-modelo de UML y el meta-modelo ORDB para dicho producto. Por último, se mostrará cómo se obtiene el código implementable de dichas reglas en el lenguaje ATL, utilizando la meta-herramienta MeTAGeM.

4.1.2 Meta-modelo ORDB para Oracle 10g

La Figura 4-3 muestra el meta-modelo ORBD para Oracle10g.

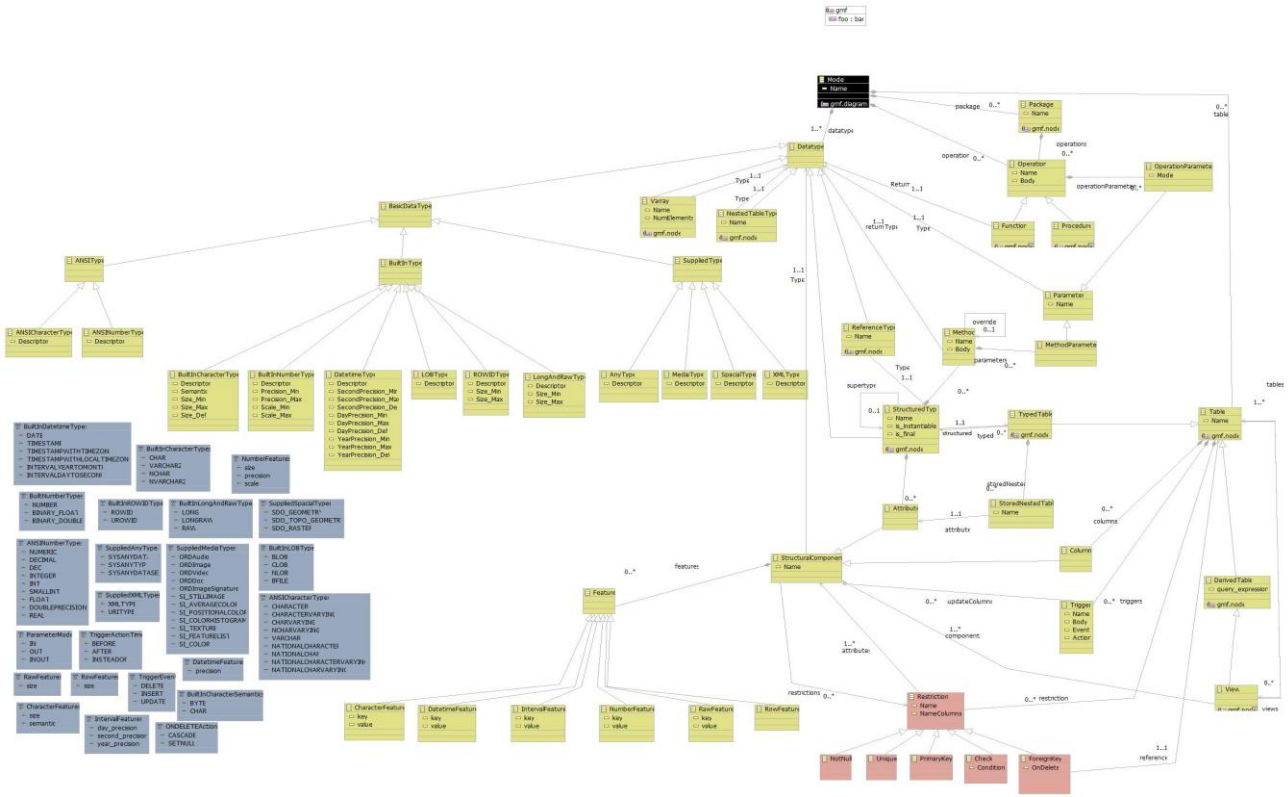

Figura 4-3. Meta-modelo ORDB4ORA

Como se ha indicado anteriormente, debido a la naturaleza XML subyacente de Ecore, cualquier meta-modelo Ecore debe incluir un elemento raíz. En el meta-modelo ORDB el elemento raíz es la meta-clase ***Model***.

Cada meta-clase ***Model*** está compuesta de: ***DataTypes*** que pueden ser *built-in* (predefinidos) o tipos definidos por el usuario; ***Operations*** que pueden ser ***Functions*** o ***Procedures***; ***Tables*** y ***Packages*** (Figura 4-4). Para ver en mayor detalle el meta-modelo ORDB consultar [165].

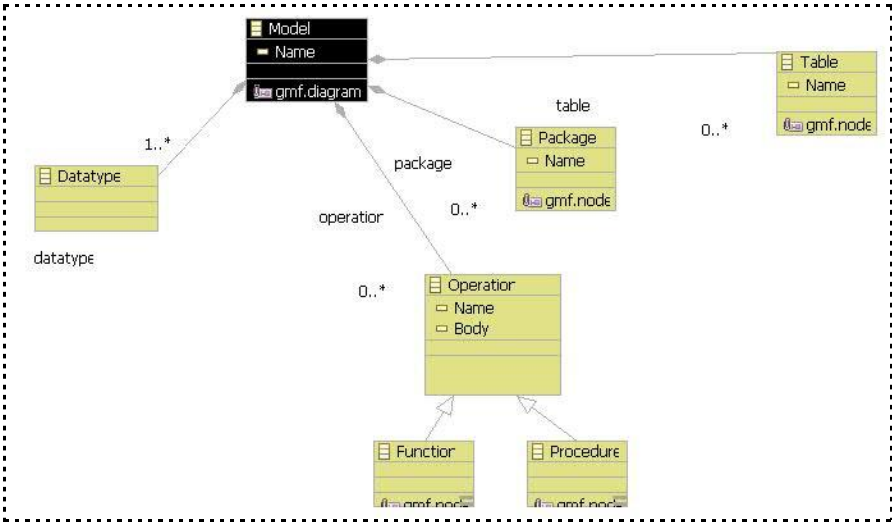

Figura 4-4. Vista Parcial del Meta-modelo ORDB4ORA para Oracle 10g. Meta-clase *Model*

4.1.3 Definición de Reglas de Transformación

En [195] se ha definido un proceso de desarrollo para BDOR que incluye un conjunto de reglas de transformación que permiten pasar del modelo conceptual de datos (PIM) al modelo OR (PSM). Dicho proceso ha sido implementado en M2DAT [191] para soportar las transformaciones de nivel PIM a PSM. En la Tabla 4-1 se muestra una tabla resumen de las reglas de transformación definidas en [195].

Las reglas de transformación establecen que un elemento de tipo *Class* a nivel PIM se transforma en un elemento de tipo *Object Type* o *Structured Type* y un elemento de tipo *Object Table* o *Typed Table*, de manera que el elemento de tipo *Typed Table* es la extensión del elemento de tipo *Structured Type*.

La transformación de los elementos de tipo *Attribute* depende de la naturaleza del elemento a transformar, por lo que se han definido distintas reglas para cada tipo de *Attribute*. Así por ejemplo, un *Multivalued Attribute* (atributo

multivaluado) se transforma utilizando un tipo colección, Oracle10g tiene dos tipos de colección el *Varray* y el *Nested Table*, por lo que un *Multivalued Attribute* se puede transformar utilizando cualquiera de los dos tipos. De la misma manera, un *Calculated Attribute* se puede transformar a un elemento de tipo *Trigger* o a un elemento de tipo *Method*.

Tabla 4-1. Reglas de Transformación de PIM a PSM de M2DAT-DB

PIM de Datos		**PSM de Datos para el Producto Oracle10g**
Package		*Package*
Class		*Object Type + Object Table*
Attrubute	*Simple*	*Attribute (column)*
	Multivalued	*Varray/Nested Table*
	Composed	*Object Type (column)*
	Calculated	*Trigger/Method*
Association	*1:1*	*Ref/Ref*
	1:M	*Ref - Nested Table/Varray*
	N: M	*Nested Table/Nested Table - Varray/Varray*
	Aggregation	*Nested Table/Varray of References*
	Composition	*Nested Table/Varray of Objects*
	Generalization	*Types/Typed Tables*

En cuanto a la transformación de los elementos de tipo *Association* es importante mencionar que se realiza de forma bidireccional y se definen diferentes reglas que permiten transformar los elementos de tipo *Association* dependiendo de la cardinalidad de las mismas. Así, por ejemplo, para transformar los elementos de tipo *Association* con cardinalidad 1:1 se crea un atributo de tipo *Ref* en cada uno de los *Structured Type* que participan en la relación, dicho atributo es una referencia al *Structured Type* del otro extremo de la relación. En el caso de los elementos de tipo *Association* con cardinalidad N:M se generan en cada uno de los *Structured Type* que participan en la relación, un elemento de tipo *Varray* o *Nested Table* que contienen elementos de tipo *Ref* al *Structured Type* del extremo opuesto de la relación.

4.1.4 Implementación utilizando MeTAGeM

A modo de resumen, en la Figura 4-5 se muestra el proceso completo de implementación de las reglas de transformación presentadas previamente utilizando MeTAGeM. Dicho proceso está representado mediante un diagrama de actividad de SPEM.

Según el proceso definido en la Figura 3-1, en primer lugar, se define el modelo de la transformación en el nivel PIM de MeTAGeM, con extensión *.amw*, donde se identifican los meta-modelos que participan en la transformación de M2DAT-DB, meta-modelo de UML y ORBD4ORA, y las relaciones existentes entre los elementos de cada meta-modelo.

A partir del modelo de las transformaciones de nivel PIM, el segundo paso es aplicar el conjunto de reglas de transformación definidas en MeTAGeM para transformar modelos de transformación definidos a nivel PIM a modelos de transformación definidos a nivel PSM. Como resultado de esta transformación se obtiene el modelo de la transformación con extensión *mm_hybrid*. Este modelo puede ser manipulado por el desarrollador para agregar nueva funcionalidad o modificar alguna existente.

Una vez refinado el modelo a nivel PSM, el tercer paso es la generación del modelo de la transformación a nivel PDM. Para esto, el desarrollador debe seleccionar el lenguaje en el que desea implementar las transformaciones, en el caso que se muestra en la figura se selecciona el lenguaje ATL, y ejecutar el conjunto de reglas de transformación correspondientes. El modelo de la transformación obtenido a nivel PDM, conforme al meta-modelo del lenguaje ATL, puede ser refinado por el desarrollador de la misma manera que el modelo del nivel anterior.

El último paso del proceso de implementación de las transformaciones utilizando la herramienta MeTAGeM es la generación de código. Por medio del mecanismo de extracción disponible en el lenguaje ATL, se obtiene el código de la transformación en el lenguaje ATL.

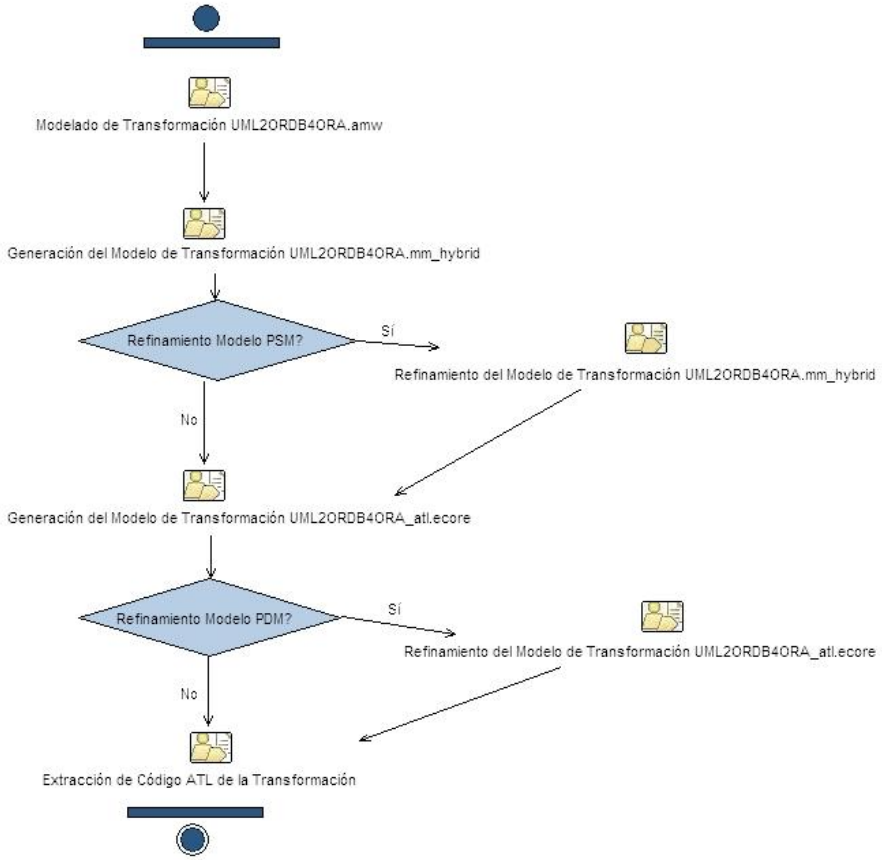

Figura 4-5. Proceso de la Transformación UML2ORDB4ORA en MeTAGeM

A continuación, se presentan los modelos de transformación de cada uno de los diferentes niveles. El meta-caso de estudio completo se encuentra en el Anexo F y en el CD adjunto.

4.1.4.1 Definición del modelo de nivel PIM

Para definir el modelo de nivel PIM de MeTAGeM se debe crear un nuevo modelo de *weaving* siguiendo los pasos mostrados en la Figura 3-39. Una vez creado el modelo se debe determinar el tipo de relación, de acuerdo a lo establecido en la sección 3.1.2.1, entre cada uno de los elementos de los meta-modelos, partiendo de las reglas de transformación mostradas en la Tabla 4-1. Así por ejemplo, las reglas de transformación establecen que un elemento de tipo *Class* del meta-modelo de UML se debe transformar en un elemento de tipo

ObjectType y un elemento de tipo *ObjectTable* del meta-modelo ORDB4ORA. Esta regla de transformación se debe expresar con una relación de 1 a N (*OneToMany*), es decir, a partir de un elemento del meta-modelo origen se generar uno o más (dos en este caso) elementos en el modelo destino. Todas las relaciones identificadas se deben especificar en el modelo de transformación de nivel PIM.

En la Figura 4-6 se muestra el modelo de transformación de nivel PIM creado. Como se puede observar, en la parte izquierda de la figura se muestran los elementos del meta-modelo origen (UML), en la parte derecha se muestran los elementos del meta-modelo destino (ORDB4ORA), y en el centro de la figura, el modelo de la transformación donde se han especificado todas las relaciones.

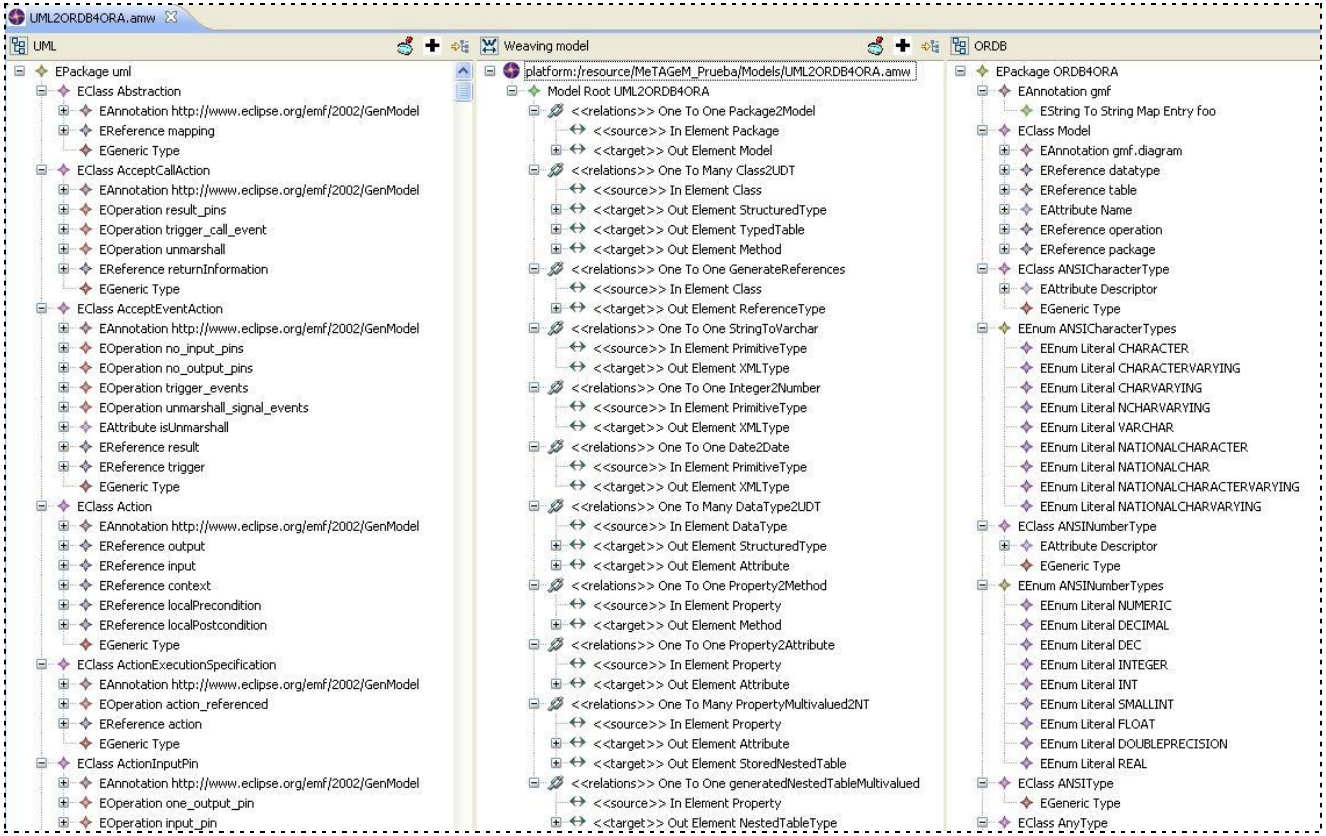

Figura 4-6. Modelo de Transformación a Nivel PIM – UML2ORDB4ORA.amw

4.1.4.2 Definición del modelo a nivel PSM

A partir de la definición del modelo de transformación independiente de plataforma entre el meta-modelo UML y ORDB4ORA, mostrado en la sección anterior, se puede generar de forma (semi-)automática, usando las facilidades de MeTAGeM, el modelo de la transformación específico de plataforma siguiendo la aproximación híbrida (Figura 4-7).

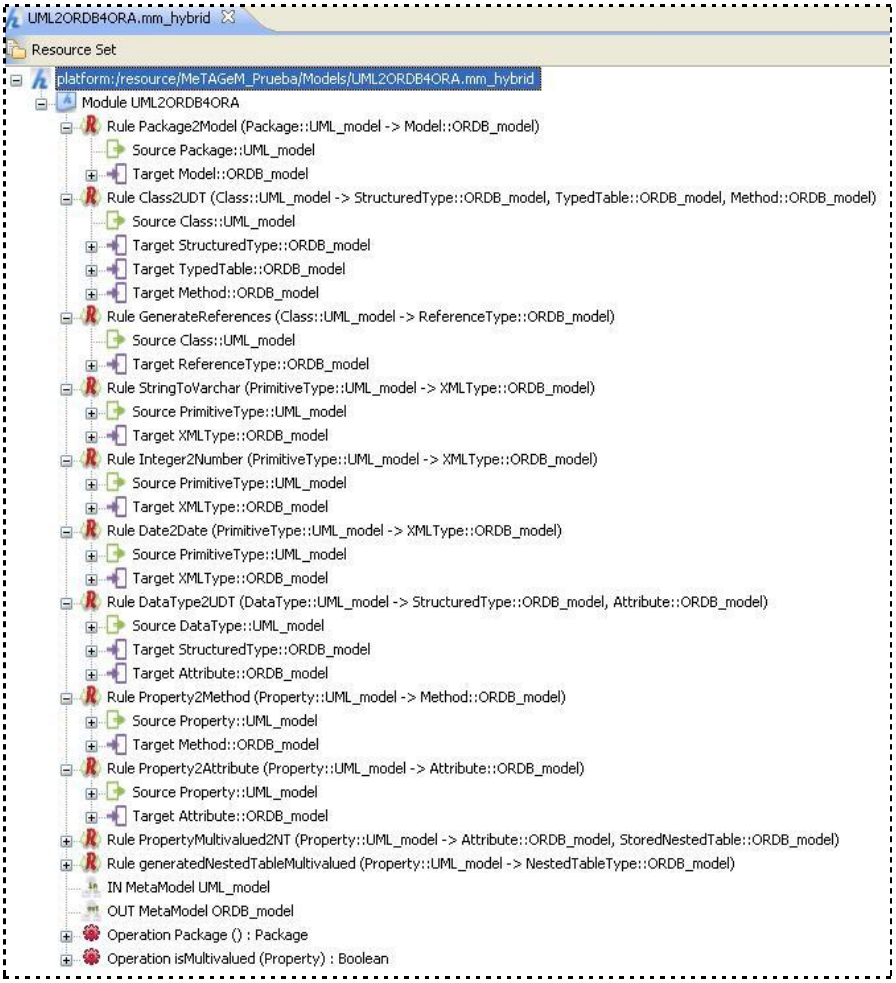

Figura 4-7. Modelo a Nivel PSM – UML2ORDB4ORA.mm_hybrid

Como se puede observar en la Figura 4-7 por cada elemento del tipo *relations* especificado en el modelo de nivel PIM se genera un elemento del tipo

Rule en el modelo destino, de acuerdo a las reglas de transformación definidas en la sección 3.1.3. De la misma manera se obtienen el resto de los elementos.

El modelo obtenido a nivel PSM puede ser refinado por el desarrollador de la transformación, ya sea modificando alguna regla generada o agregando nueva funcionalidad.

4.1.4.3 Definición del modelo a nivel PDM

A nivel PDM se puede realizar el modelado de las transformaciones de acuerdo al lenguaje ATL o al lenguaje RubyTL.

Una vez que el desarrollador haya finalizado de refinar el modelo PSM, se genera, de forma (semi-)automática, el modelo a nivel PDM seleccionando previamente el lenguaje de modelado de las transformaciones.

A continuación, se muestra los modelos de transformación obtenidos en ambos lenguajes.

4.1.4.3.1 Modelo conforme al meta-modelo de ATL

En la Figura 4-8 se muestra el modelo de transformación conforme al meta-modelo de ATL, *UML2ORDB4ORA-atl.ecore*. Como se ha mencionado anteriormente, para la manipulación de los modelos conformes al meta-modelo de ATL se utiliza el editor reflexivo de EMF.

Como se puede observar en la Figura 4-8 de acuerdo a las reglas de transformación definidas en la sección 3.1.3 se realiza la transformación de cada uno de los elementos del modelo a nivel PSM. A modo de ejemplo, por cada elemento de tipo *Rule* del modelo UML2ORBD4ORA.mm_hybrid, cuya propiedad *isMain* sea igual a *true*, se genera un elemento de tipo *MatchedRule* en el modelo destino. De la misma manera, se transforman el resto de los elementos.

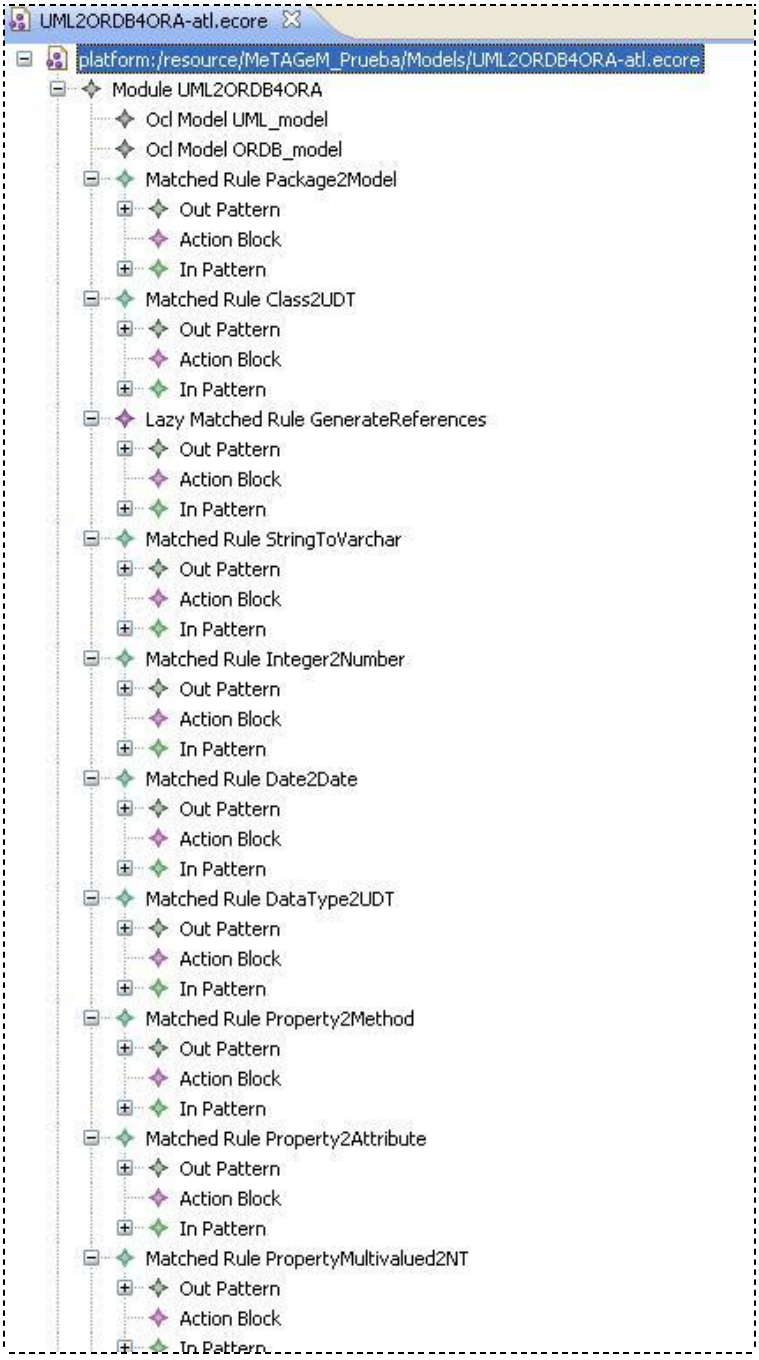

Figura 4-8. Modelo a Nivel PDM – UML2ORDB4ORA-atl.ecore

4.1.4.3.2 Modelo conforme al meta-modelo de RubyTL

En la Figura 4-9 se muestra el modelo de transformación conforme al meta-modelo de RubyTL, *UML2ORDB4ORA.rubytl*. A diferencia del modelo de ATL, para manipular el modelo conforme a RubyTL se utiliza el editor personalizado propuesto en MeTAGeM.

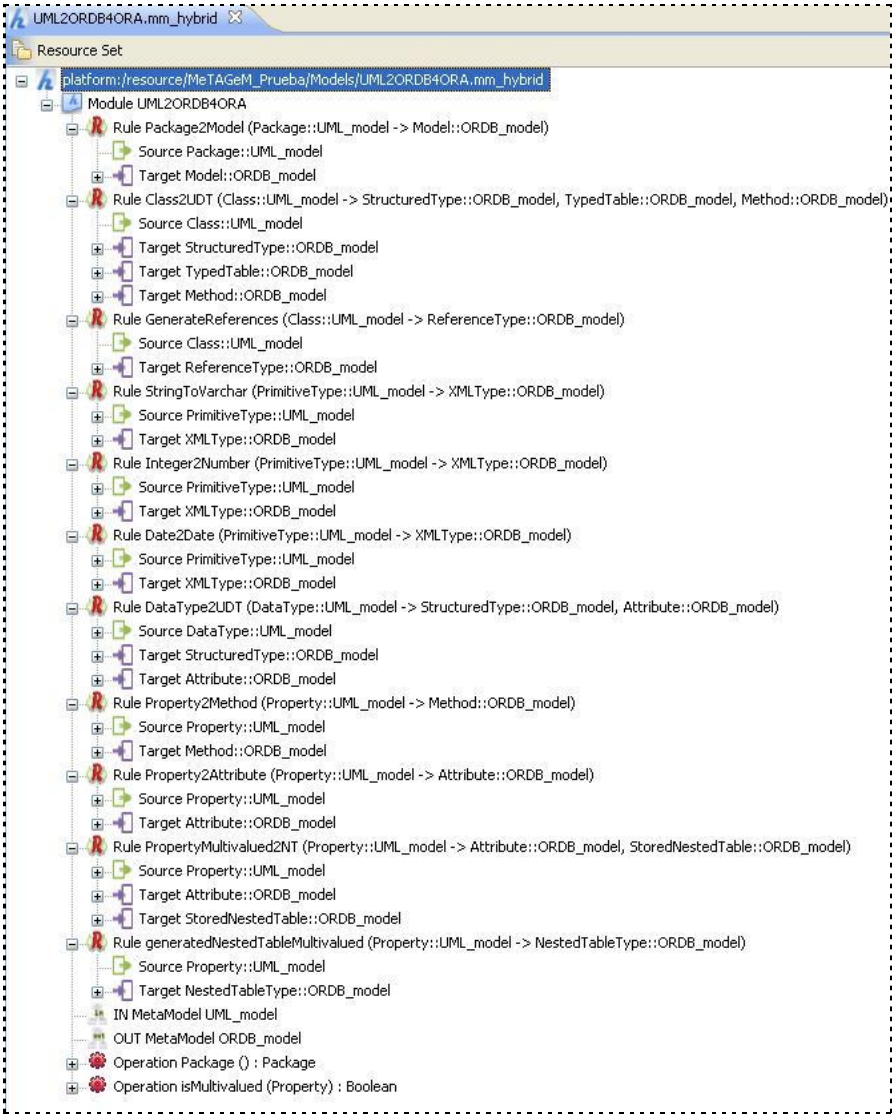

Figura 4-9. Modelo a Nivel PDM - UML2ORDB4ORA.rubyttl

Como se puede observar en la Figura 4-9 de acuerdo a las reglas de transformación definidas en la sección 3.1.3 se realiza la transformación de cada uno de los elementos del modelo a nivel PSM. A modo de ejemplo, por cada elemento de tipo *Rule* del modelo UML2ORBD4ORA.mm_hybrid, cuya propiedad *isMain* sea igual a *true*, se genera un elemento de tipo *TopRule* en el modelo destino. De la misma manera, se transforman el resto de los elementos.

4.1.4.4 Generación de Código

Según el proceso definido en la sección 3.1 el último paso es la obtención del código que implementa la transformación en el lenguaje seleccionado en el nivel PDM, en el caso de esta tesis en los lenguajes ATL y RubyTL.

4.1.4.4.1 Código de la transformación en ATL

Como se ha especificado a lo largo de este trabajo, para realizar la generación de código se utiliza la funcionalidad de extracción de código brindada por el lenguaje ATL.

En la Figura 4-10 se muesta el código de la transformación en el lenguaje ATL obtenido como resultado del proceso de modelado de transformaciones utilizando la herramienta MeTAGeM. Por cada elemento definido en el modelo de la transformación conforme al meta-modelo de ATL se genera el código que lo implementa. Así, por ejemplo, por cada elemento de tipo *MatchedRule* se genera el código que implementa la regla en el archivo *.atl* mostrado a partir de la palabra reservada *Rule*.

Es importante mencionar que el código generado no es completamente implementable. Es decir, existen situaciones donde es necesario que el desarrollador introduzca o modifique alguna sentencia. Tal es el caso de la implementación de los *helpers* (funciones auxiliares), ya que para no complicar el meta-modelo a nivel PSM con instrucciones dependientes de un lenguaje de transformación específico no se pueden recoger toda la información necesaria para poder implementar completamente el *helper*.

UML2ORDB4ORA-atl.atl

```
-- @atlcompiler atl2006
module UML2ORDB4ORA;
create ORDB_model : ORDB from UML_model : UML;

-- Comments -> This is a MatchedRule: Package2Model ->
rule Package2Model {
  from
    package_in : UML!Package
  to
    model_out : ORDB!Model (
      Name <- package_in.name
    )
  -- ActionBlock:
  do {

  }
}

- Comments -> This is a MatchedRule: Class2UDT ->
rule Class2UDT {
    from
      class_in : UML!Class
    to
      structuredtype_out : ORDB!StructuredType (
        Name <- class_in.name + '<<udt>>',
        model <- class_in.Package,
        typed <- typedtable_out,
        method <- method_out),

      typedtable_out : ORDB!TypedTable (
        Name <- class_in.name + 's<<persistent>>'),

      method_out : distinct ORDB!Method  foreach (op in class_in.ownedOperation) (
        Name <- op.name)

  -- ActionBlock:
    do {}
}

-- Comments -> This is a LazyRule: GenerateReferences ->
unique lazy rule GenerateReferences {
  from
    class_in : UML!Class
  to
    referencetype_out : ORDB!ReferenceType (
      Name <- class_in.name
    )
```

Figura 4-10. Código de la Transformación en ATL

4.1.4.4.2 Código de la transformación en RubyTL

De acuerdo a lo definido en la sección 3.2.6.1.2 para realizar la generación de código desde modelos de transformación conformes al meta-modelo de RubyTL se utiliza el mecanismo de extracción brindado por TCS. Para esto, como se ha visto en la sección antes mencionada, ha sido necesario realizar la definición de la sintaxis concreta del DSL de RubyTL utilizando TCS.

En la Figura 4-11 se muestra el código de la transformación en el lenguaje RubyTL obtenido como resultado del proceso de modelado de transformaciones utilizando la herramienta MeTAGeM. Por cada elemento definido en el modelo de la transformación conforme al meta-modelo de RubyTL se genera el código que lo implementa. Así, por ejemplo, por cada elemento de tipo *TopRule* se genera el código que implementa la regla en el archivo *.rb* mostrado a partir de la palabra reservada *top_rule*.

UML2ORDB4ORA.rb

```
transformation 'UML2ORDB4ORA'
input 'UML_model'
output 'ORDB_model'

decorator UML_model::Property do
     def  isMultivalued
          "Determina si una propiedad es multivaluada o no - ReturnType: Boolean"
     end
end

top_rule 'Package2Model' do
     from UML_model::Package
     to ORDB_model::Model
     mapping do | package_in, model_out |
          model_out.Name  =  package_in.name
     end
end

top_rule 'Class2UDT' do
     from UML_model::Class
     to ORDB_model::StructuredType, ORDB_model::TypedTable, ORDB_model::Method
     mapping do | class_in, structuredtype_out, typedtable_out, method_out |
          structuredtype_out.Name  =  class_in.name
          structuredtype_out.model  =  class_in.Package
          structuredtype_out.typed  =  typedtable_out
          structuredtype_out.method  =  method_out
          typedtable_out.Name  =  class_in.name
          method_out.Name  =  class_in.name
     end
end

rule 'GenerateReferences' do
     from UML_model::Class
     to ORDB_model::ReferenceType
     mapping do | class_in, referencetype_out |
          referencetype_out.Name  =  class_in.name
     end
end

top_rule 'StringToVarchar' do
     from UML_model::PrimitiveType
     to ORDB_model::XMLType
     filter do | primitivetype_in |
          "name = String"
     end

     mapping do | primitivetype_in, xmltype_out |
          xmltype_out.Name  =  "#VARCHAR"
          xmltype_out.model  =  primitivetype_in.Package
```

Figura 4-11. Código de la Transformación en RubyTL

Es importante mencionar, que al igual que con el caso de ATL, que el código generado no es completamente implementable. Es decir, existen situaciones donde es necesario que el desarrollador introduzca o modifique alguna sentencia; tal es el caso de la implementación de los *helpers* (funciones auxiliares), ya que para no complicar el meta-modelo a nivel PSM con instrucciones dependientes de un lenguaje de transformación específico no se pueden recoger toda la información necesaria para poder implementar completamente el *helper*.

4.2 Caso de estudio

Una vez implementadas las transformaciones entre los modelos de nivel PIM conformes al meta-modelo de UML y el modelo de nivel PSM conforme al meta-modelo ORDB4ORA usando MeTAGeM, se procede a realizar la prueba de funcionamiento de las mismas. Para esto, se realiza la ejecución de las mismas tomando como entrada un modelo conceptual específico, representado por medio de un diagrama de clases de UML.

A continuación, se muestra el modelo origen y el resultado obtenido.

4.2.1 Modelo Conceptual de Datos

El caso de estudio que se utiliza para probar las transformaciones implementadas se ha tomado de [170, pag. 5]. Este caso ha sido seleccionado por dos razones: la primera, porque el hecho de utilizar un caso de estudio "externo" impide que se adapten los modelos de acuerdo a las características de la herramienta; y la segunda es que es el mismo caso de estudio utilizado para validar las transformaciones en [191] lo que facilita la comparación de los resultados obtenidos.

La base de datos *Online Movie* (*Online Movie Database*, OMDB) está diseñada para manejar información referente a películas, actores, directores, dramaturgos, e información relacionada con las películas. Los usuarios pueden acceder a esta información en la página web de OMDB y comprar productos (por ejemplo, videos de películas, DVDs, libros, Cds y cualquier otro tipo de productos relacionados con las películas). La información de la película que se almacena, incluye el título, el director, el sitio web oficial de la película, género, estudio, una breve sinopsis, y el elenco (actores y roles de cada uno). Cada película tiene un máximo de cinco comentarios de editores externos, y el número ilimitado de comentarios de los usuarios introducidos por los mismos en línea. En la página

web OMDB se ofrecen productos para la venta incluyendo vídeos de películas y DVDs. La información que se almacena sobre los vídeos y DVDs incluye el título, la categoría, el precio de lista, fecha de lanzamiento, y otra información pertinente.

En la Figura 4-12 se muestra el modelado del caso de estudio por medio de un diagrama de clases de UML.

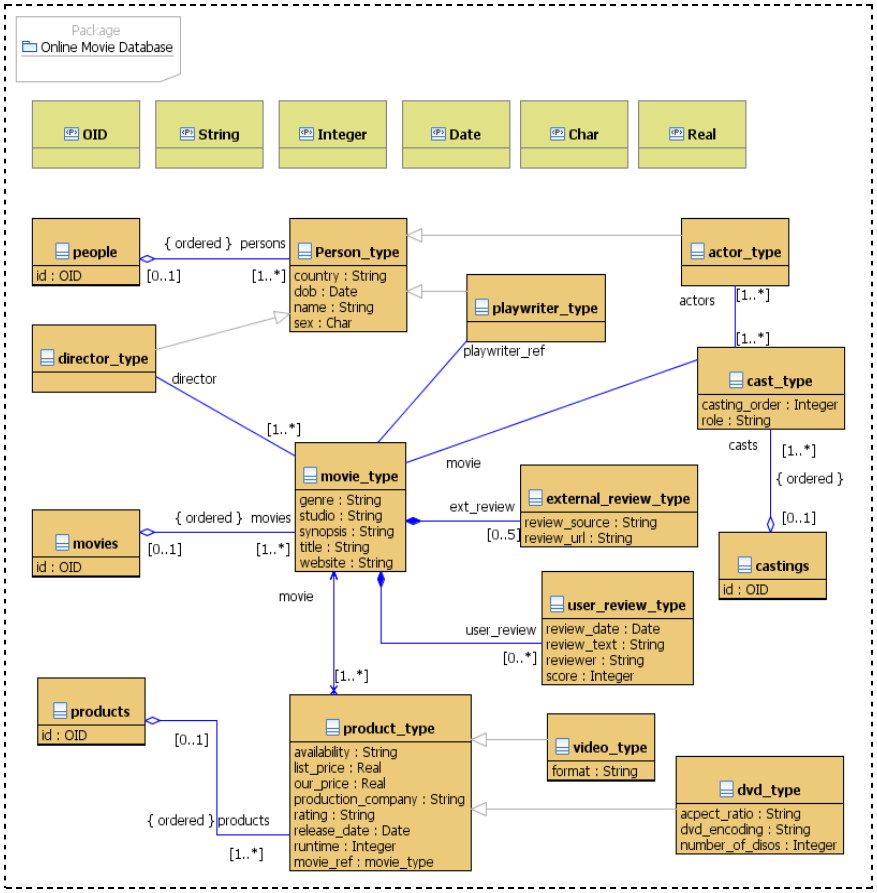

Figura 4-12. Modelo Conceptual de Datos – Caso de Estudio OMDB

4.2.2 Modelo Lógico Específico para Oracle 10g

Por medio de las las transformaciones, el modelo conceptual de datos mostrado en la Figura 4-12 se transforma en un modelo lógico conforme al meta-modelo ORDB4ORA. En la Figura 4-13 se muestra el modelo destino, conforme al meta-modelo ORDB4ORA, representado gráficamente por medio del editor de M2DAT-DB.

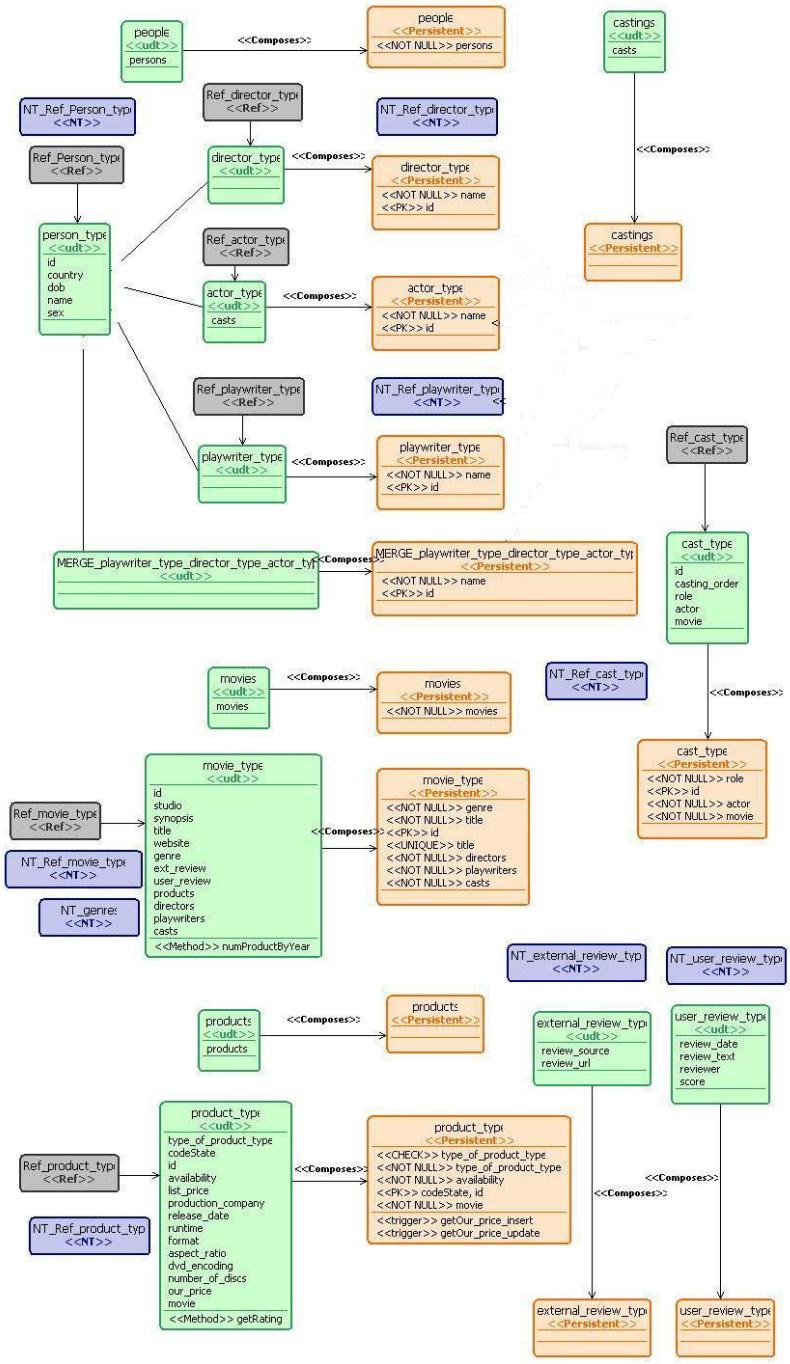

Figura 4-13. Modelo OR Representado Gráficamente – Caso de Estudio OMDB

En la Figura 4-14 se muestra el modelo con el editor en formato de árbol proporcionado en M2DAT-DB.

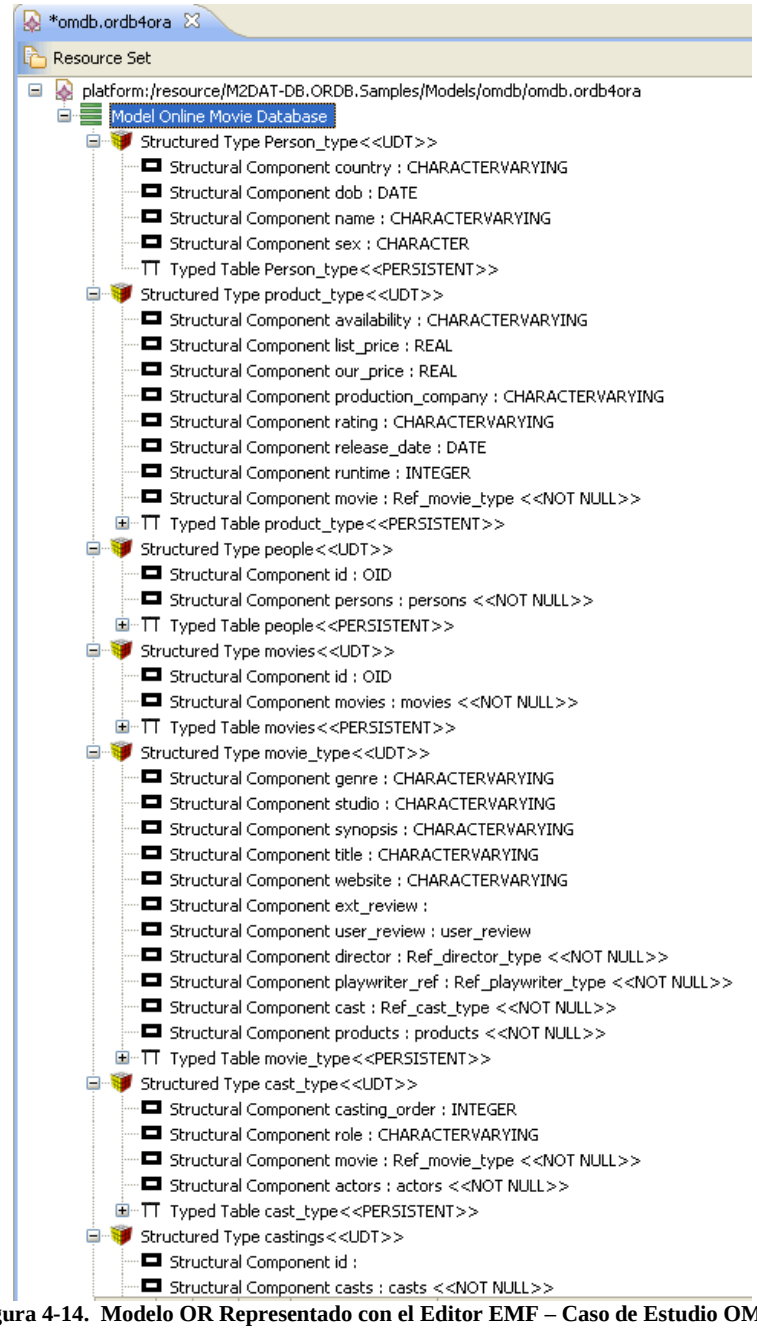

Figura 4-14. Modelo OR Representado con el Editor EMF – Caso de Estudio OMDB

Como se ha comentado anteriormente, la verificación de los resultados obtenidos con MeTAGeM se realiza desde dos puntos de vista: en primer lugar desde el punto de vista sintáctico, es decir, comparando los dos archivos de transformaciones para verificar las similitudes y diferencia; y, en segundo lugar, desde el punto de vista del funcionamiento, esto es, comprobando que la ejecución de la transformación con un modelo origen en particular genera el mismo (o similar) modelo destino en ambos casos.

En cuanto a la comparación sintáctica se puede decir que el archivo que contiene el código de la transformación generado con MeTAGeM es similar al implementado de forma manual en M2DAT-DB. Sin embargo, es importante mencionar, tal y como se puede constatar en el Anexo F, es necesario introducir algunas modificaciones al código generado con MeTAGeM, como por ejemplo agregar los estereotipos utilizados en M2DAT-DB o implementar correctamente el código de los *helpers*.

En cuanto a la comprobación de la ejecución de la transformación utilizando en ambos casos un mismo modelo origen (caso de estudio OMDB) se puede observar que el modelo destino generado es similar que el generado utilizando las transformaciones de M2DAT-DB. La diferencia radica que M2DAT-DB ofrece la posibilidad de que el usuario seleccione, por medio de anotaciones al modelo origen, el tipo de elemento colección (*Varray* o *Nested Table*) al cual transformar los elementos de tipo multivaluados (*Attribute* o *Association*). En MeTAGeM se estable una opción por defecto, donde este tipo de elementos se transforma necesariamente al tipo de colección *Nested Table*. El resto de los elementos transformados se transforman de manera similar en ambos casos.

Conclusión

A modo de conclusión de esta tesis, en este capítulo se resumen las principales contribuciones realizadas y se corrobora el cumplimiento de los objetivos establecidos al inicio de la misma. Además, se realiza un análisis de los resultados obtenidos proporcionando una enumeración de las publicaciones que sirven para contrastarlos, tanto en foros nacionales como en foros internacionales. Por último, se presentan las líneas de investigación futuras que permitirán seguir trabajando en la mejora de la propuesta presentada en esta tesis.

5.1 Análisis de la Consecución de Objetivos

Al inicio de este trabajo, en el capítulo 0 se presentaron una serie de objetivos parciales cuyo cumplimiento implica el cumplimiento del objetivo principal de esta tesis: la definición de una propuesta metodológica que, aplicando los principios del MDE, facilite el desarrollo (semi-)automático de transformaciones de modelos mediante la especificación de las mismas en un lenguaje de alto nivel independiente de plataforma y su posterior transformación a un lenguaje dependiente de plataforma. Dicho entorno deberá incluir una guía metodológica y una herramienta de soporte.

A continuación se analizará el cumplimiento de cada uno de los objetivos parciales:

O1. Estudio de Trabajos Previos

El estudio de trabajos previos se ha realizado por medio de un análisis y evaluación de por un lado, trabajos relacionados con el desarrollo de transformaciones de modelos en el ámbito de MDE, y, por otro lado, los lenguajes y herramientas de modelos que siguen la aproximación híbrida.

En la sección 2.1 se ha presentado el análisis realizado de las propuestas existentes para el desarrollo de transformaciones en el ámbito de MDE. Para facilitar y unificar los criterios a tener en cuenta durante el análisis de las diferentes propuestas se determinaron, un conjunto de característas a evaluar en cada una de ellas.

El conjunto de propuestas a evaluar se seleccionó por medio de un proceso de revisión sistemática aplicado a las principales fuentes de búsquedas del ámbito de la Ingeniería de Software.

Las principales conclusiones a las que se llegaron son:

- Si bien la mayoría de las aproximaciones evaluadas reconocen la necesidad de aplicar los principios de MDE al desarrollo de las transformaciones e incluso realizan propuestas de cómo deberían ser implementados, son muy pocas las que implementan una herramienta que soporte la definición de las transformaciones aplicando los principios de MDE. En la etapa final de la escritura de esta tesis, se encontraron dos propuestas [93, 126] en la misma línea que la presentada en esta tesis, esto contribuye a justificar la importancia de aplicar MDE al desarrollo de las transformaciones.
- Siguiendo las líneas de investigación planteadas en [39] y [79], en esta tesis se propone un entorno de desarrollo que facilite el modelado (semi-)automático de transformaciones de modelos mediante la especificación de las mismas en un lenguaje de alto nivel independiente de plataforma y su posterior transformación a un lenguaje específico de plataforma.

En la sección 2.2, en primer lugar se ha realizado la selección de la aproximación de transformaciones de modelos que se seguirá para el modelado de las transformaciones a nivel PDM. Para esto se hace un resumen de las aproximaciones existentes y se justifica la selección de la aproximación híbrida realizada. Posteriormente, se ha presentado el análisis y la evaluación realizada de los lenguajes y herramientas de transformación de modelos, que siguen la aproximación híbrida. Para facilitar y unificar los criterios a tener en cuenta durante el análisis de los diferentes lenguajes y herramientas se determinaron un conjunto de caracterísitas a evaluar en cada una de ellas.

Las principales conclusiones a las que se llegaron son:

- Los lenguajes analizados soportan la definición de diferentes tipos de reglas y de funciones auxiliares, lo que facilita la codificación de las transformaciones. En cuanto a la cardinalidad de los elementos en una regla de transformación, solo el lenguaje ATL permite definir reglas entre múltiples elementos del modelo origen y múltiples elementos del modelo destino.
- Si bien todos los lenguajes tienen un meta-modelo que los define especificado sólo el meta-modelo de ATL está implementado, lo que permite que solo en ATL se puedan definir las transformaciones de modelos como modelos en sí mismos. Otro punto a favor de ATL es el hecho de contar con mecanismos de extracción e inyección de código.

A partir de estas conclusiones se han determinado el conjunto de características que se deberán contemplar a la hora de realizar la especificación del meta-modelo que sigue la aproximación híbrida utilizado a nivel PDM.

O2. Especificación de los meta-modelos de los diferentes niveles

La principal premisa perseguida en el desarrollo de esta tesis, es que el entorno de desarrollo de transformaciones, MeTAGeM, debe ser lo suficientemente general como para poder ser aplicada a transformaciones de cualquier dominio y de manera independiente del lenguaje de transformación a utilizar finalmente. Como se ha visto, MeTAGeM propone el modelado de las transformaciones a diferentes niveles de abstracción (PIM, PSM y PDM). Para cada uno de estos niveles ha sido necesario definir el meta-modelo correspondiente.

Para realizar la especificación del meta-modelo de nivel PIM se ha considerado conveniente centrarse en la identificación de los tipos de relación existentes entre los elementos de los diferentes meta-modelos que intervienen de modo genérico en una transformación de modelos y no en los constructores específicos utilizados por los meta-modelos de los diferentes lenguajes de transformación.

A partir de la identificación de los tipos de relaciones y de los tipos de elementos a transformar, en la sección 3.1.2.1 se presenta la especificación del meta-modelo para definición de modelos de transformación de nivel PIM.

De la misma manera, en la sección 3.1.2.3 se realiza la especificación del meta-modelo que permite implementar los modelos de transformación a nivel PSM. Como primera aproximación a implementar, en esta tesis, se ha seleccionado la aproximación híbrida El meta-modelo de nivel PSM recoge los constructores comunes a los lenguajes de transformación que soportan dicha aproximación.

A nivel PDM, MeTAGeM propone el modelado de las transformaciones de acuerdo a dos lenguajes de transformación, ATL y RubyTL. Como se ha visto a lo largo de esta tesis, para el modelado de las transformaciones usando ATL se propone el uso del editor del propio lenguaje y para el modelado de las transformaciones usando RubyTL ha sido necesario la implementación de un editor. Por esto, en la sección 3.1.2.2 se especifica el meta-modelo conforme al lenguaje de RubyTL.

O3. Especificación de meta-modelos que permitan automatizar las transformaciones de modelos.

Se ha especificado un meta-transformador formado por: el conjunto de transformaciones que permiten pasar de modelos de transformación de nivel PIM a modelos de transformación de nivel PSM; por el conjunto de transformaciones que permite pasar de modelos de transformación de nivel PSM a modelos de

transformación de nivel PDM. En este último caso, se han considerado dos conjuntos de transformación diferentes, uno por cada lenguaje de transformación soportado, ATL y RubyTL.

En la sección 3.1.3 se muestra la especificación realizada de las transformaciones. Primero, se ha realizado un análisis de los meta-modelos que intervienen en la transformación, después, se han definido las reglas de transformación entre cada uno de los elementos de los meta-modelos tanto en lenguaje natural como en gramática de grafos.

O4. Construcción de la meta-herramienta

Para la construcción de la meta-herramienta MeTAGeM se ha realizado una serie de actividades:

En primer lugar, se ha realizado la especificación de la arquitectura de MeTAGeM. Para realizar dicha especificación se han seguido las directrices establecidas en [191], donde se propone considerar dos niveles de abstracción, separando, el diseño conceptual (nivel PIM) del diseño técnico (nivel PSM).

En la sección 3.2.1 se ha presentado la definición de la arquitectura a nivel PIM, donde se definen cada uno de los artefactos que formarán parte de la meta-herramienta sin contemplar las cuestiones técnicas de implementación. Posteriormente, en la sección 3.2.2 se ha presentado la arquitectura a nivel PSM. Dicha arquitectura se obtiene a partir del refinamiento de la arquitectura conceptual especificando para cada uno de los artefactos definidos a nivel PIM, la tecnología utilizada para su implementación.

La arquitectura de MeTAGeM se estructura de acuerdo a dos dimensiones ortogonales:

- Por un lado, se puede considerar a MeTAGeM como un conjunto de módulos, uno por cada nivel, en el que se pueden especificar los modelos de transformación. Cada módulo se encapsula en un DSL que comprende, el conjunto de conceptos relacionados con dicho módulo, así como las reglas de transformación que permite transformar un modelo de un nivel en otro modelo de otro nivel.
- Por otro lado, la arquitectura se define siguiendo el principio de abstracción en capas [122, 167], donde cada capa representa una vista en particular, definiéndose la capa de presentación, la capa lógica y la capa de persistencia. Esta separación en capas permite asegurar el mantenimiento de la trazabilidad entre los diferentes artefactos, la reutilización de los mismos

y el mejor control de su evolución a la hora de incorporar nuevas funcionalidades

En segundo lugar, en la sección 3.2.3 se muestra el proceso seguido para la construcción de la meta-herramienta.

Partiendo de la especificación de la arquitectura a nivel lógico se ha procedido a la implementación de cada uno de los módulos que forman MeTAGeM: implémentandose cada uno de los meta-modelos, desarrollando editores que permitan modelar las transformaciones conformes a los meta-modelos, realizando la personalización de dichos editores, implementando las transformaciones entre los modelos de los diferentes niveles de MeTAGeM y especificando el conjunto de restricciones que permite realizar la validación de los modelos en cada caso.

O5. Validación del entorno desarrollo propuesto

La validación del entorno se ha realizado de dos maneras:

- Por un lado, se valida el funcionamiento de la meta-herramienta MeTAGeM por medio del desarrollo de un meta-caso de estudio que consiste en la implementación de las reglas de transformación que permiten pasar del modelo conceptual de datos, representado con un diagrama de clases UML, a un modelo lógico espécifico representado por medio de un modelo Objeto-Relacional, para el desarrollo de bases de datos Objeto Relacionales.
- Y, por otro lado se valida el funcionamiento de dichas las reglas de transformación, realizando un caso de estudio específico.

Por todo lo expuesto, se puede concluir que se han cumplido satisfactoriamente todos los objetivos planteados y se ha probado la hípotesis propuesta.

5.2 Principales Contribuciones

Como parte de los resultados de esta tesis se ha realizado una serie de contribuciones. A continuación se presenta una descripción de las mismas:

Un análisis de las propuestas existentes sobre la utilización de los principios de MDE en el ámbito de las transformaciones de modelos.

En la sección 2.1 se ha realizado un análisis de los trabajos existentes que proponene la utilización de los principios de MDE en el desarrollo de transformaciones de modelos. En dicho análisis se evalúan los trabajos más relevantes y las diferentes herramientas que los implementan.

Para unificar la evaluación de las propuestas se han establecido un conjunto de características a analizar, lo que ha permitido realizar una comparación de las propuestas. Además, estas características pueden ser aplicadas en futuros trabajos para extender el análisis de las propuestas realizado.

Un análisis de los diferentes lenguajes de transformación de modelo existentes actualmente.

Con el objetivo de determinar el conjunto de constructores que deberán ser contemplados en el meta-modelo del nivel PSM que implementa la aproximación híbrida, en la sección 2.2, se ha realizado un análisis de los diferentes lenguajes de transformación, que siguen dicha aproximación, así como de las herramientas que implementan dichos lenguajes.

De acuerdo a la experiencia adquirida en la implementación de transformaciones en diferentes ámbitos, se han establecido un conjunto de características a evaluar en cada uno de los lenguajes y de las herramientas. Estas características han permitido detectar los constructores comunes a los lenguajes de transformación evaluados.

Especificación de una metodología para el desarrollo dirigido por modelos, de transformaciones de modelos.

Una de las principales contribuciones de esta tesis es la especificación de la metodología, como parte del entorno MeTAGeM, para el desarrollo dirigido por modelos de transformaciones de modelos. Dicha metodología se compone de:

- Un **proceso de desarrollo** de transformaciones de modelos dirigido por modelos.
- **Los diferentes meta-modelos** que soportan el proceso definido, permitiendo modelar transformaciones en los diferentes niveles propuestos en el proceso.
- La especificación de un **conjunto de reglas de transformación** que permitirán generar de forma (semi-)automática los modelos propuestos en cada nivel.

Desarrollo de la herramienta que soporta la metodología definida.

Otra de las principales contribuciones de esta tesis es el desarrollo de la **herramienta**, que soporta la metodología propuesta. Dicha herramienta se compone de:

- La **arquitectura** que define cada uno de los componentes que la forman, así como las relaciones entre ellos.

- Los **editores** gráficos, basados en árbol, para el modelado de las transformaciones en los diferentes niveles.
- Un **meta-transformador**, que implementa los diferentes conjuntos de reglas de transformación que permiten generar los modelos a diferentes niveles de abstracción: transformación de PIM a PSM, de PSM a PDM (ATL y RubyTL), y, finalmente, los extractores de código que permiten obtener el código que implementa la transformación en el lenguaje deseado.

La principal característica de MeTAGeM es su naturaleza extensible e interoperable. La extensibilidad facilita la incorporación de nuevas funcionalidades, como por ejemplo, el soporte de una nueva aproximación a nivel PSM o de un nuevo lenguaje de transformación.

Estas nuevas funcionalidades se soportarán mediante la creación de nuevos módulos en MeTAGeM, que serán directamente compatibles con los módulos existentes sin ningún esfuerzo adicional, es decir, sin la necesidad de construir puentes entre los diferentes espacios tecnológicos, ya que se utiliza la misma tecnología y el mismo proceso para desarrollarlos.

Desarrollo de Casos de Estudios.

Para la validación de MeTAGeM se ha desarrollado un meta-caso de estudio que implementa las reglas de transformación que permiten pasar del modelo conceptual de datos, representado con un diagrama de clases UML, a un modelo lógico espécifico representado por medio de un modelo Objeto-Relacional, para el desarrollo de bases de datos Objeto-Relacionales. Posteriormente estas transformaciones se han probado aplicándolas a la transformación entre modelos específicos. El meta-caso de estudio completo se encuentra disponible en el CD adjunto.

Además, mientras se implementaba la herramienta, se han realizado una serie de casos de estudios puntuales con el objetivo de probar determinadas funcionalidades de la herramienta. Dichos casos de estudios también están disponibles en el CD adjunto.

5.3 Resultados Científicos

Algunos de los resultados obtenidos durante el desarrollo de la tesis se han publicado en diferentes congresos, tanto nacionales como internacionales. A continuación se presentan las diferentes publicaciones, agrupadas por tipo de publicación. Dada la naturaleza del trabajo de esta Tesis Doctoral, para la publicación en foros de impacto, ha sido necesario esperar a finalizar la

implementación de la herramienta. Es por ello que en el momento de redactar esta memoria de Tesis se cuenta con publicaciones de menor impacto que recogen resultados parciales de la misma. En la actualidad se está trabajando en la publicación en foros de mayor impacto.

- **Artículos en Conferencias Internacionales**
  - Vara, J.M., Bollati, V.A., Irrazabal, E. y Marcos, E. (2010). *Using EMF and ATL to improve types management in MDE proposals*. 2nd International Workshop in Model Transformation with ATL (MtATL 2010), Málaga, España. (30 de Junio, 2010).
  - Vara, J.M., Vela, B., Bollati, V. A. y Marcos, E. (2009). *Supporting Model-Driven Development of Object-Relational Database Schemas: a Case Study*. **ICMT2009 - International Conference on Model Transformation**, Zurich, Switzerland. (29 y 30 de Junio, 2009) (**Porcentaje de Aceptación: 22%**).
  - Vara, J.M., Bollati, V. A., Vela, B. y Marcos, E. (2009). *Leveraging Model Transformations by means of Annotation Models*. 1st International Workshop in Model Transformation with ATL (MtATL 2009), Nantes, France. (8 y 9 de Julio, 2009).
- **Artículos en Conferencias Iberoamericanas**
  - Bollati, V.A., Vara, J.M, Vela, B. y Marcos, E. (2009). *Uso de Modelos de Anotación para automatizar el Desarrollo Dirigido por Modelos de Esquemas XML*. XII Conferencia Iberoamericana de Ingeniería de Requisitos y Ambientes de Software. (IDEAS'09), Medellín (Colombia). (13 al 17 de Abril, 2009) (**Porcentaje de Aceptación: 46%**).
  - Bollati, V.A., Vela, B., Vara, J.M. y Marcos, E. (2008). *Una Aproximación Dirigida por Modelos para el Desarrollo de Bases de Datos Objeto-Relacionales*. XIV Congreso Argentino de Ciencias de la Computación. (CACIC 2008). Chilecito (La Rioja, Argentina). (6 al 10 de Octubre, 2008).
  - Bollati, V.A., Marcos, E., Vara, J.M. y Vela, B. (2007). *Analisis de Herramientas MDA*. XIII Congreso Argentino de Ciencias de la Computación. (CACIC 2007). Corrientes y Resistencia, Argentina. (1 al 5 de Octubre, 2007)
- **Artículos en Conferencias Nacionales**
  - Jiménez, A., Vara, J. M., Bollati, V. A., Marcos, E. (2010). *Mejorando el nivel de automatización en el desarrollo dirigido por modelos de editores*

*gráficos***.** VII Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'10. Valencia, España (7 de Septiembre, 2010).

- Irrazabal, E. Vara, J. M., Bollati, V. A. y Marcos, E. (2009). *Gestion de Tipos Primitivos en Propuestas de DSDM: Aplicación al Modelado de Esquemas de Bases de Datos Objeto-Relacionales*. VI Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'09. San Sebastián, España (8 de Septiembre, 2009).
- Bollati, V. A., Victor Sánchez, Vela, B. y Marcos, E. (2009). *Análisis de QVT Operacional Mappings: un caso de estudio*. VI Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'09. San Sebastián, España (8 de Septiembre, 2009).
- Bollati, V.A., Vara, J.M., Vela, B. y Marcos, E. (2008). *Una Aproximación Dirigida por Modelos para el Desarrollo de Esquemas XML*. **XIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'08)**. Gijón, España. (7 al 10 de Octubre, 2008). (**Porcentaje de Aceptación: 25%**)
- Vara, J.M., Bollati, V., Vela, B. y Marcos, E. (2008). *Uso de Modelos de Anotación para automatizar el Desarrollo Dirigido por Modelos de Bases de Datos Objeto-Relacionales*. V Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'08. Gijón, España (7 de Octubre, 2008).
- Bollati, V. A., Vara, J.M., Vela, B. y Marcos, E. (2007). *Una Revisión de Herramientas MDA*. IV Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'07. Zaragoza, España (11 de Septiembre, 2007).

- **Registro de Propiedad Intelectual**
  - Título: *M2DAT/DB: Herramienta para el Desarrollo Dirigido por Modelos de BD*.
    - Inventores: E. Marcos, B. Vela, J.M. Vara, V. A. Bollati
    - Nº de Aplicación: M-8452/2008
    - Pais: España
    - Fecha: 24/10/2008
    - Entidad Titular: Universidad Rey Juan Carlos
    - Paises Participantes: España

5.4 Trabajos Futuros

A partir de las contribuciones realizadas en esta tesis, se han detectado varias líneas de investigación en las que seguir trabajando. Algunas de ellas simplemente no se habían considerado como objetivos de esta tesis, mientras que otras han surgido durante el desarrollo de la misma. A continuación, se resumen las principales líneas futuras de investigación.

5.4.1 Trazabilidad

Como se ha visto a lo largo de este trabajo, la trazabilidad ha sido siempre un tema importante en el ámbito de la Ingeniería de Software; con el surgimiento de MDE, y más específicamente de MDA, la administración de la trazabilidad ha cobrado cada vez mayor importancia.

El papel clave que tienen los modelos en el proceso de desarrollo de software facilita la tarea de mantener las trazas desde los elementos identificados en la especificación de los requerimientos hasta la implementación de los mismos en el código. El hecho de brindar mecanismos que permitan recuperar el elemento del meta-modelo origen a partir de un elemento del meta-modelo destino es un problema al que se le viene dando cada vez más importancia en diferentes ámbitos. Ya incluso, en *Hansel y Gretel* (Jacob y Wilhelm Grimm, Alemania, 1857) se identifica la importancia de marcar, por algún medio, el camino de retorno al lugar de origen.

En el ámbito de MDE, los objetos principales que se obtienen a lo largo del proceso de desarrollo son modelos. Por lo tanto, el soporte a la trazabilidad debería ser tan simple como crear y mantener las trazas entre los elementos de los modelos. Dichas trazas podrían ser generadas automáticamente si los modelos se conectan entre si por medio de transformaciones y si el lenguaje de transformación que se utiliza para implementarlos soporta el mantenimiento de la trazabilidad. [184]. De tal manera que, si se modifica algún elemento del modelo origen, la modificación puede ser trasladada al elemento del modelo destino correspondiente. El principal inconveniente detectado es que el soporte técnico a la trazabilidad es aún inmaduro, a pesar de ser una característica obligatoria en las transformaciones de modelos de acuerdo a lo establecido por el estándar QVT, son muy pocos los lenguajes de trasformación que brindan soporte al mecanismo de trazabilidad.

Por todo ello, uno de los trabajos futuros que se plantea es el de brindar mecanismos para el mantenimiento de la trazabilidad en MeTAGeM. Con este fin

se ha comenzado a trabajar en determinar la información relevante que debería mantenerse entre los modelos de los diferentes niveles de MeTAGeM. En esta misma línea de investigación pero a un mayor nivel de abstracción, en el marco de la tesis doctoral de Iván Santiago, miembro del Grupo Kybele, se ha comenzado a trabajar en la extracción de la información relevante de los modelos CIM y su traslado a modelos PIM y la gestión de las trazas a lo largo de las diferentes etapas del ciclo de desarrollo de software propuesto por la metodología SoD-M [68].

5.4.2 Transformaciones Bidireccionales

La bidireccionalidad en las transformaciones de modelos es un mecanismo para el mantenimiento de la coherencia entre dos (o más) fuentes de información relacionadas [67]. En el contexto de MDE, la bidireccionalidad permite sincronizar las diferentes vistas de los modelos, y debería tenerse en cuenta en el desarrollo de cualquier herramienta de soporte a las actividades del MDE si se quiere contemplar cuestiones como la evolución de meta-modelos y modelos, o el soporte a los mecanismos de ingeniería inversa. Incluso, el estándar QVT apuesta por la bidireccionalidad de los lenguajes de transformación, aunque las implementaciones del mismo no la soportan completamente.

A pesar de la utilidad e importancia de soportar el modelado de las transformaciones bidireccionales, se ha podido comprobar a lo largo del análisis realizado que los lenguajes de transformación, y tal y como se afirma en algunos trabajos como [67, 182], no brindan un soporte completo a la bidireccionalidad. De hecho, la mayoría de los lenguajes tiene deficiencias en cuestiones como depuración, verificación, o composición de reglas, que son cuestiones clave para apoyar las transformaciones bidireccionales.

Por todo lo expuesto, se considera importante permitir el modelado de las transformaciones en forma bidireccional. Para esto, se pretende implementar en MeTAGeM la bidireccionalidad de las transformaciones por medio de dos conjuntos de reglas de transformación diferentes.

A partir de la especificación de las relaciones entre los meta-modelos (nivel PIM), se pueden obtener, de forma relativamente fácil, las reglas de transformación que permitan transformar del meta-modelo origen al meta-modelo destino y viceversa. La parte superior de la Figura 5-1 muestra la transformación entre el meta-modelo MM_A (origen) y el meta-modelo MM_B (destino), donde se establece una relación entre los elementos A1 y A2 de MM_A y el elemento B1 de MM_B; el tipo de la transformación es de N a 1. Ahora bien, si se invierten los meta-modelos, es decir, se toma como meta-modelo origen a MM_B y como

meta-modelo destino MM_A, la relación se convierte en 1 a N (parte inferior de la figura). En MeTAGeM se deberán implementar dos conjuntos de transformaciones diferentes, una que tome como origen al meta-modelo MM_A y como destino al meta-modelo MM_B; y otra que tome como origen al meta-modelo MM_B, y como destino al meta-modelo MM_A. De tal manera, el desarrollador sólo establecerá las relaciones entre los meta-modelos en un sentido y, seleccionando la transformación correspondiente podrá obtener el código que implementa la transformación entre MM_A y MM_B, o, el código que implementa la transformación entre MM_B y MM_A. De esta manera, de forma transparente para el desarrollador se soporta el modelado bidireccional de las transformaciones.

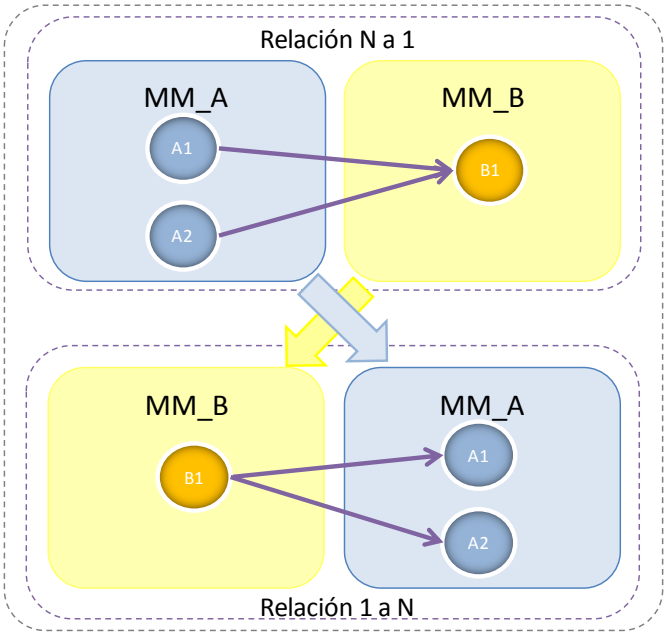

Figura 5-1. Transformaciones Bidireccionales

5.4.3 Calidad en la Definición de las Transformaciones

La calidad del software está tomando mayor importancia en las organizaciones por su influencia en los costes finales y como elemento diferenciador de la competencia y de la imagen frente a sus clientes. De tal manera, que muchas organizaciones están implantando modelos de mejora de procesos software.

Las transformaciones de modelos son uno de los pilares fundamentales en el ámbito del MDE, debido a que es una de las operaciones más importantes por medio de la cual se puede manipular modelos. En particular, un proceso basado en MDE puede ser parcialmente realizado mediante la ejecución de una serie de transformaciones de modelos. Como consecuencia, la calidad de las transformaciones de modelo afecta directamente a los artefactos obtenidos en un proceso de MDE.

Uno de los métodos más utilizados para evaluar la calidad del software en general son las métricas de software. Existen algunos trabajos en los que se ha comenzado a aplicar dicho método a las transformaciones de modelo [187, 205], en los que se definen modelos de calidad sobre los atributos en el contexto de la transformación de modelos que se propone.

El entorno MeTAGeM está formado por un proceso sistemático para el desarrollo de transformaciones. Teniendo esto en cuenta, se pueden implementar los principios tradicionales de mejora de proceso de software al mismo, como por ejemplo la norma ISO 15504, como un modo de asegurar la calidad de las transformaciones generadas.

En este sentido, actualmente se está trabajando en el análisis de las propuestas de los trabajos citados anteriormente en cuanto a calidad en las transformaciones de modelos, con el objetivo de determinar la aplicabilidad de dichos trabajos a MeTAGeM.

5.4.4 Introducir Decisiones de Diseño en las Transformaciones

Durante la implementación de la herramienta MeTAGeM se detectó que un proceso de definición de las transformaciones completamente automático no es sólo inviable, sino que es poco recomedable. En muchas situaciones es necesario introducir algunas decisiones de diseño que guíen el proceso de desarrollo, sobre todo cuando se baja a niveles dependientes de plataforma donde es necesario indicar cómo los conceptos abstractos detectados en los niveles superiores se convierten en objetos concretos de una plataforma determinada. En este sentido, se necesita poder introducir dicha información de diseño en el proceso de desarrollo de las transformaciones.

Otro factor que influye a la hora de automatizar el proceso de transformación completo es la naturaleza de algunos modelos, así como el hecho de que los modelos que participan en la transformación pertenezcan a dominios completamente diferentes, lo que implica que puedan surgir ambigüedades en la transformación de los elementos. Para evitar este tipo de inconvenientes se

necesitan definir reglas de transformación no uniformes [88], es decir, reglas que tenga diferentes comportamientos dependiendo de los parámetros recibidos.

Para soportar dicho comportamiento se propone usar transformaciones parametrizables, es decir, transformaciones donde el desarrollador pueda introducir sus decisiones en forma de parámetros en el momento de la ejecución.

En este sentido, se planea utilizar modelos de *weaving* como modelos de anotación que permitan introducir esos parámetros o decisiones de diseño. De este modo, el desarrollador, antes de ejecutar la transformación, deberá definir un nuevo modelo de *weaving* que anote el modelo origen. A partir de estos dos modelos se generaría el modelo destino. De esta manera, a partir del mismo modelo origen se podrían generar diferentes modelos destino, dependiendo del modelo de weaving o modelo de anotación utilizado.

5.4.5 Trabajos Futuros en el Contexto de MeTAGeM

En esta sección se presentan las futuras lineas de investigación en el contexto de MeTAGeM:

1. **Implementar el meta-modelo a nivel PSM para el resto de las aproximaciones existentes.** Como se especifica a lo largo de la tesis, en el nivel PSM de MeTAGeM se propone el modelado de las transformaciones siguiendo las diferentes aproximaciones de transformación de modelos existentes (declarativa, imperativa, híbrida, basada en grafos, etc). Actualmente, en este nivel se soporta el modelado siguiendo la aproximación híbrida; como futuro trabajo se propone el desarrollo de los DSLs para el resto de las aproximaciones. Esto permitirá que en el momento de realizar el modelado de la transformación, el desarrollador pueda seleccionar la aproximación que considere más adecuada para implementar su transformación. Para desarrollar el DSL se deberán analizar las diferentes propuestas que implementan cada una de las aproximaciones con el objetivo de determinar el conjunto de constructores que deberán ser definidos por el meta-modelo en cada uno de los casos.
2. **Implementar transformaciones horizontales a nivel PSM.** Al tener implementado varios DSL a nivel PSM, será interesante poder realizar transformaciones horizontales entre los modelos definidos al mismo nivel pero en diferentes aproximaciones; así, por ejemplo, pasar de forma (semi-) automática de un modelo de transformación siguiendo la aproximación híbrida, a un modelo de transformación siguiendo una aproximación

declarativa. Para esto, se deberían especificar e implementar el conjunto de reglas de transformación que permitan transformar unos modelos en otros.

3. **Mejorar el meta-modelo de RubyTL.** Como se ha visto a lo largo de la realización de esta tesis, para dar soporte a las transformaciones a nivel PDM utilizando el lenguaje RubyTL ha sido necesario desarrollar un DSL que lo implemente. Para la definición del meta-modelo de dicho DSL se han realizado numerosos casos de estudio, se han analizado casos de estudios disponibles en los foros del lenguaje, y se han consultado a los autores del mismo. Sin embargo, existen cuestiones que no han podido ser representadas en el meta-modelo como, por ejemplo, la implementación de los métodos estáticos, ya que como RubyTL es un lenguaje de transformación embebido en el lenguaje Ruby y para representar algunos constructores se utiliza el lenguaje Ruby directamente. Como trabajo futuro se propone ampliar el meta-modelo para dar soporte a las estructuras de Ruby que se utilizan en RubyTL para llevar a cabo las transformaciones.
4. **Mejorar los editores existentes.** Como se ha visto en la sección 3.2 los editores propuestos para cada uno de los DSL son editores gráficos con formato de árbol, estos editores han sido personalizados para mostrar información relevante de cada modelo. Con el objetivo de mejorar los editores existentes se está comenzando a trabajar la implementación de editores gráficos utilizando una notación similar a UML, es decir, con cajas y flechas. En esta misma línea de investigación en el marco de la tesis doctoral de David Granada, becario FPI del Grupo Kybele, se esta trabajando en el desarrollo de un meta-editor que permita realizar la implementación de editores, tanto gráficos como textuales, de forma relativamente sencilla para el desarrollador.
5. **Implementar el resto de los lenguajes de transformación, de acuerdo a cada una de las aproximaciones seleccionadas.** Para cada una de las aproximaciones seleccionadas a nivel PSM se deberían definir, como mínimo, un lenguaje de transformación de modelos, que implemente dicha aproximación, a nivel PDM. Como se ha visto en el capítulo 2, la mayoría de los lenguajes de transformación de modelos existentes no están desarrollados aplicando los principios de MDA. De hecho, en la mayoría de los casos, si bien tienen la especificación de un meta-modelo que los define, no tienen implementado dicho meta-modelo, por lo que, para que MeTAGeM soporte el lenguaje es necesario desarrollar un DSL que permita el modelado de las transformaciones conforme al lenguaje de transformación en particular. A partir de la definición de los meta-modelos a nivel PSM y PDM es necesario

definir el conjunto de reglas de transformación que permita pasar de un modelo definido a nivel PSM a un modelo definido a nivel PDM. Es importante aclarar que las reglas de transformación se definirán entre los meta-modelos de una aproximación y los meta-modelos de los lenguajes que sigan dichas aproximaciones.

6. **Obtener los modelos a partir del código.** Al realizar la definición de la sintaxis concreta de los meta-modelos del nivel PDM utilizando TCS, se puede obtener, por medio del mecanismo de inyección que proporciona TCS el modelo que representa la transformación a partir del código que la implementa. De la misma manera que con el mecanismo de extracción de código, se pretende personalizar el mecanismo de inyección de manera que sea aplicable a archivos con la extensión definida por un meta-modelo en particular.
7. **Mejorar los mecanismos de inyeccion/extracción de TCS.** Al realizar la definición de la sintaxis concreta de los meta-modelos se ha detectado que en muchas situaciones TCS no brinda todas las funcionalidades esperadas, como por ejemplo, el manejo de las variables que representan los meta-modelos en RubyTL, que no establece la correspondencia entre los elementos y los meta-modelos de forma correcta. Actualmente, se están analizando mecanismos para solucionar estos inconvenientes.
8. **Actualizar la versión de Eclipse.** Como se ha visto, la versión 1.0 de MeTAGeM está desarrollada basada en la versión 3.4.2 de Eclipse. La última versión de Eclipse disponible es la 3.6.0. MeTAGeM utiliza la herramienta AMW para el modelado de las transformaciones a nivel PIM. Cuando se comenzó el desarrollo de MeTAGeM, AMW no era compatible con la última versión de Eclipse, por lo que se ha optado por utilizar la versión más estable de AMW, que es la compatible con la versión 3.4.2 de Eclipse. Actualmente se está colaborando con el desarrollador de AMW, Marcos Didonet de la Universidad de Federal do Paraná (Brasil), para realizar la migración de la herramienta a la última versión de Eclipse. Una vez finalizada la migración de AMW se procederá a la migración de MeTAGeM.

Apéndice A: Extended Abstract

This appendix provides an extended summary of the doctoral thesis which is presented in this abstract.

In the first place, an overview of the history behind the undertaking of this thesis is offered, aiming to justify the work, while at the same time identifying those issues that were to be tackled in the production of the thesis. After that, the hypothesis is set forth, along with the main objectives of this thesis and the methodology followed throughout. That done, the solution proposed by this thesis is put forward and to conclude, the main contributions seen to emerge as products of the work are set out.

A.1 Background

The *Object Management Group* (OMG – www.omg.org) is an international consortium which groups together some 800 companies, producing open standards for a wide range of technologies. One of the main aims of the OMG group since its beginnings in 1989 has been to help computer-users to solve the problems of integration and inter-operation between and among systems, providing them with open and independent specifications from manufacturers on those issues.

At the end of the year 2000, therefore, and in pursuit of the above objective, the OMG presented *Model Driven Architecture* (MDA) [148], an architecture for the Model-Driven Software Development (MDSD). The fundamental feature of MDA is the *definition of formal models as first-class elements* for the design and implementation of systems and the *definition of the transformations for establishing relationships between models and for modifying their contents automatically.*

MDA defines three conceptual levels of modeling, depending on their level of abstraction: the business details and data of the application domain are modeled by means of Computer Independent Models (CIM).

To represent the functionality and structure of the systems, regardless of the technological details of the platform on which it is implemented, Platform Independent Models (PIM) are used.The PIM models can be refined as many times as one may wish, until a system description with the desired level of clarity and abstraction is obtained.

Finally, to combine the specifications contained in a PIM with the details of the chosen platform, Platform Specific Models (PSM) are used. Using the

different PSMs as a starting point, various implementations of the same system may be generated.

The MDA guide establishes that, at the PSM level, different models may be defined, representing different degrees of abstraction; these may be grouped into those which represent elements that are general to all the platforms, as well as those which represent elements that depend upon a specific platform. These latter type of models are classified into a new level, known as Platform Dependent Models (PDM).

In addition, we can consider the code that implements the system as another model, at a lower level of abstraction, which uses a textual notation for its definition.

One of the greatest contributions of MDA lies in the nature of all these models: they are formal models and consequently the computer can understand them. That being so, the main characteristic of this new paradigm, which from now on we shall refer to as *Model Driven Engineering*,(MDE), [26] is the role that models play in the *software* development process. In fact, MDE is a natural step in the historical tendency of software engineering to raise the level of abstraction in which the software is designed and developed. By way of example, we can point to the natural migration from the assembly languages to the fourth generation programming languages.

Up until now, the use of models was related almost exclusively to documentation tasks and, in the best of cases, to design tasks, thereby generating a skeleton of the final code (*Rational Rose* is a good example of this [95]). For that reason, the models were rejected just as soon as the corresponding development phase ended and they were not updated to reflect the changes carried out in the later models, or in the final code.

With the arrival of MDE, this situation has changed dramatically. Models have now taken on the main role in guiding the development process, by means of the definition of precise models, which capture all the requirements and specifications about the system to be built, as well as about the platform where they will be deployed. The main idea is to generate a series of models which allow us to represent the system with an ever-increasing level of abstraction. Thus, the level of detail in the models obtained in the final phases of the process will permit the (semi-) automatic generation of the code which implements the system.

The link that unites each new step in the process (each generation of a new model) is a **model transformation.** The principal aim of model transformations is to transform a model, (or various models) of the system, into another model (or

various models). These transformations should be (semi-) automatic and are implemented through the definition of transformation rules (*mappings*), between and among models.

New languages and tools [11, 41, 45, 47, 83, 85, 89, 94, 106, 121] emerge, making the automization of the transformation operation easier. These languages and tools differ from each other in a variety of ways, such as: the paradigm (declarative, imperative or hybrid); degree of generality (general purpose or designed for specific domains); level of abstraction.

This diversity of technologies brings along with it a series of problems.

1. On the one hand, the user ought to be able to choose the most suitable language for solving a specific problem and, depending on what that problem is, it may be a good idea to use one particular type of language or another. If we wish to change the language, we need to learn to use it. The time invested in doing all this is directly proportional to the complexity of the language, which means that the process of implementation of a model is really work-intensive.
2. In general, the transformation tools support a specific transformation language and there are therefore interoperability problems between them.

Taking that into account, it is considered advisable to look for solutions which will make it easier to learn about transformation languages and to use them too. Similarly, improving the interoperability of support tools, in such a way as to make migration between one language to another easier, also seems wise. Given the fact that we are in an MDE context, it seems logical to try to take advantage of the positive aspects that model-driven engineering itself provides us with, applying MDE to the process of defining transformations. To do that, we will specify a process in which the transformations are defined in a platform-independent language (at PIM level), with transformations which allow the (semi-) automatic generation of the corresponding transformations in different specific languages (at PSM level).

There is no single proposal for the specification of languages in the MDA field that is similar to MOF (*Meta Object Facility*, [149]) and which would allow us to unify the existing transformation languages. Bearing that in mind, the proposal for a development environment which is put forward in this thesis will allow us to define model transformations to a high level of abstraction, without taking into account the language of the final implementation of the transformation. The aim in all this is to solve both of the problems set out in the previous paragraphs.

1. The provision of a high level transformation, which is platform-independent and closer to the user.
2. The generation of (semi-) automatic transformations in specific platform-dependent languages, which will facilitate the development task of the transformation programmer and make it easier for there to be interoperability and migration between tools and languages.

The environment for the model-driven development of model transformations (MeTAGeM) should include:

- The definition of a methodological process which defines that environment.
- The specification of a high-level transformation meta-model which will allow us to model transformations at the PIM level.
- The specification of meta-models which conform to the different transformation approaches (declarative, imperative, hybrid, graph-based, etc.), which will enable us to model transformations at PSM level.
- The specification of a transformation meta-model which includes: the transformations between the PIM meta-model and the different PSM meta models, as well as the transformations between the PSM meta-models and the PDM meta-models.
- The implementation of a tool which supports: a) the modeling of PIM-level transformations on the basis of the meta-model proposed; b) the modeling of PSM-level transformations on the basis of the metamodels, conforms to each one of the existing approaches; c) the modeling of the transformations at PDM level, on the basis of the meta-models of each one of the transformation languages which is supported; d) a meta-transformer which allows us to obtain the transformation models in line with a specific transformation language, and taking these latter as a starting point, it enables us to get the implementable transformation code in each of the languages.

In this thesis, we focus on the PSM level, in the hybrid approach. It is for that reason that at the PDM level the ATL and RubyTL languages, which follow that approach, are chosen.

A.2 Hypothesis and Objectives

The **hypothesis** posed in this doctoral thesis is that " *it is feasible to define an environment which, by applying MDE principles, facilitates the (semi-)automatic development of models transformations, by means of the specification*

of these in a high-level platform-independent language and a subsequent transformation into a platform-specific language".

The **main objective** of this research work, derived directly from the hypothesis, is: *"the definition of an environment which, by applying MDE principles, facilitates the (semi-)automatic development of models transformations, by means of the specification of these in a high-level platform-independent language and a subsequent transformation into a platform-specific language. This environment should include a methodological guide and a support tool"*.

To reach the above objective, the following partial objectives have been posed:

O1. Study of previous work:

- **O1.1.** Analysis and evaluation of work related to the development of models transformation in the MDE field, including methodological proposals and tools.
- **O1.2.** Analysis and evaluation of model-transformation languages and tools.

O2. Specification of the meta-models at different levels.

- **O2.1.** Specification of a meta-model to define transformation models at PIM level.
- **O2.2.** Specification of meta-models to define transformation models at PSM level. At PSM level the modeling of transformations will be realized with due regard for the approach chosen by the developer in question; namely: imperative, declarative, hybrid, graph, etc. For the purposes of this particular thesis, the hybrid approach is the one chosen.
- **O2.3.** Specification of the meta-models for the definition of transformation models at PDM level. In this thesis, ATL and RubyTL are chosen as hybrid languages at PDM level. In the case of ATL, the use of the meta-model defined by the language itself is proposed. As far as RubyTL is concerned, as it has no defined meta-model, this must be specified.

O3. Specification of meta-models which allow us to automatize the model transformations:

- **O3.1.** Specification of the transformations from transformation models at PIM-level to PSM-level models.

O3.2. Specification of the transformations from transformation models at PSM-level to models at PDM-level, i.e., models dependent on a given transformation language: ATL and RubyTL:

O4. Building of the meta-tool:

O4.1. Specification of the architecture of the supporting meta-tool.

O4.2. Implementation of the meta-models defined (PIM, PSM and PDM)

O4.3. Implementation of the meta-editor which allows us to model high-level transformations on a graph.

O4.4. Implementation of the meta-transformer, which should include:

- The set of rules for transformation from models defined at PIM level to models defined at PSM level.
- The set of rules for transformation from models defined at PSM level to models defined at PDM level.
- The set of rules for transformation from model to text, which will allow us to obtain the code for a specific language (ATL and RubyTL)

O4.5. Integration of the functionality provided as a result of the tasks outlined previously. To that end, new modules will be defined, to provide a visual interface that makes it easier to execute the transformations.

O5. Validation of the environment of the proposed development.

O5.1. To achieve this, a meta-case study will be carried out, to permit us to validate both the methodological proposal and the proper working of the tool

O5.2. Moreover, a case study will be performed, consisting of testing the model transformations generated in the meta-case study.

A.3 Research Method

The different nature inherent to the various types of Engineering, when compared with the rest of the empirical and formal disciplines, makes it difficult to apply classical research methods [109], especially where Software Engineering research is concerned.

The research method followed in this thesis is an adaptation of the proposal by Marcos and Marcos in [109] for Software Engineering Research. This model is based on the hypothetical-deductive method proposed by Bunge [76] and is made up of a series of steps (see Figure A-1) which are general enough to be applied to any type of research.

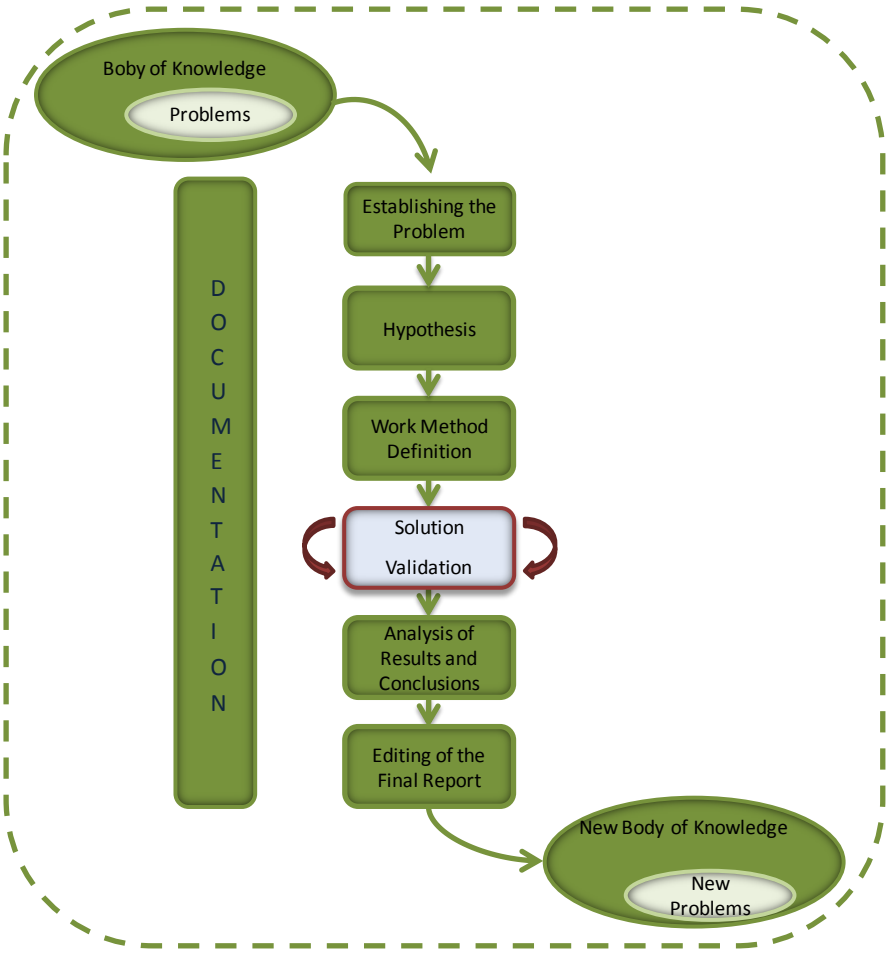

Figura A-1. Research Method

As can be seen in Figure A-1, the definition of the research method is just one more step in this same method. The authors [134] recommend defining it in this way since each piece of research is individual in nature, with its own characteristics, and it would therefore be unwise to apply one single universal research method.

We will now go on to summarize the specific method used in this thesis for the stages of documentation, solution and validation.

A.3.1 Documentation Stage

In this stage, the method of systematic review proposed in [111] has been used to carry out its purposes. This method identifies, evaluates and interprets all the information related to a particular research topic in a way that is systematic and which can be replicated. The method has come into being as a derivative of research in the field of medicine; as such it has, according to its authors [111], become a trustworthy, rigorous and auditable methodology.

The application of the systematic reviews in the field of Software Engineering enables the review of literature that is carried out to become endowed with scientific value, defining a search strategy for the literature that is to be assessed and obtaining as an end result a hypothesis that is in favor of, or against, the literature in question. To conduct this research work, the adaptation of the method for systematic reviews as related to Software Engineering, presented in [40], has been taken as a reference point. In this document a new approach is put forward, in which the process of systematic reviews is made up of four major stages: *planning, execution, analysis of the results obtained and results storing* (Figure A-2).

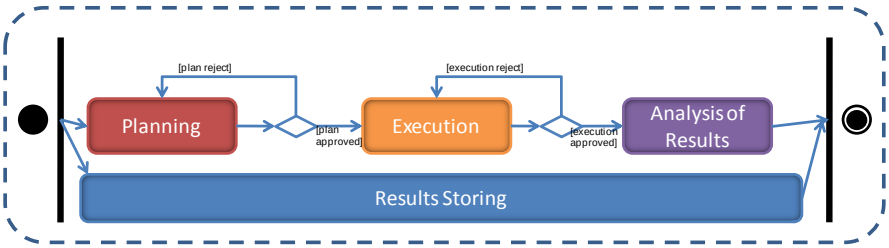

Figura A-2. Process for the System Review Method

In the **planning** phase the objective of the research should be set out clearly and the review protocol to be used needs to be specified. In other words, this is all about defining a protocol for each object to be researched, establishing which method will be used throughout the review. The criteria of inclusion and exclusion to be followed in determining the research sources and the studies (or documents) to be chosen ought also to be identified.

In the **execution** phase all that has been planned in the previous stage is carried out, so that first of all the set of studies to be evaluated should be decided

on. These studies are chosen by evaluating the criteria of inclusion and exclusion that had been determined previously with respect to each one of them.

In the **results analysis** phase, the information extracted from each study should be synthesized and assessed.

Lastly, it needs to be mentioned that the **results storing** phase is carried out during the whole process of the systematic review, since, while each phase is being conducted, the result of these should be stored.

As can be seen in Figure A-2, there are verification points throughout the whole process. The first point guarantees that the planning carried out is appropriate; in that sense the protocol defined ought to be assessed and, if any problem or inconsistency should appear, there would be a need to go back to the previous phase, i.e. planning. The second verification point should be put into play once the execution phase has been finished; as in the case of the previous point , should there be any mistake in the results in this stage, the execution needs to be realized once more.

In this thesis, two processes of systematic reviews which have contributed to the state of the art in this area have been performed: one to analyze the different pieces of work which propose the use of MDE in the development of model transformations; the other to analyze the different transformation languages and the tools which implement them in the MDE field.

As far as the analysis of the work that proposes the use of MDE in the development of model transformation is concerned, we may say that:

- The greater part of the work assessed amounts only to mere intentions. With the exception of the work presented by Didonet Del Fabro [79], none of the pieces of work validates the proposal that had been carried out using tools to implement it, or success cases which corroborate it. What is more, we should mention that the majority of the research lines assessed seem to have been abandoned by the authors, or to have been absorbed by other lines of research.
- As we see it, the pieces of work that seem to come closest to getting it right, from the point of view of proposing a method that supports meta-transformations, or, in other words, of allowing high-level transformations, are those presented by Bezivin in [39] and Didonet Del Fabro in [79]. We say this because both works are based on the idea of applying MDE to the transformations themselves, and in the case of the latter work, a tool is proposed which lets us validate the approach.

- One of the main foundational points of MDE is the automatization of the tasks in any proposal for the model-driven software development [21, 135]. Consequently, as has been evident throughout this state of the art, a series of tools has come about for the purpose of automatizing these tasks related to MDE. The greatest impact has been in the emergence of tools for defining and/or using new modeling languages. At the same time, transformations between models are one of the main operations of MDE. This has in turn led to the appearance of a variety of tools or languages which allow us to implement these model transformations. If we analyze the different languages of transformation from model to model, we may observe that the majority work properly, we should bear in mind that the success cases have been put into operation by the language developers themselves. When the languages are put into operation by an outside user, however, he/she finds himself/herself faced with two dilemmas: the first is choosing which language to use and the second is how to learn this language, once it has been selected.
- With all the above in mind and following the research lines proposed in [39] and [79], it has been deemed advisable to develop an environment which facilitates the (semi-) automatic development of model transformations by means of their specification in a high-level platform-independent language and a subsequent transformation of these into a platform-specific language. The environment will be made up of: a methodology, which defines the way in which the transformations should be carried out, along with a tool (MeTAGeM), which validates this methodology.

As regards the analysis of the transformation languages and the tools implementing them, it can be affirmed that:

- As said before, to support the development of transformations in the MDE field, different types of approach have emerged [65, 66]. In this thesis the **hybrid** approach is chosen for the modeling of the platform-specific transformations. On the basis of that selection, it is the transformation languages that follow that approach which have been analyzed.
- The analysis of the languages has been performed with the aim of establishing the set of constructors common to all of them that needs to be supported by the PSM meta-model of MeTAGeM.
- The languages analyzed support the definition of different types of rules and auxiliary functions, which makes it easier to codify the transformations. With respect to the cardinality of the elements in a transformation rule, only the ATL language allows us to establish rules between the multiple elements of

the source model and the multiple elements of the target model. The other languages permit us to generate multiple elements in the target model, but to do so they start from one single element of the source model.

- Although it is the case that all the languages have a meta-model which defines them specifically, only the ATL meta-model is actually implemented, which means that it is only in ATL that the model transformations can be defined as models in themselves. Another point in favor of ATL is the fact that it is endowed with mechanisms of code extraction and injection.

A.3.2 Resolution and Validation Stage

The method of resolution and validation followed in this thesis is the adaptation of the methods that are well-known in the field of Software Engineering as proposed in [191]: the traditional cascade method [171] and the *Rational* Unified Process [43], taking as our basis the definition of consecutive stages of the former and the iterative process of the second. The choice of these methods is based on the similarity that exists between the nature of the problem to be solved and the problems that arise in software development. There are certain problematic issues in Software Engineering research (as we comment in this thesis). They are engineering-based problems, since they have to do with the building of new objects [133]; the case we have before us here deals with the building of an environment for the (semi-) automatic development of transformations. That environment will be made up of a methodology and a tool. A software development methodology gives us the guidelines for the building of new objects (of software). That being so, software development methods can serve as a basis for the solving of engineering-based research problems in Software Engineering [133]

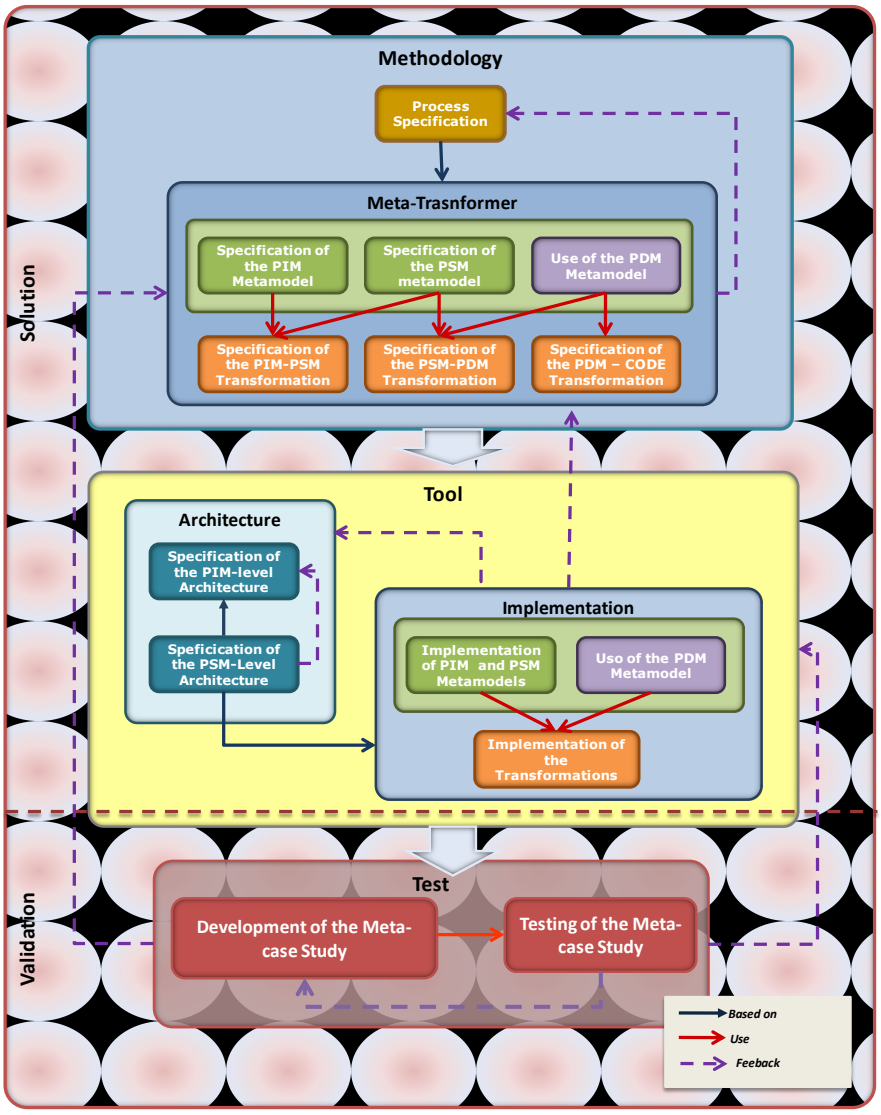

Figura A-3. Solution and Validation of the Research Method

As Figure A-3 shows, the different activities performed in the **solution** and **validation** phase are grouped together into big blocks: The block which corresponds to the specification of the **methodology** and that corresponding to the development of the **tool.**

Within the block that has to do with methodology, the first activity that is conducted is that of **specification of the process** (MeTAGeM), to build

transformation models to a high abstraction level. The definition of this process is done with the following aspects being taken into account: a) the results obtained from the studies of the already-existing approaches; b) our own experience in the definition of model transformations in different domains[42, 71, 72, 191, 193, 194, 196, 197, 198].

Once the process specification has been completed, the specification of the **meta-transformer** is carried out. This meta-transformer will be made up of two components: on the one hand, a **set of meta-models** which will allow us to conduct the modeling of the model transformations at the different levels proposed. On the other hand, there is the **set of transformation rules** which will let us realize the transformation between the models in accordance with the meta-models mentioned above.

Depending on the process defined, the meta-models corresponding to each one of the levels of MeTAGeM (PIM, PSM and PDM) should be specified. At the PIM level, a meta-model with a high abstraction level will be specified. At the PSM level, a meta-model from the hybrid approach will be specified. Finally, meta-models of each one of the existing transformation languages will be used at the PDM level; in this thesis the meta-model of the transformation language ATL will be employed and the meta-model of the RubyTL transformation language will be specified. Taking the PIM and PSM meta-models as a starting point, editors will be produced and these will make it possible to handle models which are in accordance with those meta-models. Similarly, to handle models that are in consonance with the PDM-level meta-models, the editors provided by the respective languages will be used. In the particular case of the RubyTL language, which does not provide any editor, there will be an implementation of an editor that is based on the previously-specified RubyTL meta-model.

Taking the definition of the meta-models as a foundation, the specification of the transformations between these meta-models is carried out. As a result of this sub-phase we therefore obtain: a) the transformations between models defined at PIM level and those defined at PSM level; b) the transformations between models defined at PSM level and those defined at PDM level and, lastly, c) the transformations from model to text, which make it possible to obtain the implementable code of the transformation, based on the models defined at PDM level.

In the block that has to do with the tool we find, on the one hand, the specification and implementation of the **architecture** that supports the process

proposed and, on the other hand, we have the **building** of the different meta-models, together with the transformation rules set out in the methodology block.

As we are in the MDE field, the most logical thing to do would seem to be to apply MDE in the definition and building in each of the elements defined in the methodological process. Taking into consideration too, the lessons learnt in the building of M2DAT [191], it was decided to realize the specification of the **architecture** by separating two different aspects- one being the definition at the conceptual level (PIM level) and the other being the technical details of their implementation at a low level (PSM level).

In the definition of the architecture at PIM level, the modules or elements that will make up the meta-tool are specified. Subsequently, in the definition of the architecture at PSM level, the particular technologies that will be used to implement each one of these modules or components will be determined. As well as all this, a feedback point is established, allowing modifications in the definition of the architecture at PIM level to be made.

In the **building** activity, the implementation of the MeTAGeM tool is carried out, in accordance with the architecture design, the specification of the meta-models and the transformation rules that were realized in the previous activities.

Lastly, in the **testing** phase, a meta-case study will be developed, allowing us to validate the working of the development environment, in other words, both the methodology defined and the tool that implements it. After that, the transformations obtained from the meta-case study will be validated at a later point with models which are in accordance with the meta-models on which the transformation has been performed.

It should be commented that, as the process defined is iterative, each one of the phases becomes a control point for the phases prior to it, so that it is possible to go back to them if that is deemed necessary.

A.4 Solution: MeTAGeM. An Environment for the Development of High-level Model Transformations

The environment for the model-driven development of transformations model (MeTAGeM), which is described in this thesis is made up of:

- A **methodology**, which contains
    - A process for the model-driven development of transformations models.

  - Different meta-models which support the process defined, allowing us to modeling of model transformations at the different levels proposed in the process.
  - The specification of a set of transformation rules which will make it possible to obtain (semi-) automatically the models conforms to the different meta-models proposed at each level.
- A **tool** which gives support to the methodology proposed, made up of:
  - An architecture which defines each of the components that the tool is composed of.
  - A set of editors that allows the modeling of model transformations in line with the meta-models described by the methodology.
  - A meta-transformer which permits us to realize the transformations between the models defined at the different levels of abstraction and the subsequent code-generation. The meta-transformer is made up of different sets of transformation rules.

A.4.1 Methodology

The process defined as part of the methodology of MeTAGeM is carried out in accordance with four abstraction levels: PIM, PSM, PDM and code. Figure A-4 represents the process in the form of a SPEM2 activity diagram [154].

Firstly, we propose beginning by modeling the model transformations to a high abstraction level (models at PIM level), without taking into account the technical issues of the final implementation, nor the technology used to do so. To do that, we indicate the existing relationships between the different elements of the meta-models that form part of the transformation.

From this model derived from the PIM level we generate, by means of a (semi-)automatic transformation, a model transformation specific to the platform (PSM level). This PSM level model can be refined manually by adding new elements, modifying characteristics of existing elements, o eliminating elements that are considered unnecessary.

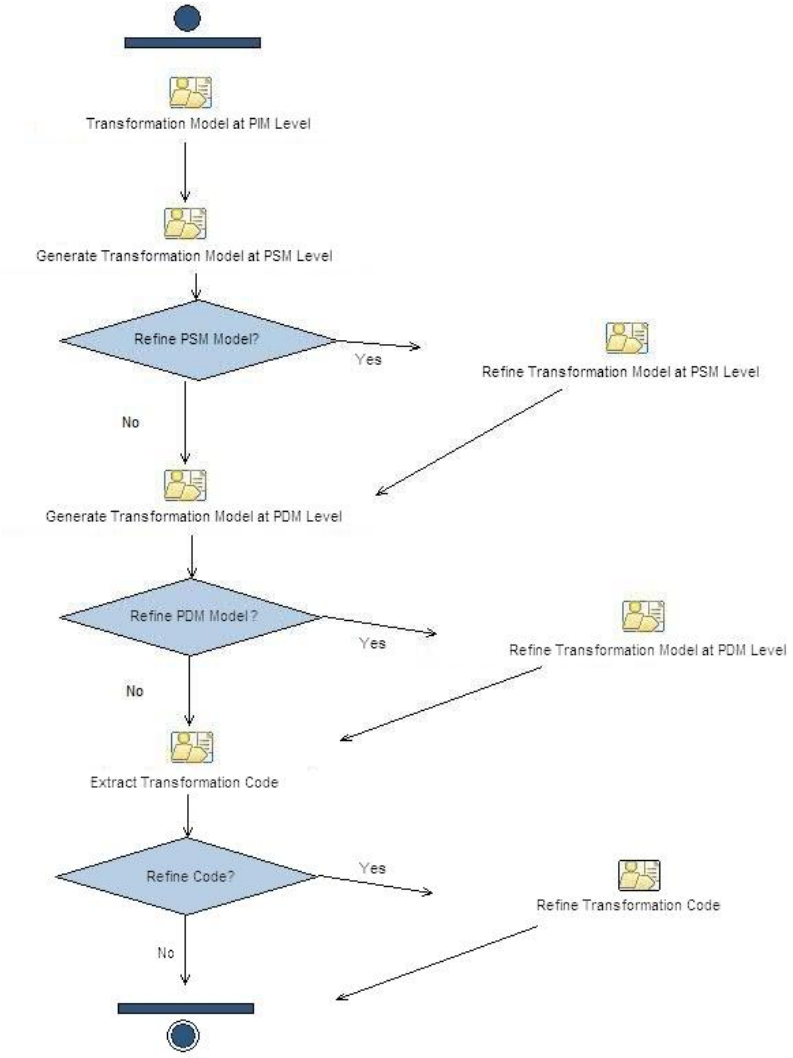

Figura A-4. MeTAGeM Process

In a similar manner, using the model obtained at PSM level and again applying a series of (semi-) automatic transformations, a model transformation is generated that is dependent on the platform (at PDM level). These models, that can also be refined, will be in accordance with the meta-model of the transformation language chosen as the final platform for implementing the transformations.

The final step is the generation of the transformation code using the models generated at PDM level. This generation will be (semi-) automatic, using transformations from models to text.

The final result of the process is to obtain the transformation code in the language that has been chosen.

In order to achieve the modeling of the transformations models at different abstraction levels one must define meta-models that allow specific models conforms to those meta-models.

Figure A-5 shows the relationship that exists between the abstraction levels defined in the process, the different meta-models needed to allow modeling of the transformations in these levels, as well as the transformation models conforms to those meta-models..

Figura A-5. MeTAGeM Metamodels

As can be observed, at **PIM level** a meta-model is defined, which conforms to MOF and which enables modeling of the model transformations to a high abstraction level, without taking into account the technical details of the final implementation of the transformations. This meta-model is formed by a group of constructors that allows the modeling of existing relationships between the original meta-models (Meta-Model In) and the final meta-models (Meta-Model Out), also defined conforms to MOF. So, when we specify the model transformations, we must define a model of model transformation in which only the existing relationships between the elements of the Meta-Model-In and Meta-Model-Out are modeled, taking into account the cardinality of the elements involved in each relationship.

At **PSM level**, different meta-models are specified, one for each of the approaches (or paradigms) of the existing model transformations. In the particular case of this thesis, the hybrid approach meta-model is specified, in the same way meta-models are specified for the rest of the approaches.

At **PDM level**, the model transformations are represented as platform-dependent. These models will be compliant with the meta-models of each one of the transformation languages that the developer might wish to use for the final transformation. For this thesis the transformation languages chosen at PDM level are: ATL and RubyTL. Thus, at PDM level we obtain a model transformation represented as an ATL model, conform to ATL meta-model; or as a RubyTL model, conform to RubyTL meta-model.

Lastly, **at code level**, using the models defined at PDM level, we obtain the code in the specific model transformation language. In this case we obtain the code in ALT or in RubyTL.

Finally, in order to allow transformations between models defined in the different MeTAGeM levels, different sets of rules are defined to enable that transformation: a) the transformation set between models at PIM level and PSM level; b) two sets of transformations between PSM level models and those of PDM, one for ATL language and the other for RubyTL; and c) the set of model transformations to text format that enables the code that implements the transformation in the selected language to be obtained.

A.4.2 Tool

To support the methodology defined we propose the development of a tool, MeTAGeM that is made up of a collection of DSLs that will enable the modelling

of model transformation to a high abstraction level, as well as transformations between models at different levels and the subsequent code production.

In order to define the framework of the MeTAGeM we take as the base [191] where it is proposed that the different levels of abstraction are taken into account, separating on the one hand the conceptual design (PIM level) where each one of the elements that make up the tool is defined without considering the technical questions for carrying them out; and on the other hand, the technical design (PSM level) that is obtained by refining the conceptual framework, specifying for each one of the elements defined at PIM level the tools used for its implementation. Moreover, it is proposed that the process of construction should be iterative and incremental, allowing the updating of the functionality of the tool and the continuous incorporation of improvements as well as enabling new approaches and /or transformation languages to be supported in a relatively easy manner.

A.5 Conclusions

In section A.2 a series of partial objectives were presented, the fulfilment of which imply reaching the main objective of the thesis: definition of a methodological proposal that, by applying the principles of MDE, would facilitate the (semi-)automatic development of models transformations by means of specifying these in a platform-specific language. This environment should include a methodological guide and a support tool.

The following is an analysis of the realization of each one of the partial objectives:

O1. Study of Previous Work

The study of previus work was carried out by means of the analysis and assessment of, on the one hand, work related to the development of model transformations in the area of MDE, and on the other hand, the model languages and tools that follow the hybrid approach.

To analyze the existing work on the development of model transformations in the area of MDE, a set of characteristics to be evaluated in each one was determined, aiming to facilitate the work and unify the criteria to be kept in mind during that analysis.

The set of works to be assessed was selected by means of a process of systematic revision applied to the main search sources in the area of Software Engineering.

The main conclusions arrived at are:

- Whilst the majority of approaches evaluated recognize the need to apply MDE principles to the transformations development and even make suggestions as to how to bring them about, very few use a tool that supports the definition of the transformation by applying MDE principles. In the final stage of writing this thesis, two such proposals were found [93, 126] along the lines of those presented in this thesis; this fact in itself helps to justify the importance of applying MDE to the development of transformations.
- Following the lines of investigation suggested in [39] and [79], this thesis proposes a development environment that facilitates the (semi-)automatic modeling of model transformations by means of specifying these in a high level platform-independent language and then by their subsequent transformation into a platform-specific language.

Before commencing the analysis of the model transformation languages and the existing tools, it was considered advisable to select the approach for model transformations that was to be followed for the modeling of the transformations at the PDM level of MeTAGeM. This was done by analyzing the existing approaches, selecting a hybrid approach and then justifying this selection. Following that, an analysis and evaluation of the model transformation languages and tools that follow the hybrid approach was carried out. To make the work easier and more efficient and to unify the criteria to be taken into account during the analysis of the different languages and tools, a series of characteristics to be evaluated was established for each one of them.

The main conclusions arrived at are the following:

- The languages analyzed support the definition of different types of rules and auxiliary functions, which in turn facilitates the codifying of the transformations. With regards to the cardinality of the elements in a transformation rule, only the ATL language allows the definition of rules between multiple elements of the original model and of the destination model.
- Whilst all the languages have a specified defining meta-model, it is only the ATL meta-model that is implemented, meaning that only in ATL can we define the transformation of models as models in their own right. Another point in favor of ATL is being able to count on mechanisms for the extraction and injection of code.

As a result of these conclusions, a series of features that must be taken into consideration when specifying the meta-model that follows the hybrid approach used at PDM level.

O2. Specification of the meta-models of the different levels

The principal premise in the development of this thesis is that the transformation development enviroment, MeTAGeM, should be sufficiently general as to be able to be applied to transformations of any domain, regardless of the final transformation language used. As has been seen, MeTAGeM proposes the modeling of the transformations at different levels of abstraction (PIM, PSM, PDM). For each of these levels it was necessary to define a corresponding meta-model.

To bring about the specification of the PIM level meta-model, it was thought best to concentrate on identifying the types of relationship existing between the different meta-models that in overall terms form a part of a model transformation, and not on the specific constructor used by the meta-models of the different transformation languages.

Having identified the types of relationship and the types of elements to be transformed, we proceed to specifying the meta-model for the PIM level model transformations.

Subsequently, a meta-model has been specified that enables us to implement model transformations at PSM level. As has been mentioned previously, the first approach to be used in this thesis has been the hybrid approach. The PSM-level meta-model reflects the constructors that are common to the transformation languages that support such an approach.

At the PDM level, MeTAGeM proposes the modeling of the transformations according to two transformation languages: ATL and RubyTL. For modeling transformations using ATL it is proposed that the language's own editor be used, and for modeling transformations using RubyTL an editor had to be implemented. To that end, a meta-model was specified, conform to RubyTL language.

O3. Specification of meta-models that enable the automating of model transformation.

A meta-transformer has been specified that is made up of: a set of transformations that allow the mapping between PIM model transformations to PSM model transformations; a set of transformations that allow the mapping between PSM model transformations to PDM model transformations. In this last case, two different transformation sets have been used, one for each transformation language, ATL and RubyTL.

To produce the transformation specifications, it was firstly necessary to carry out an analysis of the meta-models involved in the transformation; subsequently, the transformation sets between each of the elements of the meta-model were defined, both in natural language and in graph grammar.

O4. Construction of the meta-tool

A series of activities was necessary in order to construct the MeTAGeM meta-tool:

First of all, the architecture of the MeTAGeM was specified. To carry out this specification, the guidelines laid out in [191] were followed, where it is proposed that two levels of abstraction be considered, separating the conceptual design (PIM level) from the technical design (PSM level).

In the framework at PIM level each one of the elements that forms a part of the meta-tool is defined, without considering the technical issues concerning its implementation. The architecture at PSM-level is arrived at by means of refining the conceptual framework, specifying for each one of the elements defined at PIM-level the technology used for its implementation.

The architecture of MeTAGeM is structured according to two orthogonal dimensions:

- On the one hand, MeTAGeM can be considered as a collection of modules, one for each level, in which transformation models can be specified. Each module is encapsulated in a DSL that includes the set of concepts related to that module, as well as the rules of transformation that enable the mapping of a model at one level to another model at another level.
- At the same time, the architecture is defined according to the separation of concerns principle [122, 167] where each layer represent a particular view, defined as the presentation layer, the logical layer and the persistence layer. This separation of layers allows us to maintain traceability between the different artifacts, the reuse of these elements and a greater control of their evolution when incorporating new functions.

Secondly, a process followed for the building of the meta-tool is demonstrated. Starting with the specification of the framework at logical level, we go on to implement each one of the modules that makes up MeTAGeM: implementing each one of the meta-models; developing editors allow modeling of the transformations conform to the meta-models; personalizing those editors; implementing the mappings between the models of the different levels of

MeTAGeM; and specifying the set of restrictions that allow us to realize our validation of the models in each case.

O5. Validation of proposed development environment

The validation of the environment takes place in two ways:

- On the one hand, the functioning of the MeTAGeM meta-tool is validated by developing a meta-case study that consists in the implementing transformation rules that allow the map to the conceptual data model, represented in a UML class diagram, to a logical-specific model represented by means of an object-relational model, so as to develop Object-Relational databases.
- And, on the other hand, the functionality of those transformation rules is validated by carrying out a specific case study.

A.5.1 Main Contributions

One of the results of this thesis is the number of contributions that it has given rise to. We will give a description of these in the following lines:

An analysis of the existing proposals on the use of the principles of MDE in the area of model transformations.

An analysis was performed of the existing work which proposes the use of MDE principles in the model transformations development. In this analysis the most relevant work is assessed, along with the different tools that implement them.

To unify the assessment of the proposals, a set of characteristics to be analyzed has been established, which has allowed us to make a comparison of the proposals. In addition, these features can be applied in future work to extend the analysis done on the proposals.

An analysis of the different model transformation languages currently avaible.

To determine the set of constructors that should be contemplated in the meta-model at PSM-level that implements the hybrid approach, an analysis is performed of the different transformation languages that follow that approach, as well as the tools that implement those languages.

From the experience adquired in the implementation of transformations in the different fields, a set of characteristics to evaluate in each one of the languages and tools has been established. These features have enabled us to detect the

building elements that are common in the transformation languages being assessed.

Specification of a methodology for the model-driven development of model transformation.

One of the main contributions of this thesis is the specification of the methodology, as part of the MeTAGeM environment, for model-driven development of model transformation. That methodology is made up of:

- A model-driven **development process** of model transformation
- **The different meta-models** which support the process defined, allowing us to model transformations at the different levels proposed in the process.
- The specification of a **set of transformation rules** which make it possible to generate the models proposed at each level (semi-) automatically.

Development of a tool that supports the methodology designed.

Another of the main contributions of this thesis is the development of the **tool,** which supports the methodology proposed. This tool is composed of:

- The **architecture** that defines each one of the components that it is comprised of, as well as the relationships between them.
- Graphic **editors,** which are tree-liked, for the modeling of the transformations in the different levels.
- A **meta-transformer,** which implements the different sets of transformation rules that allow us to generate the models at different abstraction levels: transformation of PIM to PSM, of PSM to PDM (ATL and RubyTL) and finally, the code extractors which make it possible to obtain the code which implements the transformation in the language desired.

The main feature of MeTAGeM is that it is extensible and interoperable. Its extensible nature makes it easier to incorporate new functionalities, as for example, the support of a new approach to the PSM level or of a new transformation language.

These new functionalities will be supported through the creation of new modules in MeTAGeM, which will be directly compatible with the existing modules with no extra effort involved; i.e. with no need to build bridges between the different technological spaces, since the same technology and the same process are used to develop them.

Development of Case Studies.

To validate MeTAGeM, a meta-case study has been carried out, implementing the transformation rules, which means that it is possible to pass from a conceptual model of data, represented by a UML class diagram, to a specific logical model, represented by an object-relational model, creating an Object-Relational data base. These transformations were later tested, by applying them to the transformation between specific models.

In addition, while the tool was being implemented, a series of case studies was carried out, aiming to test given functionalities of the tool.

A.5.2 Scientific Results

Some of the results obtained while this thesis was being produced have been published in various congresses, both at a national and international level. We set out the different publications below, classified according to the type of publication. Given the nature of the work of this Doctoral Thesis, it has been necessary to wait until the implementation of the tool was finished before work could be published in significant and high-impact forums. That is why, at the time of writing the abstract of the thesis, only publications of low-impact, which contain partial results of the work, can be cited. That said, work is currently being done towards publication in forums with a greater degree of impact.

- **Articles in International Conferences**
  - Vara, J.M., Bollati, V.A., Irrazabal, E. y Marcos, E. (2010). *Using EMF and ATL to improve types management in MDE proposals*. 2nd International Workshop in Model Transformation with ATL (MtATL 2010), Málaga, España. (30 June, 2010).
  - Vara, J.M., Vela, B., Bollati, V. A. y Marcos, E. (2009). *Supporting Model-Driven Development of Object-Relational Database Schemas: a Case Study*. **ICMT2009 - International Conference on Model Transformation**, Zurich, Switzerland. (29 and 30 June, 2009) **(Acceptance Ratio: 22%)**
  - Vara, J.M., Bollati, V., Vela, B. & Marcos, E. (2009). *Leveraging Model Transformations by means of Annotation Models*. 1st International Workshop in Model Transformation with ATL (MtATL 2009), Nantes, France. (8 and 9 July, 2009).

- **Articles in Latin American Conferences**
  - Bollati, V.A., Vara, J.M, Vela, Belén & Marcos, E. *Uso de Modelos de Anotación para automatizar el Desarrollo Dirigido por Modelos de Esquemas XML*. XII Conferencia Iberoamericana de Ingeniería de Requisitos y Ambientes de Software. (IDEAS'09), Medellín (Colombia). (13 to 17 April , 2009) **(Acceptance Ratio: 46%)**
  - Bollati, V.A., Vela, B., Vara, J.M. & Marcos, E. *Una Aproximación Dirigida por Modelos para el Desarrollo de Bases de Datos Objeto-Relacionales*. XIV Congreso Argentino de Ciencias de la Computación. (CACIC 2008). Chilecito (La Rioja, Argentina). (6 to 10 October, 2008).
  - Bollati, V.A., Marcos, E., Vara, J.M. &. Vela, B. *Analisis de Herramientas MDA*. XIII Congreso Argentino de Ciencias de la Computación. (CACIC 2007). Corrientes y Resistencia, Argentina. (1 to 5 October, 2007)
- **Articles in National Conferences**
  - Jiménez, A., Vara, J. M., Bollati, V. A., Marcos, E. *Mejorando el nivel de automatización en el desarrollo dirigido por modelos de editores gráficos***.** VII Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'10. Valencia, España (7 September, 2010).
  - Irrazabal, E. Vara, J. M., Bollati, V. A. y Marcos, E. *Gestion de Tipos Primitivos en Propuestas de DSDM: Aplicación al Modelado de Esquemas de Bases de Datos Objeto-Relacionales*. VI Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'09. San Sebastián, España (8 September, 2009).
  - Bollati, V. A., Victor Sánchez, Vela, B. y Marcos, E. *Análisis de QVT Operacional Mappings: un caso de estudio*. VI Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'09. San Sebastián, España (8 September, 2009).
  - Bollati, V.A., Vara, J.M., Vela, B. & Marcos, E. *Una Aproximación Dirigida por Modelos para el Desarrollo de Esquemas XML*. **XIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'08)**. Gijón, España. (7 to 10 October, 2008). **(Acceptance Ratio: 25%)**
  - Vara, J.M., Bollati, V., Vela, B. & Marcos, E. *Uso de Modelos de Anotación para automatizar el Desarrollo Dirigido por Modelos de Bases de Datos Objeto-Relacionales*. V Taller sobre Desarrollo de

Software Dirigido por Modelos. DSDM'08. Gijón, España (7 October, 2008).

- Bollati, V. A., Vara, J.M., Vela, B. y Marcos, E. *Una Revisión de Herramientas MDA*. IV Taller sobre Desarrollo de Software Dirigido por Modelos. DSDM'07. Zaragoza, España (11 September, 2007).

- **Patents**
  - Título: *M2DAT/DB: Herramienta para el Desarrollo Dirigido por Modelos de BD*.
    - Inventors: E. Marcos, B. Vela, J.M. Vara, V. A. Bollati
    - Application Number : M-8452/2008
    - Country: Spain
    - Date: 24/10/2008
    - Entity: Universidad Rey Juan Carlos
    - Participating Countries : Spain

Apéndice B: Revisiones Sistemáticas

Como se ha visto en el capítulo 1 para llevar a cabo la etapa de documentación del método de investigación que se usa en esta tesis se utiliza una adaptación del método de revisiones sistemáticas propuesto en [111]. Este método sirve para identificar, evaluar e interpretar toda la información relativa a un tema de investigación en particular, de un modo sistemático y reproducible. Surge de la investigación en el campo de la medicina, por lo que, según sus autores [111], se ha convertido en una metodología confiable, rigurosa y auditable. Para el desarrollo de este trabajo de investigación, se ha tomado como referencia la adaptación del método de revisiones sistemáticas para Ingeniería de Software presentado en [40].

A continuación se detallará brevemente las características del método, se presentarán las plantillas utilizadas para llevar a cabo la revisión sistemática y se mostrarán los resultados de aplicar las revisiones sistemáticas a la etapa de documentación en esta tesis.

B.1 ¿Qué es una Revisión Sistemática?

El término Revisión Sistemática (RS) se usa para referirse a un método específico de investigación, desarrollado con el fin de recopilar y evaluar los datos disponibles relativos a un tema en particular.

A diferencia de la revisión literaria tradicional, una RS sigue una secuencia estricta y bien definida de pasos metodológicos, de acuerdo con un protocolo elaborado a priori, que garantiza el alto valor científico de los resultados obtenidos. Este proceso se construye con un tema principal, que representa el núcleo de la investigación, y que se expresa mediante conceptos y términos específicos, que deben ser abordados respecto a la información relacionada con el tema en cuestión. Cada uno de los pasos metodológicos del protocolo, las estrategias para recuperar las evidencias del tema y el enfoque con el que se aborda el tema, se definen explícitamente, de manera que otros profesionales puedan reproducir el mismo protocolo y ser capaces de obtener el mismo resultado.

La principal razón para llevar a cabo una RS es incrementar la probabilidad de detectar más resultados reales en el área de interés que los obtenidos con una revisión menos formal. Una RS requiere un esfuerzo considerablemente mayor en

comparación con una revisión tradicional, pero, a cambio, se obtiene una revisión profunda y completa de un determinada área de interés.

B.1.1 Proceso de Revisión Sistemática

El concepto de RS apareció en el área de la medicina, y su adaptación a la Ingeniería del Software se presenta en [111]. La aplicación de la RS al ámbito de la Ingeniería de Software permite dar un valor científico a la revisión de la literatura que se hace, definir una estrategia de búsqueda de la literatura a evaluar y obtener finalmente una hipótesis a favor o en contra de dicha literatura. Para el desarrollo de este trabajo de investigación, se ha tomado como referencia la adaptación del método de RS para Ingeniería de Software presentado en [40]. En este trabajo los autores proponen una nueva aproximación, en la cual el proceso de las RSs está compuesto por cuatro grandes fases: *planificación*, *ejecución*, *análisis de los resultados* y *almacenamento de los resultados* obtenidos (Figura B– 1).

Figura B-1. Método de Revisiones Sistemáticas

En la fase de **planificación** se debe establecer claramente el objetivo de la investigación y definir el protocolo de revisión a utilizar. Es decir, definir un protocolo para cada tema a ser investigado, estableciendo el método que será utilizado a lo largo de la realización de la revisión. Además se deben identificar los criterios de inclusión y exclusión que se seguirán para determinar las fuentes de investigación y los estudios (o documentos) a seleccionar.

En la fase de **ejecución** se lleva a cabo lo planificado en la etapa anterior, por lo que en primer lugar, se debe determinar el conjunto de documentos a evaluar. Estos estudios se seleccionan a través de la evaluación, para cada uno de ellos, de los criterios de inclusión y exclusión determinados perviamente.

En la fase de **análisis de resultados**, se debe sintetizar y evaluar la información extraída de cada documento.

Por último, se debe mencionar que la fase de **almacenamiento de resultados** se realiza durante todo el proceso de la RS, ya que a medida que se ejecutan cada una de las fases, el resultado de las mismas debe ser almacenado.

Como se puede ver en la Figura B– 1, existen puntos de verificación a lo largo del proceso, representado por medio de rombos entre cada una de las fases. Estos puntos de verificación permiten realizar un análisis de cada una de las etapas para corroborar su correcta definición o funcionamiento. El primer punto garantiza que la planificación realizada es la adecuada; para ello, se debe evaluar el protocolo definido y, si hubiera algún problema o incongruencia, se debería volver a la fase anterior, la planificación. El segundo punto de verificación debe realizarse una vez acabada la fase de ejecución; de la misma manera que en el punto anterior, si hubiera algún error en los resultados de esta etapa se debería volver a realizar la ejecución.

B.1.2 Plantilla de Revisión Sistemática

La realización de RS es una tarea compleja, ya que durante su ejecución se utilizan conceptos y términos específicos que pueden ser desconocidos para los investigadores que realizan revisiones informales. Además, requiere un esfuerzo adicional de seguimiento. El proceso completo debe ser planificado antes de la ejecución y debe ser completamente documentado.

En [40] se presenta una plantilla con las principales cuestiones a tener en cuenta durante la realización de una RS. Con el objetivo de facilitar la planificación y la ejecución del protocolo de RS en esta tesis se realiza una adaptación de dicha plantilla centrándonos en los puntos más relevantes para, desde nuestro punto de vista, los temas a revisar. La plantilla presentada en [40] esta más enfocada a realizar RS basadas en estudios empíricos, las RS realizadas en esta tesis se centran más en el análisis de documentos y de verificación de funcionamiento de herramientas, por lo que muchas de las cosas que se plantean en la plantilla original son innecesarias; así como por ejemplo, la determinación de métricas para realizar la medición de los resultados o determinar la manera de realizar el cálculo estadístico de los mismos.

Es importante mencionar que las plantillas que se presentan a continuación se definen sugiriendo las actividades que se deben tener en cuenta cuando se lleva a cabo una RS. Dependiendo del tipo de RS que se esté realizando estas actividades pueden variar.

En la Tabla B- 1 se presentan las actividades relativas a la etapa de planificación de la revisión y a la evaluación de la misma definidas en la Figura B– 1.

Tabla B-1. Plantilla Revisiones Sistemáticas – Etapa Planificación

Planificación de la Revisión	**Formulación de la Pregunta**	**Pregunta Principal**	Debe enfocar el tema sobre el que se realiza la RS, identificando claramente el objetivo principal de la misma.	
		Calidad y Amplitud	Problema	Identificar claramente cuál es el problema que motiva a la RS.
			Palabras Claves y Sinónimos	Lista de los principales términos que forman parte de la RS y que se utilizarán durante la ejecución de la RS.
			Intervención	De qué manera se llevará a cabo la RS
			Efecto	Se define cuál es el resultado que se espera obtener al finalizar la RS.
	Selección de Fuentes	**Definición de Criterios**	Define los criterios que se utilizaran para evaluar los resultados obtenidos en la RS.	
		Identificación de Fuentes	Método de Búsqueda	Describe la forma en que se realizará la búsqueda de estudios primarios (búsqueda manual, búsqueda en bases de datos, búsqueda en la web, etc)
			Cadena de Búsqueda	Definición de las diferentes cadenas de búsquedas para los motores.
			Lista de Fuentes	Es la lista de fuentes inicial, en la cual se realizará la búsqueda.
			Chequeo de Referencia	Una vez definidas las fuentes de búsquedas, éstas deben ser aprobadas por expertos en el tema que se evalúa.
	Selección de Estudios	**Definición**	Definición de Tipos de Estudios	Define el tipo de estudios que se realizará durante la RS. Por ejemplo, estudios cuantitativos, cualitativos, de observación, etc.
			Procedimiento para la selección de los estudios	Se define la manera en la que se llevará a cabo el proceso de estudio y/o selección de los diferentes resultados (Figura A-2).
Evaluación de la Planificación		Antes de ejecutar la RS se debe evaluar la planificación realizada. Una forma de realizar dicha evaluación es preguntar a los expertos para revisar el protocolo definido. Otra forma de evaluar la misma es realizar una prueba de ejecución del protocolo, esto es, realizar la RS con un conjunto reducido de fuentes seleccionadas. Si los resultados obtenidos no son adecuados, se debe revisar el protocolo y crear una nueva versión del mismo.		

En la Figura B-2 se muestra el procedimiento, definido como un diagrama de flujo, para la selección de estudios.

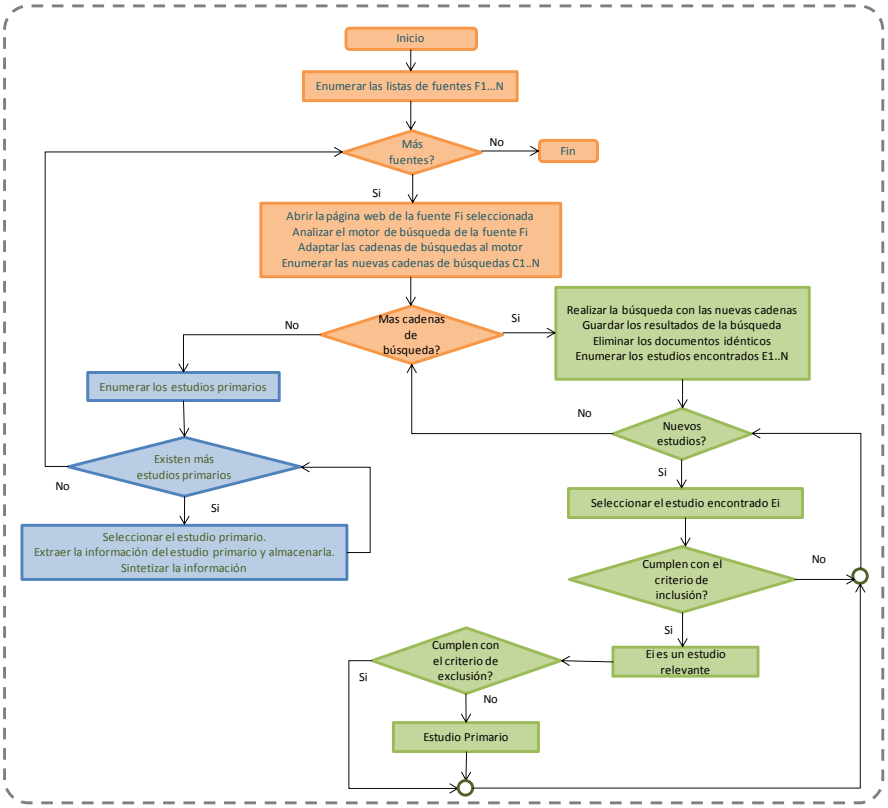

Figura B-2. Procedimiento para la Selección de Estudios

Como se puede observar se comienza enumerando las fuentes de búsquedas, para cada una de ellas se realiza la búsqueda adaptando las cadenas de búsquedas definidas al motor de la fuente. Si a partir de la adaptación de las cadenas surgen nuevas cadenas de búsqueda se vuelve a realizar la búsqueda con esas nuevas cadenas.

Con los documentos obtenidos se realiza un análisis inicial para verificar que no existan documentos repetidos. Posteriormente, para cada uno de los documentos encontrados se analiza el criterio de inclusión definido, si no cumple con el criterio de inclusión se descarta el documento; por el contrario, si cumple con el criterio de inclusión se analiza el criterio de exclusión. De similar manera, si cumple con el criterio de exclusión se descarta el documento y si no cumple se lo selecciona como estudio primario y pasa a la fase de análisis de resultados.

En la Tabla B-2 se muestran las actividades correspondientes a la fase de ejecución de la revisión y el punto de verificación de esta etapa.

En la Tabla B-3 se muestran las actividades correspondientes a la fase de análisis de los resultados.

Tabla B-2. Plantilla de Revisiones Sistemáticas – Etapa Ejecución

Ejecución de la Revisión	**Ejecución de la Selección**	**Selección de Estudios Iniciales**	Una vez aprobado el protocolo se ejecuta la RS. Los resultados obtenidos se listan para una primera evaluación
		Evaluación de la Calidad de los Estudios	A partir de la selección de los estudios iniciales, se deben evaluar los mismos aplicando los criterios de inclusión y exclusión definidos.
		Análisis de la Selección	Se debe revisar la selección de los estudios para garantizar que en el paso anterior no se hayan eliminado estudios relevantes. Es conveniente que esta actividad la realicen personas ajenas al protocolo de RS.
	Extracción de Información	**Formas para la Extracción de Resultados**	Se define la forma en la cual se recopilará la información. Depende siempre del tipo de RS que se esté realizando
		Ejecución de la Extracción	Cúal es el proceso con el que se extraen los resultados obtenidos de la RS.
Evaluación de la Ejecución	Durante la ejecución pueden surgir varios problemas, principalmente con los métodos de búsqueda de cada motor. Por lo tanto es necesaria una evaluación de los motores de búsqueda durante la ejecución para comprobar si son capaces de ejecutar las cadenas definidas en la fase de planificación		

Tabla B-3. Plantilla de Revisiones Sistemáticas – Etapa Análisis de los Resultados

Análisis de Resultados	**Resumen de los Resultados**	**Presentación en Tablas de Resultados**	Es conveniente mostrar los resultados obtenidos de la RS en tablas para facilitar el análisis de los mismos.	
		Comentarios Finales	Número de Estudios	La cantidad de estudios obtenidos y analizados
			Aplicación de los resultados	Define cómo los resultados obtenidos por la RS pueden ser aplicados
			Recomendaciones	Sugerencias de cómo se debe aplicar la RS

A partir de la adaptación de las plantillas que guian el proceso de la RS, en las siguientes secciones se presentan las planificaciones realizadas para cada una de las RS llevadas a cabo en esta tesis.

B.2 Revisión Sistemática de Propuestas Metodológicas para el Desarrollo de las Transformaciones

En esta sección se muestra el proceso de RS realizado para analizar las propuestas metodológicas para el desarrollo de transformaciones en el ámbito de MDE.

A continuación se muestran las actividades realizadas en cada etapa:

B.2.1 Etapa de Planificación de la Revisión

El principal objetivo de esta RS es realizar un estudio de las aproximaciones existentes que propongan el uso de MDE para el desarrollo de las transformaciones de modelos y, si las hubiera, de las herramientas que las implementan. Los documentos que se obtengan como resultado de la RS se analizarán y estudiarán para determinar el alcance y grado de aplicación de las mismas.

En la Tabla B-4 se muestra la planificación realizada para esta RS.

Tabla B-4. Planificación de la Revisión de Propuestas Metodológicas

Planificación de la Revisión	**Formulación de la Pregunta**	**Pregunta Principal**	Realizar un estudio de las aproximaciones existentes que propongan el uso de MDE para el desarrollo de las transformaciones de modelos y, si las hubiera, de las herramientas que las implementan.	
		Calidad y Amplitud	Problema	Evaluar se existen aproximaciones que propongan el uso de MDE para el modelado de las transformaciones. Es decir aproximaciones que permitan realizar el modelado de las transformaciones de modelos como modelos en sí mismos.
			Palabras Claves y Sinónimos	*Transformations*, *Meta-Transformations*, *High Order Transformations*, *Models*, *Tools*, *MDA* y *MDE*
			Intervención	En primer lugar realizar la búsqueda en cada una de las fuentes identificadas. A los resultados obtenidos aplicar los criterios de ínclusión y exclusión. Por último, realizar el análisis de los documentos obtenidos.
			Efecto	Como resultado se espera obtener el conjunto de documentos y/o herramientas que representen a las aproximaciones que propongan el uso de MDE en las transformaciones.
	Selección de Fuentes	**Definición de Criterios**	Criterio de inclusión: realizar un análisis del título, el resumen y las palabras clave de los documentos obtenidos en la búsqueda. De esta manera, se determina si los documentos pertenecen al tema de la revisión sistemática e incluirlos como estudios primarios de la misma. A partir del análisis de cada uno de los estudios primarios seleccionados surjen nuevos documentos que deben ser incluidos en la revisión sistemática. Criterio de exclusión: En este caso el criterio de exclusión está condicionado al criterio de inclusión, con lo cual no se lo define.	
		Identificación de Fuentes	Método de Búsqueda	La búsqueda se realiza en diferentes motores de bases de datos del ámbito de la Ingeniería de Software y en las Web y Actas de Congresos y Conferencias en el ámbito de MDE.
			Cadena de Búsqueda	Para obtener las cadenas de búsqueda se ha procedido a la concatenación de las palabras claves definidas..

			Lista de Fuentes	La lista de las fuentes iniciales para realizar la búsqueda son: ○ Science@Direct ○ IEEE Digital Library ○ SpringerLink ○ ACM Digital Library ○ Journal of Computer Science ○ Actas de congresos y workshops en el ámbito de MDE
			Chequeo de Referencia	Las fuentes han sido aprobadas por la directora y la co-directora de esta tesis
	Selección de Estudios	**Definición**	Definición de Tipos de Estudios	El estudio que se realizará es un estudio cualitativo y de observación en cuanto a los documentos que se seleccionen. En cuanto a las herramientas se verificará el funcionamiento por medio de la elaboración de casos de estudio.
			Procedimiento para la selección de los estudios	El proceso de selección es el mismo que el definido en la Figura B-2.
Evaluación de la Planificación		Una vez terminada la planificación de la RS se ha procedido a su evaluación, para esto, se realizó una prueba de las cadenas de búquedas en una de las fuentes. Como resultado se han obtenido una serie de estudios iniciales a los que se les ha aplicado el criterio de inclusión y exclusión.		

B.2.2 Etapa de Ejecución de la Revisión

Durante la ejecución de la RS, se han obtenido una serie de documentos sobre los que se deben aplicar el criterio de inclusión, para determinar cuales deberían ser incluidos en el proceso de análisis.

Como se ha dicho, el criterio de inclusión utilizado es realizar un análisis del título, el resumen y las palabras clave de los documentos obtenidos en la búsqueda. De esta manera, se determina si los documentos pertenecen al tema de la RS e incluirlos como estudios primarios de la misma. A partir del análisis de cada uno de los estudios primarios seleccionados surjen nuevos documentos que deben ser incluidos como resultados de la RS. En la Tabla B – 5 se presentan los resultados obtenidos luego de realizar la búsqueda en cada una de las fuentes.

Tabla B-5. Distribución de Estudios Seleccionados por Fuente

Fuentes	**Estudios**				**%**
	Encontrados	**No Repetidos**	**Relevantes**	**Primarios**	
IEEE Digital Library	258	62	10	6	20
Science Direct	151	70	20	10	33
SpringerLink	29	10	5	5	17
Google Academic	1526	34	10	6	20
ACM Digital Librery	20	7	3	3	10
Total	**1984**	**183**	**48**	**30**	**100**

En la Tabla B-6 se muestran las actividades establecidas para la etapa de ejecución de la RS de las propuestas metodológicas existentes.

Tabla B-6. Ejecución de la Revisión de Propuestas Metodológicas

Ejecución de la Revisión	**Ejecución de la Selección**	**Selección de Estudios Iniciales**	Aplicando el proceso de selección de estudios establecidos en la Figura B-2 se obtienen los primeros estudios iniciales.
		Evaluación de la Calidad de los Estudios	En cada uno de los estudios iniciales obtenidos en el paso anterior, se aplica el criterio de inclusión para determinar si deben ser incluidos en el conjunto de estudios a analizar.
		Análisis de la Selección	Los estudios seleccionados luego de aplicar el criterio de inclusión deben ser analizados nuevamente para asegurar su correcta inclusión.
	Extracción de Información	**Formas para la Extracción de Resultados**	En cuanto a la forma en la que se recopilará la información relativa a cada estudio, se propone realizar un resumen de cada propuesta y se definen una serie de características que deberan ser evaluadas en los estudios seleccionados. El resultado de dicha evaluación también se debe incluir en el resumen.
		Ejecución de la Extracción	Analizar cada estudio resaltando las características de cada uno.
Evaluación de la Ejecución	Durante la ejecución de la RS ha sido necesario redefinir algunas cadenas de búsqueda para adaptarlas a los diferentes motores con el objetivo de restringir los resultados obtenidos.		

B.2.3 Etapa de Análisis de Resultados

El análisis de los resultados obtenidos a partir del proceso de revisión sistemática realizado se presenta en la sección 2.1, donde se presenta un resumen de cada uno de los artículos seleccionados, evaluando en cada caso un conjunto de características determinadas.

B.3 Revisión Sistemática de Lenguajes de Transformación

En esta sección se muestra el proceso de RS realizado para analizar los lenguajes de transformación de modelos híbridos y las herramientas que los implementan en el ámbito de MDE.

A continuación, se muestran las actividades realizadas en cada etapa:

B.3.1 Etapa de Planificación de la Revisión

El principal objetivo de esta RS es evaluar los lenguajes de transformación de modelos, existentes en el ámbito del MDE, y las herramientas que los implementan. Los resultados esperados al finalizar esta RS, serán, entre otros, el de contar con un conjunto de lenguajes y herramientas de transformaciones que puedan ser evaluados aplicando las características definidas anteriormente.

En la Tabla B-7 se muestra la planificación realizada para esta RS.

Tabla B-7. Planificación de la Revisión de Lenguajes de Transformación Modelos

Planificación de la Revisión	**Formulación de la Pregunta**	**Pregunta Principal**	Evaluar los lenguajes de transformación de modelos, y las herramientas que los implementan, existentes en el ámbito del MDE.	
		Calidad y Amplitud	Problema	Evaluar los lenguajes de transformación de modelos y las herramientas que los implementan.
			Palabras Claves y Sinónimos	*Approach, hybrid, language, transformation, models, tools, MDA, MDE.*
			Intervención	La forma en la que se planea llevar a cabo esta RS es: en primer lugar realizar la búsqueda en cada una de las fuentes identificadas. A los resultados obtenidos aplicar los criterios de inclusión y exclusión. Por último, realizar el análisis de los documentos obtenidos.
			Efecto	Como resultado se espera obtener el conjunto de documentos y/o herramientas que presenten los lenguajes de transformación de modelos que siguen la aproximación híbrida y el conjunto de herramientas que los implementan.
	Selección de Fuentes	**Definición de Criterios**	Criterio de inclusión: realizar un análisis del título, el resumen y las palabras clave de los artículos obtenidos en la búsqueda. Esto sirve, en primera instancia, para determinar si los artículos pertenecen al tema que se desea evaluar e incluirlos como estudios primarios para la revisión. Criterios de exclusión: se han definido dos tipos de criterios: el primero que el lenguaje debe seguir la aproximación híbrida y el segundo es el hecho de que el lenguaje debe tener una herramienta que lo implemente, ya que el objetivo final de esta revisión sistemática es la de analizar los lenguajes de transformación que siguen la aproximación híbrida y las herramientas que los implementan para determinar el conjunto de características a las que MeTAGeM deberá dar soporte.	
		Identificación de Fuentes	Método de Búsqueda	La búsqueda se realiza en diferentes motores de bases de datos del ámbito de la Ingeniería de Software y en las Web y Actas de Congresos y Conferencias en el ámbito de MDE.
			Cadena de Búsqueda	Para obtener las cadenas de búsqueda se ha procedido a la concatenación de las palabras claves definidas.

			Lista de Fuentes	La lista de las fuentes iniciales para realizar la búsqueda son: ○ Science@Direct ○ IEEE Digital Library ○ SpringerLink ○ ACM Digital Library ○ Journal of Computer Science ○ Actas de congresos y workshops en el ámbito de MDE
			Chequeo de Referencia	Las fuentes han sido aprobadas por la directora y la co-directora de esta tesis
	Selección de Estudios	**Definición**	Definición de Tipos de Estudios	El estudio que se realizará es un estudio cualitativo y de observación en cuando a los documentos que se seleccionen. En cuanto a las herramientas se verificará el funcionamiento por medio de la elaboración de casos de estudios.
			Procedimiento para la selección de los estudios	El proceso de selección es el mismo que el definido en la Figura B-2.
Evaluación de la Planificación		Una vez terminada la planificación de la RS se procede a su evaluación, para esto, se ha realizado una prueba de las cadenas de búquedas en una de las fuentes. El resultado obtenido ha sido el esperado.		

B.3.2 Etapa de Ejecución de la Revisión

Como se ha dicho en la sección 1.4.1, para realizar esta tarea se propone un proceso iterativo e incremental. Es iterativo, porque la ejecución de la RS (búsqueda, extracción de la información y visualización de los resultados) se realiza para cada una de las fuentes seleccionadas y con cada una de las cadenas de búsqueda determinadas. Es incremental, en el sentido de que el resultado final obtenido, incrementa en cada iteración hasta obtener la versión definitiva.

A partir de la búsqueda realizada se ha obtenido numerosos documentos, por lo que es necesario definir criterios de inclusión y de exclusión que permitan acotar los resultados obtenidos, es decir determinar los criterios a partir de los cuales un lenguaje será incluido, o no, en el proceso de evaluación. En la Tabla B-8 se presentan los resultados obtenidos luego de realizar la búsqueda en cada una de las fuentes.

Tabla B-8. Distribución de Estudios Seleccionados por Fuente

Fuentes	**Estudios**				**%**
	Encontrados	**No Repetidos**	**Relevantes**	**Primarios***	
IEEE Digital Library	812	757	50	10	20
Science Direct	20.252	19.262	70	6	12
SpringerLink	50.996	30.562	63	7	14
Google Academic	1.150.000	948.635	77	15	30
ACM Digital Librery	14.185	9.623	25	12	24
Total	**1.236.245**	**1.008.839**	**3.831**	**50**	**100**

Es importante aclarar que a partir del análisis de cada uno de los estudios primarios seleccionados se han incluido nuevos artículos de lenguajes a analizar.

En la Tabla B-9 se muestra el protocolo establecido para la etapa de ejecución de la RS de los lenguajes de transformación y de las herramientas que los implementan.

Tabla B-9. Ejecución de la Revisión de Lenguajes de Transformación Híbridos

Ejecución de la Revisión	**Ejecución de la Selección**	**Selección de Estudios Iniciales**	Se aplica el proceso de selección de estudios establecidos en la Figura B-2 y se obtienen los primeros estudios iniciales.
		Evaluación de la Calidad de los Estudios	En cada uno de los estudios iniciales obtenidos en el paso anterior, se aplica los criterios de inclusión y de exclusión establecidos para determinar si deben ser incluidos como estudios a analizar.
		Análisis de la Selección	Los estudios seleccionados después de aplicar el criterio de inclusión se analizarán nuevamente para asegurar su correcta inclusión.
	Extracción de Información	**Formas para la Extracción de Resultados**	En cuanto a la forma en la que se recopilará la información relativa a cada estudio, se realiza un resumen de cada lenguaje incluyendo las particularidades de la herramienta que lo implemente. Además se definen una serie de características que deberan ser evaluadas en las herramientas.
		Ejecución de la Extracción	Analizar cada estudio resaltando las características de cada uno.
Evaluación de la Ejecución	Durante la ejecución de la RS ha sido necesario redefinir algunas cadenas de búsqueda para adaptarlas a los diferentes motores con el objetivo de restringir los resultados obtenidos.		

B.3.3 Etapa de Análisis de Resultados

El análisis de los resultados obtenidos a partir del proceso de revisión sistemática realizado se presenta en la sección 2.2, donde se presenta un resumen de cada uno de los artículos seleccionados, evaluando en cada caso un conjunto de características determinadas.

Apéndice C: Nuevos Trabajos Relacionados con el Estado del Arte

Como se ha comentado en el Capítulo 2 en la etapa final de la escritura de esta tesis, se encontraron dos artículos, publicados ambos en el año 2010 relevantes para el tema que se investiga en la presente tesis. Debido al avanzado estado de la tesis sólo se ha podido realizar un análisis de los artículos encontrados en relación con la propuesta realizada en cada artículo, pero no se han podido obtener las herramientas que los respaldan ni hacer un estudio de cada una de ellas; por lo que en este anexo se presentan las características de cada propuesta y las conclusiones a las que se llegaron en ambos casos.

C.1 Guerra, E., de Lara, J., Kolovos, D., Paige, R. y dos Santos O. M.

En [93] se presenta una familia de lenguajes de modelado, llamado *transML*, que abarca el ciclo de vida completo del desarrollo de transformaciones: desde la definición de los requisitos hasta las pruebas de las mismas. Según sus autores, *transML* se puede utilizar usando cualquier lenguaje de implementación de transformación. Además, al proponer un enfoque basado en MDE, la propuesta soporta el desarrollo parcialmente automático de los modelos de transformación y la generación de código en diferentes lenguajes de transformación. También soporta la reingeniería de transformaciones y facilitan la migración entre diferentes plataformas de transformación.

Básicamente, para el desarrollo de esta propuesta, los autores se basan en la idea de que, si las transformaciones de modelos se comienzan a utilizar a nivel industrial, es decir para desarrollar software con fines comerciales, deberían ser construidas utilizando los principios de ingeniería. Por lo tanto, el proceso de desarrollo de transformaciones debe incluir, además de la etapa de codificación y prueba, las etapas de: determinación de requisitos, análisis, diseño arquitectónico, diseño de alto nivel y diseño detallado. La notación utilizada en cada una de las fases tiene en cuenta las necesidades del desarrollo de los modelos de transformación.

C.1.1 Propuesta

Si bien la propuesta incluye un conjunto de lenguajes que permiten, por medio de diferentes tipos de diagramas, especificar los modelos de cada una de las etapas, los autores consideran que la utilización de los modelos definidos en cada etapa depende de las características de la transformación que se esté realizando. Sin embargo, el uso de estos modelos en forma combinada es el punto fuerte de *transML*.

A continuación se describen brevemente cada una de las etapas propuestas en *transML*.

- ***Determinación de Requerimientos***. Con el objetivo de mantener la trazabilidad desde los requerimientos hasta los elementos generados a partir de los mismos en las etapas posteriores, *transML* incluye una representación de los requisitos en forma de diagramas, similares a los requisitos diagramas SysML (http://www.omg.org/spec/SysML/1.1/). Para esto, definen un meta-modelo que permite agrupar los requerimientos en forma jerárquica, clasificarlos, realizar el refinamiento de los mismos y mantener la trazabilidad con los elementos generados posteriormente. En cuanto a la clasificación de los requerimientos se hace desde dos puntos de vista: por un lado, en cuanto a la funcionalidad de los mismos (funcionales y no funcionales); y, por otro lado, en cuanto a si pertenencen al modelo origen, al modelo destino o a la transformación en sí.
- ***Análisis.*** Para realizar el análisis de los requerimientos en este trabajo se propone la adaptación de las técnicas de análisis de requerimientos de la Ingeniería del Software. Para cada uno de los requerimientos (ya aprobados) se debe definir una transformación representativa (*transformation by example*, de forma similar al rol de los casos de uso en UML). Estos escenarios se llaman ***casos de transformación*** y describen cómo se debe transformar un modelo origen en concreto en un modelo destino. La definición de los casos de transformación tiene dos propósitos: por un lado, ser utilizados para comprender y razonar sobre lo que la transformación tiene que hacer; y, por otro lado, que puedan ser utilizados aplicando las técnicas de transformación de modelos por ejemplo (*model transformation-by-example*, [199]) que derivan en un esbozo de la transformación, permitiendo su uso como casos de prueba de la implementación de la transformación.
- ***Arquitectura de las Transformaciones.*** La propuesta incluye un lenguaje de modelado para el diseño de la arquitectura que permite realizar la definición

de la arquitectura por medio de unidades funcionales. Esto es muy útil para especificar transformaciones complejas, que deben ser divididas en diferentes módulos y permitir realizar la conexión entre dichos módulos.

- ***Diseño de Alto Nivel.*** El diseño de las transformaciones a alto nivel se realiza por medio de la definición de un diagrama de mapeo (*mapping diagram*), en el que se especifican las relaciones existentes entre los elementos de los diferentes modelos que participan en las transformaciones. Por medio de este diagrama se proporciona una idea de lo que se transforma y en qué se transforma sin dar detalles de cómo se debe realizar la transformación. Este diagrama se obtiene por medio de una transición entre el análisis y el diseño de la transformación. Para permitir el diseño de las transformaciones a alto nivel los autores han definido un meta-modelo que permite representar los conceptos específicos de este nivel de modelado.
- ***Diseño de Bajo Nivel.*** Los diagramas de diseño de bajo nivel indican cómo se debe implementer la transformación, haciendo una separación entre su estructura y su comportamiento. De esta manera se definen diagramas para indicar la estructura de cada una de las reglas de transformación y se anexan diagramas que indican qué se debe hacer en cada caso, es decir, definen su comportamiento. Para permitir el diseño de las transformaciones a un bajo nivel la propuesta define un meta-modelo que permite especificar la estructura de cada una de las reglas que la componen (indicando patrones de origen y destino), la ejecución de las mismas y su dependencia, dentro de la transformación, con otras reglas. Los diagramas de las reglas redefinen los diagramas de mapeos indicando como se deben realizar dichos mapeos.
- ***Implementación y Pruebas.*** *transML* no incluye ningún lenguaje propio de implementación de las transformaciones, pero brinda mecanismos para utilizar los lenguajes de transformación existentes (QVT, ATL, ETL, etc.). Siguiendo la filosofía MDE, se puede generar el código para diferentes plataformas a partir de los diagramas especificados en los niveles anteriores, en concreto a partir de los diagramas de reglas (de estructura y de comportamiento).

C.1.2 Herramienta de Soporte

Para soportar la propuesta, los autores han implementado los meta-modelos definidos en cada una de las etapas por medio de EMF, además han definido una serie de transformaciones de modelos y varios generadores de código que

permiten la transformación entre los diagramas de las diferentes etapas. Por medio de las transformaciones se brinda una automatización parcial del proceso desde la determinación de los requisitos hasta la generación de código. Por ejemplo, a partir de un diagrama de mapeo se puede generar el esqueleto del diagrama de reglas, que debe ser completado con la definición de su comportamiento, por medio de transformaciones definidas. Estas transformaciones se han implementado con el lenguaje ETL y para la generación de código se utiliza EGL.

C.1.3 Conclusión

Realizando una analogía con las características evaluadas (Tabla 2-1) en cada una de las aproximaciones en el capítulo 2, se puede decir que, desde nuestro punto de vista en la propuesta se permite el **modelado de las transformaciones a nivel PIM**, ya que en la etapa de diseño de las transformaciones a un alto nivel se realiza el mapeo entre los elementos de los modelos que intervienen en la transformación sin especificar los detalles de implementacion de los mismos. De la misma manera soporta el **modelado de las transformaciones a nivel PSM**, ya que en la etapa de diseño de las transformaciones a bajo nivel se redefinen los diagramas realizados en la etapa anterior brindando información de cómo se implementa la transformación con detalles específicos de la plataforma.

Ademas se definen reglas de transformación entre los diagramas de mapeos y los diagramas de reglas, por lo que permite realizar **transformaciones entre modelos definidos a nivel PIM y modelos definidos a nivel PSM**.

En cuanto al **modelado gráfico** de las transformaciones, según lo especificado en el artículo, en la etapa de análisis se pueden especificar los requerimientos de forma visual, para el resto de las etapas no se dice nada sobre su soporte gráfico.

Se permite la **generación automática del código** de la transformación, aunque según la documentación encontrada actualmente sólo se brinda soporte al lenguaje ETL. Además brinda **soporte a la validación** de las reglas de transformación por medio del OCL.

Se debe mencionar que si bien en la documentación estudiada los autores presentan, además de la propuesta, la herramienta que la valida y un caso de estudio en el cual se ha aplicado; no se ha podido tener acceso a la herramienta para corroborar su funcionamiento.

Por último, es importante recalcar que el artículo analizado a sido seleccionado como mejor artículo en el congreso Models2010 (*Model Driven Engineering Languages and Systems* 2010).

C.2 Kusel, A.

En [126] se propone un *framework* para el desarrollo de transformaciones de modelos llamado TROPIC (*Transformations on Petri Nets in Color*). Dicho *framework* permite especificar las transformaciones de modelos en diferentes niveles de abstracción, proporcionando una visión abstracta del mapeo de los elementos y una visión concreta de la transformación, resolviendo, de esta manera, las heterogeneidades estructurales entre los meta-modelos que participan en las transformaciones. Además, facilita la reutilización de transformaciones, proporcionando una biblioteca de componentes reutilizables que conduce a una mayor productividad en el desarrollo de transformaciones y a una mayor calidad en las mismas.

C.2.1 Propuesta

TROPIC propone el modelado de las transformaciones desde dos puntos de vista: una vista de mapeo, donde se realiza en un alto nivel de abstracción la correspondencia semántica entre los elementos que intervienen en la transformación; y una vista de transformación, donde se especifican los detalles de la lógica de la transformación.

- ***Vista de Mapeo***. Esta vista incluye los operadores de mapeo que permiten conectar elementos del meta-modelo origen con elementos del meta-modelo destino. Estos operadores de mapeo encapsulan la lógica de transformación reutilizable por medio de una biblioteca de componentes extensibles. Como formalismo de representación se utiliza los diagramas de componentes de UML2.
- ***Vista de Transformación.*** A partir de la vista de mapeo se genera la vista de la transformación ejecutable. Para ello, cada operador de mapeo, de la vista de mapeo, debe tener asociada una semántica operacional bien definida que indique la lógica de la transformación en la vista de transformación. Como formalismo de representación en esta vista se usan las redes de Petri Coloreadas (*Coloured Petri Nets*) llamadas redes de transformación.

Por último, se propone la definición de un compilador que realice la generación del código de la transformación a partir de la definición de los *mappings* en forma declarativa.

C.2.2 Herramienta de Soporte

Para dar soporte al *framework* propuesto se ha desarrollado un prototipo llamado TROPIC, que permite editar, ejecutar y depurar las redes de transformación. La implementación de TROPIC está basada en EMF y GMF e incluye dos editores, uno para la especificación de la transformación y otro que muestra la representación gráfica de dicha especificación en términos de redes de transformación. Además, proporciona funcionalidades de depuración, como la depuración paso a paso y otras funcionalidades de edición.

A partir de TROPIC, se están desarrollando una serie de patrones de transformación reutilizables llamados operadores de mapeo (*Mapping Operators*, MOP). Como primer prototipo utilizan AMW para probar el funcionamiento de los MOPs.

C.2.3 Conclusión

En cuanto a las caracteríticas propuestas en la Tabla 2-1 del capítulo 2, se puede decir que brindan soporte para el **modelado de las transformaciones a nivel PIM**, ya que en la vista de mapeo permite indicar las relaciones existentes entre los elementos de los meta-modelos origen y destino, sin indicar la forma en que se implementan posteriormente. Además, soporta el **modelado de las transformaciones a nivel PSM** a traves de la generación de la vista de transformación, donde se indica la lógica de la transformación.

Si bien, en la propuesta se dice que la transformación entre la vista de mapeo y la vista de transformación es automática, no se indica el conjunto de reglas de transformación entre ambas ni se indica con qué lenguaje se implementan.

En cuanto al **modelado gráfico de las transformaciones**, según lo especificado en el artículo en la vista de mapeo se utilizan los diagramas de componentes de UML y en la vista de transformación se utiliza las redes de transformación con una notación similar a las redes de Petri para representar las transformaciones.

Es importante mencionar que si bien la herramienta se encuentra disponible en la página del proyecto TROPIC (http://www.modeltransformation.net/) por cuestiones de tiempo no fue posible realizar una evaluación detallada de la misma, ya que como se dijo anteriormente en el momento de descubrir esta herramienta, el estado de la presente tesis ya estaba muy avanzado.

Apéndice D: Implementación de Reglas de Transformación

Como se ha visto a lo largo de esta tesis, MeTAGeM propone el modelado de las transformaciones en diferentes niveles de abstracción: independiente de plataforma (PIM), específico de plataforma (PSM), dependiente de plataforma (PDM) y por último, la generación de código. Para realizar la transformación entre los modelos de los diferentes niveles de abstracción se definen un conjunto de reglas de transformación que permiten automatizar esta tarea.

Estas reglas de transformación se han implementado utilizando el lenguaje de transformación ATL. En este apéndice se muestra la implementación realizada de las mismas.

D.1 Transformaciones de M-TIP a M-LTH

En el capítulo 3 se ha presentado el proceso de especificación de las reglas de transformación entre el meta-modelo de transformación independiente de plataforma (M-TIP) y el meta-modelo de transformación específico de plataforma siguiendo la aproximación híbrida (M-LTH). En la Tabla D-1 se muestra dicha especificación.

Tabla D-1. Transformaciones de M-TIP a M-LTH

Meta-modelo M-TIP		**Meta-modelo M-LTH**
ModelRoot		*Module*
InModelTransf		*InMetaModel*
OutModelTransf		*OutMetaModel*
Relations	*Sin dependencia de otros elementos*	*Rule*
	Con dependencia de otros elementos	*ElementIncluded*

Meta-modelo M-TIP		**Meta-modelo M-LTH**
OneToOne	*Sin dependencia de otros elementos*	- *Rule*: *in* con cardinalidad = 1 *out* con cardinalidad = 1 - *SourceElementRule* - *OutElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* -*LeftPattern*
OneToZero		- *Rule*: *in* con cardinalidad = 1 - *SourceElementRule*
ZeroToOne	*Sin dependencia de otros elementos*	- *Rule* *out* con cardinalidad 1 - *TargetElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* (sin elementos) - *LeftPattern*
OneToMany		- *Rule*: *in* con cardinalidad = 1 *out* con cardinalidad = N - *SourceElementRule* - *N OutElementRule*
ManyToOne	*Sin dependencia de otros elementos*	- *Rule*: *in* con cardinalidad = N *out* con cardinalidad = 1 - *N SourceElementRule* - *OutElementRule*
	Con dependencia de otros elementos	- *ElementIncluded* - *RightPattern* (solo 1 *RightPattern* con N *sourceElements* en su interior) -*LeftPattern*

Meta-modelo M-TIP	Meta-modelo M-LTH
ManyToMany	- *Rule*: *in* con cardinalidad = N *out* con cardinalidad = N - *N SourceElementRule* - *N OutElementRule*
InElement (con *GuardCondition*)	- *SourceElementRule* - *Guard*
InElement (sin *GuardCondition*)	- *SourceElementoRule*
OutElement	- *TargetElementRule*

A continuación se muestra como se han implementado cada una de las reglas usando el lenguaje ATL, a modo de establecer un orden en la presentación de las reglas se sigue el establecido en la Tabla D-1.

Para comenzar a implementar las reglas usando el lenguaje ATL, en primer lugar es necesario definir la cabecera del del archivo (Figura D-1).

Como se ha comentado anteriormente, las reglas en ATL se definen a nivel de meta-modelo, por lo que es necesario indicar en la cabecera del archivo ATL los meta-modelos entre los que se definen las reglas de transformación, indicando en cada caso el rol de cada uno. En la Figura D-1 se especifica que como meta-modelo origen (*from*) se tiene: el meta-modelo M-TIP, que es una extensión del meta-modelo base de AMW y los meta-modelos entre los que se establecen las relaciones (*left* y *right*) que, como se utiliza EMF, se definen de tipo MOF, de esta manera se asegura la compatibilidad con modelos definidos por otras herramientas. Como meta-modelo destino (*create*) el meta-modelo *MM_Hybrid*.

```
--@atlcompiler atl2006
module MeTAGeM2Hybrid;
create OUT : MM_Hybrid from IN : AMW, left : MOF, right : MOF;
```

Figura D-1. Cabecera Módulo ATL - MeTAGeM 2Hybrid

Una vez definida la cabecera del módulo que contiene las reglas de transformación se comienza con la implementación de cada una de las reglas.

En primer lugar se realiza la implementación de la regla que transforma los elementos de tipo *ModelRoot* a elementos de tipo *Module* (Figura D-2), completándose la propiedad *name* y las referencias a los elementos *inMetaModel*, *outMetaModel* y *Rule*. De acuerdo al meta-modelo *MM_Hybrid* especificado, M-

LTH a partir de ahora, un elemento del tipo *Module* está compuesto por elementos de tipo *inMetaModel*, *outMetaModel*, *Rule* y *Operation*.

Los elementos de tipo *Operation* no tiene su correspondencia a nivel PIM, ya que el meta-modelo de nivel PIM sólo recoge las relaciones entre los elementos de los meta-modelos participantes. El desarrollador podrá agregar elementos del tipo *Operation* al modelo generado con las transformaciones en caso de identificar la necesidad de las mismas.

```
rule Module {
   from
      amw : AMW!ModelRoot
   to
      hybrid : MM_Hybrid!Module (
               name_module <- amw.name,
               inMM <- amw.inputModel,
               outMM <- amw.outputModel,
               "rule" <- amw.relations)
}
```

Figura D-2. Regla de Transformación ATL – *Module*

La Figura D-3 muestra la definición de las reglas que transforman los elementos de tipo *InModelTransf* y *OutModelTransf* en los elementos *InMetaModel* y *OutMetaModel*, respectivamente. Estos elementos representan a los meta-modelos origen y destino que participan en la transformación final. Para cada uno de los elementos generados se especifica el nombre del meta-modelo (*name_mm*) y el tipo del meta-modelo (*type_mm*).

```
rule inModel{
   from
      in_MM_amw: AMW!InModelTransf
   to
      in_MM_hybrid: MM_Hybrid!InMetaModel(
             name_mm <- in_MM_amw.name+'_model',
             type_mm <- in_MM_amw.name)
}

rule outModel{
   from
      out_MM_amw: AMW!OutModelTransf
   to
      out_MM_hybrid: MM_Hybrid!OutMetaModel(
             name_mm <- out_MM_amw.name+'_model',
             type_mm <-  out_MM_amw.name)
}
```

Figura D-3. Regla de Transformación ATL – *inModel* y *outModel*

Los elementos de tipo *Relations* se pueden transformar a un elemento de tipo *Rule* o a un elemento de tipo *ElementIncluded*, de acuerdo al valor que retorna el *helper IsIncluded*. Además, en el meta-modelo M-TIP, la meta-clase *Relations* se define como abstracta, por lo que para realizar la transformación de los elementos de tipo *Relations* se ha considerado conveniente realizar dos reglas abstractas *Relations2Rule* (Figura D-4) y *Relations2ElementIncluded* (Figura D-5) que permita transformar los elementos generales a todos los tipos de relaciones. Estas regla serán extendida por las reglas específicas para cada tipo.

```
abstract rule Relations2Rule{
   from
      relation:AMW!Relations(relation.isNotIncluded())
   to
      r_hybrid:MM_Hybrid!Rule(
            name_rule <- relation.getRuleName(),
            isAbstract <- relation.typeRelation=#IsAbstract,
            isMain <- relation.typeRelation=#IsMain,
            "extends" <- relation.extend,
            isExtended <- relation.isExtend,
            typeAttribute <- relation.typeAttri,
            typeElement <- relation.typeE)
}
```

Figura D-4. Regla de Transformación ATL – *Relations2Rule*

Al tener dos reglas de transformación definidas sobre un mismo elemento es necesario definir correctamente la condición de guarda de las reglas. De manera que no exista ambigüedad en el momento de que el motor de transformación seleccione la regla.

```
abstract rule Relations2ElementIncluded{
   from
    relation_in:AMW!Relations(relation_in.isIncluded())
   to
      elementIncluded:MM_Hybrid!ElementIncluded(
            typeAttribute <- relation_in.typeAttri,
            typeElement <- relation_in.typeE)
}
```

Figura D-5. Regla de Transformación ATL – *Relations2ElementIncluded*

Para definir la condición de guarda de ambas reglas se han definido dos funciones auxiliares llamadas *helpers*: *isIncluded* e *isNotIncluded*. Como se puede observar en la Figura D-6 el *helper isIncluded()* comprueba el tipo del elemento *Relations*, ya que los únicos elementos *Relations* que pueden estar incluidos dentro de otro elemento son los de tipo: *OneToOne*, *ManyToOne* y *ZeroToOne*,

con lo que si se cumple esa condición se evalúa la cardinalidad de la referencia *outElement* que indica si el elemento *Relations* depende de un elemento *OutElement*. Si la cardinalidad sea mayor que cero indica que está incluido en otro elemento. En el segundo caso, el *helper isNotIncluded***,** controla que el valor del *helper isIncluded* sea *false*, es decir, que el elemento de tipo *Relations* no esté incluido dentro de otro elemento de tipo *Relations*.

```
helper context AMW!Relations def: isIncluded() : Boolean =
   if ((self.oclIsTypeOf(AMW!OneToOne))
    or  (self.oclIsTypeOf(AMW!ManyToOne))
    or (self.oclIsTypeOf(AMW!ZeroToOne))) then
        self.outElement.asSequence().size()>0
   else
        false
   endif;

helper context AMW!Relations def: isNotIncluded() : Boolean =
   not self.isIncluded();
```

Figura D-6. *Helper* ATL –*isIncluded e isNotIncluded*

Además se han definido otros *helpers*, para entre otras cosas, ayudar a mejorar la modularidad de las reglas, como por ejemplo el *helper getRuleName* (Figura D-7) que permite recuperar el nombre de la regla que se esta generando.

```
helper context AMW!Relations def : getRuleName () : String =
      if self.name.oclIsUndefined() then
            self.getInOutPatternName()
      else
            if self.name = '' then
                  self.getInOutPatternName()
            else
                  self.name
            endif
      endif;
```

Figura D-7. *Helper* ATL – *getRuleName*

En caso de que el desarrollador no haya especificado dicho nombre en el modelo a nivel PIM, por medio de otro *helper getInOutPatternName* (Figura D-8) se genera un nombre por defecto. Dicho nombre se genera a partir de la concatenación de los nombres de los elementos de tipo *InElement* y *OutElement* que participan en la relación. De esta manera se asegura que todas las reglas tengan un nombre, lo que evita errores en la generación de los siguientes modelos. Por último, si la regla no tiene definido el nombre de los elementos *InElement* o

OutElement, entonces se genera un nombre de la forma RN, donde N es el número de la regla que se autoincrementa cada vez que se especifica una nueva regla, esto se logra mediante la sentencia *thisModule.countRules.toString()*.

```
helper context AMW!Relations def : getInOutPatternName () : String =
  if ((self.oclIsTypeOf(AMW!ZeroToOne))
      or  (self.oclIsTypeOf(AMW!OneToZero))) then
      'R' + thisModule.countRules.toString()
  else
    if not self.source.asSequence()->first().oclIsUndefined() and not
      self.target.asSequence()->first().oclIsUndefined() then
        self.source.asSequence()->first().name + '_2_'+
        self.target.asSequence()->first().name
    else
      'R' + thisModule.countRules.toString()
    endif
  endif;
```

Figura D-8. *Helper* ATL – *getInOutPatternName*

De la misma manera que en el caso e los elementos de tipo *Relations*, los elementos de tipo *OneToOne* se pueden transformar en elementos de tipo *Rule* o elementos de *ElementIncluded*, para ello se definen dos tipos de reglas diferentes.

En la Figura D-9 se muestra la transformación de los elementos de tipo *OneToOne* en elementos de tipo *Rule*. Como se puede observar, esta regla extiende la regla *Relations2Rule*, por medio de la palabra reservada *extends* y se establece el valor de las propiedades *in* y *out* con la asignación de las referencias a los elementos *SourceElementRule* y *OutElementRule* que se generaron a partir de los elementos *InElement*, cuya referencia se encuentra en la propiedad *source*, y *OutElement*, cuya referencia se encuentra en la propiedad *target*. En este caso la cardinalidad de las propiedades *in* y *out* será de uno en ambos casos.

```
rule OneToOne2rule extends Relations2Rule{
   from
      relation: AMW!OneToOne
   to
      r_hybrid: MM_Hybrid!Rule(
            "in" <- relation.source.asSequence(),
            out <- relation.target.asSequence())
   do {
      thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-9. Regla de Transformación ATL – *OneToOne2Rule*

En la Figura D-10 se muestra la transformación de los elementos de tipo *OneToOne* en elementos de tipo *ElementIncluded*. Esta regla extiende la regla *Relations2ElementIncluded*, además se establecen las referencias en las propiedades *right* y *left* con los valores de los elementos de tipo *rightPattern* y *leftPattern* creados en la misma regla. El elemento *rightPattern* se genera a partir de los elementos referenciados en la propiedad *source* del elemento *OneToOne* y los elementos de tipo *leftPattern* se generan a partir de los elementos referenciados en la propiedad *target* del elemento *OneToOne*. La cardinalidad de los elementos *right* y *left* será de uno en ambos casos.

```
rule OneToOne2ElementIncluded extends Relations2ElementIncluded{
   from
     relation_in:AMW!OneToOne
   to
     elementIncluded: MM_Hybrid!ElementIncluded (
        right <- rightPattern,
        left <- leftPattern),

     rightPattern:MM_Hybrid!RightPattern(
        name_pattern <- relation_in.getRuleName().concat('_rightPattern'),
        sourceElement <- relation_in.source.asSequence(),
        "rule" <- relation_in.getRuleInvoked()),

     leftPattern:MM_Hybrid!LeftPattern(
        name_pattern  <- relation_in.getRuleName().concat('_leftPattern'),
        targetElement <- relation_in.target)
   do {
      thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-10. Regla de Transformación ATL – *OneToOne2ElementIncluded*

En la Figura D-11 se muestra la regla que transforma elementos de tipo *OneToZero* a elementos de tipo *Rule*, esta regla tambien extiende la regla *Relations2Rule*. Un elemento de tipo *OneToZero* está compuesto solo por un elemento de tipo *SourceElementRule*, por lo que el elemento de tipo *Rule* generado a partir del mismo solo tendrá definida una referencia a un elemento en la propiedad *in*.

```
rule OneToZero2rule extends Relations2Rule{
   from
     relation: AMW!OneToZero
   to
     r_hybrid: MM_Hybrid!Rule(
       "in" <- relation.source.asSequence())
   do {
     thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-11. Regla de Transformación ATL – *OneToZero2Rule*

Los elementos de tipo *ZeroToOne* se pueden transformar en elementos de tipo *Rule* o elementos de *ElementIncluded*, para ello se definen dos tipos de reglas diferentes.

En la Figura D-12 se muestra la regla de transformación entre elementos de tipo *ZeroToOne* a elementos de tipo *Rule*, que extiende a la regla *Relations2Rule*. De forma contraria a la regla anterior, un elemento de tipo *ZeroToOne* está compuesto solo de un elemento de tipo *TargetElementRule*, por lo que el elemento de tipo *Rule* generado a partir del mismo solo tendrá definida una referencia a un elemento en la propiedad *out*.

```
rule ZeroToOne2rule extends Relations2Rule{
   from
     relation: AMW!ZeroToOne
   to
      r_hybrid: MM_Hybrid!Rule(
            out <- relation.target.asSequence())
   do {
      thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-12. Regla de Transformación ATL –*ZeroToOne2Rule*

En la Figura D-13 se muestra la regla de transformación entre elementos de tipo *ZeroToOne* a elementos de tipo *ElementIncluded*, que extiende a la regla *Relations2ElementIncluded*. En este caso se establecen las referencias en las propiedades *right* y *left* con los valores de los elementos de tipo *rightPattern* y *leftPattern* creados en la misma regla. A diferencia de la regla *OneToOne2ElementIncluded*, en el momento que se crea el elemento *rightPattern* solo se establece la propiedad *name* del mismo. Los elementos de tipo *leftPattern* se generan de la misma manera que en la regla anterior.

```
rule ZeroToOne2ElementIncluded extends Relations2ElementIncluded{
   from
     relation_in:AMW!ZeroToOne
   to
      elementIncluded: MM_Hybrid!ElementIncluded (
         right <- rightPattern,
         left <- leftPattern),

      rightPattern:MM_Hybrid!RightPattern(
         name_pattern <- relation_in.getRuleName().concat('_rightPattern')),

      leftPattern:MM_Hybrid!LeftPattern(
         name_pattern  <- relation_in.getRuleName().concat('_leftPattern'),
         targetElement <- relation_in.target)

   do {
      thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-13. Regla de Transformación ATL –*ZeroToOne2ElementIncluded*

La Figura D-14 muestra la regla que permite la transformación entre elementos de tipo *OneToMany* y elementos de tipo *Rule*. Esta regla extiende la regla *Relations2Rule* y su definición es similar a la regla *OneToOne*.

```
rule OneToMany2rule extends Relations2Rule{
   from
     relation: AMW!OneToMany
   to
     r_hybrid: MM_Hybrid!Rule(
        "in" <- relation.source.asSequence(),
         out <- relation.target.asSequence())
   do {
     thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-14. Regla de Transformación ATL – *OneToMany2Rule*

Los elementos de tipo *ManyToOne* se pueden transformar en elementos de tipo *Rule* o elementos de tipo *ElementIncluded*, para ello se definen dos tipos de reglas diferentes.

La Figura D-15 muestra la regla de transformación entre los elementos de tipo *ManyToOne* y los elementos de tipo *Rule*. Esta regla extiende a la regla *RelationsToRule* y su definición es similar a la regla *OneToOne*.

```
rule ManyToOne2rule extends Relations2Rule{
   from
     relation: AMW!ManyToOne
   to
     r_hybrid: MM_Hybrid!Rule(
      "in" <- relation.source.asSequence(),
       out <- relation.target.asSequence())
do {
     thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-15. Regla de Transformación ATL –*ManyToOne2Rule*

La Figura D-16 muestra la regla que permite la transformación entre elementos de tipo *ManyToOne* y elementos de tipo *ElementIncluded* y que extiende a la regla *Relations2ElementIncluded*. Como se puede observar la definición es similar a la de la regla *OneToOne2ElementIncluded*.

```
rule ManyToOne2ElementIncluded extends Relations2ElementIncluded{
   from
    relation_in:AMW!ManyToOne
   to
    elementIncluded: MM_Hybrid!ElementIncluded (
      right <- rightPattern,
      left <- leftPattern),

    rightPattern:MM_Hybrid!RightPattern(
      name_pattern <- relation_in.getRuleName().concat('_rightPattern'),
      sourceElement <- relation_in.source.asSequence(),
      "rule" <- relation_in.getRuleInvoked()),

    leftPattern:MM_Hybrid!LeftPattern(
      name_pattern  <- relation_in.getRuleName().concat('_leftPattern'),
      targetElement <- relation_in.target)

   do {
     thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-16. Regla de Transformación ATL –*ManyToOne2ElementIncluded*

En la Figura D-17 se muestra la regla que permite transformar elementos del tipo *ManyToMany* a elementos del tipo *Rule*, como se puede observar se extiende la regla *Relations2Rule* y su definición es similar a la regla *OneToOne*.

```
rule ManyToMany2rule extends Relations2Rule{
   from
      relation: AMW!ManyToMany
   to
      r_hybrid: MM_Hybrid!Rule(
            "in" <- relation.source.asSequence(),
            out <- relation.target.asSequence())
      do {
            thisModule.countRules <- thisModule.countRules + 1;}
}
```

Figura D-17. Regla de Transformación ATL – *ManyToMany2Rule*

Los elementos de tipo *InElement* se transforman de dos maneras diferentes, dependiendo si tienen definido el valor de la propiedad *guardCondition*. Por este motivo se han especificado dos reglas diferentes para contemplar ambas posibilidades.

En la Figura D-18 se muestra la regla que permite la transformación entre elementos de tipo *InElement*, que no tengan el valor de la propiedad *guardCondition* definido, y elementos de tipo *SourceElementRule*. Para esto en la guarda de la regla se comprueba que el valor de dicha propiedad sea vacio. En el elemento *SourceElementRule* se completan la propiedad *name_element*, con el

valor de la propiedad name del elemento *InElement*, y la propiedad *metamodel* con el valor de la propiedad *element.modelRef* del elemento *InElement*.

```
rule InElement2SourceElementRuleWithoutGuard{
   from
     inElem: AMW!InElement (inElem.guardCondition.oclIsUndefined()
     and not( inElem.element.oclIsUndefined()))
   to
     sourceElem: MM_Hybrid!SourceElementRule(
       name_element <- inElem.name,
       metamodel <- inElem.element.modelRef)
}
```

Figura D-18. Regla de Transformación ATL – *InElement2SourceElementRuleWithoutGuard*

En la Figura D-19 se muestra la regla que permite la transformación entre elementos de tipo *InElement*, que tengan el valor de la propiedad *guardCondition* definido, y elementos de tipo *SourceElementRule*. A diferencia del caso anterior, en esta regla además de generar un elemento de tipo *SourceElementRule* se genera un elemento de tipo *Guard*, donde se establece que el valor de su propiedad *value* sea igual al valor de la propiedad *guardCondition* del elemento *InElement*.

```
rule InElement2SourceElementRuleWithGuard{
   from
     inElem: AMW!InElement (not inElem.guardCondition.oclIsUndefined()
     and not( inElem.element.oclIsUndefined()))
   to
     sourceElem: MM_Hybrid!SourceElementRule(
       name_element <- inElem.name,
       metamodel <- inElem.element.modelRef,
       guard <- sourceGuard),
     sourceGuard: MM_Hybrid!Guard(
       value <- inElem.guardCondition)
}
```

Figura D-19. Regla de Transformación ATL – *InElement2SourceElementRuleWithGuard*

En la Figura D-20 se muestra la regla de transformación entre elementos de tipo *OutElement* y elementos de tipo *TargetElementRule*. Cuando se crea el elemento *TargetElementRule* se establece el valor de sus propiedades: *name_element*, *metamodel* e *included*.

```
rule OutElement2TargetElementRule{
   from
     outElem: AMW!OutElement
   to
     targetElem: MM_Hybrid!TargetElementRule(
     name_element <- outElem.name,
     metamodel <- outElem.element.modelRef,
     included <- outElem.relationsIncluded())
}
```

Figura D-20. Regla de Transformación ATL – *OutElement2TargetElementRule*

A nivel PDM MeTAGeM permite realizar el modelado de las transformaciones con dos lenguajes de transformación diferentes: ATL y RubyTL, por lo que para permitir la transformación entre el modelo definido a nivel PSM y los modelos definidos a nivel PDM es necesario definir dos conjunto de reglas de transformación diferentes: las reglas que permiten transformar a modelos conformes al meta-modelo de ATL y las reglas que permiten transformar a modelos conformes al meta-modelo RubyTL.

A continuación se explican cada conjunto de reglas por separado.

D.2 Transformaciones de M-LTH a ATL

En el capítulo 3 se ha presentado el proceso de especificación de las reglas de transformación entre el meta-modelo de transformación espécifico de plataforma siguiendo la aproximación híbrida (M-LTH) y el meta-modelo de ATL para el modelado de las transformaciones de forma dependiente de plataforma. En la Tabla D-2 se muestra dicha especificación.

Tabla D-2. Transformaciones de M-LTH a ATL

Meta-modelo M-LTH		**Meta-modelo ATL de Nivel PDM**	
Module		*Module*	
InMetaModel		*OclModel (Input)*	
OutMetaModel		*OclModel (Output)*	
Rule	*isMain = true* y *in > 0*	*Rule*	*Matched Rule*
	isAbstract = true y *isExtended = defined* y *in > 0*		*MatchedRule (superRule= true)*

Meta-modelo M-LTH	Meta-modelo ATL de Nivel PDM
isMain = false and in > 0	*LazyMatchedRule*
(isMain = false and in > 0 and typeAttribute = #unique)	*LazyMatchedRule (unique = true)*
in = 0	*CalledRule*
SourceElementRule	*SimpleInPatternElement*
TargetElementRule	*SimpleOutPatternElement*
ElementIncluded *- LeftPattern* *- RightPattern*	*Binding* *- propertyName (Left side)* *- value (Right side)*
Operation *- Return.datatype* *- Boolean* *- Integer* *- String* *- Element*	*Helper* *- Operation.returnType* *- BooleanType* *- IntegerType* *- StringType* *- OclModelElement*

A continuación se muestra como se han implementado cada una de las reglas usando el lenguaje ATL, a modo de establecer un orden en la presentación de las reglas se sigue el establecido en la Tabla D-2.

En la Figura D-21 se muestra la cabecera del módulo ATL, en el que se definen las reglas de transformación. Como se puede verificar se define como meta-modelo origen (*from*) el meta-modelo MM_Hybrid y como meta-modelo destino (*create*) al meta-modelo ATL.

```
module Hybrid2ATL;
create OUT : ATL from IN : MM_Hybrid;
```

Figura D-21. Regla de Transformación ATL – *Hybrid2ATL*

Una vez definida la cabecera del módulo que contiene las reglas de transformación se comienza con la implementación de cada una de las reglas.

La Figura D-22 muestra la regla que implementa la transformación entre los elementos del tipo *Module* del meta-modelo MM_Hybrid y los elementos de tipo *Module* del meta-modelo ATL. Se completan las propiedades *isRefining* y *name* y las referencias: *inModels*, que indica los meta-modelos origen que participan en la transformación, *outModels*, que indica los meta-modelos destino

de la relación, *elements*, que representa a las reglas de transformación que forman parte del módulo y *commentsBefore*, que permite establecer un comentario antes de comenzar con la definición de la regla.

```
rule Module {
   from
      mm_hybrid : MM_Hybrid!Module
   to
      atl : ATL!Module (
            isRefining <- false,
            name <- mm_hybrid.name_module,
            inModels <- mm_hybrid.inMM,
            outModels <- mm_hybrid.outMM,
            elements <- mm_hybrid."rule",
            commentsBefore <- Set {'-- @atlcompiler atl2006'})
}
```

Figura D-22. Regla de Transformación ATL – *Module*

Para la transformación de los elementos de tipo meta-modelo origen (*InMetaModel*) y meta-modelo destino (*OutMetaModel*) se definen dos reglas (Figura D-23) que generan el correspondiente elemento de tipo *OCLModel* en el modelo destino conforme al meta-modelo ATL.

```
rule inMM{
   from
      inMM_hybrid : MM_Hybrid!InMetaModel
   to
      inMM_ATL : ATL!OclModel(
          name <- inMM_hybrid.name_mm,
          metamodel <- ametamodelinMM),

      ametamodelinMM : ATL!OclModel (
          name <- inMM_hybrid.type_mm)
}

rule outMM{
   from
      outMM_hybrid : MM_Hybrid!OutMetaModel
   to
      outMM_ATL : ATL!OclModel(
           name <- outMM_hybrid.name_mm,
           metamodel <- ametamodeloutMM),

      ametamodeloutMM : ATL!OclModel(
           name <- outMM_hybrid.type_mm)
}
```

Figura D-23. Regla de Transformación ATL – *inMM* y *OutMM*

La meta-clase *Rule* en ATL es de tipo abstracta, por lo que se implementa por tres meta-clases diferentes: *MatchedRule*, *LazyMatchedRule* y *CalledRule*. Cada una de estas meta-clases representa el tipo de reglas que es posible especificar con ATL. De esta manera, las *MatchedRules* sirven para especificar explícitamente qué elemento del meta-modelo origen se transforma en qué elemento del meta-modelo destino; este tipo de regla se ejecuta siempre que el motor de ATL encuentre una coincidencia en el modelo origen que se transforma. Las *LazyMatchedRules* tienen un comportamiento similar, aunque la diferencia radica en que su ejecución debe ser realizada explícitamente. Por último, las *CalledRules* son las reglas que se utilizan para generar elementos en el modelo destino de manera imperativa, es decir, no existe un elemento del modelo origen a partir del cual se genera el elemento del modelo destino.

Para la transformación de los elementos de tipo *Rule* a sus correspondientes elementos de tipo *Rule* en ATL, es necesario definir tres tipos de reglas diferentes, una por cada tipo de regla en ATL. La determinación de a qué tipo de regla de ATL se debe transformar debe estar en forma explícita en la condición de guarda de cada una de las reglas definidas.

En la Figura D-24 se muestra la transformación de los elementos de tipo *Rule* a elementos de tipo *MatchedRule*. Como se puede observar en la condición de guarda se evalúa el valor de la propiedad *isMain* del elemento, además, se especifica una función especial, el *helper getSizeIP* que verifica la cantidad de elementos del tipo *SourceElementRule* que dependen del elemento *Rule*. De esta manera, si el valor de la propiedad *isMain* es *true* y el valor que retorna el *helper getSizeIP* es mayor que cero, es decir, la regla tiene elementos de tipo *SourceElementRule*, entonces el elemento *Rule* se transforma a un elemento de tipo *MatchedRule*.

Como se puede observar en la Figura D-24, además del elemento de tipo *MatcheRule*, se crean los elementos de tipo *InPattern* y *OutPattern* como referencias a los elementos que dependan del elemento de tipo *Rule*, en las propiedades *in* y *out* respectivamente.

```
rule createRule2MatchedRule{
   from
    mm_hybrid_rule : MM_Hybrid!Rule
    (mm_hybrid_rule.isMain=true and mm_hybrid_rule.getSizeIP()>0)
   to
    atl : ATL!MatchedRule (
      name <- mm_hybrid_rule.name_rule,
      isAbstract <- mm_hybrid_rule.isAbstract,
      isRefining <- false,
      isNoDefault <- false,
      superRule <- mm_hybrid_rule."extends",
      inPattern <- inPattern,
      outPattern <- outPattern,
      commentsBefore <- Set {'-- Comments -> This is a MatchedRule: '
      + mm_hybrid_rule.name_rule + ' -> ' + mm_hybrid_rule.getComment()}),

    inPattern : ATL!InPattern (
      elements <- mm_hybrid_rule."in".asSequence(),
      filter <- mm_hybrid_rule.getFilter()),

    outPattern : ATL!OutPattern(
      elements <- mm_hybrid_rule.out.asSequence())
}
```

Figura D-24. Regla de Transformación ATL – *createRule2MatchedRule*

En la Figura D-25 se muestra la definición del *helper getSizeIP()*, que retorna la cantidad de elementos tipo *in* que tiene un elemento de tipo *Rule*.

```
helper context MM_Hybrid!Rule def : getSizeIP () : Integer =
      self."in".size();
```

Figura D-25. Regla de Transformación ATL – *getSizeIP()*

Existen dos tipos *LazyMatchedRule* en ATL, la *LazyMatchedRule* normal y la *UniqueLazyRule*, la diferencia entre ambos tipos es que, la segunda se ejecuta una sola vez y en las llamadas siguientes devuelve la referencia al elemento transformado la primera vez. Para permitir estos dos tipos de transformaciones se especifican dos reglas diferentes.

En la Figura D-26 se muestra la transformación de los elementos de tipo *Rule* a elementos de tipo *LazyMatchedRule*, en la condición de guarda de esta regla se controla: que el valor de la propiedad *isMain* sea *false*, que el valor de la propiedad *typeAttribute* sea distinto de "*#isUnique*" y que el valor que retorna el *helper getSizeIP* sea mayor que cero.

En este caso se crea el elemento de tipo *LazyMatchedRule*, estableciéndose el valor de la propiedad *isUnique* como *False*. Además, al igual que en el caso de las *MatchedRule* también se crean los elementos de tipo *InPattern* y *OutPattern*.

```
rule createRule2LazyRule{
   from
    mm_hybrid_rule : MM_Hybrid!Rule
    (mm_hybrid_rule.isMain=false and not
    (mm_hybrid_rule.typeAttribute = #"unique")
     and mm_hybrid_rule.getSizeIP()>0)
   to
    atl : ATL!LazyMatchedRule (
      name <- mm_hybrid_rule.name_rule,
      isAbstract <- mm_hybrid_rule.isAbstract,
      isRefining <- false,
      isNoDefault <- false,
      isUnique <- false,
      inPattern <- inPattern,
      outPattern <- outPattern,
      commentsBefore <- Set {'-- Comments -> This is a LazyRule: ' +
      mm_hybrid_rule.name_rule + ' -> ' + mm_hybrid_rule.getComment()}),

    inPattern : ATL!InPattern (
      elements <- mm_hybrid_rule."in".asSequence(),
      filter <- mm_hybrid_rule.getFilter()),

    outPattern : ATL!OutPattern(
      elements <- mm_hybrid_rule.out.asSequence())
}
```

Figura D-26. Regla de Transformación ATL – *creatRule2LazyRule*

En la Figura D-27 se muestra la transformación de los elementos de tipo *Rule* a elementos de tipo *LazyMatchedRule* que se estable como *Unique*, en la condición de guarda de esta regla se controla que el valor de la propiedad *isMain* sea *false*, que el valor de la propiedad *typeAttribute* sea igual a "*#isUnique*" y que el valor que retorna el *helper getSizeIP* sea mayor que cero.

En este caso se crea el elemento de tipo *LazyMatchedRule* estableciéndose el valor de la propiedad *isUnique* como *True*, esta propiedad permite identificar el tipo de *LazyMatchedRule* y antepone a la regla la palabra reservada *Unique* cuando se crea el código de la transformación. Además, al igual que en el caso de las *MatchedRule* también se crean los elementos de tipo *InPattern* y *OutPattern*

```
rule createRule2UniqueLazyRule{
   from
     mm_hybrid_rule : MM_Hybrid!Rule
     (mm_hybrid_rule.isMain=false
     and mm_hybrid_rule.typeAttribute = #"unique"
     and mm_hybrid_rule.getSizeIP()>0)
   to
     atl : ATL!LazyMatchedRule (
      name <- mm_hybrid_rule.name_rule.debug('Unique LazyRule'),
      isAbstract <- mm_hybrid_rule.isAbstract,
      isRefining <- false,
      isNoDefault <- false,
      isUnique <- true,
      inPattern <- inPattern,
      outPattern <- outPattern,
      commentsBefore <- Set {'-- Comments -> This is a LazyRule: ' +
      mm_hybrid_rule.name_rule + ' -> ' + mm_hybrid_rule.getComment()}),

     inPattern : ATL!InPattern (
      elements <- mm_hybrid_rule."in".asSequence(),
      filter <- mm_hybrid_rule.getFilter()),

     outPattern : ATL!OutPattern(
      elements <- mm_hybrid_rule.out.asSequence())
}
```

Figura D-27. Regla de Transformación ATL – *creatRule2LazyRule*

Por último, si el valor de la propiedad *isMain* es *true*, pero el valor que retorna el *helper getSizeIP* es cero, es decir, no existen elementos del tipo *SourceElementRule* que dependan del elemento *Rule*, entonces se transforma a un elemento de tipo *CalledRule* (Figura D-28). Como se puede observar, en este caso se generan, además del elemento de tipo *CalledRule*, elementos de tipo *OutPattern* y elementos de tipo *ActionBlock* que representa la parte imperativa de la regla ATL, donde el usuario podría agregar código específico.

```
rule createRule2CalledRule{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()=0)
   to
     atl : ATL!CalledRule (
      name <- mm_hybrid_rule.name_rule.debug('CalledRule'),
      outPattern <- outPattern,
      actionBlock <- anAction,
      commentsBefore <- Set {'-- Comments -> This is a CalledRule: ' +
      mm_hybrid_rule.name_rule + ' -> ' + mm_hybrid_rule.getComment()}),

     outPattern : ATL!OutPattern(
      elements <- mm_hybrid_rule.out.asSequence()),

     anAction : ATL!ActionBlock(
      commentsBefore <- Set {'-- ActionBlock: '})
}
```

Figura D-28. Regla de Transformación ATL – *creatRule2CalledRule*

En la Figura D-29 se muestra la transformación entre elementos de tipo *SourceElementRule* y elementos de tipo *SimpleInPatternElement*. Como se puede observar, además se genera un elemento de tipo *OclModelElement* que define el *type* del elemento *SimpleInPatternElement*.

```
rule InPatternElement {
   from
     inPattern : MM_Hybrid!SourceElementRule
     (inPattern.refImmediateComposite().oclIsTypeOf(MM_Hybrid!Rule))
   to
     atl : ATL!SimpleInPatternElement (
       varName <- inPattern.name_element.toLower()+'_in',
       type <- aType),

     aType : ATL!OclModelElement(
       name <- inPattern.name_element,
       model <- thisModule.resolveTemp(inPattern.metamodel,'ametamodelinMM'))
}
```

Figura D-29. Regla de Transformación ATL – *InPatternElement*

En la Figura D-30 se muestra la transformación entre elementos de tipo *TargetElementRule* y elementos de tipo *SimpleOutPatternElement*. De igual manera que en el caso anterior, se genera un elemento de tipo *OclModelElement* que define el *type* del elemento *SimpleOutPatternElement*. Además se generan, en la propiedad binding del elemento *SimpleOutPatternElement* las referencias a los elementos de tipo *outPattern* que estén incluidos en el elemento *TargetElementRule*.

```
rule OutPatternElement {
   from
      outPattern : MM_Hybrid!TargetElementRule
      (outPattern.refImmediateComposite().oclIsTypeOf(MM_Hybrid!Rule))
   to
      atl : ATL!SimpleOutPatternElement (
        varName <- outPattern.name_element.toLower()+'_out',
        type <- aType,
        bindings <- outPattern.included),

      aType : ATL!OclModelElement(
        name <- outPattern.name_element,
        model <- thisModule.resolveTemp(outPattern.metamodel,'ametamodeloutMM'))
}
```

Figura D-30. Regla de Transformación ATL – *OutPatternElement*

En la Figura D-31 se muestra la transformación entre los elementos de tipo *ElementIncluded*, que son los elementos que incluidos dentro de un elemento *OutElement*, y los elementos de tipo *Binding*, que representan las relaciones entre los elementos origen y destino en una regla ATL.

El elemento *ElementIncluded* del meta-modelo M-LTH esta formado por elementos de tipo *LeftPattern* y elementos de tipo *RightPattern* que se corresponden con el lado izquierdo y derecho de una regla (*LeftPattern* ← *RightPattern*). A su vez, en ATL un elemento (lado izquierdo de la regla) puede estar formado a partir de varios elementos, así por ejemplo, se puede acceder a las propiedades del elemento del modelo de origen (*SourceElementRule.property*) o invocar una operación (*SourceElementRule.operation)*, de la misma manera se puede invocar a un elemento de tipo *TargetElementRule* o asignar un valor concreto. Para permitir cada una de estas posibilidades, en la propiedad *value* del elemento *Binding* se invoca al *helper getBindingSource()*.

```
rule Bindings {
  from
   elemInc : MM_Hybrid!ElementIncluded
  to
   atl : ATL!Binding (
   --Left side of formula, that will receive the value
    propertyName <- elemInc.left.targetElement.asSequence().first().name_element,
   --Right side of formula, that has the value - issues
    value <- elemInc.getBindingSource())
}
```

Figura D-31. Regla de Transformación ATL – *Binding*

La Figura D-32 muestra el *helper getBindingSource()* que se puede invocar a partir de un elemento de tipo *ElementIncluded*. Este *helper* permite manejar todas las posibles situaciones comentadas anteriormente. De esta manera, para cada una de las situaciones se realizan operaciones diferentes. Por ejemplo, si el valor de la propiedad *typeAttribute* del elemento de tipo *ElementIncluded* es *"#concatenation"* se invoca a la regla *CreateConcatBinding()* que permite realizar la concatenación de los elementos que recibe como parámetros.

```
helper context MM_Hybrid!ElementIncluded def : getBindingSource() : ATL!OclExpression =
  if self.typeAttribute = #"concatenation" then
    thisModule.CreateConcatBinding(self,
    self.right.sourceElement.asSequence()->collect(i | i.name_element))
  else
    if (self.right."rule".asSequence().first().oclIsUndefined() and
      self.right.operation.asSequence().first().oclIsUndefined() and
      self.right.sourceElement.asSequence().first().oclIsUndefined() and
      self.right.reference.oclIsUndefined()) then
            thisModule.getConcreteBinding(self)
    else
      if (not self.right.reference.oclIsUndefined()) then
        if self.right.reference.oclIsTypeOf(MM_Hybrid!SourceElementRule) then
            thisModule.getComplexBinding(self)
        else
            thisModule.getSimpleBinding(self)
        endif
      else
            thisModule.getComplexBinding(self)
      endif
    endif
  endif;
```

Figura D-32. Regla de Transformación ATL – *getBindingSource()*

En la Figura D-33 se muestra la regla *CreateConcatBinding* que permite la creación de elementos concatenados. La regla se define como una regla de tipo *CalledRule*, por lo que debe ser invocada explícitamente para que se ejecute. Como se puede observar, en primer lugar se crea un elemento de tipo *OperatorCallExp* al que se le asigna el símbolo "+"; en el segmento imperativo de la regla se realiza la concatenación de cada uno de los elementos incluidos en el *RightPattern*, y que recibe como parámetro, con el símbolo "+" creado anteriormente, de esta manera el resultado tendrá la forma de: *elemento1* + *elemento2*.

```
rule CreateConcatBinding (elemInc : MM_Hybrid!ElementIncluded, attrRefs : Sequence(String)){
  to
   operation : ATL!OperatorCallExp (
            operationName <- '+')
  do {
    operation.source <- thisModule.CreateReferredConcatElement(elemInc, attrRefs.first());
      if(attrRefs->size() = 2) {
        operation.arguments <- thisModule.CreateReferredConcatElement
        (elemInc, attrRefs->last());}
      else {
        operation.arguments <- thisModule.CreateConcatBinding(elemInc,
        attrRefs->subSequence(2, attrRefs->size()));}
    operation;}
}
```

Figura D-33. Regla de Transformación ATL – *CreateConcatBinging*

En la Figura D-34 se muestra la regla de transformación entre los elementos de tipo *Operation* y los elementos de tipo *Helper*. Además se generan elementos de tipo *OclFeatureDefinition*, que se asigna a la propiedad *definition* y establece el contexto en el cual se ejecuta el *helper*, y elementos de tipo

Operation, que definen la operación a realizar indicando el valor de retorno de la misma.

```
rule createOperation2Helper {
   from
    oper : MM_Hybrid!Operation
   to
    atl : ATL!Helper (
     "module" <- oper.refImmediateComposite(),
     definition <- adefinition,
     commentsBefore <- Set {'-- Comments -> This is a Helper: ' +
     oper.name_operation},
     commentsAfter <- Set {'-- Body: ' + oper.body}),

    adefinition : ATL!OclFeatureDefinition (
      feature <- afeature,
      context_ <- oper.getContext()),

     afeature : ATL!Operation (
      name <- oper.name_operation,
      returnType <- oper.getReturnType())
}
```

Figura D-34. Regla de Transformación ATL – *createOperation2Helper*

La Figura D-35 muestra la definición del *helper getContext()* que permite recuperar el contexto, en el cual se aplica un *helper* definido.

```
helper context MM_Hybrid!Operation def : getContext () : ATL!OclExpression =
      if self."context".oclIsUndefined() then
            OclUndefined
      else
            thisModule.getSimpleContext(self)
      endif;
```

Figura D-35. Regla de Transformación ATL – *getContext()*

En la Figura D-36 se muestra la definición del *helper getReturnType()* que permite recuperar el valor de retorno que se asigna a un *helper* en particular. Como se puede observar, se evalúa el valor de la propiedad *datatype* del elemento origen *Operation* y se devuelve el correspondiente valor del meta-modelo destino. Es conveniente mencionar que el código ATL que se genera a partir del elemento de tipo *Helper* no es completamente implementable, ya que no es posible obtener la implementación de la función a realizar dentro del *helper* sin contaminar el modelo de nivel anterior con cuestiones propias de la implementación del lenguaje de transformación de modelos que se utiliza a nivel PDM, en este caso el lenguaje ATL.

```
helper context MM_Hybrid!Operation def : getReturnType () : ATL!OclExpression =
      if self.returnType.oclIsUndefined() then
            OclUndefined
      else
            if self.returnType.datatype = #"String" then
                  thisModule.getReturnStringType(self)
            else
                  if self.returnType.datatype = #"Integer" then
                        thisModule.getReturnIntegerType(self)
                  else
                        if self.returnType.datatype = #"Boolean" then
                              thisModule.getReturnBooleanType(self)
                        else
                              thisModule.getReturnElementType(self)
                        endif
                  endif
            endif
      endif;
```

Figura D-36. Regla de Transformación ATL – *getReturnType*

D.3 Transformaciones de M-LTH a RubyTL

En el capítulo 3 se ha presentado el proceso de especificación de las reglas de transformación entre el meta-modelo de transformación espécifico de plataforma siguiendo la aproximación híbrida (M-LTH) y el meta-modelo de RubyTL para el modelado de las transformaciones de forma dependiente de plataforma. En la Tabla D-3 se muestra dicha especificación.

Tabla D-3. Transformaciones de M-LTH a RubyTL

Meta-modelo M-LTH		**Meta-modelo de RubyTL de Nivel PDM**	
Module		*Transformation*	
InMetaModel		*MetaModel(Input)*	
OutMetaModel		*MetaModel (Output)*	
Rule	*isMain = true y in= 1*	*Rule*	*TopRule*
	isMain = true y in> 1		*TopRule (uso del método allObjects en Rule.filter)*
	isMain = false, typeAttribute <>#unique y in= 1		*CopyRule*
	isMain = false, typeAttribute <>#unique y in> 1		*CopyRule (uso del métdo allObjects en Rule.filter)*
	(isMain = false y typeAttribute = #unique y in = 1)		*NormalRule*

Meta-modelo M-LTH		Meta-modelo de RubyTL de Nivel PDM	
	(isMain = false y *typeAttribute = #unique y in > 1)*		*NormalRule (uso del métdo allObjects en Rule.filter)*
	in = 0		*Método Estático de Ruby*
SourceElementRule		*FromElement*	
TargetElementRule		*ToElement*	
ElementIncluded *- LeftPattern* *- RightPattern*		*Binding* *- ExpGet (Left side)* *- ExpGet (Right side)*	
Operation *- Return.datatype*		*Decorator* *- Decorator.body + dataType. toString()*	

A continuación se muestra como se han implementado cada una de las reglas usando el lenguaje ATL, a modo de establecer un orden en la presentación de las reglas se sigue el establecido en la Tabla D-3.

En la Figura D-37 se muestra la cabecera del módulo ATL en el que se definen las reglas de transformación, como se puede verificar se define como meta-modelo origen (*from*) al meta-modelo MM_Hybrid y como meta-modelo destino (*create*) al meta-modelo RubyTL.

```
module Hybrid2RubyTL;
create OUT : RubyTL from IN : MM_Hybrid;
```

Figura C-37. Regla de Transformación ATL – *Hybrid2RubyTL*

Una vez definida la cabecera del módulo que contiene las reglas de transformación se comienza con la implementación de cada una de las reglas

La Figura D-38 muestra la regla que implementa la transformación entre los elementos del tipo *Module* y los elementos de tipo *Transformation*. Se completan la propiedad *name* y las referencias: *sourceMetamodels*, que indica los meta-modelos origen que participan en la transformación, *targetMetamodels*, que indica los meta-modelos destino de la relación y *rules*, que representa las reglas de transformación que forman parte del módulo y *decorators*, que representa las funciones que pueden definirse.

```
rule Module {
   from
      mm_hybrid : MM_Hybrid!Module
   to
      rubytl : RubyTL!Transformation (
        name <- mm_hybrid.name_module,
        sourceMetamodels <- mm_hybrid.inMM,
        targetMetamodels <- mm_hybrid.outMM,
        rules <- mm_hybrid."rule",
        decorators <- mm_hybrid.operations)
}
```

Figura D-38. Regla de Transformación ATL – *Module*

Para la transformación de los elementos de tipo meta-modelo origen, *InMetaModel*, y meta-modelo destino, *OutMetaModel*, su definen dos reglas (Figura D-39) que generan los correspondientes elementos de tipo *Metamodel* en el modelo destino conforme al meta-modelo RubyTL.

```
rule inMM{
   from
      inMM_hybrid : MM_Hybrid!InMetaModel
   to
      inMM_rubytl : RubyTL!Metamodel(
            name <- inMM_hybrid.name_mm)
}

rule outMM{
   from
      outMM_hybrid : MM_Hybrid!OutMetaModel
   to
      outMM_rubytl : RubyTL!Metamodel(
            name <- outMM_hybrid.name_mm)
}
```

Figura D-39. Regla de Transformación ATL – *inMM* y *OutMM*

La meta-clase *Rule* en RubyTL es de tipo abstracta, y se implementa por tres meta-clases diferentes: *TopRule*, *NormalRule* y *CopyRule*. Cada una de estas meta-clases representa el tipo de reglas que es posible especificar con RubyTL. Debido a esto, la transformación de los elementos de tipo *Rule* del meta-modelo M-LTH a sus correspondientes elementos de tipo *Rule* en RubyTL se realiza por

medio de tres tipos de reglas diferentes, una por cada tipo de regla, de la misma manera que en el caso de la transformación a ATL, es necesario especificar correctamente la condición de guarda en cada una de las reglas.

Cuando se comenzaron a codificar las reglas de transformación entre el meta-modelo M-LTH de nivel PSM y el meta-modelo de RubyTL a nivel PDM, ha sido necesario tener en cuenta algunas restricciones propias del meta-modelo de RubyTL, como, por ejemplo, el hecho de que RubyTL no soporta la definición de múltiples elementos de origen en un elemento de tipo Rule. A nivel PSM, en el modelo de la transformación conforme al meta-modelo M-LTH se puede definir un elemento de tipo *Rule* que tenga más de un *SourceElementRule*. Esta definición debe ser respetada en el modelo de la transformación conforme al meta-modelo de RubyTL, por lo que ha sido necesario definir dos tipos de reglas de transformación diferentes, una para el caso donde tenga solo un elemento de tipo *SourceElementRule* dependiendo del elemento *Rule* (Figura D-40) y otro para el caso de que tenga más de un elemento de tipo *SourceElementRule* (Figura D-41). De manera similar, se procede con el resto de las reglas que permiten la transformación del elemento *Rule*.

Como se puede observar en la Figura D-40, donde se transforma el elemento *Rule* a un elemento *TopRule*, en la condición de guarda se evalúa el que el valor de la propiedad *isMain* del elemento sea *true*. Además, se invoca una función especial, el *helper getSizeIP*, que verifica que la cantidad de elementos del tipo *SourceElementRule* que dependen del elemento *Rule*, sea igual a 1. Si se cumple la condición de guarda, entonces se crea un elemento de tipo *TopRule* y un elemento de tipo *Mapping*, donde se agruparan los sub-elementos que compongan la regla.

Para el elemento de tipo *TopRule* se establecen los valores de las propiedades *name*, que se corresponde con el nombre de la regla; *from*, que indica el elemento origen a partir de los que se activa la regla; *to*, que indica los elementos destino que genera la regla; *comment*, donde se puede escribir un comentario y *mapping*, que agrupa las propiedades del elemento destino.

```
rule createRule2TopRule{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()=1 and
     mm_hybrid_rule.isMain=true)
   to
     rubytl : RubyTL!TopRule (
        name <- mm_hybrid_rule.name_rule,
        "from" <- mm_hybrid_rule."in".asSequence().first(),
        "to" <- mm_hybrid_rule.out.asSequence(),
        comment <- mm_hybrid_rule.getComment(),
        mapping <- amapping),

     amapping : RubyTL!Mapping (
        bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-40. Regla de Transformación ATL – *createRule2TopRule*

En la Figura D-41 se muestra la transformación del elemento de tipo *Rule* a un elemento de tipo *TopRule* cuando tengo más de un elemento *SourceElementRule* en *Rule* (*getSizeIP > 1*). La diferencia con el caso anterior es que es necesario crear un elemento de tipo *Filter*, donde se realice la comprobación de la existencia de todos los elementos de tipo *SourceElementRule*, para esto se define un *helper getFilterMultiIN()*.

```
rule createRule2TopRuleMulti{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()>1
     and mm_hybrid_rule.isMain=true)
   to
     rubytl : RubyTL!TopRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      filter <- afilter,
      mapping <- amapping),

     afilter : RubyTL!Filter(
      expression <- mm_hybrid_rule.getFilterMultiIN()),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-41. Regla de Transformación ATL – *createRule2TopRuleMulti*

En la Figura D-42 se muestra la regla de transformación entre elementos de tipo *Rule* y elementos de tipo *CopyRule* cuando el valor que retorna el *helper getSizeIP()* es igual a uno. Además, como se puede observar, se comprueba si el valor de la propiedad *isMain* es igual a *False* y el valor de la propiedad *typeAttribute* es distinto de *IsUnique*. El resto de las propiedades se completan de manera similar a la regla *createRule2TopRule*.

```
rule createRule2CopyRule{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()=1
     and mm_hybrid_rule.isMain=false
     and not(mm_hybrid_rule.typeAttribute = #"unique"))
   to
     rubytl : RubyTL!CopyRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      mapping <- amapping),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-42. Regla de Transformación ATL – *creatRule2CopyRule*

La Figura D-43 muestra la transformación entre elementos de tipo *Rule* y elementos de tipo *CopyRule* cuando el valor que retorna el *helper getSizeIP()* es mayor a uno. Además, se comprueba que el valor de la propiedad *isMain* sea igual a *False* y que el valor de la propiedad *typeAttribute* sea distinto de *IsUnique*. El resto de las propiedades se completan de manera similar a la regla *createRule2TopRuleMulti*.

```
rule createRule2CopyRuleMulti{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()>1
     and mm_hybrid_rule.isMain=false
     and not(mm_hybrid_rule.typeAttribute = #"unique"))
   to
     rubytl : RubyTL!CopyRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      filter <- afilter,
      mapping <- amapping),

     afilter : RubyTL!Filter(
      expression <- mm_hybrid_rule.getFilterMultiIN()),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-43. Regla de Transformación ATL – *creatRule2CopyRuleMulti*

La Figura D-44 muestra la transformación entre elementos de tipo *Rule* y elementos de tipo *NormalRule* cuando el valor que retorna el *helper getSizeIP()* es igual a uno. Además, se comprueba que el valor de la propiedad *isMain* sea igual a

False y que el valor de la propiedad *typeAttribute* sea igual a *IsUnique*. Las propiedades se generan de manera similar a la regla *createRule2TopRule*.

```
rule createRule2NormalRule{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()=1
     and mm_hybrid_rule.isMain=false
     and mm_hybrid_rule.typeAttribute = #"unique")
   to
     rubytl : RubyTL!NormalRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      mapping <- amapping),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-44. Regla de Transformación ATL – *creatRule2NormalRule*

La Figura D-45 muestra la transformación entre elementos de tipo *Rule* y elementos de tipo *NormalRule* cuando el valor que retorna el *helper getSizeIP()* es mayor a uno. Además, se comprueba que el valor de la propiedad *isMain* sea igual a *False* y que el valor de la propiedad *typeAttribute* sea igual a *IsUnique*. El resto de las propiedades se completan de manera similar a la regla *createRule2TopRuleMulti*.

```
rule createRule2NormalRuleMulti{
   from
     mm_hybrid_rule : MM_Hybrid!Rule (mm_hybrid_rule.getSizeIP()>1
     and mm_hybrid_rule.isMain=false
     and mm_hybrid_rule.typeAttribute = #"unique")
   to
     rubytl : RubyTL!NormalRule (
      name <- mm_hybrid_rule.name_rule,
      "from" <- mm_hybrid_rule."in".asSequence().first(),
      "to" <- mm_hybrid_rule.out.asSequence(),
      comment <- mm_hybrid_rule.getComment(),
      filter <- afilter,
      mapping <- amapping),

     afilter : RubyTL!Filter(
      expression <- mm_hybrid_rule.getFilterMultiIN()),

     amapping : RubyTL!Mapping (
      bindings <- mm_hybrid_rule.out.asSequence()->collect(i | i.included))
}
```

Figura D-45. Regla de Transformación ATL – *creatRule2NormalRuleMulti*

La Figura D-46 muestra la transformación entre elementos de tipo *SourceElementRule* y elementos de tipo *FromElement*. Estos elementos se corresponden con los elementos origen de las reglas de transformación.

```
rule source2from{
   from
     inPattern : MM_Hybrid!SourceElementRule
     (inPattern.refImmediateComposite().oclIsTypeOf(MM_Hybrid!Rule))
   to
     fromElement : RubyTL!FromElement (
      name <-  inPattern.name_element.toLower()+'_in',
      classname <- inPattern.name_element,
      metamodel <- thisModule.resolveTemp(inPattern.metamodel,'inMM_rubytl'))
}
```

Figura D-46. Regla de Transformación ATL – *source2from*

En la Figura D-47 se muestra la regla de transformación entre elementos de tipo *TargetElementRule* y elementos de tipo *ToElement*. Estos elementos se corresponden con los elementos destino de las reglas de transformación.

```
rule target2to{
   from
     outPattern : MM_Hybrid!TargetElementRule
     (outPattern.refImmediateComposite().oclIsTypeOf(MM_Hybrid!Rule))
   to
     toElement : RubyTL!ToElement (
      name <-  outPattern.name_element.toLower()+'_out',
      classname <- outPattern.name_element,
      metamodel <- thisModule.resolveTemp(outPattern.metamodel,'outMM_rubytl'))
}
```

Figura D-47. Regla de Transformación ATL – *target2to*

En la Figura D-48 se muestra la regla de transformación que permite generar elementos de tipo *Binding* a partir de elementos de tipo *ElementIncluded*. Como se puede observar se generan además elementos de tipo *ExpGet*, *ExpVariable*, y *ToElement*.

```
rule Bindings {
   from
     elemInc : MM_Hybrid!ElementIncluded
   to
     rubytl : RubyTL!Binding (
      --Right side of formula, that has the value - issues
      source <- elemInc.defineBinding(),
      --Left side of formula, that will receive the value
      target <- atargetvalue),

     atargetvalue : RubyTL!ExpGet(
      --property of target
      property <- elemInc.left.targetElement.asSequence().first().name_element,
      source <- asourcename),

     asourcename : RubyTL!ExpVariable(
      variable <- avariabletrg),

     avariabletrg : RubyTL!ToElement (
      name <- elemInc.refImmediateComposite().name_element.toLower()+'_out')
}
```

Figura D-48. Regla de Transformación ATL – *Bindings*

En la Figura D-49 se muestra la transformación entre elementos de tipo *Operation* en elementos de tipo *Decorator*, completándose las propiedades *name*, con el valor *name_operation*, *body*, con el valor *body_operation* más el valor de retorno de la misma y *context*, con el valor del elemento a partir del cual se ejecuta el *Decorator*. Al igual que con el código en ATL, el código RubyTL que se genera a partir del elemento de tipo *Decorator* no es completamente implementable, ya que no es posible obtener la implementación de la función a realizar dentro del mismo sin contaminar el modelo de nivel anterior con cuestiones propias de la implementación del lenguaje de transformación de modelos que se utiliza a nivel PDM, en este caso el lenguaje RubyTL.

```
rule createOperation2Decorator {
   from
     oper : MM_Hybrid!Operation
   to
     rubytl : RubyTL!Decorator (
      name <- oper.name_operation,
      body <- oper.body + oper.getReturnType(),
      "context" <- acontext),

     acontext : RubyTL!FromElement (
      classname <-  oper."context".name_element,
      name <- oper."context".name_element.toLower()+'_in',
      metamodel <- ametamodel),

     ametamodel : RubyTL!Metamodel(
      name <- oper."context".metamodel.name_mm)
}
```

Figura D-49. Regla de Transformación ATL – *createOperation2Decorator*

Apéndice E: Manual de Usuario de MeTAGeM

En esta sección se presenta el manual de usuario de MeTAGeM, la herramienta que permite realizar el modelado de transformaciones en un alto nivel de abstracción aplicando los principios de MDE.

Para ilustrar cada uno de los pasos que se deben seguir en el modelado de las transformaciones utilizando MeTAGeM se utiliza como caso de estudio el modelado de las transformaciones entre el meta-modelo de SQL:2003 [98] y el meta-modelo ORDB para el producto espercífico de Oracle10g [165], en el ámbito de las bases de datos Objeto Relacionales.

E.1 Requisitos e Instalación del Sistema

A continuación se detallan los pre-requisitos necesario para el correcto funcionamiento de MeTAGeM y el proceso de instalación.

E.1.1 Requisitos

Para poder instalar la versión 1.0 de MeTAGeM es necesario:

- Tener instalada la máquina virtual de JAVA versión 1.6, que se puede descargar en: *http://www.java.com/es/download/index.jsp*.
- Tener instalado el intérprete del lenguaje de programación Ruby, que se puede descargar en: *http://rubyforge.org/frs/download.php/18566/ruby186-25.exe*
- Por último, es necesario tener en cuenta que Eclipse debe tener instalados los siguientes *plug-ins*:
    - Age 0.3.4, disponible en http://gts.inf.um.es/trac/age
    - AM3 0.4.0, disponible en http://www.eclipse.org/gmt/am3/download
    - AMW 1.0.0, disponible en http://www.eclipse.org/gmt/amw/download
    - ATL 2.0.0, disponible en http://www.eclipse.org/atl/download
    - EMF 2.4.0, disponible en http://www.eclipse.org/modeling/emf/downloads
    - Epsilon 0.8.6, disponible en http://www.eclipse.org/gmt/epsilon/download

- RubyTL 0.3.2, disponible en http://rubyforge.org/projects/rubytl
- TCS 0.8.0, disponible en http://atlanmod.emn.fr/www/download/updates/tcs

E.1.2 Instalación MeTAGeM

En cuanto a la instalación de MeTAGeM existen dos posibilidades:

a) se pueden bajar los *plug-ins* de MeTAGeM junto con Eclipse, como se explicará a continuación, o

b) si ya se tiene un Eclipse funcionando, se puede bajar sólo los *plug-ins* de MeTAGeM del sitio http://www.kybele.etsii.urjc.es/members/vbollati/thesis/software/plug-insMeTAGeM, asegurándose de tener instalados el resto de *plug-ins* necesarios para el correcto funcionamiento de MeTAGeM.

En caso de seleccionar la opción a) para la instalación de MeTAGeM, opción que se recomienda, se debe descargar el *framework* de Eclipse con los *plug-ins* de MeTAGeM incluidos de la siguiente dirección: http://www.kybele.etsii.urjc.es/members/vbollati/thesis/Software/EclipseAMMA_MeTAGeM.zip.

Esta versión de Eclipse, tiene incluidos además todos los *plug-ins* necesarios para el correcto funcionamiento de MeTAGeM, detallados anteriormente.

Una vez finalizada la descarga, sólo se debe descomprimir el archivo, con lo cual obtendremos dos carpetas, una llamada *eclipse* y otra llamada *RubyTL*.

MeTAGeM soporta la generación de reglas de transformación conformes al lenguaje RubyTL, por lo que para poder ejecutar dichas reglas de transformación es necesario tener instalado el intérprete del lenguaje de programación Ruby. Para instalar dicho intérprete se debe ejecutar el archivo *ruby186-25.exe* que se encuentra en la carpeta RubyTL y seguir las instrucciones de instalación.

Una vez finalizado el proceso de instalación de Ruby, se debe configurar en Eclipse el intérprete de Ruby. Para esto, inicializamos Eclipse, ejecutando dentro de la carpeta Eclipse, el archivo *eclipse.exe*. Una vez inicializado Eclipse, se debe configurar la ruta donde se encuentra instalado el intérprete de Ruby. Para ello, en el menú *Windows→Preferences→Ruby→Intalled Interpreters* se debe añadir un nuevo intérprete e indicar la ruta de instalación del mismo (Figura E-1).

Este procedimiento se debe hacer una sola vez, cuando se inicia Eclipse por primera vez.

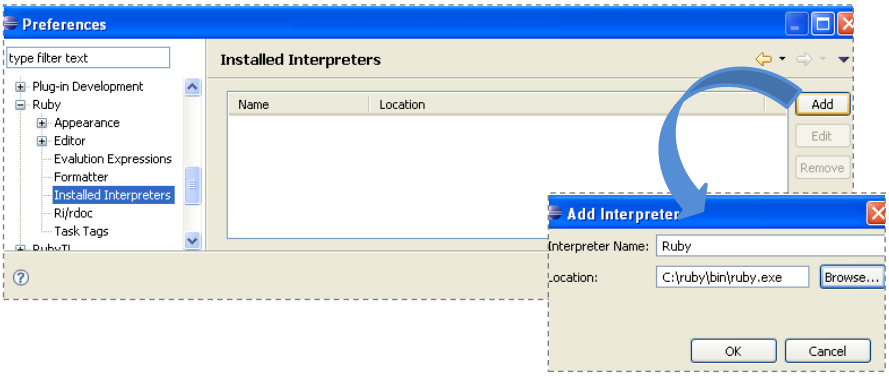

Figura E-1. Configuración Intérprete de Ruby

Una vez finalizada la instalación, se puede observar que en el espacio de trabajo (*workspace*) de Eclipse existen dos proyectos, *Class2Table* y *Families2Person*, cada uno de ellos representa un caso de estudio sencillo que muestra el funcionamiento de MeTAGeM, el primero muestra la generación de código en el lenguaje RubyTL y el segundo en el lenguaje ATL.

Una vez finalizada la configuración de Eclipse, se puede comenzar con la definición de nuevos proyectos, para modelar las transformaciones.

MeTAGeM permite realizar el modelado de las transformaciones en cuatro niveles de abstracción tal y como se detalla en el capítulo 3: a nivel independiente de plataforma, donde el usuario debe indicar las relaciones existentes entre los elementos de los meta-modelos que intervienen en la transformación; a nivel espécifico de plataforma, siguiendo la aproximación híbrida; a nivel dependiente de plataforma, donde se puede seleccionar como lenguaje de transformación ATL o RubyTL y, por último, la generación de código que implementa la transformación en el lenguaje seleccionado. A continuación se explica cómo realizar los modelos de transformación en cada uno de los diferentes niveles de abstracción.

E.2 Modelado de las Transformaciones Independientes de Plataforma

Para el modelado de las transformaciones independiente de plataforma, MeTAGeM propone el uso de un editor de modelos que permite representar las

relaciones existentes entre los elementos de los meta-modelos que intervienen en la transformación.

Para comenzar con el modelado de las transformaciones en este nivel, en primer lugar se debe definir un modelo conforme al meta-modelo de *weaving* extendido para MeTAGeM. Para esto, se debe crear un nuevo tipo de modelo a traves del menú *File→New→Other→Model Weaver→Weaving Model* (Figura E-2).

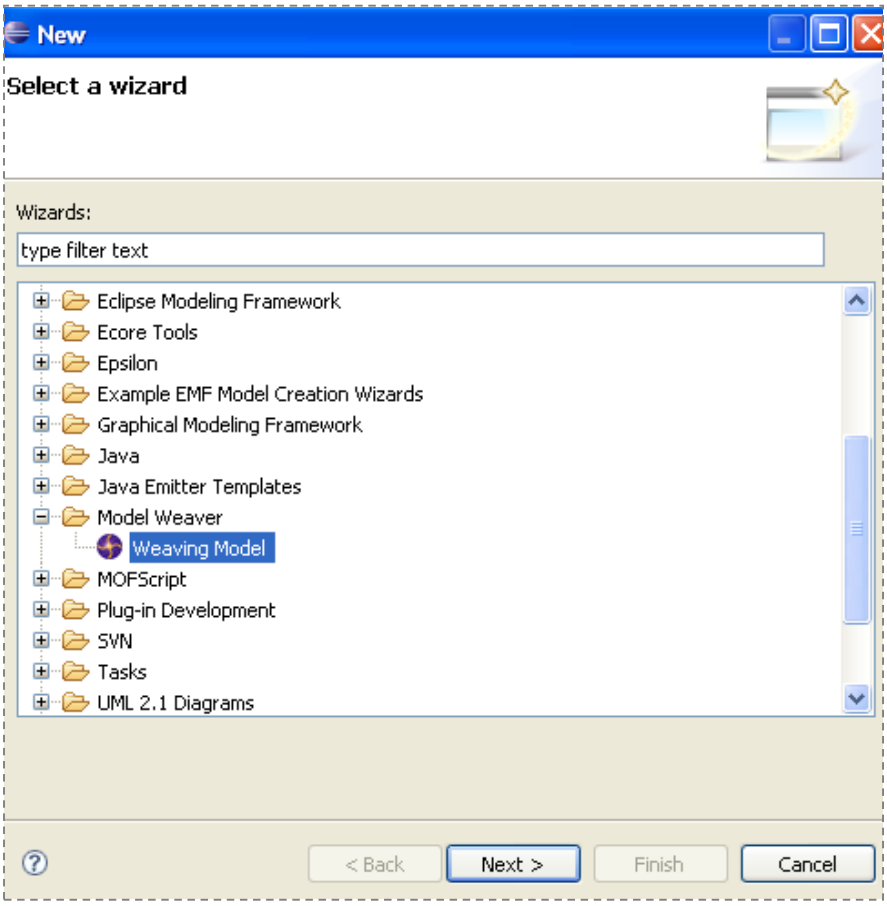

Figura E-2. Selección de *Wizard* para Modelos de *Weaving*

Una vez seleccionado el tipo de modelo a crear, el *plug-in* de *Weaver* proporciona un *Wizard* que guía en la definición inicial del modelo. En la Figura E-3 se muestra el proceso completo. En primer lugar se debe seleccionar el *plug-in* que se utilizará para la definición del modelo, en este caso se selecciona el *plug-in* de MeTAGeM. Después se debe indicar el lugar donde se almacenará el nuevo

modelo (*Container*), el nombre del fichero (*FileName*) y el tipo de panel en el que se quiere trabajar. En este caso, para el modelado de las transformaciones, se debe seleccionar la opción *TransformationWeavingPanelExtension*.

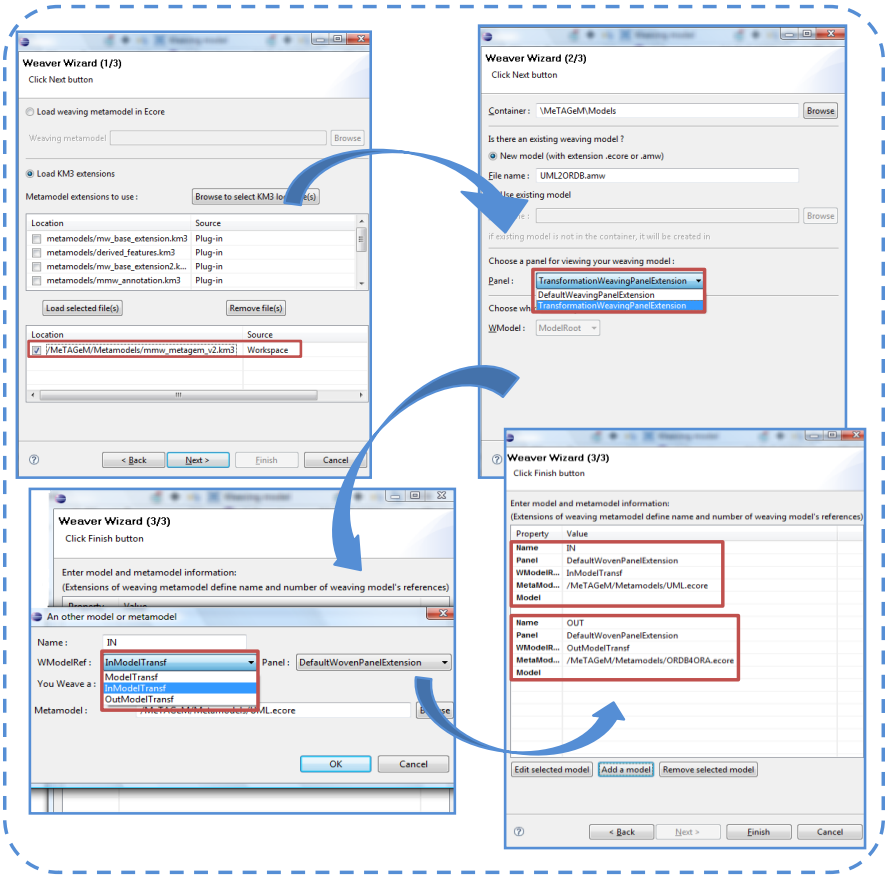

Figura E-3. Inicialización de Modelo de Transformación Independiente de Plataforma

El último paso para la creación del modelo, es la selección de los meta-modelos que participarán en la transformación, es decir, el meta-modelo origen y el meta-modelo destino. A partir de esto se obtiene el modelo donde se pueden comenzar a definir las relaciones entre los elementos. Como se puede ver en la Figura E-4 en el panel izquierdo se muestra el meta-modelo origen de la transformación, en este caso el elegido es el meta-modelo SQL2003; en el panel derecho se muestra el meta-modelo destino de la transformación, en este caso el meta-modelo ORDB4ORA; y en el centro se muestra el modelo donde se establecerán las relaciones entre los diferentes elementos de ambos meta-modelos.

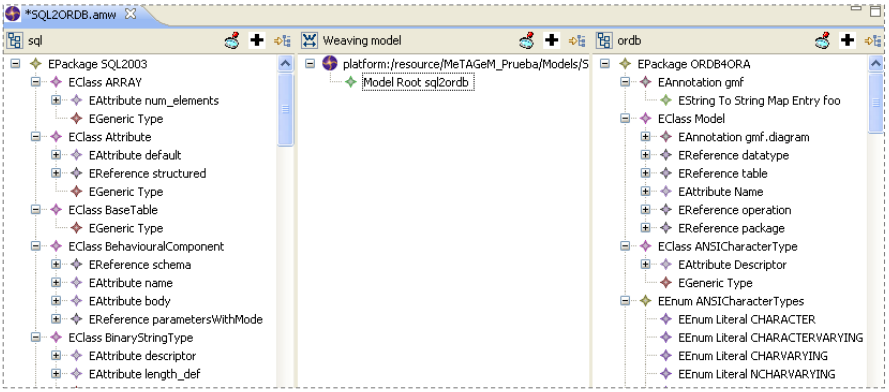

Figura E-4. Modelo de *Weaving*

El siguiente paso en la definición del modelo de transformación es la especificación de las diferentes relaciones entre los elementos de ambos meta-modelos. Para esto se debe hacer clic con el botón derecho sobre el elemento raíz del modelo (*Model Root sql2ordb*) y en la opción *New child* seleccionar el tipo de relación que se quiere especificar (Figura E-5).

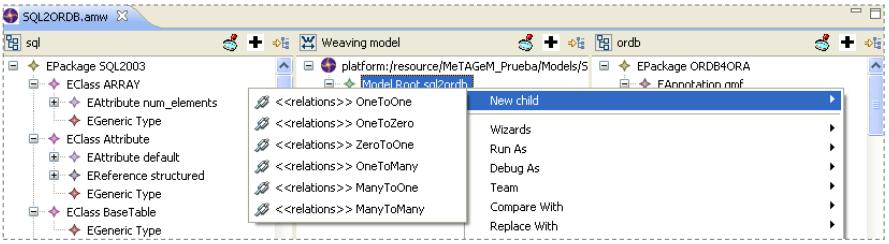

Figura E-5. Tipos de Relaciones

Según lo definido en el capítulo 3, MeTAGeM soporta diferentes tipos de relaciones que varian de acuerdo a la cardinalidad de los elementos de ambos meta-modelos que se relacionan. Así por ejemplo, el tipo de relación *OneToOne* sirve para indicar que un único elemento del meta-modelo origen se relaciona con un único elemento del meta-modelo destino, de la misma manera, la relación *ManyToOne* sirve para indicar que a partir de varios elementos del meta-modelo origen se genera un único elemento en el meta-modelo destino. A modo de ejemplo en la Figura E-6 se muestra la especificación de una relación *OneToOne*. Tras seleccionar el tipo de relación se deben indicar los elementos (*inElement*, *OutElement*) que participan en la relación. Para esto, se selecciona el elemento en el meta-modelo origen (*Schema*) y se arrastra hasta el elemento *relations* del modelo del centro, al momento de soltar el elemento se debe seleccionar el rol que

tiene en la relación (*InElement* en este caso). Es importante mencionar que al definir una relación de tipo *OneToOne* la herramienta sólo deja seleccionar un elemento origen y un elemento destino por lo que después de la selección se deshabilita la posibilidad de seguir agregando más elementos.

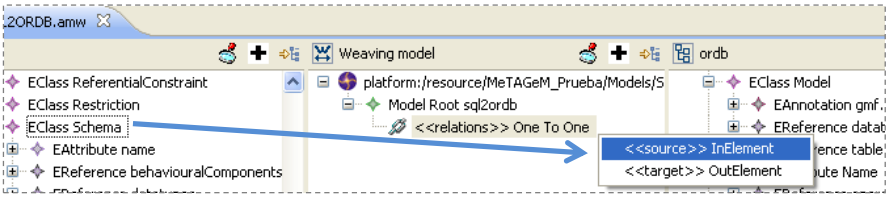

Figura E-6. Relación *OneToOne* - *InElement*

Como se puede ver en la Figura E-7, el elemento *OutElement*, del modelo que se está creando (*sql2orbd*), puede tener también relaciones que dependan de él. Esto es para representar las relaciones que existen entre las propiedades de los elementos de la relación padre de la que depende. Los tipos de relaciones que se pueden especificar en este nivel son un sub-conjunto de la totalidad de tipos de relaciones disponibles en MeTAGeM.

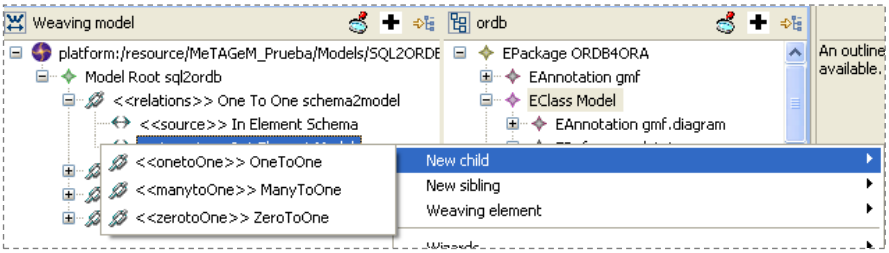

Figura E-7. Relación *OneToOne*, sub-relaciones

En la Figura E-8 se muestra el resultado de la especificación completa de la relación *OneToOne*, como se puede observar se han añadido tres elementos de tipo *OneToOne* que dependen de *OutElement* y que permiten establecer las relaciones entre las propiedades de los elementos *Schema* y *Model*. De esta manera, por ejemplo, se especifica que la propiedad *Name* del elemento *Model* se genera a partir de la propiedad *name* del elemento *Schema*.

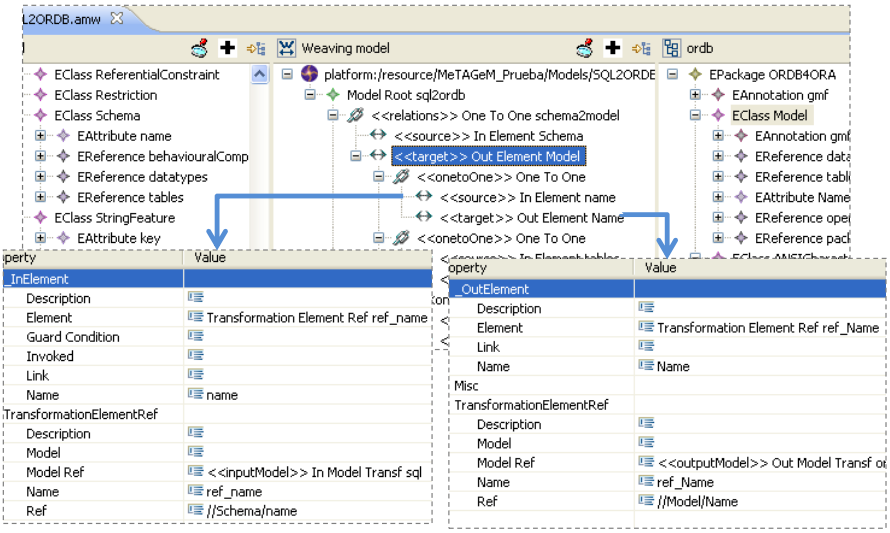

Figura E-8. Definición de Relaciones Dependientes de *OutElement*

De la misma manera se procede a la especificación de cada uno de los tipos de relaciones que proporciona MeTaGeM.

E.3 Modelado de las Transformaciones Específicas de Plataforma

A partir del modelo de nivel independiente de plataforma definido en la sección anterior, se obtiene el modelo de transformación específico de plataforma. Como se ha comentado anteriormente, actualmente, a nivel PSM MeTAGeM propone el modelado de las transformaciones siguiendo la aproximación híbrida.

Para obtener el modelo de la transformación a nivel PSM se debe hacer clic con el botón derecho sobre el modelo especificado a nivel PIM, *SQL2ORDB.amw*, y seleccionar *Run As → 1 MeTAGeM → Hybrid* (parte superior de la Figura D-9). Se debe mencionar que, en la opción *Run As*, la herramienta MeTAGeM solo muestra las opciones de transformaciones disponibles para el tipo de modelo sobre el que se quiere ejecutar la transformación, en este caso la única transformación disponible para un modelo con extención *.amw* es la *1 MeTAGeM → Hybrid*. Como resultado de esta transformación se obtiene un modelo conforme al meta-modelo de nivel PSM siguiendo la aproximación híbrida, *SQL2ORDB.mm_hybrid* (parte inferior de la Figura E-9).

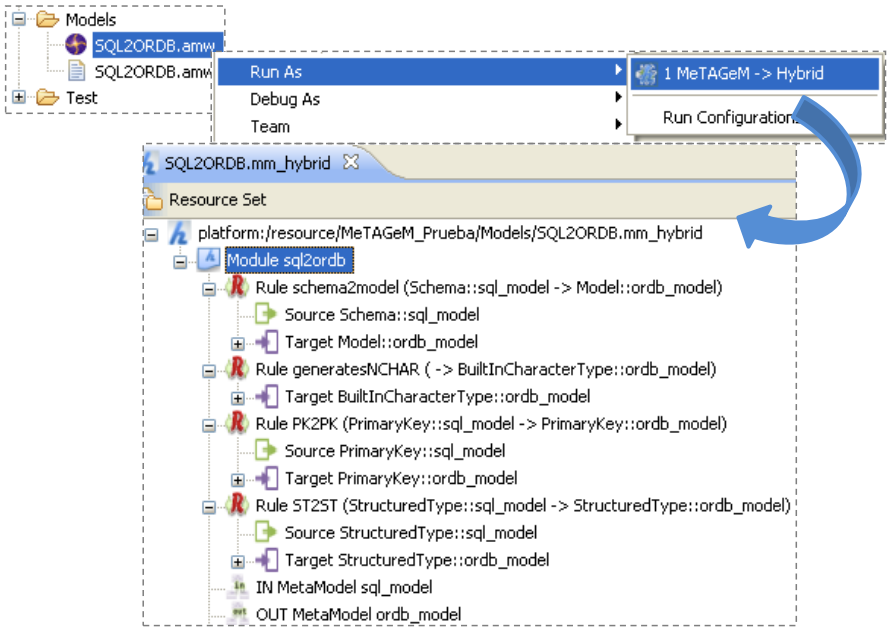

Figura E-9. Transformación *MeTAGeM → Hybrid*

La primera vez que se realiza la ejecución de la transformación es necesario configurar los modelos sobre los que se realiza la transformación. La Figura E-10 muestra los campos que se deben configurar: el modelo de la izquierda (*Left Model URI*), que en este caso será el meta-modelo de SQL:2003, y el modelo de la derecha (*Right Model URI*), que en este caso será el meta-modelo ORBD, ambos modelos son conformes a MOF.

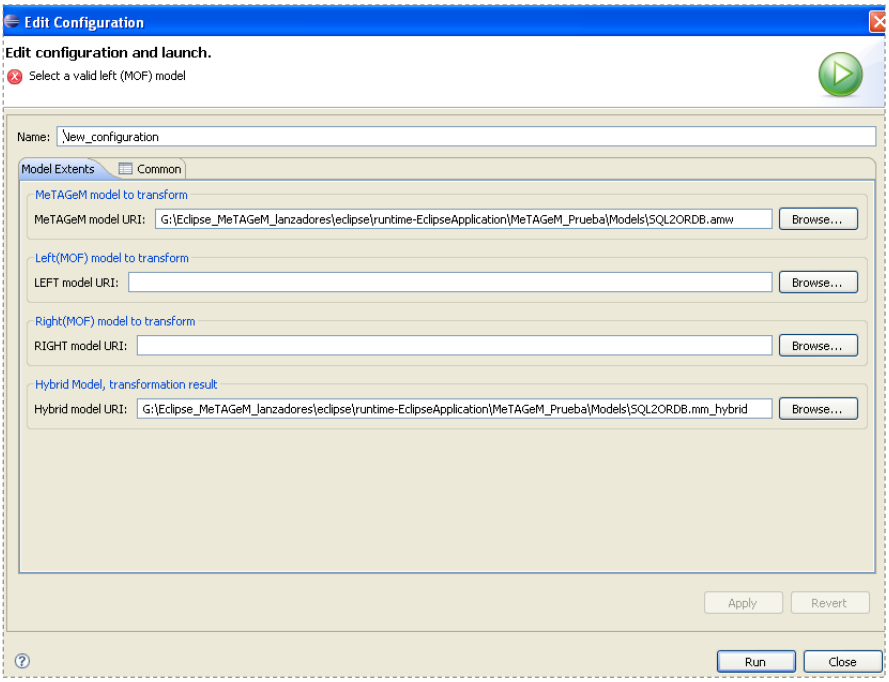

Figura E-10. Configuración Transformación *MeTAGeM→Hybrid*

MeTAGeM brinda la posibilidad de modificar el modelo generado a nivel PSM, ya sea agregando nuevos elementos, modificando los existentes o eliminando alguno de ellos.

Como se ha visto en la sección anterior, a nivel PIM sólo se establecen las relaciones entre los elementos de los diferentes meta-modelos, por lo que el modelo generado a nivel PSM sólo contiene elementos de tipo *Rule*. Pero muchas veces es necesario definir algún tipo de función especial que genere o recupere algún tipo de información. Esto se hace por medio de la creación de un elemento de tipo *Operation*.

En la Figura E-11 se muestra la forma de agregar un elemento de tipo *Operation*, para esto, se debe hacer clic con el botón derecho sobre el elemento raíz del modelo (*Module sql2ordb*) y seleccionar la opción *New Child → Operation*. Una vez creado el elemento se debe completar las propiedades: *Body*, donde se puede indicar cúal debería ser la función a implementar o simplemente agregar un comentario; *Context*, donde se indica a partir de que elemento es invocada la función; y *Name*, estableciendo simplemente el nombre de la función. Además se debe indicar cúal es el valor que se deberá obtener al ejecutar la función, para esto se debe hacer clic con el botón derecho sobre el elemento

Operation y se selecciona la opción de *Return Value*, indicándose si el valor de retorno es de tipo *DataType* o de tipo *Element*. En el ejemplo que se muestra en la Figura E-11 se crea un elemento de tipo *Operation*, a cuya propiedad *Name* se le asigna el valor de *"isBoolean"*; en la propiedad *Context* se asigna la referencia al elemento del modelo origen, *sql_model* y en la propiedad *Body* se escribe un comentario de lo que debe hacer la operación. Además se especifica que el valor de retorno de la mismas es de tipo *Boolean*.

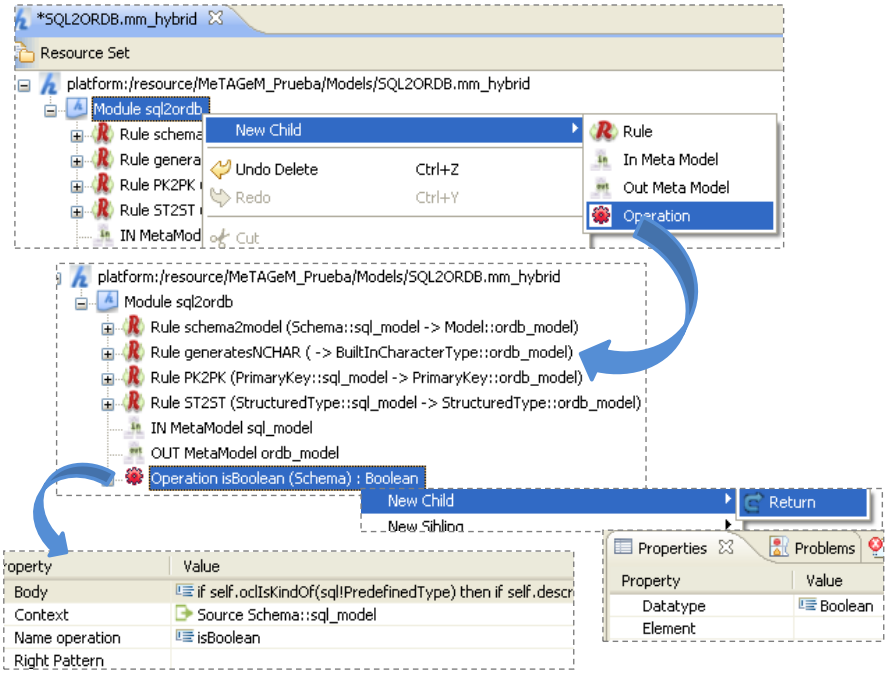

Figura E-11. Elemento de Tipo *Operation*

También se puede personalizar el comentario asociado a un elemento de tipo *Rule* creado. Para esto se selecciona el elemento en cuestión y en la propiedad *Comment* del mismo se introduce el comentario deseado (Figura E-12).

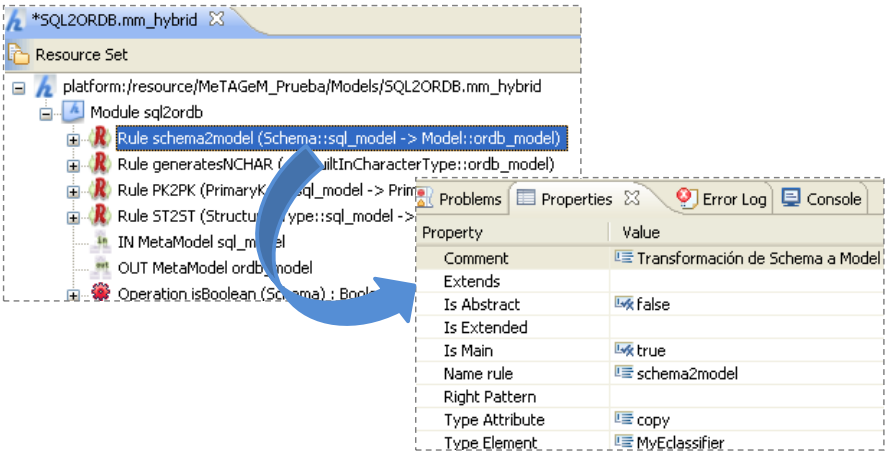

Figura E-12. Personalización de la propiedad *Comment*

Otra de las cosas que se debe personalizar es la propiedad *Reference* del elemento *RightPattern*. Esta propiedad permite establecer una referencia con otro elemento de tipo *Element* y se utiliza cuando el lenguaje en el que se va a implementar la transformación es ATL. Como esta es una propiedad que depende exclusivamente del lenguaje del nivel PDM no se puede recoger a nivel PIM, por lo que en caso de ser necesario se debe establecer su valor manualmente en el modelo de nivel PSM.

De acuerdo al tipo de *Element* al que se haga referencia varía el significado de la propiedad *Reference*:

- Cuando en una misma regla tengo más de un elemento de tipo *SourceElementRule*, en el momento que se definen los elementos de tipo *RightPattern* es necesario indicar con cúal de los elementos *SourceElementRule* está relacionado. Para esto se establece el valor de la propiedad *Reference* como una referencia al *SourceElementRule* correspondiente. Por ejemplo, en la parte izquierda de la Figura E-13, se muestra como se establece la propiedad *Reference* del elemento de tipo *RightPattern* con una referencia al elemento de tipo *SourcePattern NameMMIN*.
- Existen situaciones en las que un elemento es una referencia a un elemento generado dentro de la misma regla. Esto se puede modelar estableciendo el valor *Reference* del elemento *RightPattern* como una referencia al elemento de tipo *TargetElementRule*. Por ejemplo en la parte derecha de la Figura D-13 se muestra como se establece la propiedad *Reference* del elemento de tipo

RightPattern con una referencia al elemento de tipo *TargetPattern NameMMOUT*.

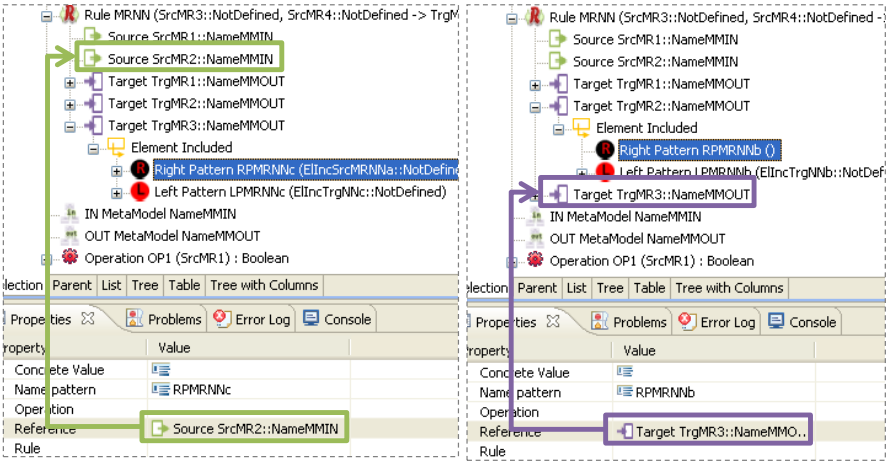

Figura E-13. Propiedad *Reference*

MeTAGeM proporciona dos formas de validación de los modelos generados: a) una validación individual del modelo, en este caso la validación debe ser realizada de forma explícita por el usuario, y, b) una validación automática que se realiza en el momento de la ejecución de la transformación; en este caso se valida el modelo origen de acuerdo al modelo destino elegido.

Para realizar la validación individual del modelo se debe hacer clic con el botón derecho sobre el elemento raíz del modelo y seleccionando la opción *Validate* (Figura E-14).

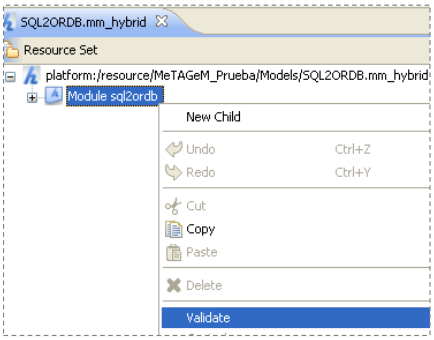

Figura E-14. Validación de los Modelos

Los errores detectados durante el proceso de validación se comunican por medio de mensajes, donde se describe el tipo de error. Al hacer clic en el botón

OK del mensaje, automáticamente la herramienta se posiciona en objeto que produce el error para que el mismo sea subsanado. Por ejemplo en la parte superior de la Figura E-15 se muestra un mensaje de error que indica que un elemento de tipo *RightPattern* debe tener por asociado algún elemento de tipo *SourceElementRule*, *Reference*, *Rule*, *Operation* o *ConcreteValue*, al hacer clic en el botón *OK*, se posicionaría en el elemento de tipo *RightPattern* y nos mostraría las propiedades del mismo (parte inferior de la Figura E-15).

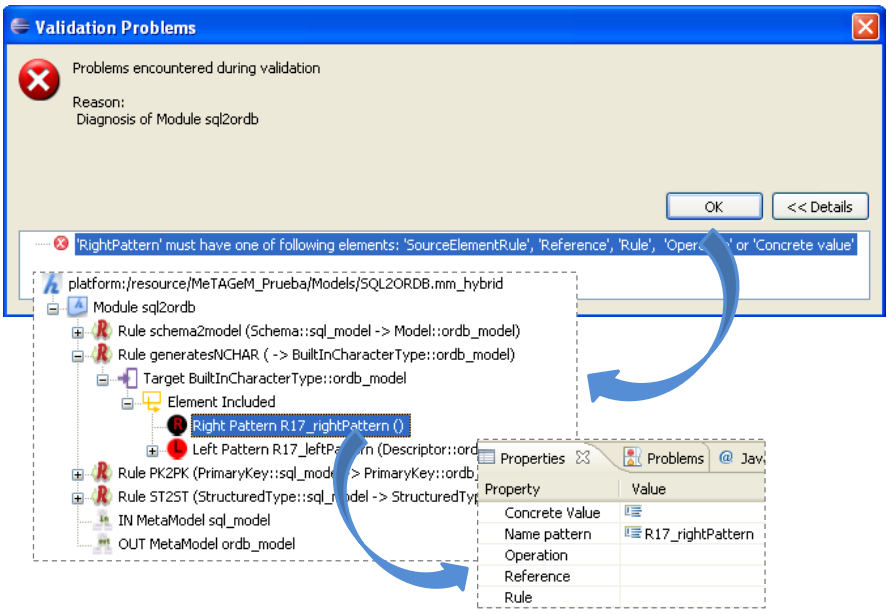

Figura E-15. Problemas de Validación

El proceso de validación del modelo consiste en controlar los siguientas características:

- Todos los elementos deben tener la propiedad *Name* definida, y el valor de la misma debe comenzar con una letra.
- El elemento de tipo *Module* debe contener al menos un meta-modelo origen y un meta-modelo destino.
- El elemento de tipo *Module* debe contener al menos un elemento de tipo *Rule*.
- Se deben definir los tipos de los meta-modelos origen y destino.
    - Las propiedades *isAbstract*, *isMain*, *typeAttribute* y *typeElement* de los elementos de tipo *Rule* deben estar definidas.

- Las propiedades *body*, *context* y *returnType* de los elementos de tipo *Operation* deben estar definidas.
- En los elementos de tipo *Return* debe estar definida la propiedad *datatype* o la propiedad *element*, pero ambas son excluyentes entre sí.
- Las propiedades *typeAttribute*, *typeElement*, *right* y *left* de los elementos de tipo *ElementIncluded* deben estar definidas.
- Los elementos de tipo *RightPattern* deben tener definido al menos una propiedad *Reference*, *Rule*, *Operation* o *sourceElementRule*. Además, se controla que no se puedan presentar las siguientes combinaciones: *Rule-Operation*, *Rule-Reference*, *Operation-Reference* o *SourceElementRule-Reference*. Por último, si la propiedad *ConcreteValue* está definida, el resto de las propiedades no deben estar definidas.

Una vez finalizadas todas las modificaciones necesarias o deseadas en el modelo del nivel PSM, se puede seguir con el procedimiento de modelado de la transformación realizando la transformación a modelos del nivel PDM.

E.4 Modelado de las Transformaciones Dependientes de Plataforma

A nivel PDM, actualmente, MeTAGeM propone el modelado de las transformaciones de acuerdo a dos lenguajes de transformación que siguen la aproximación híbrida: ATL y RubyTL. Por lo tanto, el usario debe seleccionar en que lenguaje desea implementar las transformaciones antes de realizar la transformación. Como se puede ver en la Figura E-16, al invocar la transformación, MeTAGeM proporciona dos opciones diferentes: *1 Hybrid → ATL* ó *2 Hybrid → RubyTL*.

Figura E-16. Transformación de PSM a PDM

E.4.1 Transformación de modelo a nivel PSM a modelo ATL

Si selecciona la opción *1 Hybrid→ATL* se obtiene un modelo de transformación conforme al lenguaje ATL.

Antes de ejecutarse la transformación se realiza de forma automática el proceso de validación del modelo definido a nivel PSM, teniendo en cuenta que el modelo destino es conforme a ATL. En este, caso la validación consiste en controlar las siguientes restricciones:

- Los elementos de tipo *Operation* deben tener definido un *returnType* y la propiedad *Context*.
- Si los elementos de tipo *Rule* no contiene ningún elemento de tipo *SourceElement*, los elementos de tipo *RightPattern* solo pueden tener definida la propiedad *ConcreteValue*.

La primera vez que se ejecuta la transformación es necesario indicar la ruta en la cual se almacenará el modelo generado por la transformación y el nombre de la misma (Figura E-17). Al hacer clic en el botón *Run* se genera el modelo destino (*SQL2ORDB-atl.ecore*) en la ubicación seleccionada.

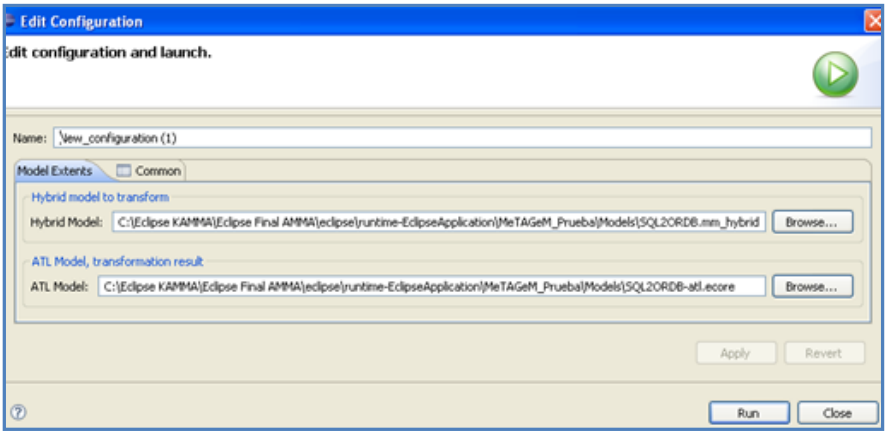

Figura E-17. Configuración de la Transformación

En la Figura E-18 se muestra el modelo de transformación, conforme al lenguaje ATL, resultante de la transformación realizada.

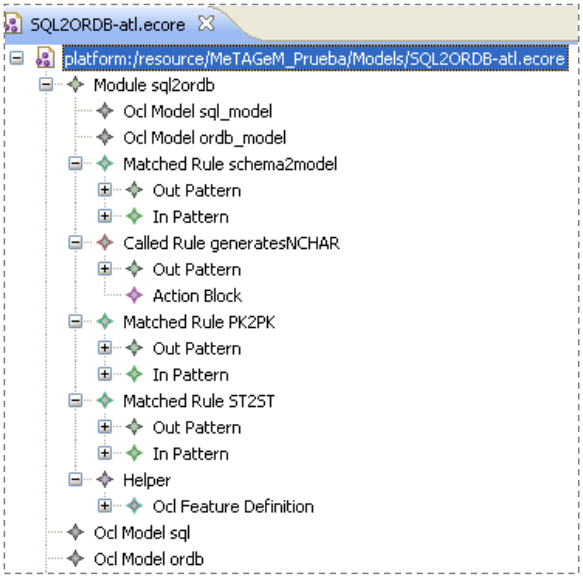

Figura E-18. Modelo de la Transformación *SQL2ORDB-atl.ecore*

E.4.2 Transformación de modelo a nivel PSM a modelo RubyTL

Si se selecciona la opción *2 Hybrid→RubyTL* se obtiene un modelo de transformación conforme al lenguaje RubyTL.

Antes de ejecutarse la transformación se realiza en forma automática el proceso de validación del modelo definido a nivel PSM, teniendo en cuenta que el modelo destino es conforme a RubyTL. Si hubiera alguna inconsistencia o error en el modelo, la herramienta mostraría el error que deberá ser solucionado antes de ejecutarse la transformación. En la Figura E-19 se puede ver como la herramienta informa del error y como al hacer clic en el botón *Run* muestra el tipo de error. El procedimiento de corrección del error es el explicado en las secciones anteriores.

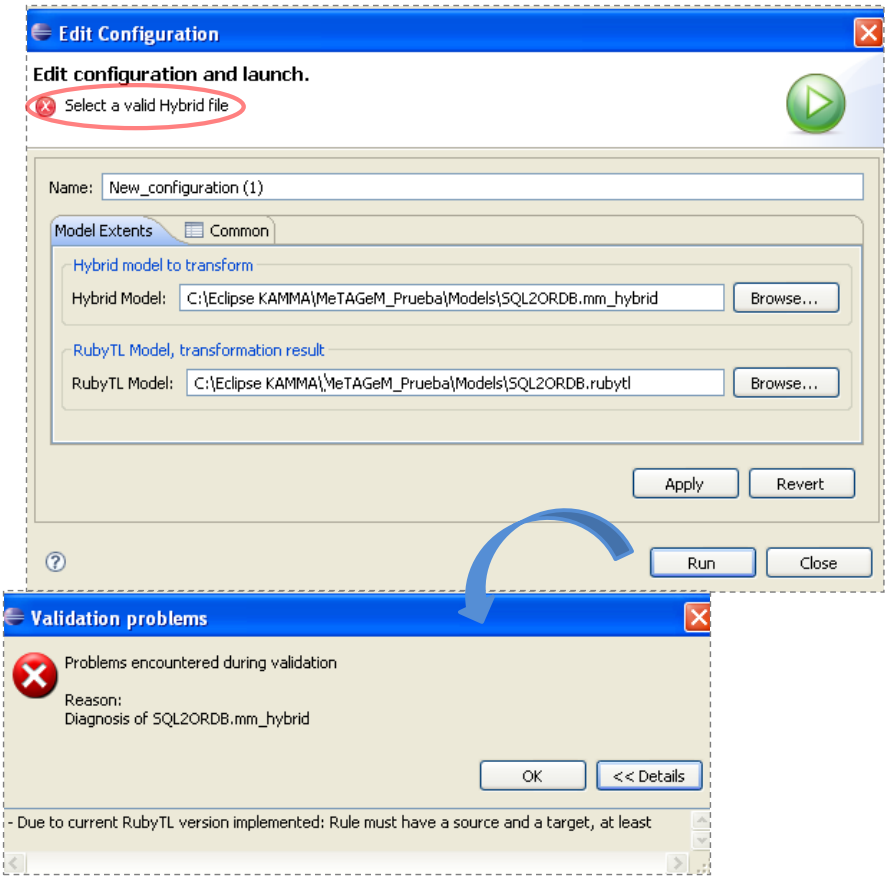

Figura E-19. Validación del Modelo en al Transformación

La validación que se realiza en esta transformación consiste en verificar las siguientes restricciones:

- Los elementos de tipo *Rule* deben tener una cardinalidad de 1 a N, es decir, deben tener solo un elemento de tipo *From* y pueden tener uno o varios elementos de tipo *To*.
- Los elementos de tipo *Operation* deben tener definido la propiedad *Context*.
- La propiedad *Reference* de un elemento de tipo *RightPattern* no puede ser del tipo *TargetElementRule*.

De la misma manera que en los casos anteriores, la primera vez que se ejecuta la transformación es necesario configurar la ruta de almacenamiento del modelo destino generado.

En la Figura E-20 se muestra el modelo de transformación conforme al lenguaje RubyTL, resultante de la transformación realizada.

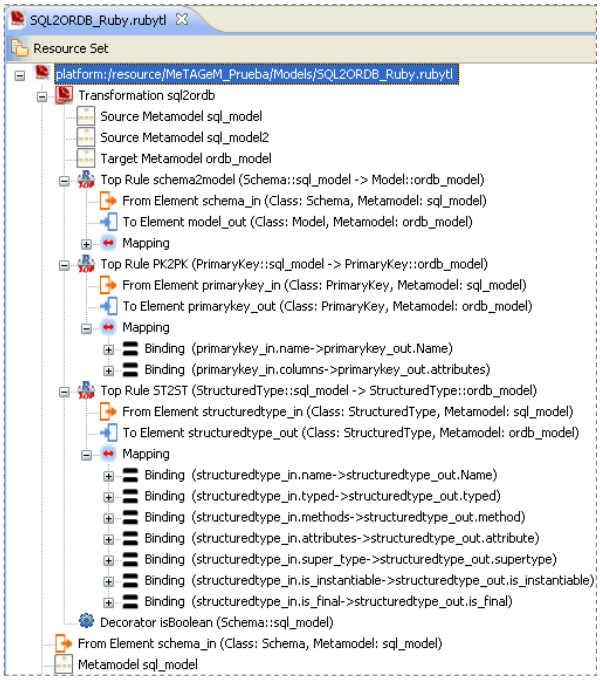

Figura E-20. Modelo de la Transformación *SQL2ORDB_Ruby.rubytl*

E.5 Generación de Código

El código que implementa la transformación depende del modelo de nivel PDM del cúal se parte.

E.5.1 Código ATL

A partir del modelo ATL (*SQL2ORDB-atl.ecore*) mostrado en la Figura D-18 se genera el código de la transformación en el lenguaje ATL. Para esto se debe hacer clic con el botón derecho sobre dicho modelo y seleccionar la opción *Extract ATL-0.2 model to ATL-0.2 file*. Como resultado de este proceso se obtiene un archivo con extensión atl (*SQL2ORBD.atl*) que contiene el código que implementa la transformación en el lenguaje ATL (Figura E-21).

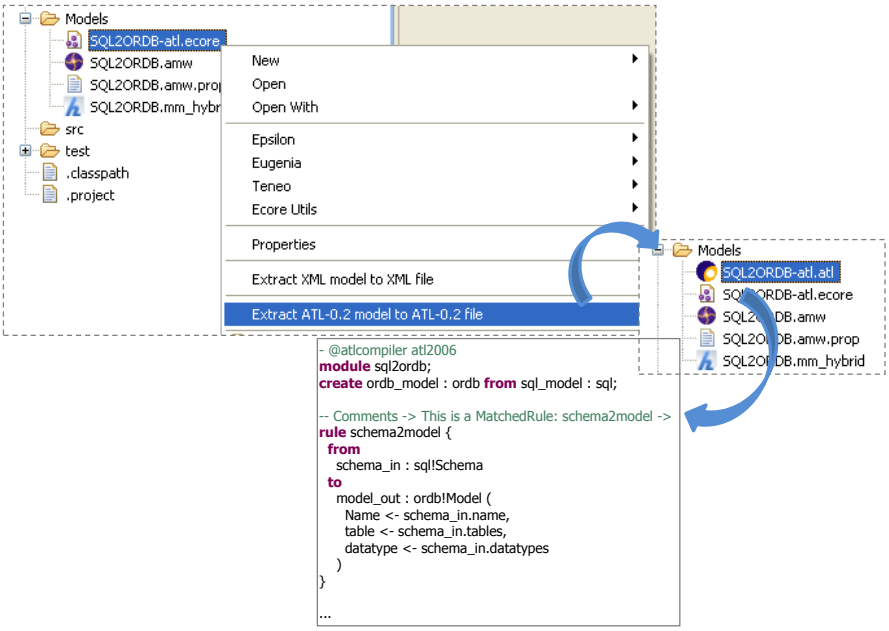

Figura E-21. Generación de Código ATL a partir del Modelo ATL

E.5.2 Código RubyTL

De la misma manera, a partir del modelo RubyTL (*SQL2ORDB_Ruby.rubytl*) mostrado en la Figura E-20 se genera el código de la transformación en el lenguaje RubyTL. Para esto se debe hacer clic con el botón derecho sobre dicho modelo y seleccionar la opción *Extract RubyTL model to RubyTL file (MeTAGeM)*. Como resultado de este proceso se obtiene un archivo con extensión rb (*SQL2ORBD.rb*) que contiene el código que implementa la transformación en el lenguaje RubyTL (Figura E-22).

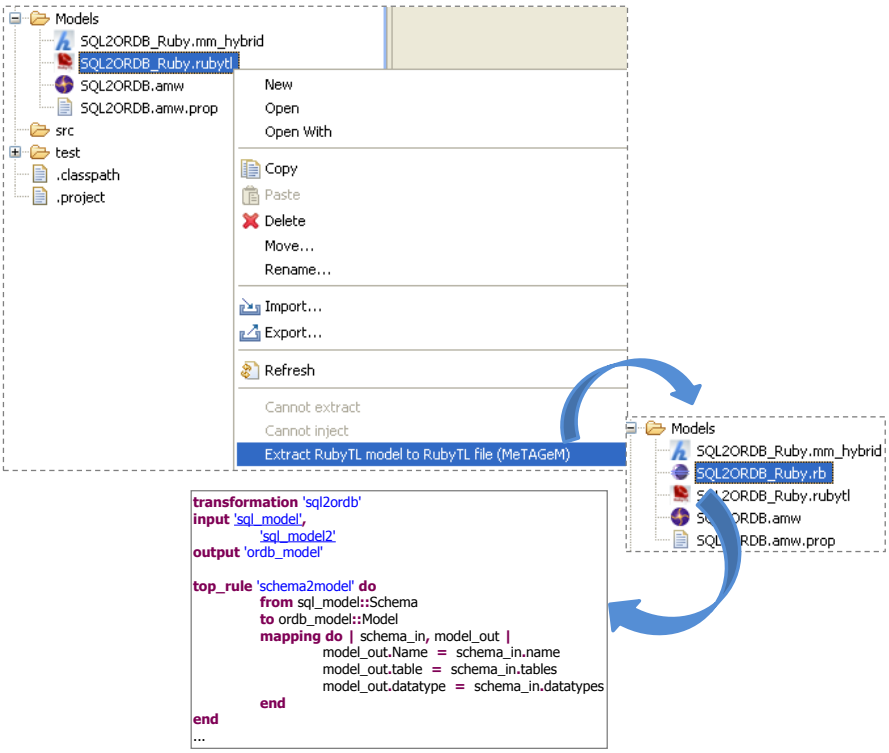

Figura E-22. Generación de Código RubyTL a partir del Modelo RubyTL

El código generado en ambos casos no es un código completamente operativo. En algunos casos es necesario realizar algún tipo de modificación ya que existen muchas situaciones en las que, para asegurar cierto grado de independencia de los modelos, no se puede recoger en el modelo de nivel PSM cuestiones de implementación de los lenguajes que se utilicen para implementar las transformaciones normalmente. Una vez realizadas las modificaciones necesarias, la transformación puede ser utilizada para transformar modelos conformes a los meta-modelos sobre los cuales se definió la transformación en cuestión.

Apéndice F: Caso de Estudio Completo

En el capítulo 4 se presenta el desarrollo de un meta-caso de estudio a modo de validación de MeTAGeM. Dicho meta-caso de estudio consiste en la implementación de las reglas de transformación propuestas en la herramienta M2DAT-DB para realizar el modelado de Bases de Datos Objeto-Relacionales. Estas reglas definen la transformación entre el modelo conceptual de datos definido a nivel PIM, representado por medio de un diagrama de clases de UML, y el modelo lógico específico para bases de datos Oracle10g definido a nivel PSM.

En la Tabla F-1 se muestra una tabla resúmen de las reglas de transformación entre ambos meta-modelos definidas en [195].

Tabla F-1. Reglas de Transformación de PIM a PSM de M2DAT-DB

PIM de Datos		**PSM de Datos para el Producto Oracle10g**
Package		*Package*
Class		*Object Type + Object Table*
Attrubute	*Simple*	*Attribute (column)*
	Multivalue	*Varray/Nested Table*
	Composed	*Object Type (column)*
	Calculated	*Trigger/Method*
Association	*1:1*	*Ref/Ref*
	1:M	*Ref - Nested Table/Varray*
	N: M	*Nested Table/Nested Table - Varray/Varray*
	Aggregation	*Nested Table/Varray of References*
	Composition	*Nested Table/Varray of Objects*
	Generalization	*Types/Typed Tables*

Como se puede observar, se definen las reglas de transformación para cada elemento del nivel PIM; así, por ejemplo, a partir de un elemento del tipo *class* definido en el modelo de nivel PIM se genera, a nivel PSM, dos elementos: un *object type* y un *typed table*.

Para comenzar con el modelado de las transformaciones usando MeTAGeM se debe seguir el proceso definido en la Figura 3-1. Según este

proceso, en primer lugar, se define el modelo de la transformación en el nivel PIM de MeTAGeM, con extensión *.amw*, donde se identifican los meta-modelos que participan en la transformación de M2DAT-DB, meta-modelo de UML y ORBD4ORA, y las relaciones existentes entre los elementos de cada meta-modelo.

A partir de esto, el segundo paso es aplicar el conjunto de reglas de transformación definidas en MeTAGeM para transformar modelos definidos a nivel PIM a modelos definidos a nivel PSM, estos modelos de nivel PSM se definen siguiendo la aproximación híbrida.

Como resultado de esta transformación se obtiene el modelo de la transformación con extensión *.mm_hybrid*, este modelo puede ser manipulado por el desarrollador para agregar nueva funcionalidad o modificar alguna existente.

Una vez refinado el modelo a nivel PSM, el tercer paso es la generación del modelo de la transformación a nivel PDM. Para esto el desarrollador debe seleccionar el lenguaje en el que desea implementar la transformación, ATL o RubyTL y ejecutar el conjunto de reglas de transformación correspondientes a cada lenguaje. El modelo de la transformación obtenido a nivel PDM, conforme al meta-modelo de del lenguaje seleccionado, puede ser refinado por el desarrollador de la misma manera que el modelo del nivel anterior.

El último paso del proceso de implementación de las transformaciones utilizando la herramienta MeTAGeM es la generación de código. Por medio del mecanismos de extracción disponible en el lenguaje ATL y en el editor desarrollado para RubyTL se obtiene el código de la transformación en el lenguaje seleccionado.

A continuación se muestra la implementación de las reglas definidas usando la herramienta MeTAGeM.

F.1 Definición del Modelo de nivel PIM

Para definir el modelo de nivel PIM de MeTAGeM se debe crear un nuevo modelo de *weaving* siguiendo los pasos mostrados en la Figura 3-39. En primer lugar, se deben seleccionar el meta-modelo origen (meta-modelo de UML) y el meta-modelo destino (meta-modelo ORDB4ORA). En la Figura F-1 se muestra el modelo de transformación independiente de plataforma. En la parte izquierda de la figura se muestran los elementos del meta-modelo origen (UML), en la parte derecha se muestran los elementos del meta-modelo destino (ORDB4ORA), y en

el modelo del centro se muestran las relaciones entre los diferentes elementos de ambos meta-modelos.

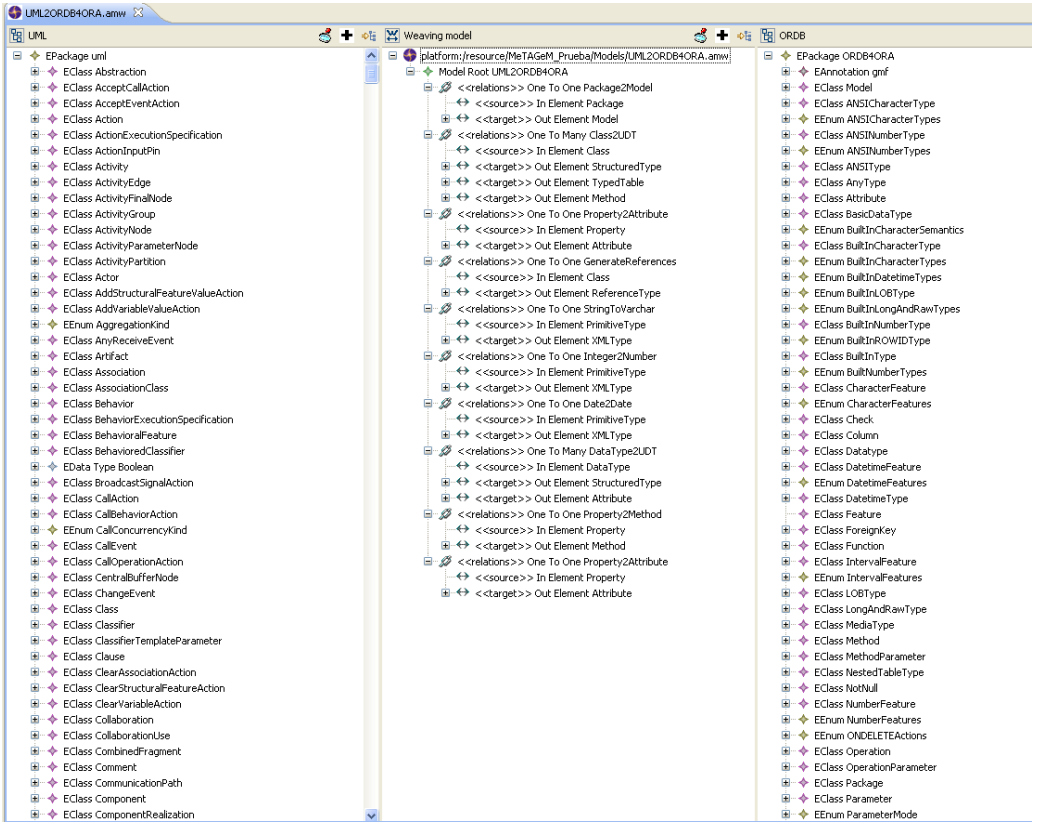

Figura F-1. Modelo de Transformación a Nivel PIM – UML2ORDB4ORA.amw

A continuación se detallará como se han establecido cada una de las relaciones entre los elementos de cada meta-modelo.

En la Figura F-2 se muestra la relación entre el elemento *Package* del meta-modelo *UML* y el elemento *Model* del meta-modelo *ORDB4ORA*. Esta relación se representa con un elemento de tipo *OneToOne*, estableciéndose como *source* el elemento *Package* y como *target* el elemento *Model*. Además, es necesario determinar la relación que existe entre las propiedades del elemento *target* con las propiedades a partir de las que se generan. En este caso se establece una relación de tipo *OneToOne* entre la propiedad *name* del elemento *Package* y la propiedad *Name* del elemento *Model*.

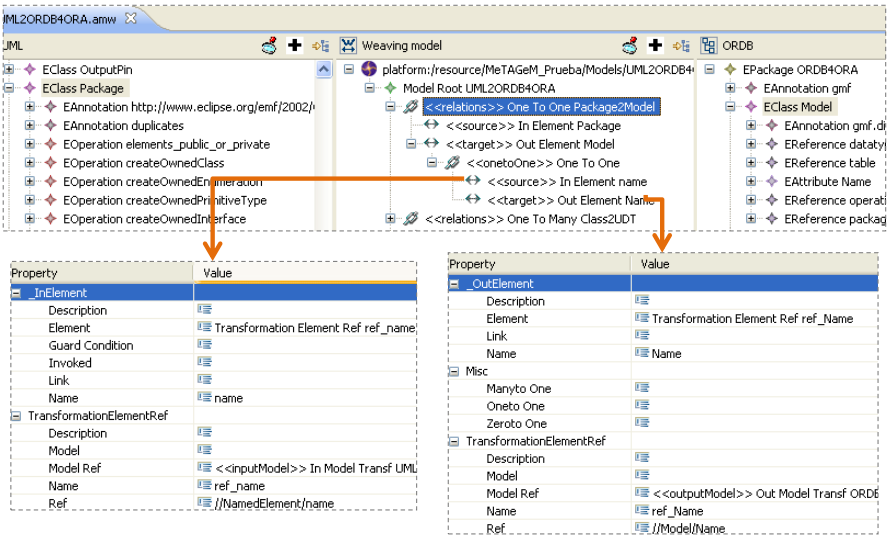

Figura F-2. Relacion *Package2Model* – UML2ORDB4ORA.amw

La Figura F-3 muestra la implementación de la regla de transformación entre elementos de tipo *class* del meta-modelo de UML a elementos de tipo *StructuredType* y *Typed Table* del meta-modelo de ORDB4ORA. Esta relación se representa con un tipo de elemento *OneToMany*, ya que, a partir de un elemento del meta-modelo origen (el elemento *class*) se generan dos elementos en el meta-modelo destino (el elemento *Structured Type* y el elemento *Typed Table*). Por último, las reglas de transformación definidas establecen que los elementos de tipo *Method* de UML deben ser transformados a elementos de tipo *Method* en ORDB4ORA; por lo que, es necesario establecer la relación entre los elementos de tipo *Method* que dependan del elemento *Class* y el elemento de tipo *Method* de ORDB4ORA que se dependan del tipo *StructuredType*.

Como se puede observar, además, se establecen las relaciones existentes entre las propiedades de los elementos destino y las propiedades del elemento origen. Por lo que, en cada *OutElement* se añaden dichas relaciones. Por ejemplo, el elemento *StructuredType* tiene la propiedad *Name*, que indica su nombre que se obtienen a partir de la propiedad *Name* del elemento *Class* respectivamente.

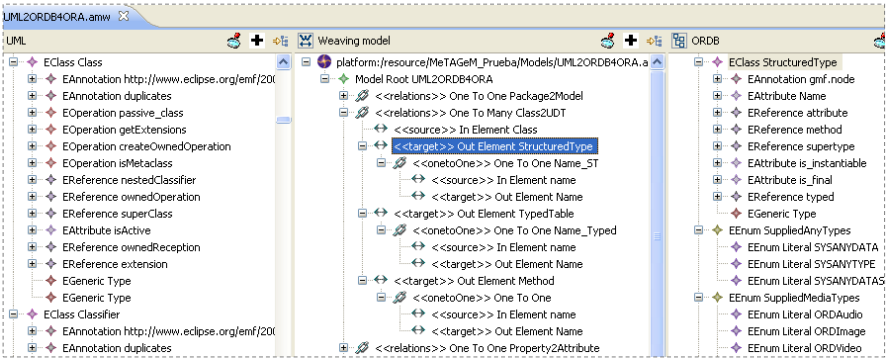

Figura F-3. Relacion *Class2UDT* – UML2ORDB4ORA.amw

El elemento de tipo *Property* del meta-modelo UML puede ser transformado en diferentes elementos del modelo destino, dependiendo del valor de sus propiedades. Para representar cada una de estas situaciones se establecen diferentes relaciones y se determina las condiciones que deben cumplir en cada caso.

En la Figura F-4 se muestra la relación entre los elementos de tipo *Property* del meta-modelo de UML y los elementos de tipo *Attribute* del meta-modelo ORDB4ORA, con esto se representa la transformación de los atributos simples. Esta relación se representa con un tipo de relación *OneToOne*, y para que se cumpla esta relación se estable como condición (*Guard Condition*) que no debe ser un atributo derivado (propiedad *isDefined* = *false*) y no debe ser un atributo multivaluado (para determinar esto último será necesario definir una función auxiliar).

Al igual que en las relaciones presentadas anteriormente, en este caso, también se establecen las relaciones existentes entre las propiedades de ambos elementos, para esto se agregan dos relaciones de tipo *OneToOne* que dependen del elemento de tipo *OutElement*.

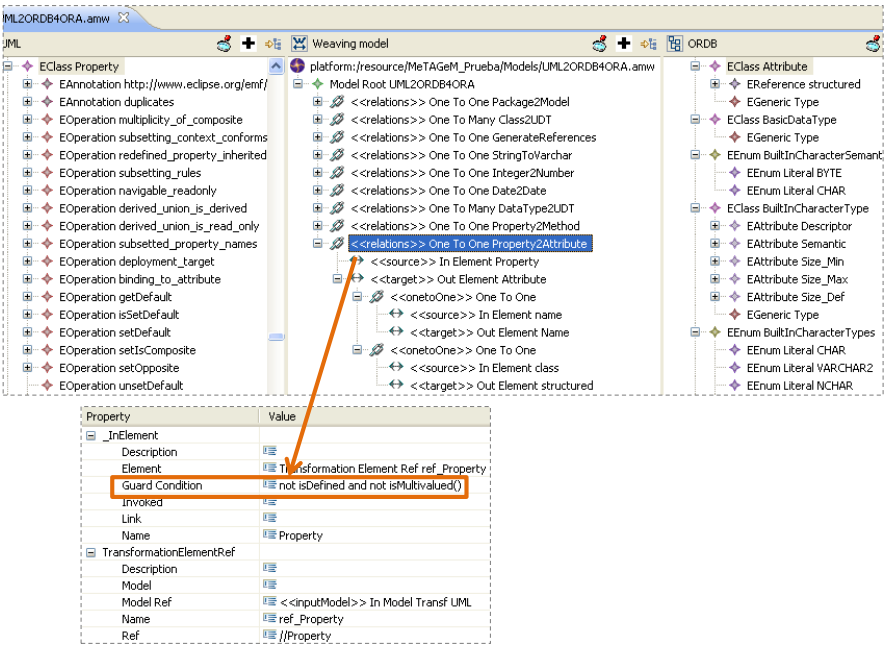

Figura F-4. Relación *Property2Attribute* – UML2ORDB4ORA.amw

En la Figura F-5 se muestra la relación entre los elementos de tipo *Property* del meta-modelo de UML y los elementos de tipo *Method* del meta-modelo ORDB4ORA, con esto se representa la transformación de los atributos derivados. Esta relación se representa con un tipo de relación *OneToOne*, y para que se cumpla esta relación se establece como condición (*Guard Condition*) que debe ser un atributo derivado (propiedad *isDefined* = *true*).

Al igual que en las relaciones presentadas anteriormente, en este caso, también se establecen las relaciones existentes entre las propiedades de ambos elementos, para esto se agregan dos relaciones de tipo *OneToOne* que dependen del elemento de tipo *OutElement*.

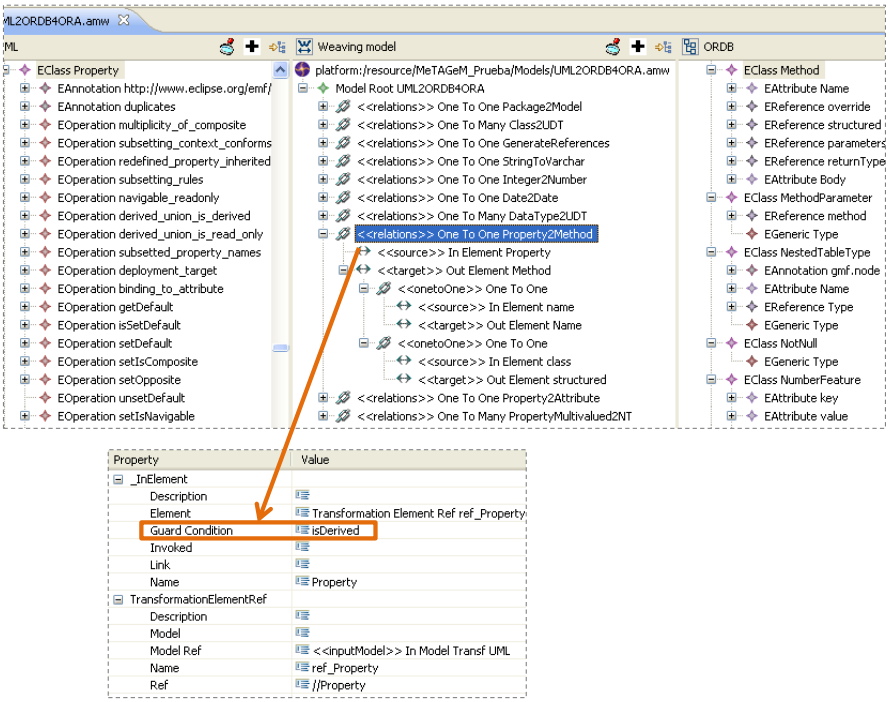

Figura F-5. Relación *Property2Method* – UML2ORDB4ORA.amw

En la Figura F-6 se muestra la relación entre los elementos de tipo *Property* del meta-modelo de UML y los elementos de tipo *Attribute* y *StrodeNestedTable* del meta-modelo ORDB4ORA. Esta relación especifica la transformación de los atributos multivaluados y se representa con un tipo de relación *OneToMany*. Para que se cumpla esta relación se estable como condición (*Guard Condition*) que no debe ser un atributo derivado (propiedad *isDefined = false*) y debe ser un atributo multivaluado (en esta caso se utiliza la misma función auxiliar definida para la relación *Property2Attribute*).

Al igual que en las relaciones presentadas anteriormente, en este caso, también se establecen las relaciones existentes entre las propiedades de ambos elementos, para esto se agregan dos relaciones de tipo *OneToOne* que dependen del elemento de tipo *OutElement*.

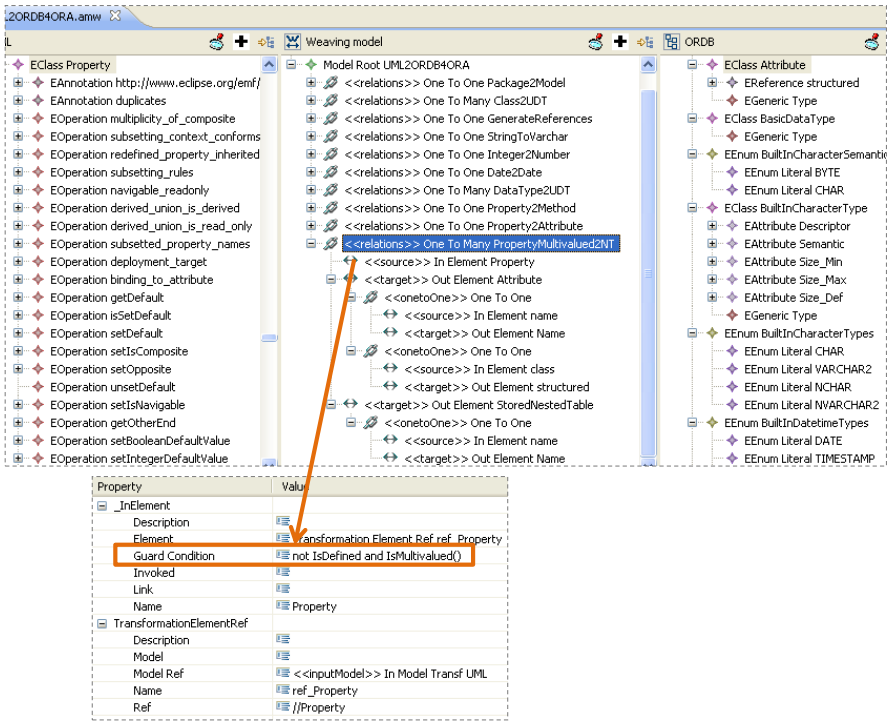

Figura F-6. Relación *PropertyMultivalued2NT*– UML2ORDB4ORA.amw

El elemento *PrimitiveType* del meta-modelo de UML agrupa los tipos de datos que proporciona UML, es decir: *String*, *Integer*, *Date*, etc. Para realizar la transformación de un elemento de tipo *PrimitiveType* de UML a un elemento de tipo *XMLType* de ORDB4ORA es necesario establecer correctamente cuales son las condiciones para que se cumpla la relación. Para esto se han creado tres relaciones, usando un elemento de tipo *OneToOne*, que permite transformar de acuerdo al valor de la propiedad *name* del tipo *PrimitiveType* (Figura F-7). En la relación *String2VarChar* se comprueba que la propiedad *name* del elemento *PrimitiveType* sea igual a *"String"*, de igual manera, en la relación *Integer2Number* se comprueba que la propiedad *name* del elemento *PrimitiveType* sea igual a *"Integer"*. Por último, en la relación *Date2Date* se comprueba que la propiedad *name* del elemento *PrimitiveType* sea igual a *"Date"*.

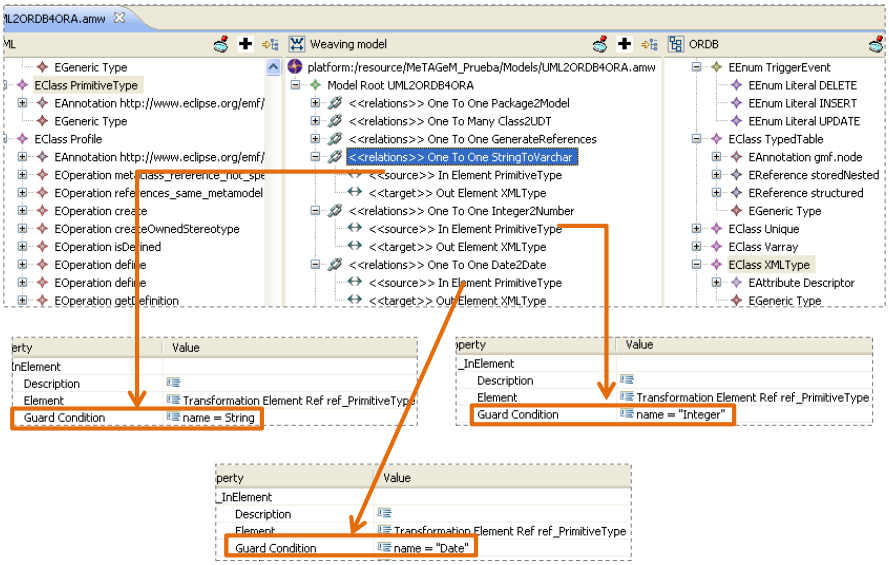

Figura F-7. Relación *String2Varchar, Integer2Number* y *Date2Date*– UML2ORDB4ORA.amw

En la Figura F-8 se muestra la relación entre elementos de tipo *DataType* y elementos de tipo *StructuredType*. Uno de los usos de los elementos de tipo *DataType* es para representar tipos de datos especiales definidos por el usuario. Esta relación permite transformar ese tipo de datos especial, por lo que es necesario controlar que la propiedad *name* sea diferente de los tipos de datos transformados en las relaciones anteriores; es decir en la condición de guarda del elemento de tipo *DataType* se establece que *name* <> *"String"*, *name* <> *"Integer"* y *name* <> *"Date"*. Como se puede observar en la Figura F-8 esta relación se representa con el tipo *OneToMany*, ya que además del elemento *StructuredType* se deben crear los elementos de tipo *Attribute* que dependan del *StructuredType*.

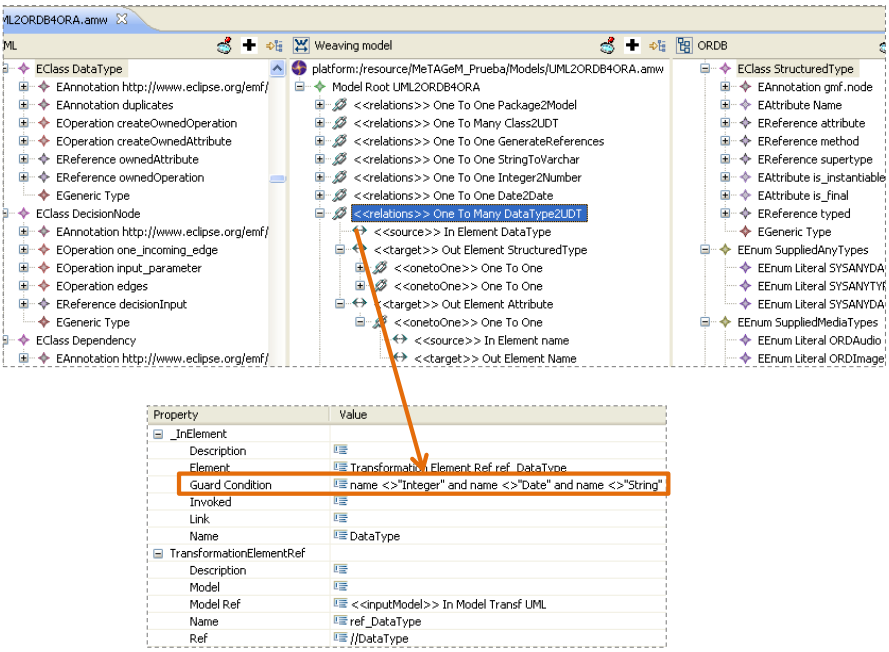

Figura F-8. Relación *DataType2UDT* – UML2ORDB4ORA.amw

F.2 Definición del Modelo a nivel PSM

A partir de la definición del modelo de transformación independiente de plataforma entre el meta-modelo UML y ORDB4ORA, realizado en la sección anterior, se puede generar, de forma (semi-)automática, por medio de la ejecución de la transformación propuesta en MeTAGeM, el modelo de la transformación específico de plataforma siguiendo la aproximación híbrida (Figura F-9).

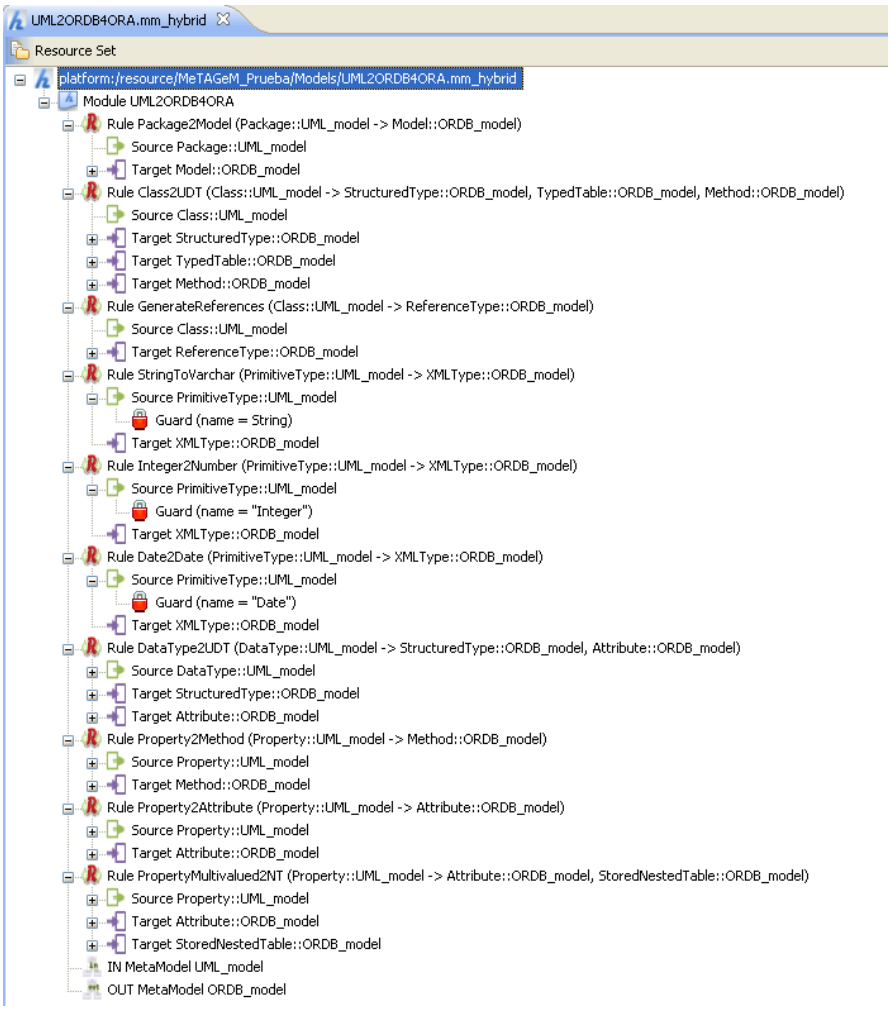

Figura F-9. Modelo a Nivel PSM – UML2ORDB4ORA.mm_hybrid

Como se puede observar en la Figura F-9 por cada uno de los elementos de tipo *Relations* definidos en el modelo de la transformación a nivel PIM se genera un elemento de tipo *Rule* en el modelo de la transformación a nivel PSM (*UML2ORDB4ORA.mm_hybrid*).

El modelo *UML2ORDB4ORA.mm_hybrid* puede, (en muchos casos, debe) ser refinado por el desarrollador de la transformación; ya sea modificando alguna regla generada, o agregando nueva funcionalidad.

En nuestro caso de estudio es necesario agregar algunas funciones auxiliares, como la de *isMultivalued*, o establecer algunos tipos de relaciones especiales.

Una de las necesidades detectadas es tener un mecanismo que nos permita recuperar fácilmente el elemento de tipo *Package* del cual dependen todos los elementos del modelo de UML. Para esto se crea un elemento de tipo *Operation* llamado *Package()*. En esta *Operation* se especifica que su valor de retorno es un elemento de tipo *Package* (Figura F-10).

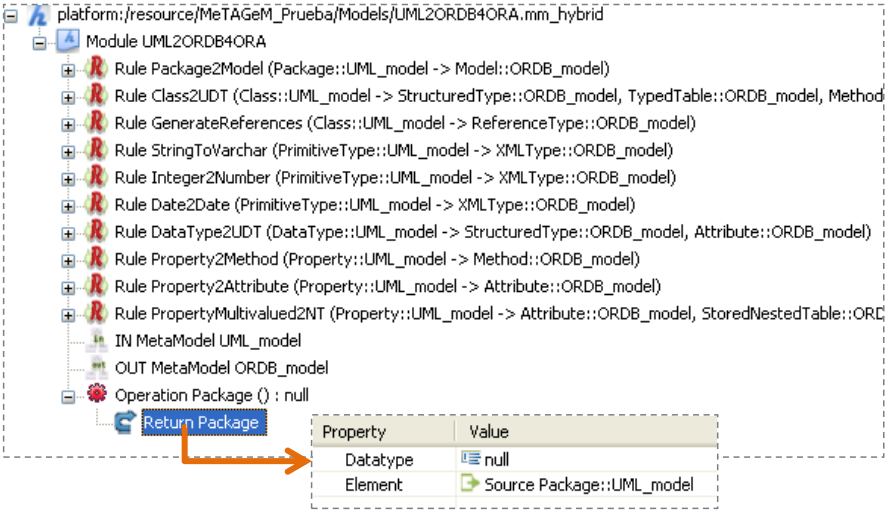

Figura F-10. Operation *Package*– UML2ORDB4ORA-mm_hybrid

A partir de la definición del elemento de tipo *Operation* es necesario indicar las reglas desde los que puede ser invocado. En este caso se utiliza para devolver el elemento del modelo destino (*Model*) en el que fue transformado el elemento de tipo *Package*.

En la Figura F-11 se muestra la utilización del mismo, el elemento de tipo *StructuredType* del meta-modelo ORDB4ORA tiene una referencia al elemento *Model* al cual pertenece; para esto se agrega un nuevo tipo de elemento *ElementIncluded* y se especifican que el valor del elemento de tipo *RightPattern* es una llamada al elemento de tipo *Operation Package()* y que este elemento deberá ser asignado al elemento de tipo *Model* especificado en el elemento *LeftPattern*. De la misma manera se procede en todas las situaciones donde es necesario invocar la función auxiliar.

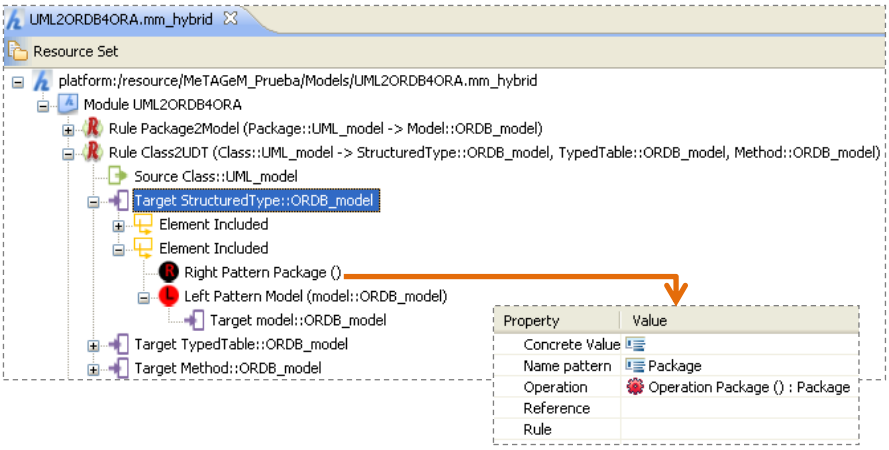

Figura F-11. Asignación de *Operation Package()* – UML2ORDB4ORA.mm_hybrid

El elemento *StructuredType* tiene dos referencias que son necesarias indicar, *typed* que es una referencia al elemento *TypedTable* que lo define, y *method* que es una referencia al elemento *Method*. Los elementos *TypedTable* y *Method* se crean en la misma regla en la que se crea el elemento *StructuredType*, por lo que estas referencias son necesarias agregarlas en el modelo del nivel PSM de forma explícita (Figura F-12).

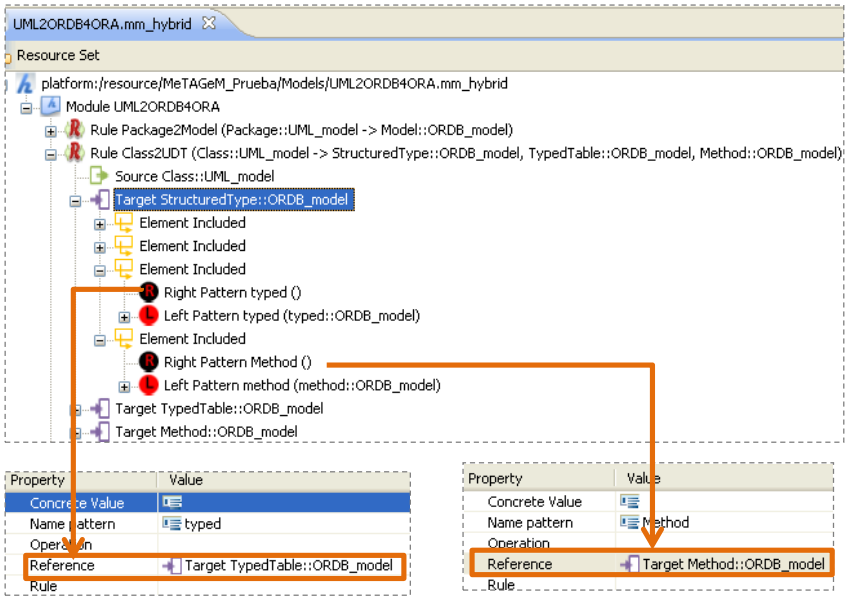

Figura F-12. Asignación de *typed* y *method*– UML2ORDB4ORA.mm_hybrid

Existen situaciones donde es necesario crear un elemento asignándole a alguna de sus propiedades un valor concreto, tal es el caso de la regla *Strin2Varchar* donde a la propiedad *name* del elemento de tipo *DataType* es necesario asignar el valo *"#VARCHAR2"* (Figura F-13). Esta misma situación se repite en las reglas *Integer2Number* y *Date2Date*.

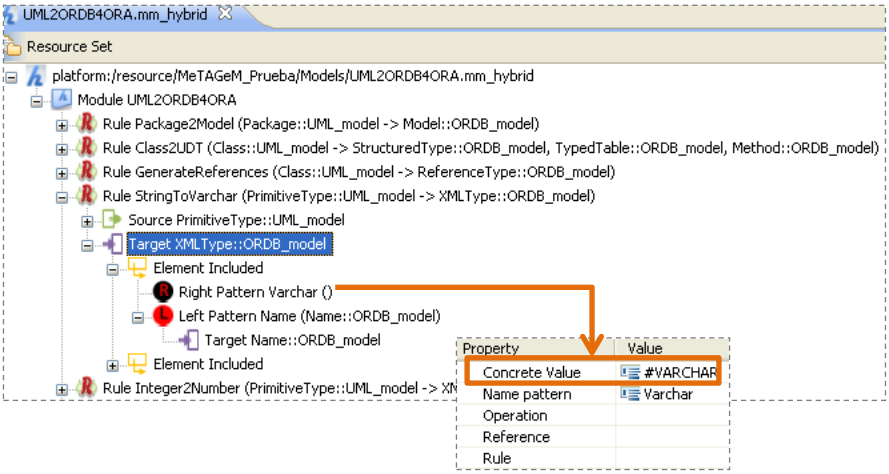

Figura F-13. Asignación *Name* en *Strin2Varchar* – UML2ORDB4ORA.mm_hybrid

F.3 Definición del Modelo a nivel PDM

A nivel PDM se puede realizar el modelado de las transformaciones de acuerdo al lenguaje ATL o al lenguaje RubyTL.

Una vez que el desarrollador haya finalizado de refinar el modelo PSM, se genera, de forma (semi-)automática, el modelo a nivel PDM seleccionando, previamente el lenguaje de modelado de las transformaciones.

A continuación se muestra el desarrollo del caso de estudio en ambos lenguajes.

F.3.1 Modelo Conforme al Meta-modelo de ATL

En la Figura F-14 se muestra el modelo de transformación conforme al meta-modelo de ATL, *UML2ORDB4ORA-atl.ecore*. Es importante mencionar que, como se ha mencionado anteriormente, para la manipulación de los modelos conformes al meta-modelo de ATL se utiliza el editor reflexivo de EMF.

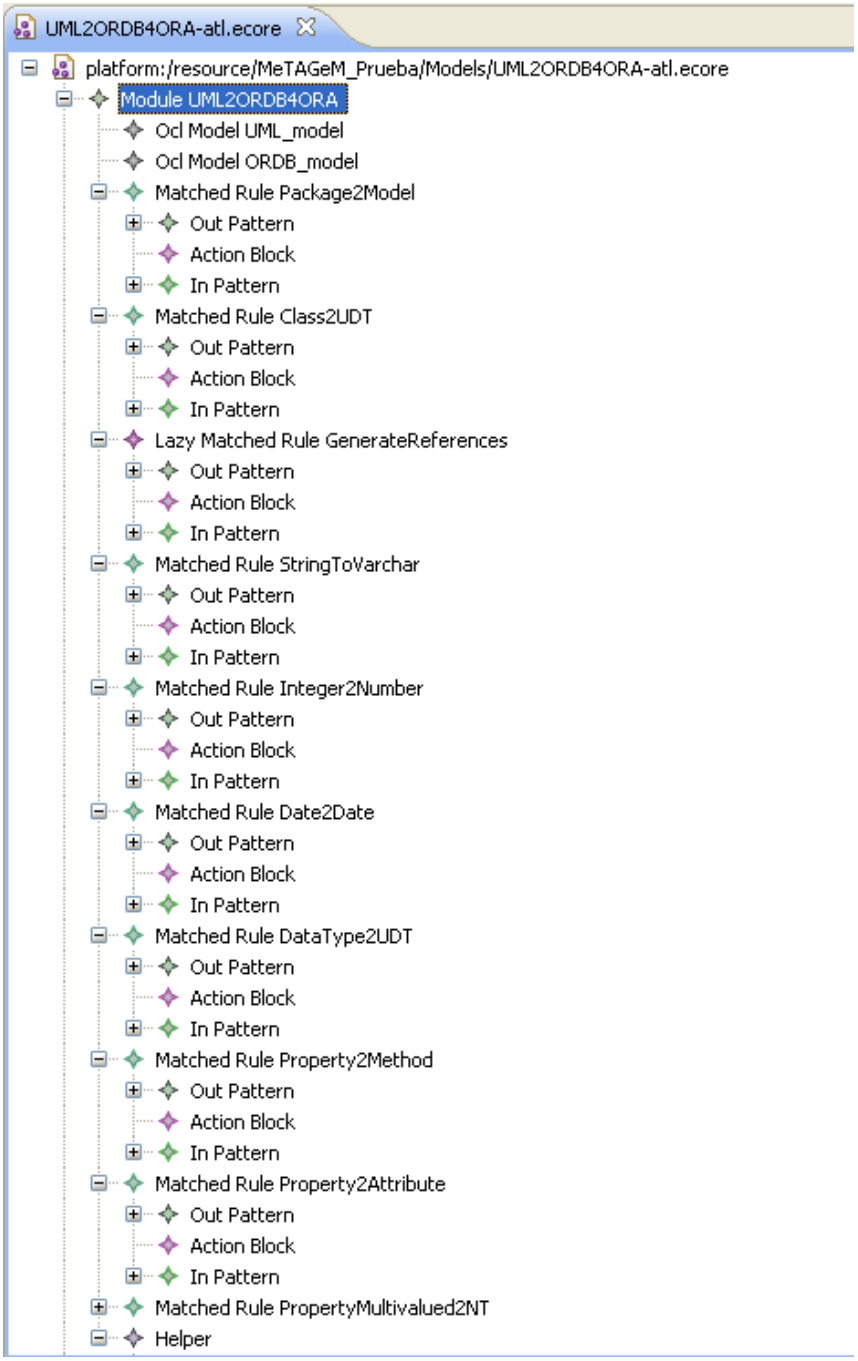

Figura F-14. Modelo a Nivel PDM – UML2ORDB4ORA-atl.ecore

En la Figura F-15 se muestra la transformación entre cada uno de los elementos del modelo de nivel PSM y los elementos del modelo de transformación conforme al meta-modelo de ATL. Por cada elemento de tipo *Rule* del modelo origen, cuya propiedad *isMain* sea igual a *true* (parte superior de la figura), se genera un elemento de tipo *MatchedRule* en el modelo destino (parte inferior de la figura).

De la misma manera, por cada elemento *SourceElementRule* y *TargetElementRule* del modelo origen se generan, en el modelo destino, los elementos *InPattern* y *OutPattern* respectivamente. Por último, por cada elemento de tipo *ElementIncluded* se generan elementos de tipo *Binding* por medio de los cuales se define el cuerpo de cada una de las reglas.

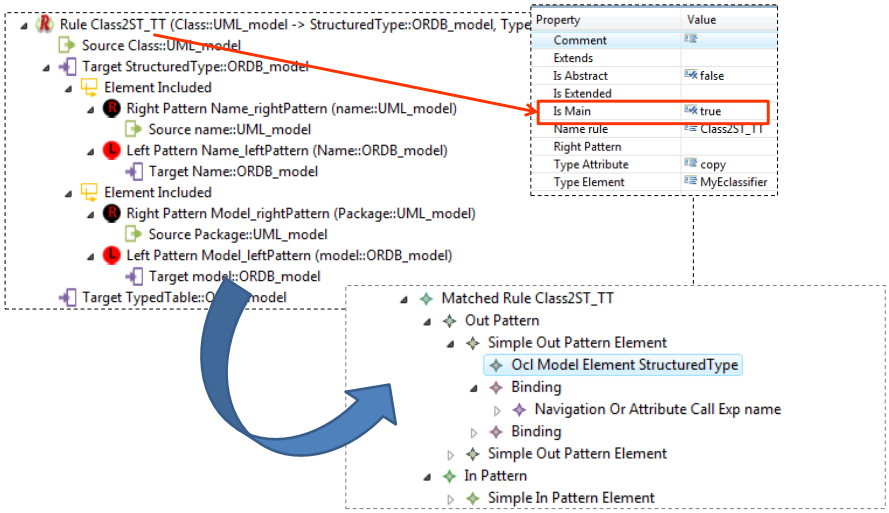

Figura F-15. UML2ORDB4ORA-atl.ecore - Transformación

F.3.2 Modelo Conforme al Meta-modelo de RubyTL

En la Figura F-16 se muestra el modelo de transformación conforme al meta-modelo de RubyTL, *UML2ORDB4ORA.rubytl.* A diferencia que en el modelo de ATL para manipular el modelo conforme al meta-modelo de RubyTL se utiliza el editor personalizado propuesto en MeTAGeM.

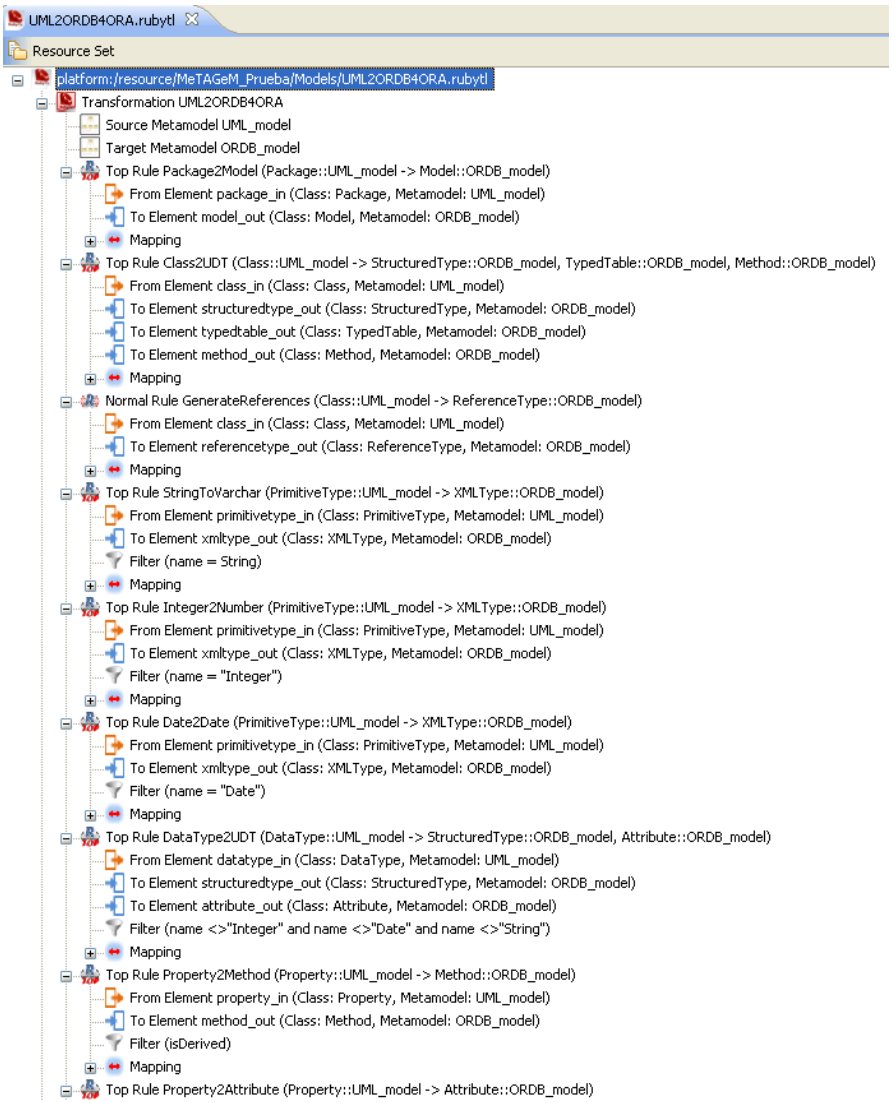

Figura F-16. Modelo a Nivel PDM - UML2ORDB4ORA.rubytl

En la Figura F-17 se muestra la transformación entre cada uno de los elementos del modelo de nivel PSM y los elementos del modelo de transformación conforme al meta-modelo de RubyTL. Por cada elemento de tipo *Rule* del modelo origen, cuya propiedad *isMain* sea igual a *true* (parte superior de la figura), se genera un elemento de tipo *TopRule* en el modelo destino (parte inferior de la figura).

De la misma manera, por cada elemento *SourceElementRule* y *TargetElementRule* del modelo origen se generan, en el modelo destino, los elementos *FromElement* y *ToElement* respectivamente. Por último, por cada elemento de tipo *ElementIncluded* se generan un elemento de tipo *Mapping* que contiene elementos de tipo *Binding* por medio de los cuales se define el cuerpo de cada una de las reglas.

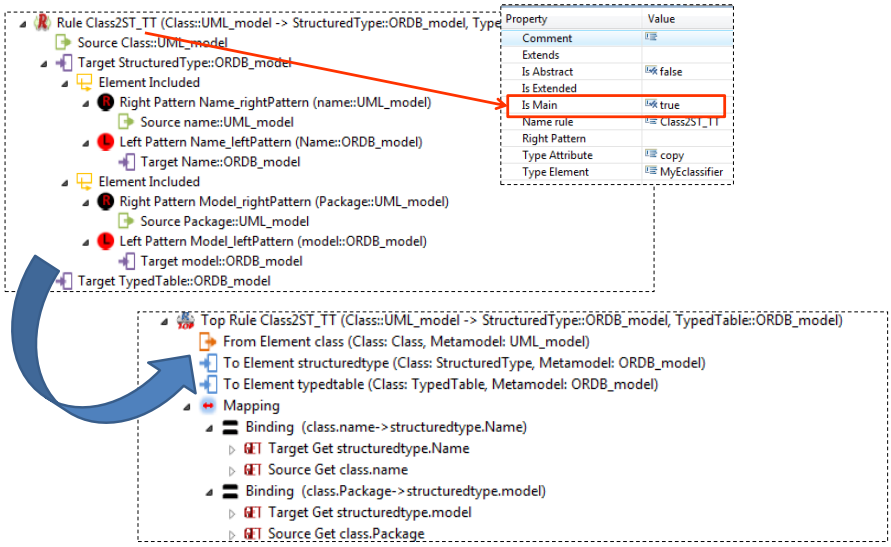

Figura F-17. Modelo a Nivel PDM – Reglas de Transformación

F.4 Generación de Código

Según el proceso definido en la sección 3.1 el último paso es la obtención del código que implementa la transformación en el lenguaje seleccionado en el nivel PDM, en el caso de esta tesis en los lenguajes ATL y RubyTL.

F.4.1 Código de la Transformación en ATL

Para realizar la generación de código se utiliza la funcionalidad de extracción código brindada por el lenguaje ATL. Es necesario aclarar que el código generado no es implementable completamente, hay algunos detalles que deben ser modificados por el desarrollador.

En la Figura F-18 se muestra la cabecera del módulo que contiene el código de la transformación en el lenguaje ATL. Como se puede observar se

define el nombre de la transformación, UML2ORDB4ORA, y los meta-modelos destino, ORDB, y origen, UML.

```
-- @atlcompiler atl2006
module UML2ORDB4ORA;
create ORDB_model : ORDB from UML_model : UML;
```

Figura F-18. Cabecera de la Transformación en ATL

En la Figura F-19 se muestra la regla que implementa la transformación entre un elemento de tipo *Package* de UML y un elemento de tipo *Model* de ORDB.

```
-- Comments -> This is a MatchedRule: Package2Model ->
rule Package2Model {
   from
    package_in : UML!Package
   to
    model_out : ORDB!Model (
       Name <- package_in.name)
  -- ActionBlock:
  do { }
}
```

Figura F-19. Código ATL – *Rule Package2Model*

En la Figura F-20 se muestra el código generado que implementa la transformación entre elementos de tipo *Class* del meta-modelo de UML y elemento de tipo *StrcuturedType* del meta-modelo de ORDB. Como se puede observar, además se generan los elementos de tipo *TypedTable* y *Method*.

```
-- Comments -> This is a MatchedRule: Class2UDT ->
rule Class2UDT {
   from
    class_in : UML!Class
   to
    structuredtype_out : ORDB!StructuredType (
       Name <- class_in.name,
       model <- class_in.Package,
       typed <- typedtable_out,
       method <- method_out),

    typedtable_out : ORDB!TypedTable (
       Name <- class_in.name),

    method_out : ORDB!Method (
      Name <- class_in.name)

  -- ActionBlock:
  do {}
}
```

Figura F-20. Código ATL – *Rule Class2UDT*

En el código mostrado en la Figura F-20 es necesario introducir algunas modificaciones, en primer lugar es necesario agregar en la propiedad *name* del elemento *StructuredType* y del elemento *TypedTable* los estereotipos definidos en M2DAT-DB. En segundo lugar es necesario modificar la sentencia con la que se crea el elemento *Method*, ya que este elemento se genera a partir de los elementos referenciados en la propiedad *ownedOperation* del elemento *Class*, con lo cual es necesario acceder a dichos elementos para generar el elemento de tipo *Method* correctamente. En la Figura F-21 se muestra el código luego de las modificaciones realizadas.

```
- Comments -> This is a MatchedRule: Class2UDT ->
rule Class2UDT {
   from
     class_in : UML!Class
   to
     structuredtype_out : ORDB!StructuredType (
       Name <- class_in.name + '<<udt>>',
       model <- class_in.Package,
       typed <- typedtable_out,
       method <- method_out),

     typedtable_out : ORDB!TypedTable (
       Name <- class_in.name + 's<<persistent>>'),

     method_out : distinct ORDB!Method  foreach (op in class_in.ownedOperation) (
       Name <- op.name)

  -- ActionBlock:
   do {}
}
```

Figura F-21. Código Modificado ATL – *Rule Class2UDT*

En la Figura F-22 se muestra la regla que implementa la transformación entre los elementos de tipo *Property* del meta-modelo de UML y los elementos de tipo *Attribute* del meta-modelo ORDB.

```
-- Comments -> This is a MatchedRule: Property2Attribute ->
rule Property2Attribute {
   from
     property_in : UML!Property ( 'not isDefined and not isMultivalued()')
   to
     attribute_out : ORDB!Attribute (
       Name <- property_in.name,
       structured <- property_in.class)
  -- ActionBlock:
   do {}
}
```

Figura F-22. Código ATL – *Rule Property2Attribute*

En el código de la Figura F-22 también es necesario introducir algunas modificaciones. La condición de guarda de la regla se genera como un *string*, por lo que es necesario indicar correctamente. En la Figura F-23 se muestra el código modificado.

```
-- Comments -> This is a MatchedRule: Property2Attribute ->
rule Property2Attribute {
   from
     property_in : UML!Property (not property_in.isDefined and not property_in.isMultivalued())
   to
     attribute_out : ORDB!Attribute (
       Name <- property_in.name,
       structured <- property_in.refImmediateCompisite())
  -- ActionBlock:
   do { }
}
```

Figura F-23. Código Modificado ATL – *Rule Property2Attribute*

En la Figura F-24 se muestra el código modificado que implementa la transformación entre elementos de tipo *Property* de UML y elementos de tipo *Method* de ORDB. Las modificaciones que se realizaron son las mismas que las explicadas anteriormente.

```
-- Comments -> This is a MatchedRule: Property2Method ->
rule Property2Method {
   from
     property_in : UML!Property (property_in.isDerived)
   to
    method_out : ORDB!Method (
       Name <- property_in.name,
       structured <- property_in.refImmediateComposite())
  -- ActionBlock:
   do {}
}
```

Figura F-24. Código Modificado ATL – *Rule Property2Method*

En la Figura F-25 se muestra el código modificado que implementa la transformación entre elementos de tipo *Property* de UML y elementos de tipo *Attribute* y *Stored Nested Table* de ORDB, en este caso en la condición de guarda se controla, por medio de la llamada al *helper IsMultivalued*, que el atributo sea multivaluado. En esta regla se invoca la creación de elementos de tipo *Nested Table* por medio de la invocación a la regla de tipo *Lazy generateNestedTableMultivalued*.

```
-- Comments -> This is a MatchedRule: PropertyMultivalued2NT ->
rule PropertyMultivalued2NT {
  from
    property_in : UML!Property  not property_in.IsDefined and property_in.IsMultivalued())
  to
    attribute_out : ORDB!Attribute (
      Name <- property_in.name,
      structured <- property_in.refimmediateComposite(),
      Type <- thisModule.generatedNestedTableMultivalued(property_in)),

    storednestedtable_out : ORDB!StoredNestedTable (
      Name <- property_in.name,
      attribute <-    attribute_out,
      typed <- thisModule.resolveTemp(property_in ->refImmediateComposite,'tt'))
 -- ActionBlock:
  do { }
}
```

Figura F-25. Código Modificado ATL – *Rule PropertyMultivalued2NT*

En la Figura F-26 se muestra la regla de transformación *generateNestedTableMultivalued*. Esta regla es de tipo *Lazy*, es decir que para que se ejecute debe ser invocada explícitamente y permite la creación de elementos de tipo *Nested Table* a partir de elementos de tipo *Property*.

```
-- Comments -> This is a LazyRule:
generatedNestedTableMultivalued ->
lazy rule generatedNestedTableMultivalued {
  from
    property_in : UML!Property
  to
    nestedtabletype_out : ORDB!NestedTableType (
      Name <- property_in.name,
      model <- property_in.Package)
  -- ActionBlock:
  do {}
}
```

Figura F-26. Código Modificado ATL – *Lazy Rule generateNestedTableMultivalued*

En La Figura F-27 se muestra el código generado para la transformación entre elementos de tipo *PrimitiveType* y elementos de tipo *XMLType*, donde en la condición de guarda se establece que el valor de la propiedad *name* del tipo *PrimitiveType* debe ser *"String"*. En la regla se establece la propiedad *name* del tipo *XMLType* como *"VARCHAR2"*.

```
-- Comments -> This is a MatchedRule: StringToVarchar ->
rule StringToVarchar {
   from
    primitivetype_in : UML!PrimitiveType ( 'name = String')
   to
    xmltype_out : ORDB!XMLType (
      Name <- "#VARCHAR",
      model <- primitivetype_in.Package)
 -- ActionBlock:
 do {}
}
```

Figura F-27. Código ATL – *Rule String2VarChar*

En el código de la Figura F-27 es necesario introducir algunas modificaciones, como por ejemplo la condición de guarda de la regla. En la Figura F-28 se muestra el código modificado de la regla de transformación.

```
-- Comments -> This is a MatchedRule: StringToVarchar ->
rule StringToVarchar {
  from
    primitivetype_in : UML!PrimitiveType( primitivetype_in.name = 'String')
  to
    xmltype_out : ORDB!XMLType (
      Name <- "#VARCHAR",
      model <- primitivetype_in.Package)
  -- ActionBlock:
  do {}
}
```

Figura F-28. Código Modificado ATL – *Rule String2VarChar*

La Figura F-29 muestra el código modificado que implementa la transformación entre elementos de tipo *PrimitiveType* del meta-modelo de UML y elementos de tipo *XMLType* del meta-modelo ORDB4ORA. Como se puede observar en la condición de guarda se controla que el valor de la propiedad *name* del elemento *PrimitiveType* debe ser *"Integer"*, si se cumple dicha condición se genera un elemento de tipo *XMLType* cuya propiedad *name* tiene el valor *"NUMBER"*.

```
-- Comments -> This is a MatchedRule: Integer2Number ->
rule Integer2Number {
  from
    primitivetype_in : UML!PrimitiveType (primitivetype_in.name = 'Integer')
  to
    xmltype_out : ORDB!XMLType (
      Name <- "#NUMBER",
      model <- primitivetype_in.Package)
  -- ActionBlock:
  do {}
}
```

Figura F-29. Código Modificado ATL – *Rule Integer2Number*

En la Figura F-30 se muestra el código modificado que implementa la regla de transformación entre elementos de tipo *PrimitiveType* y elementos de tipo *XMLType*. En esta regla, en la condición de guarda se controla que el valor de la propiedad *name* del elemento *PrimitiveType* debe ser *"Date"*, si se cumple dicha condición se genera un elemento de tipo *XMLType* cuya propiedad *name* tiene el valor *"DATE"*.

```
-- Comments -> This is a MatchedRule: Date2Date ->
rule Date2Date {
  from
    primitivetype_in : UML!PrimitiveType(primitivetype_in.name = 'Date')
  to
    xmltype_out : ORDB!XMLType (
      Name <- "#DATE",
      model <- primitivetype_in.Package)
  -- ActionBlock:
  do {}
}
```

Figura F-30. Código Modificado ATL –*Rule Date2Date*

En la Figura F-31 se muestra la transformación entre elementos de tipo *DataType* y elementos de tipo *StructuredType*, donde además se generan elementos de tipo *Attribute* que dependen del *StructuredType*. En esta regla ha sido necesario modificar la condición de guarda del elemento de tipo *DataType* y agregar la condición de guarda en la generación del elemento de tipo *Attribute*.

```
-- Comments -> This is a MatchedRule: DataType2UDT ->
rule DataType2UDT {
  from
    datatype_in : UML!DataType (datatype_in.name <>'Integer'
    and datatype_in.name <>'Date' and datatype_in.name <>'String')
  to
    structuredtype_out : ORDB!StructuredType (
      Name <- datatype_in.name + '<<udt>>',
      Model <- datatype_in.Package,
      attribute <-attribute_out),

    attribute_out : distinct ORDB!Attribute foreach (at in
      datatype_in.ownerattribute)(
        Name <- datatype_in.name)
  -- ActionBlock:
  do {}
}
```

Figura F-31. Código Modificado ATL – *Rule DataType2UDT*

Por último, se muestra el código que implementa los *helpers* que se generaron a partir de los elementos de tipo *Operation*. Como se ha mencionado

anteriormente, el código que implementa los *helpers* está incompleto, ya que para no complicar el meta-modelo a nivel PSM con instrucciones dependientes de un lenguaje de transformación específico no se puede recoger toda la información necesaria para poder implementarlo completamente. Por esto, solo se genera la cabecera, (*context*, *name* y *returnValue*) más un comentario donde se especifica que función debería realizar el *helper* (Figura F-32).

```
-- Comments -> This is a Helper: Package
helper def: Package() : UML!Package =
  ; -- Body: Recupera el elemento de tipo Package de UML
```

Figura F-32. Código ATL –*Helper Package*

En la Figura F-33 se muestra él código modificado del *helper Package()*. Este *helper* permite recuperar el primer elemento de tipo *Package* en el modelo origen.

```
-- Comments -> This is a Helper: Package
helper def: Package : UML!Package =
 UML!Package.allInstances()->asSequence()->first();
```

Figura F-33. Código Modificado ATL –*Helper Package()*

En la Figura F-34 se muestra el código modificado que implementa el *helper isMultivalued()*. Este *helper* verifica el valor de la propiedad *upperValue* del elemento de tipo *Property*. Si dicho valor está entre 0 y -1, lo que indica que el elemento *Property* es multivaluado, el *helper* retorna el valor *true*, caso contrario retorna *false*.

```
-- Comments -> This is a Helper: isMultivalued
helper context UML!Property def: isMultivalued() : Boolean =
  if (self.upperValue.oclIsUndefined()) then
       false
  else
       if (self.upperValue.value =(0-1)) then
             true
       else
             false
       endif
endif;
```

Figura F-34. Código Modificado ATL –*Helper isMultivalued()*

F.4.2 Código de la Transformación en RubyTL

Para realizar la generación de código desde modelos de transformación conformes al meta-modelo de RubyTL se utiliza el mecanismo de extracción brindado por TCS. Para esto, como se ha visto anteriormente, ha sido necesario realizar la definición de la sintaxis concreta del DSL de RubyTL utilizando TCS.

En la Figura F-35 se muestra la cabecera del módulo que contiene el código de la transformación en el lenguaje RubyTL. Como se puede observar se define el nombre de la transformación, UML2ORDB4ORA, y los meta-modelos destino, ORDB, y origen, UML.

```
transformation 'UML2ORDB4ORA'
input 'UML_model'
output 'ORDB_model'
```

Figura F-35. Cabecera de la Transformación en RubyTL

En la Figura F-35 se muestra la regla que implementa la transformación entre un elemento de tipo *Package* de UML y un elemento de tipo *Model* de ORDB.

```
top_rule 'Package2Model' do
    from UML_model::Package
    to ORDB_model::Model
     mapping do | package_in, model_out |
        model_out.Name  =  package_in.name
     end
end
```

Figura F-35. Código RubyTL–*Rule Package2Model*

En la Figura F-36 se muestra el código generado que implementa la transformación entre elementos de tipo *Class* del meta-modelo de UML y elemento de tipo *StructuredType* del meta-modelo de ORDB. Como se puede observar, además se generan los elementos de tipo *TypedTable* y *Method*.

```
top_rule 'Class2UDT' do
   from UML_model::Class
   to ORDB_model::StructuredType, ORDB_model::TypedTable, ORDB_model::Method
    mapping do | class_in, structuredtype_out, typedtable_out, method_out |
      structuredtype_out.Name  =  class_in.name + '<<udt>>'
      structuredtype_out.model  =  class_in.Package
      structuredtype_out.typed  =  typedtable_out
      structuredtype_out.method  =  method_out
      typedtable_out.Name  =  class_in.name + 's+<<persistent>>'
      method_out.Name  =  class_in.name
    end
end
```

Figura F-36. Código Modificado RubyTL – *Rule Class2UDT*

El código mostrado en la Figura F-36 se ha modificado, en primer lugar se agregaron en la propiedad *name* del elemento *StructuredType* y del elemento *TypedTable* los estereotipos definidos en M2DAT-DB

En la Figura F-37 se muestra la regla, modificada, que implementa la transformación entre los elementos de tipo *Property* del meta-modelo de UML y los elementos de tipo *Attribute* del meta-modelo ORDB. En particular se ha modificado la parte *filter* de la regla especificando correctamente la condición a cumplir.

```
top_rule 'Property2Attribute' do
   from UML_model::Property
   to ORDB_model::Attribute
   filter do | property_in |
    property_in.isDefined and !property_in.isMultivalued()
   end
    mapping do | property_in, attribute_out |
      attribute_out.Name  =  property_in.name
      attribute_out.structured  =  property_in.class
    end
end
```

Figura F-37. Código Modificado RubyTL – *Rule Property2Attribute*

En la Figura F-38 se muestra el código modificado que implementa la transformación entre elementos de tipo *Property* de UML y elementos de tipo *Method* de ORDB. Las modificaciones que se realizaron son las mismas que las explicadas anteriormente.

```
top_rule 'Property2Method' do
   from UML_model::Property
   to ORDB_model::Method
   filter do | property_in |
      property_in.isDerived
   end
    mapping do | property_in, method_out |
     method_out.Name  =  property_in.name
     method_out.structured  =  property_in.class
    end
end
```

Figura F-38. Código Modificado RubyTL – *Rule Property2Method*

En la Figura F-39 se muestra el código modificado que implementa la transformación entre elementos de tipo *Property* de UML y elementos de tipo *Attribute* y *Stored Nested Table* de ORDB, en este caso en la condición de guarda se controla, por medio de la llamada al *helper IsMultivalued*, que el atributo sea multivaluado. En esta regla se invoca la creación de elementos de tipo *Nested Table* por medio de la invocación a la regla de tipo *Lazy generateNestedTableMultivalued*.

```
top_rule 'PropertyMultivalued2NT' do
   from UML_model::Property
   to ORDB_model::Attribute, ORDB_model::StoredNestedTable
   filter do | property_in |
    !property_in.IsDefined and property_in.IsMultivalued()
   end
    mapping do | property_in, attribute_out, storednestedtable_out |
     attribute_out.Name  =  property_in.name
     attribute_out.structured  =  property_in.class
     attribute_out.Type  =  property_in.generatedNestedTableMultivalued
     storednestedtable_out.Name  =  property_in.name
     storednestedtable_out.attribute  =  attribute_out
    end
end
```

Figura F-39. Código Modificado RubyTL– *Rule PropertyMultivalued2NT*

En la Figura F-40 se muestra la regla de transformación *generateNestedTableMultivalued.* Esta regla es de tipo *Copy*, es decir que para que se ejecute debe ser invocada explícitamente y permite la creación de elementos de tipo *Nested Table* a partir de elementos de tipo *Property*.

```
copy_rule 'generatedNestedTableMultivalued' do
   from UML_model::Property
   to ORDB_model::NestedTableType
    mapping do | property_in, nestedtabletype_out |
     nestedtabletype_out.Name  =  property_in.name
     nestedtabletype_out.model  =  property_in.Package
    end
end
```

Figura F-40. Código Modificado RubyTL– *Copy Rule generateNestedTableMultivalued*

En La Figura F-41 se muestra el código modificado para la transformación entre elementos de tipo *PrimitiveType* y elementos de tipo *ANSICharacterType*, donde en la condición de guarda se establece que el valor de la propiedad *name* del tipo *PrimitiveType* debe ser *"String"*. En la regla se establece la propiedad *name* del tipo *XMLType* como *"VARCHAR2"*.

```
top_rule 'StringToVarchar' do
   from UML_model::PrimitiveType
   to ORDB_model::ANSICharacterType
   filter do | primitivetype_in |
     primitivetype_in.name = = 'String'
   end
    mapping do | primitivetype_in, ansicharactertype_out |
      ansicharactertype_out.Descriptor  =  '#VARCHAR'
      ansicharactertype_out.model  =  primitivetype_in.Package
    end
end
```

Figura F-41. Código RubyTL – *Rule String2VarChar*

La Figura F-42 muestra el código modificado que implementa la transformación entre elementos de tipo *PrimitiveType* del meta-modelo de UML y elementos de tipo *ANSICharacterType* del meta-modelo ORDB4ORA. Como se puede observar en la condición de guarda se controla que el valor de la propiedad *name* del elemento *PrimitiveType* debe ser *"Integer"*, si se cumple dicha condición se genera un elemento de tipo *ANSICharacterType* cuya propiedad *name* tiene el valor *"NUMBER"*.

```
top_rule 'Integer2Number' do
   from UML_model::PrimitiveType
   to ORDB_model::ANSINumberType
   filter do | primitivetype_in |
     primitivetype_in.name == 'Integer'
   end
    mapping do | primitivetype_in, ansinumbertype_out |
      ansinumbertype_out.Descriptor  =  '#NUMBER'
      ansinumbertype_out.model  =  primitivetype_in.Package
    end
end
```

Figura F-42. Código Modificado RubyTL – *Rule Integer2Number*

En la Figura F-43 se muestra el código modificado que implementa la regla de transformación entre elementos de tipo *PrimitiveType* y elementos de tipo *ANSICharacterType*. En esta regla, en la condición de guarda se controla que el valor de la propiedad *name* del elemento *PrimitiveType* debe ser *"Date"*, si se cumple dicha condición se genera un elemento de tipo *ANSICharacterType* cuya propiedad *name* tiene el valor *"DATE"*.

```
top_rule 'Date2Date' do
   from UML_model::PrimitiveType
   to ORDB_model::DatetimeType
   filter do | primitivetype_in |
      primitivetype_in.name == 'Date'
   end
    mapping do | primitivetype_in, datetimetype_out |
      datetimetype_out.Descriptor  =  '#DATE'
      datetimetype_out.model  =  primitivetype_in.Package
   end
end
```

Figura F-43. Código Modificado RubyTL–*Rule Date2Date*

En la Figura F-44 se muestra la transformación entre elementos de tipo *DataType* y elementos de tipo *StructuredType*, donde además se generan elementos de tipo *Attribute* que dependen del *StructuredType*. En esta regla ha sido necesario modificar la condición de guarda del elemento de tipo *DataType* y agregar la condición de guarda en la generación del elemento de tipo *Attribute*.

```
top_rule 'DataType2UDT' do
   from UML_model::DataType
   to ORDB_model::StructuredType, ORDB_model::Attribute
   filter do | datatype_in |
    datatype_in.name!='Integer' and datatype_in.name!='Date'
    and datatype_in.name!='String'
   end
    mapping do | datatype_in, structuredtype_out, attribute_out |
      structuredtype_out.Name  =  datatype_in.name
      structuredtype_out.Model  =  datatype_in.Package
      attribute_out.Name  =  datatype_in.name
    end
end
```

Figura F-44. Código Modificado RubyTL – *Rule DataType2UDT*

Apéndice G: Implementación de Meta-Modelos

En el capítulo 3 se presenta de forma resumida la especificación e implementación de los meta-modelos propuestos en cada uno de los niveles de MeTAGeM. A continuación se presenta la especificación e implementación completa.

G.1 Especificación de M-TIP

A partir de la identificación de los tipos de elementos a transformar y de las relaciones entre los mismos, se procede a definir el meta-modelo que permitirá modelar transformaciones de modelos a nivel PIM. En la Figura G-1 se muestra de forma gráfica el meta-modelo definido. A continuación se explican cada uno de los elementos definidos:

- *ModelRoot*. Es la meta-clase que representa el elemento principal del meta-modelo. Está compuesta por tres tipos de elementos: *InModelTransf*, *OutModelTransf* y *Relations*. Las relaciones de composición entre la meta-clase *ModelRoot* y las meta-clases *InModelTransf*, *OutModelTransf* y *Relations* (*inputModel*, *outputModel* y *relations*, respectivamente) se definen con cardinalidad 1 a N lo que significa que se podrían tener varios (meta-)modelos origen, varios (meta-)modelos destino y se podrían identificar varios tipos de relaciones entre los elementos.
- *ModelTransf*. Es una meta-clase, definida en forma abstracta, que agrupa los meta-modelo origen (*InModelTransf*) y destino (*OutModelTransf*).
- *InModelTransf*. Es la meta-clase que identifica el meta-modelo origen de la transformación. Es una especialización de la meta-clase *ModelTransf* y está compuesta por elementos de tipo *InElement*.
- *OutModelTransf*. Es la meta-clase que identifica el meta-modelo destino de la transformación. Es una especialización del elemento *ModelTransf* y está compuesta por elementos de tipo *OutElement*.
- *Relations.* Es la meta-clase que permite establecer las relaciones de transformación entre los elementos de cada uno de los meta-modelos origen y destino. Tiene propiedades definidas: la propiedad *typeE* esta definida como un enumerado de tipo *MyTypeElement*, y permite identificar los tipos de elementos de los meta-modelos origen y destino que intervienen en la relación; los valores que puede tomar son: *MyEClassifier*, *MyEAttribute*,

MyEReference. La propiedad *typeRelation*, definida como un enumerado de tipo *MyTypeRelation*, permite que el desarrollador determine si la relación que está definiendo es una relación principal (relación que se establece entre dos elementos y que no depende de ninguna otra relación) en cuyo caso el valor de la propiedad será *isMain*, secundaria (relación condicionada a la existencia de otra relación) en cuyo caso el valor de la propiedad será *isSecondary*; o abstracta (que deberá ser implementada posteriormente) en cuyo caso el valor de la propiedad será *isAbstract*. Para este último caso, se define una relación recursiva en la clase *Relation* que permitirá indicar qué *Relation* extiende a qué *Relation*.

- *OnetoOne*, *OnetoZero*, *ZerotoOne*, *OnetoMany*, *ManytoOne* y *ManytoMany*. Son las meta-clases definidas a partir de la identificación del tipo de relación entre los elementos de los meta-modelos que participan en la transformación. Se definen como una especialización de la meta-clase *Relations*. Cada una de estas meta-clases esta compuestas por elementos del meta-modelo origen, *InElement*, y elementos del meta-modelo destino, *OutElement*; en cada caso, dependiendo del tipo de relación, variará la cardinalidad de los elementos relacionados.
- *InElement.* Es la meta-clase que representa a cada uno de los elementos del meta-modelo origen que participan en cada relación. Tiene una propiedad *GuardCondition* de tipo *String* que permite indicar consideraciones especiales que se deben satisfacer para que se establezca la relación. Además, se define una relación (*invoked*) con la meta-clase *Relation* que permitirá indicar si es necesario la llamada a algún tipo de relación especial para obtener el elemento del meta-modelo destino.
- *OutElement*. Es la meta-clase que representa cada uno de los elementos del meta-modelo destino que participan en cada relación. Puede estar compuesta por elementos de las meta-clases *OnetoOne*, *ZerotoOne* o *OnetoMany*, lo que permite realizar la transformación de nuevos elementos a partir de un elemento generado; el ejemplo más representativo de esto es el de la transformación de un elemento del tipo *Classifier*, que implica la generación de los elementos del tipo *Property* o *Association* relacionados.
- *MyAttributetype*, *MyTypeElement*, *MyTypeRelation*. Son meta-clases definidas como enumerados, con diferentes opciones, que son utilizados como tipos de datos por algunas de las meta-clases definidas.

Es necesario aclarar que en el momento de la implementación del meta-modelo será necesario incluir restricciones que permitan realizar el control y

validación de los modelos. Así, por ejemplo, se debe poder controlar que una meta-clase del tipo *Relation* sólo puede extender a meta-clases del tipo *Relation* con la propiedad *typeRelation* en *isAbstract*.

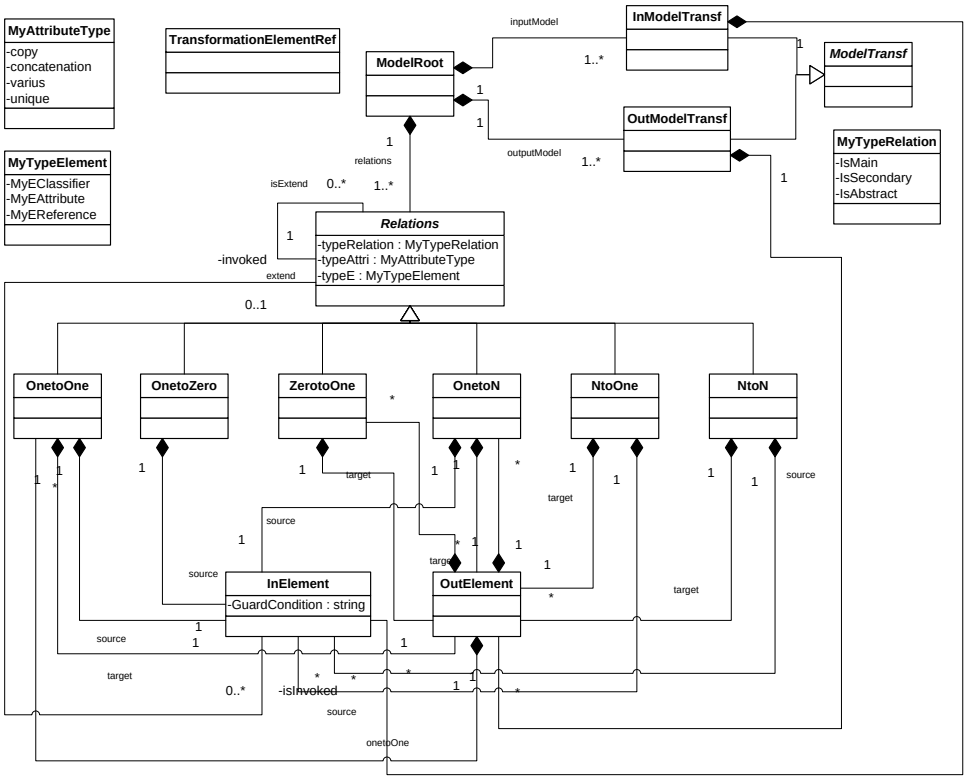

Figura G-1. Meta-modelo M-TIP

G.2 Especificación de M-LTH

En la Figura G-2 se muestra el meta-modelo para el modelado de las transformaciones de modelos a nivel PSM siguiendo la aproximación híbrida. A continuación se explican cada uno de los elementos.

- *Module*. Es la meta-clase principal que agrupa al resto de las meta-clases. Está compuesta por cuatro meta-clases: *InMetaModel*, *OutMetaModel*, *Rule* y *Operation*. Las relaciones de composición con cada meta-clase, *inMM*, *outMM*, *rule* y *operations*, estan definidas con cardinalidad máxima de N.
- *InMetaModel*. Es la meta-clase que permite identificar el (o los) meta-modelo/s origen de la transformación. Tiene la propiedad de *name_mm*, definida de tipo *String*, donde se especifica el nombre del meta-modelo origen y una propiedad *type_mm*, también de tipo *String*, donde se especifica el tipo del meta-modelo origen. Además, tiene una relación *elements* con la meta-clase *SourceElementRule*, definida con cardinalidad de 0 a N que permite identificar los elementos que pertenecen al meta-modelo origen.
- *OutMetaModel*. Es la la meta-clase que permite identificar el (o los) meta-modelo/s destino de la transformación. Tiene la propiedad de *name_mm*, definida de tipo *String*, donde se especifica el nombre del meta-modelo destino y una propiedad *type_mm*, también de tipo *String*, donde se especifica el tipo del meta-modelo destino. Además, tiene una relación *elements* con la meta-clase *TargetElementRule*, definida con cardinalidad de 0 a N que permite identificar los elementos que pertenecen al meta-modelo destino.
- *Rule*. Es la meta-clase donde se definen las reglas de transformación. Tiene diferentes propiedades: *name_rule*, que permite especificar el nombre de la regla; *isAbstract* que permite definir la la regla como abstracta; *isMain*, que permite definir la regla como principal; las propiedades *typeAttribute* y *typeElement* que permiten identificar el tipo de elementos que se relacionan y por último la propiedad *comment* permite que el usuario establezca algún tipo de definición a modo de comentario sobre la regla. Cada regla está compuesta por elementos del meta-modelo origen (*in)* y elementos del meta-modelo destino (*out*). A su vez, una regla cuyo atributo *isAbstract* tenga el valor verdadero debe ser extendida por otra regla (relación reflexiva *isExtended*).

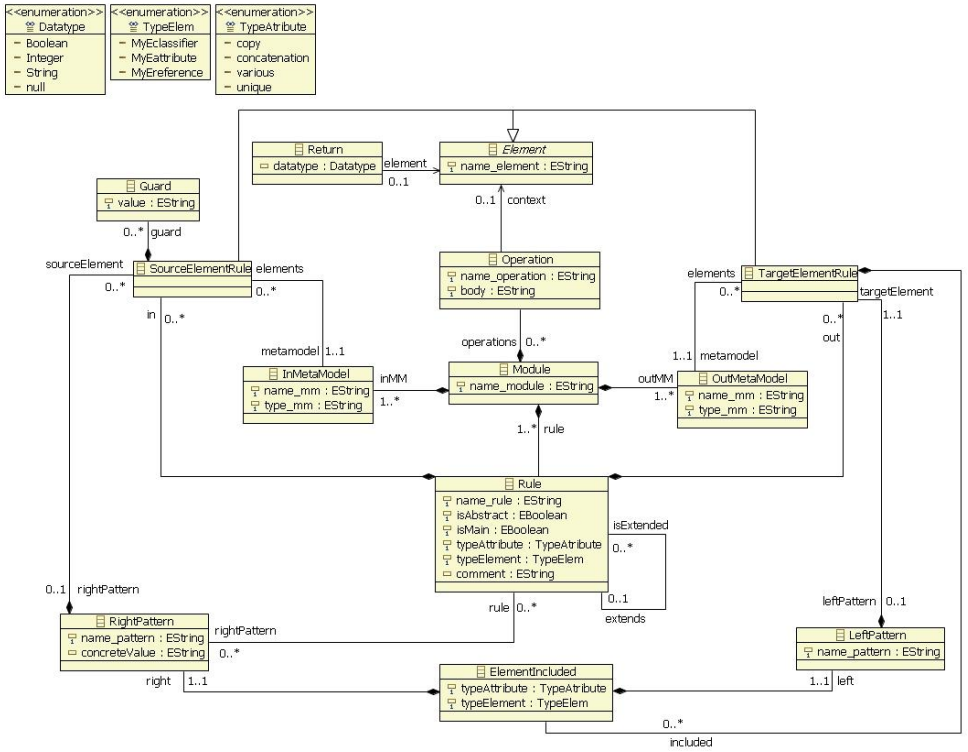

Figura G-2. Meta-modelo M-LTH

- *Operation*. Es la meta-clase que permite definir funciones. Tiene las propiedades *name_operation*, donde se especifica el nombre de la función y *body* donde se puede detallar la operación que se debe implementar por medio de la función. Además tiene una relación, *context*, con la meta-clase *Element* que permite definir el elemento a partir del cual se realiza la función. Por último, está compuesto, *returnType*, por elementos de la meta-clase *ReturnValue* donde se estable el tipo de valor que retorna la función.
- *Element.* Es la meta-clase, definida como abstracta, que agrupa los elementos que participan en las reglas de transformación. Tiene la propiedad *name_element* donde se especifica el nombre del elemento.
- *ReturnValue.* Es la meta-clase que representa los elementos de retorno en una función (*Operation*). El valor de retorno de una función puede ser o bien un tipo de dato, representado por la propiedad *DataType*, o bien un elemento, representado por la relación *element* con la meta-clase *Element*.
- *SourceElementRule*. Representa cada uno de los elementos del meta-modelo origen que participa en la relación. Se define como una especialización de la meta-clase *Element* y tiene definido una relación del tipo composición, *guard*, con la meta-clase *Guard*, esta relación permite definir condiciones que deberán cumplirse en el elemento del meta-modelo origen para que se ejecute la regla.
- *TargetElementRule*. Representa cada uno de los elementos del meta-modelo destino que participa en la relación. Se define como una especialización de la meta-clase *Element* y esta compuesta por elementos de la meta-clase *ElementIncluded*, que permite realizar la transformación de los sub-elementos dependientes del elemento del meta-modelo destino que se esté transformando.
- *ElementIncluded*. Permite identificar los elementos que se transforman a partir de un elemento de tipo *TargetElementIncluded*. Cada meta-clase *ElementIncluded* está compuesta por elementos de tipo *RightPattern* y elementos de tipo *LeftPattern*.
- *RightPattern* y *LeftPattern*. Forman parte de los elementos del tipo *ElementIncluded* y permiten identificar la parte derecha y la parte izquierda de cada regla. Están compuestos por elementos del tipo *SourceElementRule* y *TargetElementRule*, respectivamente.

- *TypeAttribute, TypeElem, DataType*. Son meta-clases definidas como enumerados, con diferentes opciones, que son utilizados como tipos de datos por algunas de las meta-clases definidas.

Es importante mencionar, que si bien, para el desarrollo de esta tesis se selecciona la aproximación híbrida para el modelado de las transformaciones a nivel PSM, como futuro trabajo se pretende implementar el resto de las aproximaciones.

G.3 Implementacion de M-TIP

El meta-modelo M-TIP se define como una extensión del meta-modelo base de AMW. Para realizar la definición de la sintaxis abstracta de M-TIP se utiliza el lenguaje KM3 (*Kernel MetaMetaModel*, [25]) que es el lenguaje utilizado para realizar la extensión del meta-modelo base de AMW. De acuerdo a la especificación de M-TIP realizada (Figura G-1) a continuación se muestra la implementación de las meta-clases más importantes de dicho meta-modelo, la implementación completa se puede consultar en el CD adjunto:

- *Meta-Clase ModelRoot*: es el elemento raíz del meta-modelo, del cúal dependen el resto de los elementos. En la Figura G-3 se muestra el código correspondiente a la definición de la meta-clase. Esta meta-clase se define como una extensión de la clase *WModel* del meta-modelo base de AMW por lo que hereda todas las propiedades del mismo. Además tiene definido0s varios elementos de tipo referencia: a) *inputModel*, referencia a la meta-clase *InModelTransf* que representa el/los meta-modelos origen de la relación; b) *outputModel*, referencia a la meta-clase *OutModelTransf* que representa el/los meta-modelos destino de la relación y c) *relations*, referencia a la meta-clase *Relations* que agrupa todos los tipos de relaciones definidas en MeTAGeM.

```
class ModelRoot extends WModel{
  -- @subsets wovenModel
  reference inputModel [1-*] container : InModelTransf;
  -- @subsets wovenModel
  reference outputModel [1-*] container : OutModelTransf;
  -- @subsets ownedElement
  reference relations [1-*] container : Relations;}
```

Figura G-3. Definición de la Meta-clase *ModelRoot*

- *Meta-clase ModelTransf*: representa los meta-modelos que participan en la transformación, se define como una meta-clase abstracta que hereda de la meta-clase *WModelRef* del meta-modelo base de AMW. Esta meta-clase es instanciada por medio de dos meta-clases: *InModelTransf* y *OutModelTransf* que representan a los meta-modelos origen y destino respectivamente (Figura G-4).

```
-- @welementRefType TransformationElementRef
abstract class ModelTransf extends WModelRef {}

-- @welementRefType TransformationElementRef
class InModelTransf extends ModelTransf{}

-- @welementRefType TransformationElementRef
class OutModelTransf extends ModelTransf{}

-- @wmodelRefType ModelTransf
class TransformationElementRef extends WElementRef {}
```

Figura G-4. Definición de la Meta-clase *ModelTransf*

- *Meta-clase Relations:* define todos los tipos de relaciones identificadas entre los elementos de los meta-modelos origen y destino (sección 3.1.2.1), esta definida como una meta-clase abstracta que hereda de la meta-clase *WLink* del meta-modelo base de AMW (Figura G-5). La meta-clase *Relations*, o mejor dicho, las meta-clases que la implementen, pueden ser invocadas desde cero o muchos elementos del tipo *InElement*, esto se indica a través de la referencia *isInvoked* definido como opuesto (*oppositeOf*) a la referencia *invoked* de la meta-clase *InElement*. El atributo *typeRelation* definido como de tipo *MyTypeRelation* se utiliza para identificar el tipo de relación que se representa, de esta manera, al momento de definir las transformaciones entre los modelos de transformación de nivel PIM y los modelos de transformación de nivel PSM, este atributo servirá para identificar a que tipo de elemento se debe transformar. El atributo *typeRelation* puede tomar los siguientes valores: a) *isMain*, significa que la relación que se representa es una relación principal, cuya existencia no esta condicionada por la existencia de ninguna otra relación; b) *isSecondary*, significa que la existencia de la relación está condicionada a la existencia de otra relación y c) *isAbstract*, significa es una relación abstracta y que debe ser extendida por otro tipo de relación. Para este último caso, se define la referencia *isExtend*, para indicar la relación que la

extiende y la referencia *extend* para indicar a que relación extiende, ambos definidos como opuestos entre sí.

```
abstract class Relations extends WLink{
  attribute typeAttri : MyAttributeType;
  attribute typeE: MyTypeElem;
  attribute typeRelation : MyTypeRelation;
  -- @subsets child
  reference isInvoked[0-*]: InElement oppositeOf invoked;
  -- @subsets child
  reference extend [0-1]: Relations oppositeOf isExtend;
  reference isExtend [*]: Relations oppositeOf extend;}
```

Figura G-5. Definición de la Meta-clase *Relations*

A partir de la especificación de la sintaxis abstracta del meta-modelo M-TIP se procede a realizar la especificación de su sintaxis concreta. Para esto se utiliza la herramienta AMW.

Referencias

1. Acerbis, R., Bongio, A., Brambilla, M. & Butti, S. (2007). *WebRatio 5: An Eclipse-Based CASE Tool for Engineering Web Applications*, in Web Engineering, 2007, 501-505.
2. Acerbis, R., Bongio, A., Brambilla, M., Butti, S., Ceri, S. & Fraternali, P. (2008). *Web Applications Design and Development with WebML and WebRatio 5.0*. In S. B. Heidelberg (Ed.), Objects, Components, Models and Patterns. 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings (Vol. 11, pp. 392-411).
3. Acuña, C., *PISA – Arquitectura de integración de portales Web: un enfoque dirigido por modelos y basado en servicios Web semánticos*. PhD Thesis. Universidad Rey Juan Carlos, Octubre 2007.
4. Acuña, C. Minoli, M. Vara, J.M. *Model Driven Development of Semantic Web Services using Eclipse.* 2009 Mexican International Conference on Computer Science, ENC'09. Mexico City, Mexico. (Submitted).
5. Agrawal, A. (2003). *Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck*. Paper presented at the 18th IEEE International Conference on Automated Software Engineering (ASE 2003) Montreal, Canada.
6. Amelunxen, C., Königs, A., Rötschke, T., & Schürr, A. (2006). *MOFLON: A Standard-Compliant Metamodelling Framework with Graph Transformations*. Paper presented at the Second European Conference on Model Driven Architecture – Foundations and Applications, ECMDA-FA 2006 Bilbao, Spain.
7. Allilaire, F. Jouault, F. *ATL Basic Examples and Patterns*. Retrieved 8 January, 2009, from: http://www.eclipse.org/m2m/atl/basicExamples_Patterns/.
8. Altova. (2008). *XML Schema* [Software]. Available from http://www.altova.com/dev_portal_xml_schema.html.
9. Ambler, S. (2006). *Comparing the Various Approaches to Modelling in Software Development*. Retrieved 23 August, 2007, from: http://www.agilemodeling.com/essays/modelingApproaches.htm
10. Amelunxen, C., Knigs, A., Rtschke, T., & Schrr, A. (2006). *MOFLON: A Standard-Compliant Metamodelling Framework with Graph Transformations*. Paper presented at the Model Driven Architecture - Foundations and Applications: Second European Conference (ECMDA-FA 2006), Bilbao, Spain.
11. AndroMDA. (2008). *AndroMDA.org* - Home. Retrieved 16 October 2008, from http://www.andromda.org/
12. Arsenault, S. (2007). *Contributing Actions to the Eclipse Workbench*. Eclipse Corner Articles, January, 2007. Retrieved from http://www.eclipse.org/articles/article.php?file=Article-action-contribution/index.html.
13. *ArcStyler 5.5*, Interactive Objects Software GmbH (iO GmbH). Freiburg, Germany.
14. Atkinson, C., & Kuhne, T. (2003). *Model-driven development: a metamodelling foundation*. IEEE Software, 20(5)
15. *ATL VM Code Generator* (ACG), http://wiki.eclipse.org/ACG.
16. Atzeni, P., Cappellari, P. y Bernstein, P. *A. Model-Independent Schema and Data Translation*. En EDBT 2006, LNCS 3896, pp. 368–385. Springer, 2006.
17. Atzeni, P., Cappellari, P. y Bernstein, P. A. *Modelgen: Model independent schema translation*. En ICDE, Tokyo, pp. 1111–1112. IEEE Computer Society, 2005.
18. Atzeni, P. Ceri, S. Paraboschi, S. & Torlone, R. (1999). *Database Systems. Concepts, Languages and Architectures*, McGraw-Hill.
19. Atzeni, P., Cappellari, P., Torlone, R., Bernstein, P., A., & Gianforme, G. (2008). *Model-independent schema translation*. The VLDB Journal, 17(6), 1347-1370.

20. ATLANMOD. (2009). *Metamodel Zoos*. Retrieved, March 20, 2009, from http://www.emn.fr/x-info/atlanmod/index.php/Zoos.
21. Balasubramaniam, R., Curtis, S., Timothy, P., & Michael, E. (1997). *Requirements traceability: Theory and practice*. Annals of Software Engineering, 3, 397-415.
22. Balogh, A., & Varro, D. (2006). *Advanced model transformation language constructs in the VIATRA2 framework*. Paper presented at the ACM symposium on Applied computing (SAC 2006), Dijon, France.
23. Baresi, L. y Heckel, R. 2002. *Tutorial Introduction to Graph Transformation: A Software Engineering Perspective*. En Proceedings of the First international Conference on Graph Transformation). LNCS 2505. Springer-Verlag, London, pp. 402-429, 2002.
24. Bézivin, J. (2001). *From Object Composition to Model Transformation with the MDA*. Paper presented at the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), Washington DC, USA.
25. Bézivin, J., & Jouault, F. (2006, April, 2006). *KM3: a DSL for metamodel specification.* Paper presented at the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006), Bologna, Italy.
26. Bézivin, J. (2004). *In search of a Basic Principle for Model Driven Engineering.* Novatica/Upgrade, V(2), 21-24.
27. Bézivin, J. (2005). *On the unification power of models. Journal of Software and Systems Modeling*, 4(2), 171-188.
28. Bézivin, J., Joualt, F., Rosenthal, P., & Valduriez, P. (2005). *Modeling in the large and modeling in the small*. Paper presented at the Model Driven Architecture, Europen MDA Workshops: Foundations and Applications (MDA-FA: 2003-2004), Twente, The Netherlands.
29. Bézivin, J., Hillairet, G., Jouault, F., Piers, W., & Kurtev, I. (2005). *Bridging the MS/DSL Tools and the Eclipse Modeling Framework*. Paper presented at the OOPSLA2005 International Workshop on Software Factories, San Diego, USA.
30. Bézivin, J., Jouault, F., Brunette, C., Chevrel, R., & Kurtev, I. (2005). *Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF)*. Paper presented at the OOPSLA2005 International Workshop on Best Practices for Model Driven Software Development San Diego, California, USA.
31. Bézivin, J., Rumpe, B., Schürr, A., & Tratt, L. (2005). *Mandatory Example Specification. CFP of the Model Transformations in Practice*. Workshop at MoDELS 2005, Montego Bay, Jamaica.
32. Bézivin, J. (2005). *Some Lessons Learnt in the Building of a Model Engineering Platform.* Paper presented at the 4th Workshop in Software Model Engineering (WISME), Montego Bay, Jamaica.
33. Bezivin, J., Jouault, F., Touzet, D. (2005). *An introduction to the ATLAS Model Management Architecture*. (LINA Research Report Nº05.01). Nantes: University of Nantes. Retrieved June 20, 2007, from http://www.sciences.univ-nantes.fr/lina/atl/www/papers/RR-LINA2005-01.pdf.
34. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., & Lindow, A. (2006). *Model Transformations? Transformation Models!*. Paper presented at the 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2006, Genève, Italy.
35. Bezivin, J., Pierantonio, A. & Vallecillo, A. (2006). *Special track on model transformation (MT 2006).* In Proceedings of the 2006 ACM symposium on Applied computing, Dijon (France), 2006, pp. 1186-1187.
36. Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M. P., Jouault, F., Kolovos, D., et al. (2006). *A Canonical Scheme for Model Composition.* Paper presented at the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'06), Bilbao, Spain.
37. Bézivin, J., Vallecillo, A., García-Molina, J., & Rossi, G. (2008). *MDA at the Age of Seven: Past, Present and Future.* UPGRADE, IX(2), 4-7.

38. Bézivin, J. (2009). *Advances in Model-Driven Engineering.* Invited Talk at JISBD 2009, Jornadas de Ingeniería del Software y Bases de Datos. San Sebastian, Spain.
39. Bezivin, J., Farcet, N., Jezequel, J.M., Langlois, B. and Pollet, D. *Reflective Model Driven Engineering.* In UML, volume 2863 of LNCS, pages 175-189, 2003.
40. Biolchini, J., Gomes Mian, P., Cruz Natali, A.C., Horta Travasso, G. *Systematic Review in Software Engineering.* Technical Report ES 679/05. May, 2005
41. Bohlen, M. (2006). *QVT und Multi-Metamodell-Transformationen in MDA.* OBJEKTspektrum. Vol. 2 (March/April 2006). Translated in: http://www.andromda.org/jira/secure/attachment/10744/bohlen_OS_02_06_k4.pdf.
42. Bollati, V. A., Sánchez, V., Vela, B. y Marcos, E. *Análisis de QVT Operational Mappings: un caso de estudio* (2009). En Actas del VI Taller sobre Desarrollo de Software Dirigido por Modelos, MDA y Aplicaciones (DSDM'09). : V. Pelechano, O. Avila-García, J. R. Romero. ISSN: 1988–3455. San Sebastían (España).
43. Booch, G., Brown, A. W., Iyengar, S., Rumbaugh, J., & Selic, B. (2004). *An MDA Manifesto*. Business Process Trends/MDA Journal.
44. Borland. (2008). *Software Architecture Design, Visual UML & Business Process Modeling from Borland*. Retrieved 16 October 2008, from http://www.borland.com/us/products/together/index.html
45. Boronat, A., Carsí, J., & Ramos, I. (2006). *Algebraic Specification of a Model Transformation Engine.* Paper presented at the Fundamental Approaches to Software Engineering (FASE'06), Vienna, Austria.
46. Boronat, A. (2007). *Ph. D Thesis. A formal framework for model management*. Technical University of Valencia, Valencia.
47. Braun, P., & Marschall, F. (2003). *The Bi-directional Object-Oriented Transformation Language* (No. TUM-I0307). Munich, Germany: Technische UniversitätMünchen.
48. Bruel, J-M. (ed.) (2006). *Model Transformations in Practice Workshop*. In Proceedings of Satellite Events at the MoDELS 2005 Conference, LNCS, Vol. 3844 (Springer, 2006).
49. Brun, C. (2007). *EMF Compare: One year later*. Eclipse Summit Europe, Eclipse Modeling Symposium. Ludwisburg, Germany. [PDF Document]. Retrieved from: http://www.eclipsecon.org/summiteurope2007/presentations/ESE2007_EMFCompare.pdf, May, 20, 2009.
50. Budinsky, F., Merks, E., & Steinberg, D. (2008). *Eclipse Modeling Framework 2.0 (2nd Edition):* Addison-Wesley Professional.
51. Bunge, M. (1979). *La Investigación Científica*. Barcelona: Ariel.
52. Burstall, R. M., & John, D. (1977). *A Transformation System for Developing Recursive Programs.* Journal of the ACM, 24(1), 44-67.
53. Buttner, F., & Gogolla, M. (2004). *Realizing UML Metamodel Transformations with AGG*, In Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004), Barcelona, Spain.
54. Cabot, J., & Teniente, E. (2006). *Constraint Support in MDA Tools: A Survey*. Paper presented at the Second European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain.
55. Cabot, J., Clarisó, R., Guerra, E., & de Lara, J. (2008). *An Invariant-Based Method for the Analysis of Declarative Model-to-Model Transformations.* Paper presented at the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2008, Toulouse, France.
56. Cáceres, P., De Castro, V., Vara, J. M., & Marcos, E. (2006, April 23 - 27, 2006). *Model Transformations for Hypertext Modeling on Web Information Systems*. Paper presented at the ACM Symposium on Applied computing (SAC), Dijon, France.
57. Cicchetti, A., Ruscio, D. D., Eramo, R., & Pierantonio, A. (2008). *Automating Co-evolution in Model-Driven Engineering*. Paper presented at the 12th International IEEE Enterprise Distributed Object Computing Conference - EDOC 2008, München, Germany.

58. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al. (1999). *The Maude System*. Paper presented at the 10th International Conference on Rewriting Techniques and Applications, Trento, Italy.
59. *Codagen Architect*, Codagen Technologies Corp., Montreal, Canada.
60. Cook, S., Jones, G., Kent, S., & Cameron Wills, A. (2007). *Domain-Specific Development with Visual Studio DSL* Tools Addison-Wesley Professional.
61. Cook, S., & Kent, S. (2008). *The Domain-Specific IDE*. UPGRADE, IX(2), 17-21.
62. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., & Padberg, J. (1996). *The Category of Typed Graph Grammars and its Adjunctions with Categories*. Paper presented at the 5th International Workshop on Graph Gramars and Their Application to Computer Science, Williamsburg, VA, USA.
63. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A. & Varro, D. (2002). *VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models*. In Proceedings of 17th IEEE International Conference on Automated Software Engineering (ASE'02), IEEE Computer Society, Los Alamitos, CA, USA, 2002.
64. Czarnecki, K., & Eisenecker, U. (1999). *Components and Generative Programming.* Paper presented at the 7th European Software Engineering Conference / 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering — ESEC/FSE '99, Toulouse, France.
65. Czarnecki, K., & Helsen, S. (2003). *Classification of Model Transformation Approaches.* Paper presented at the 2nd OOPSLA Workshop on Generative Techniques in the Context of MDA, Anaheim, USA.
66. Czarnecki, K., & Helsen, S. (2006). *Feature-based survey of model transformation approaches*. IBM Systems Jorunal, 45(3), 621-645.
67. Czarnecki, K., Natahan foster, J., Hu, Z., Lämmel, R., Schürr, A. and Terwiligger, J. F. *Bidirectional Transformations: a cross-discipline perspective. GRACE meeting notes, state of the art and outlook.* Proceedings of the International Conference on Model-Transformations (ICMT 2009). 29-30 June 2009, Zurich (Switzerland).
68. de Castro, M. V, *Una aproximación MDA para el desarrollo orientado a servicios de sistemas de información Web: del modelo de negocio al modelo de composición de servicios Web.* Tesis Doctoral. Universidad Rey Juan Carlos, Marzo 2007.
69. de Castro, V., Marcos, E., & Cáceres, P. (2004). *A User Service Oriented Method to Model Web Information Systems.* In WISE (Vol. 3306, pp. 41-52): Springer.
70. de Castro, V., Marcos, E., & Lopez Sanz, M. (2006). A model driven method for service composition modelling: a case study. International Jorunal on Web Engineering Technology, 2(4), 335-353.
71. de Castro, V., Vara, J. M., & Marcos, E. (2007). *Model Transformation for Service-Oriented Web Applications Development*. Paper presented at the 3rd International Workshop on Model-Driven on Web Engineering (MDWE 2007), Como, Italy.
72. de Castro, V., Vara, J. M., Herrmann, E., & Marcos, E. (2008). *A Model Driven Approach for the Alignment of Business and Information Systems Models*. Paper presented at the Proceedings of the 2008 Mexican International Conference on Computer Science, ENC'08. Mexicali, Baja California, *Mexico*
73. de Castro, V., Vara, J. M., Herrmann, E., & Marcos, E. (2008, September 2008). *From Real Computational Independent Models to Information System Models: An MDE approach.* Paper presented at the 4th International Workshop on Model-Driven Web Engineering (MDWE 2008), Toulouse, France.
74. De Lara, J., Vangheluwe, H. & Alfonseca, M. (2004). *Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM3*, Journal on Software and Systems Modelling, Vol 3(3).
75. Den Han, J. (2008, June 11). *MDA MDD MDE MDSD MDSE: help!*. Retrieved from http://www.theenterprisearchitect.eu/
76. Den Han, J. (2009, January 9). *MDE - Model Driven Engineering - reference guide.* Retrieved from http://www.theenterprisearchitect.eu/

77. Didonet Del Fabro, M., Bézivin, J., & Valduriez, P. (2006). *Model-Driven Tool Interoperability: An Application in Bug Tracking.* Paper presented at the OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France.
78. Didonet Del Fabro, M., Bézivin, J. & Valduriez P. (2006). *Weaving Models with the Eclipse AMW plugin.* Eclipse Modeling Symposium, Eclipse Summit Europe, Esslingen, Alemania.
79. Didonet Del Fabro, M. (2007). *Metadata management using model weaving and model transformation.* Ph.D. Thesis University of Nantes, Nantes, France.
80. Eclipse Foundation. (2009). *GMF Tutorial*. Retrieved November, 2008, from: http://wiki.eclipse.org/GMF_Tutorial.
81. Efftinge, S., Friese, P., & Köhnlein, J. (2008). *Best Practices for Model-Driven Software Development*. InfoQ. Retrieved from http://www.infoq.com/articles/model-driven-dev-best-practices.
82. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation.*Vol(1). World Scientific.
83. Estévez, A., Padrón, J., Sánchez Rebull, V., & Roda, J. L. (2006). *ATC: A Low-Level Model Transformation Language.* Paper presented at the II International Workshop on Model-Driven Enterprise Information Systems 2006 (MDEIS 2006) in 8th International Conference on Enterprise Information Systems (ICEIS) 2006, Pahos, Chipre.
84. Favre, J. (2004). *Towards a Basic Theory to Model Model Driven Engineering.* Paper presented at the Workshop on Software Model Engineering, WISME 2004, joint event with UML2004, Lisbon, Portugal.
85. France Telecom, F. (2008). *SmartQVT - A QVT implementation*. Retrieved 16 October 2008, from http://smartqvt.elibel.tm.fr/
86. Friese, P., Efftinge, S., & Köhnlein, J. (2008). *Build your own textual DSL with Tools from the Eclipse Modeling Project*. Eclipse Corner Articles. Retrieved from http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/
87. Garcia, M., & Sentosa, P. (2008). *Generation of Eclipse-based IDEs for Custom DSLs.* Hamburg, Germany: Software Systems Institute (STS), Technische Universit at Hamburg-Harburg.
88. Gerber, A., Lawley, M., Raymond, K., Steel, J., & Wood, A. (2002). *Transformation: The missing link of MDA.* Graph Transformation: First International Conference, ICGT 2002, 90-105.
89. Glineur, Q. *M2M/Relational QVT Language (QVTR).* Retrieved March, 2009, from: http://wiki.eclipse.org/M2M/Relational_QVT_Language_(QVTR).
90. Gronback, R. C. (2009). Eclipse *Modeling Project: A Domain-Specific Language (DSL) Toolkit*: Addison-Wesley Professional.
91. Grose, T. J., Doney, G. C., & Brodsky, S. A. (2002). *Mastering XMI: Java Programming with XMI, XML, and UML*: Wiley.
92. Grunske, L., Geiger, L., & Lawley, M. (2005). *A Graphical Specification of Model Transformations with Triple Graph Grammars*. Paper presented at the First European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2005), Nuremberg, Germany.
93. Guerra, E., de Lara, J., Kolovos, D., Paige, R. F. y dos Santos, O. M. *transML: A Family of Languages to Model Model Transformations*. In Dorina Petriu, Nicolas Rouquette, Øystein Haugen, editors, Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010. Proceedings. Lecture Notes in Computer Science, Springer, 2010.
94. IBM. (2004). *IBM Model Transformation Framework 1.0.2 Programmer's Guide.* Retrieved, July 23, 2006, from: http://www.alphaworks.ibm.com/tech/mtf
95. IBM. (2009). *Rational Rose Product line.* Retrieved 20 January, 2009, from: http://www-01.ibm.com/software/awdtools/developer/rose/index.html

96. ikv++ technologies. (2008). *mediniQVT*. Retrieved 16 October 2008, from http://www.ikv.de/index.php?option=com_content&task=view&id=75&Itemid=77&lang=en.
97. IRISA (2009). *Kermeta workbench*: http://www.kermeta.org/.
98. ISO (International Standards Organization for Standardization) & IEC (International Electrotechnical Commission) (2003). ISO/IEC 9075:2003 Information technology – Database languages – SQL:2003.
99. Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley Professional.
100. Jézéquel, J-M. *Model Transformation Techniques.* Rennes: France: IRISA-Universite de Rennes. Retrieved December, 2007, from: http://modelware.inria.fr/rubrique21.html.
101. Jouault, F., Laarman, A., Allilaire, F. & Glineur, Q. (2005). *Specification of the ATL Virtual Machine.* Retrieved 20 January, 2009, from: http://www.eclipse.org/m2m/atl/doc/ATL_VMSpecification[v00.01].pdf
102. Jouault, F., & Bézivin, J. (2006). *KM3: A DSL for Metamodel Specification.* Paper presented at the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2006, Bologna, Italy.
103. Jouault, F., & Kurtev, I. (2006). *On the architectural alignment of ATL and QVT*. Paper presented at the ACM symposium on Applied computing (SAC 2006), Dijon, France.
104. Jouault, F., Bezivin, J. and Kurtev, I. (2006). *TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering*. In: GPCE'06: Proceedings of the fifth international conference on Generative programming and Component Engineering, Portland, Oregon, USA, pages 249--254.
105. Jouault, F., & Kurtev, I. (2006). *Transforming Models with ATL.* In Satellite Events at the MoDELS 2005 Conference (pp. 128-138).
106. Jouault, F. & Piers, W. (2009). *ATL User Guide*. Retrieved 15 January, 2009, from http://wiki.eclipse.org/ATL/User_Guide.
107. Kalnins, A., Celms, E., & Sostaks, A. (2006). *Model Transformation Approach Based on MOLA.* Paper presented at the Model Transformations in Practice Workshop - Satellite Events at the MoDELS 2005 Conference, Montego Bay, Jamaica.
108. Kelly, S., Lyytinen, K., & Rossi, M. (1996). *MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment*. In 8th International Conference on Advanced Information Systems Engineering, CAiSE'96 (pp. 1-21). Heraklion, Crete: Springer.
109. Kerlinger, F. S. (1975). *Investigación del Comportamiento. Técnicas y Metodologías*. Interarmericana. México. 1979.
110. Kern, H. (2008). *The Interchange of (Meta)Models between MetaEdit+ and Eclipse EMF*. Paper presented at the 8th ooPSLA Workshop on Domain-Specific Modelling at OOPSLA 2008, Birmingham, USA.
111. Kitchenham, B. *Procedures for Performing Systematics Reviews*. Keele University Technical Report TR/SE-0401. ISSN:1353-7776. NICTA Technical Report 0400011T.1. July, 2004
112. Klatt, B. (2007). *Xpand: A Closer Look at the model2text Transformation Language.* Retrieved 10/16/2008, from http://www.bar54.de/publikationen.html.
113. Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*: Addison-Wesley.
114. Knapp, A., Koch, N., Moser, F., & Zhang, G. (2003). *ArgoUWE: A Case Tool for Web Applications.* Paper presented at the First International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE'03), Geneva, Switzerland.
115. Koch, N. (2006). *Transformation Techniques in the Model-Driven Development Process of UWE.* In Workshop Proc. of the 6th Int. Conf. on Web Engineering (ICWE 2006), ACM Vol. 155, Palo Alto, California, 2006.
116. Koch, N., Meliá, S., Moreno, N., Pelechano, V., Sánchez, F., & Vara, J. M. (2008). *Model-Driven Web Engineering.* UPGRADE, IX(2), 40-45.

117. Koch, N. (2006). *Transformation Techniques in the Model-Driven Development Process of UWE*. In Workshop Proc. of the 6th Int. Conf. on Web Engineering (ICWE 2006), ACM Vol. 155, Palo Alto, California, 2006.
118. Kolovos, D., Paige, R., & Polack, F. (2006). *Eclipse Development Tools for Epsilon, Eclipse* Summit Europe, Eclipse Modeling Symposium. Esslingen, Germany.
119. Kolovos, D., Paige, R., & Polack, F. (2006). *The Epsilon Object Language (EOL).* Paper presented at the Model Driven Architecture - Foundations and Applications: Second European Conference (ECMDA-FA 2006), Bilbao, Spain.
120. Kolovos, D., Paige, R., Rose, L., & Polack, F. (2008). *The Epsilon Book*. Retrieved 20 December, 2008, from: http://epsilonlabs.svn.sourceforge.net/svnroot/epsilonlabs/org.eclipse.epsilon.book/EpsilonBook.pdf
121. Kolovos, D., Paige, R., & Polack, F. *The EpsilonTransformation Language.* In Proc. 1st International Conference on Model Transformation, Zurich, Switzerland, July 2008.
122. Kulkarni, V. & Reddy, S. (2003). *Separation of Concerns in Model-Driven Development.* IEEE Software, 20(5), 64-69.
123. Kurtev, I., Bezivin, J., & Aksit, M. (2002). *Technological Spaces: An Initial Appraisal*. Paper presented at the Confederated International Conferences DOA, CoopIS and ODBASE 2002, Industrial Track, Irvine, California, USA.
124. Kurtev, I. (2008). *State of the Art of QVT: A Model Transformation Language Standard.* Paper presented at the Third International Symposium on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007 Kassel, Germany.
125. Küster, J.M. y Abd-El-Razik, M. *Validation of Model Transformations –First Experiences Using a White Box-Approach.* MoDELS 2006 Conference. LNCS 4364 (pp.193-204). Springer-Verlag Berlin Heidelberg. 2007.
126. Kusel, A., *TROPIC - A Framework for Building Reusable Transformation Components*.In Proceedings of the Doctoral Symposium at MODELS 2009, School of Computing, Queen's University, Denver, October 2009.
127. Küster, J.M., Gschwind, T. y Zimmermann, O. *Incremental Development of Model Transformations Chains Usin Automated Testing*. MoDELS 2009 Conference. LNCS 5795 (pp.733-747). Springer-Verlag Berlin Heidelberg. 2009.
128. Küster, J. M., Ryndina, K. and R. Hauser. *A Systematic Approach to Designing Model Transformations*. Report RZ 3621, IBM, Zurich, July 2005.
129. Lawley, M., & Steel, J. (2006). *Practical Declarative Model Transformation with Tefkat.* In Satellite Events at the MoDELS 2005 Conference (pp. 139-150).
130. Langlois, B., Jitia, C. E., and Jouenne, E. (2007). *DSL Classification.* In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling.
131. Lemesle, R. (1998). *Transformation Rules Based on Meta-modeling.* Paper presented at the 2nd International Enterprise Distributed Object Computing (EDOC'98), La Jolla, San Diego, California.
132. Lucas, F. J., & Toval, J. A. (2008). *Model Transformations powered by Rewriting Logic.* Paper presented at the CAiSE Forum at CAiSE'08, Montpellier, France.
133. Marcos, E. (2005) S*oftware engineering research versus software development.* ACM SIGSOFT Software Engineering Notes. Vol. 30 Issue 4. Julio 2005. ISSN:0163-5948
134. Marcos, E. & Marcos, A. (1998). *An Aristotelian Approach to the Methodological Research: a Method for Data Models Construction.* In: Information Systems - The Next Generation. L. Brooks and C. Kimble (Eds.). McGraw-Hill. 1998.
135. Marcos, E., Acuña, C. y Cuesta, C.E. (2006), *Integrating Software Architecture into a MDA Framework.* Lecture Notes in Computer Science (ISSN 0302-9743). Software Architecture, Third European Workshop (ISBN 3-540-69271-1). Vol. 4344. Pp:127-143. Ed. Springer Verlag. Heidelberg, Alemania Diciembre 2006
136. Marcos, E., & Marcos, A. (2001). *A Philosophical Approach to the Concept of Data Model: Is a Data Model, in Fact, a Model?* Information Systems Frontiers, 3(2), 267 - 274.

137. Marcos, E., Vela, B., & Cavero, J. (2004). *A methodological approach for object-relational database design using UML*. Informatik - Forschung und Entwicklung, 18(3), 152-164.
138. Marcos, E., De Castro, V., Vela, B. (2004). *Representing Web Services with UML: A Case Study*, presented at First International Conference on Service Oriented Computing (ICSOC'03), Trento, Italy.
139. Marcos, E., Vela, B., & Cavero, J. M. (2001). *Extending UML for Object-Relational Database Design.* In UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings (Vol. 2185, pp. 225-239): Springer.
140. Mazón, J.-N., & Trujillo, J. (2008). *An MDA approach for the development of data warehouses.* Decission Support Systems, 45(1), 41-58.
141. Mens, T. y Van Gorp, P., *A Taxonomy of Model Transformation.* Primer Workshop on Graph and Model Transformation (GraMoT) 2005.
142. Mernik, M., Heering, J., & Sloane, A. M. (2005). *When and how to develop domain-specific languages*. ACM Computer Surveys, 37(4), 316-344.
143. MetaCase. *MetaEdit+* [Software]. Available at http://www.metacase.com/mep/.
144. Miller, J., Mukerji, J. (Eds.), *MDA Guide Version 1.0*, OMG Document - omg/2003-05-01.
145. *MOFLON development team*. MOFLON website, 2007. available ay http://www.moflon.org.
146. Molina, F., Lucas, F. J., Toval, J. A., Vara, J. M., Cáceres, P., & Marcos, E. (2008). *Toward Quality Web Information Systems Through Precise Model Driven Development*. In C. Calero, M. A. Moraga & M. Piattini (Eds.), Handbook of Research on Web Information Systems Quality (pp. 344-363). Hershey PA, USA: Information Science Reference - IGI Global.
147. Moore, B., Dean, D., Gerber, A., Wagenknecht, G., & Vanderheyden, P. (2004). *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*: IBM.COM.
148. OMG, *Model Driven Architecture. A technical perspective.* OMG document -ormsc/01-07-01.
149. OMG/MOF *Meta Object Facility (MOF) 1.4*. Final Adopted Specification Document. formal/02-04-03, 2002.
150. OMG. *The Meta Object Facility (MOF) Core Specification*, Version 2.0. OMG Document - formal/06-01-01
151. OMG, *MOF Model to Text Transformation Language (MOFM2T)*, 1.0. OMG Document - formal/08-01-16
152. OMG, *MOF Model to Text Transformation Language*. RFP. Retrieved from http://www.omg.org/cgi-bin/doc?ad/04-04-07
153. OMG, *Object Constraint Language Specification (OCL)*, version 2.0. OMG Document - formal/2006-05-01
154. OMG. *Software Process Engineering Meta-Model (SPEM)*, version 2.0. OMG Document - formal/2008-04-01
155. OMG. *UML 2.1.1 Formal Specification.* OMG Document - formal/07-02-03.
156. OMG. *UML Diagram Interchange*, v1.0. OMG Document - formal/2006-04-04
157. OMG. *UML 1.5 Formal Specification*. OMG Document - formal/03-03-01.
158. OMG. *MOF 2.0 Query/View/Transformation (QVT)*, V1.0. OMG Document - formal/08-04-03.
159. OMG. *MOF 2.0 Query/Views/Transformations* RFP, OMG document ad/2002-04-10 (2002).
160. OMG - *XML Metadata Interchange (XMI) specification* V2.1.1. OMG Document - formal/2007-12-01.
161. openArchitectureWare (2008*). openArchitectureWare User guide* (Version 4.3) Retrieved February, 4, 2009, from http://www.openarchitectureware.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf.

162. openMDX. (2008). *openMDX - the leading open source MDA platform* [Software]. Available at http://www.openmdx.org/
163. Oracle Corporation. *Oracle XML DB.* Technical White Paper. Retrieved from: www.otn.com, January, 2003.
164. Oracle Corporation. *Oracle10g. SQL Reference*, 2000.
165. Oracle Corporation. (2008). *Oracle Designer 10g Release 2*. [Software] Available from http://www.oracle.com/technology/products/designer/index.html
166. Oracle Corporation (2008). *Oracle Database 11g* [Software]. Available from http://www.oracle.com/global/lad/database/index.html.
167. Parnas, D. L. (1972). *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12), 1053-1058.
168. Partsch, H., & Steinbrüggen, R. (1983). *Program Transformation Systems*. ACM Computing Surveys, 15(3), 199-236.
169. Pilato, C., Collins-Sunsman, B., & Fitzpatrick, B. (2008). *Version Control with Subversion*: O'Reilly Media.
170. Ravi, M. & Sandeepan, B. (2003). *XML schemas in Oracle XML DB.* Paper presented at the Proceedings of the 29th international conference on Very Large Data Bases (VLDB 2003), BErling, Germany.
171. Royce, W. W. (1987). *Managing the development of large software systems: concepts and techniques.* Paper presented at the 9th international conference on Software Engineering, Monterey, California, United States.
172. Ruby, S., Thomas, D., & Heinemier Hansson, D. (2009). *Agile Web Development with Rails (Third ed.):* The Pragmatic Bookshelf.
173. Sánchez, V. (2008). *Making a case for supporting a byte-code model transformation approach.* In OMG (Ed.), Symposium on Eclipse Open Source Software and OMG Open Specifications. Ottawa, Ontario, Canada: OMG Document omg/08-06-48.
174. Sánchez, D. M., Cavero, J. M., & Marcos, E. (2009). *The concepts of model in information systems engineering: A proposal for an ontology of models*. The Knowledge Engineering Review, 24(1), 5-21.
175. Sanchez-Barbudo, A., Sanchez, V., Roldán, V., Estévez, A., & Roda-García, J. L. (2008). *Providing an Open Virtual-Machine-based QVT Implementation.* Paper presented at the V Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM 2008) in XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2008), Gijon, Spain.
176. Sánchez Cuadrado, J., García Molina, J., & Menarguez Tortosa, M. (2006). *RubyTL: A Practical, Extensible Transformation Language.* Paper presented at the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006), Bilbao, Spain.
177. Sánchez Cuadrado, J., & García Molina, J. (2007). *Building Domain-Specific Languages for Model-Driven Development.* IEEE Software, 24(5), 48-55.
178. Sánchez Cuadrado, J., & García Molina, J. (2008). *Approaches for Model Transformation Reuse: Factorization and Composition.* Paper presented at the 1st International conference on Theory and Practice of Model Transformations (ICMT 2008), Zurich, Switzerland.
179. Selic, B. (2003). *The pragmatics of Model-Driven development*, IEEE Software, 20(5), 19-25.
180. Sendall, S., & Kozaczynski, W. (2003). *Model Transformation–the Heart and Soul of Model-Driven Software Development*. IEEE Software, 20(5), 42-45.
181. Skinner, C. (2008, June 25). *DSL + UML = Pragmatic Modeling.* Retrieved from http://blogs.msdn.com/camerons/default.aspx.
182. Stevens, P. (2008). *A Landscape of Bidirectional Model Transformations*. Paper presented at the International Summer School on Generative and Transformational Techniques in Software Engineering II, GTTSE 2007, Braga, Portugal.
183. Sun Microsystems. *Java Metadata Interface (JMI) Specification*. (June, 2002). Retrieved, January, 2006, from: http://java.sun.com/products/jmi/index.jsp.

184. Tratt, L. (2005). *Model transformations and tool integration.* Journal of Software and Systems Modeling, 4(2), 112-122.
185. Tratt, L y Clark, T. *Model Transformations in Converge*. Technical Report. Department of Computer Science, King's College London. Septiembre, 2003.
186. TRDCC. (2007). *ModelMorf: a model transformer.* Retrieved 16 October 2008, from http://www.tcs-trddc.com/ModelMorf/index.htm
187. van Amstel, M. F., Lange, C. F. J. y van den Brand, M. G. J. *Metrics for Analyzing the Quality of Model Transformations.* In G. Falcone, Y.-G. Gueh_eneuc, C. F. J. Lange, Z. Porkolab, and H. A. Sahraoui, editors, 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2008), pages 41{51, Paphos, Cyprus, July 2008.
188. Van Gorp, P. (2008). *Model-Driven Development of Model Transformations.* Paper presented at the 4th International Conference on Graph Transformations ICGT 2008, Leicester, United Kingdom.
189. Van Gorp, P., Keller, A., & Janssens, D. (2009). *Transformation Language Integration Based on Profiles and Higher Order Transformations*. Paper presented at the First International Conference on Software Language Engineering, SLE 2008, Toulouse, France.
190. Vanhooff, B., Ayed, D., & Berbers, Y. (2006). *A Framework for Transformation Chain Design Processes*. Paper presented at the First European Workshop on Composition of Model Transformations - CMT 2006; European Conference on Model Driven Architecture (ECMDA-FA 2006), Bilbao, Spain.
191. Vara, J.M, *M2DAT: a Technical Solution for Model-Driven Development of Web Information Systems*. PhD Thesis. University Rey Juan Carlos, November 2009.
192. Vara, J. M., Acuña, C. J., Marcos, E., & Lopez Sanz, M. (2004). *Desarrollo de un Sistema de Información web: una experiencia con Oracle XMLDB.* CUORE. Círculo de Usuarios de Oracle España, 27, 3-12.
193. Vara, J.M. De Castro, V., Marcos, E. (2005). *WSDL Automatic Generation from UML Models in a MDA Framework*. International Journal of Web Services Practices, 1 (1-2), 1-12.
194. Vara, J. M., Vela, B., & Marcos, E. (2006). *Oracle XML DB como repositorio integrado para herramientas CASE. Aplicación al desarrollo de MIDAS-CASE, una herramienta MDA.* Paper presented at the XVI Congreso Nacional Usuarios de Oracle (CUORE).
195. Vara, J. M., Vela, B., Cavero, J. M., & Marcos, E. (2007). *Model Transformation for Object-Relational Database development.* SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, 1012-1019.
196. Vara, J.M., Didonet Del Fabro, M., Joualt, F. & Bezivin, J. (2008). *Model Weaving Support for Migrating Software Artifacts from AUTOSAR 2.0 to AUTOSAR 2.1*. Int. Conf. on Embedded Real Time Software (ERTS 2008), Toulose (France), 2008.
197. Vara, J. M., De Castro, V., Didonet Del Fabro, M., & Marcos, E. (2009). *Using Weaving Models to automate Model-Driven Web Engineering proposals.* International Journal of Computer Applications in Technology (To be published).
198. Vara, J. M., Vela, B., Bollati, V., & Marcos, E. (2009). *Supporting Model-Driven Development of Object-Relational Database Schemas: a Case Study* Paper presented at the ICMT2009 - International Conference on Model Transformation, Zurich, Switzerland.
199. Varró, D. *Model transformation by example.* In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
200. Vela B., Marcos E. (2003). *Extending UML to represent XML Schemas.* The 15th Conference On Advanced Information Systems Engineering. CAISE'03 FORUM. Ed: J. Eder, T. Welzer. Short Paper Proceedings. Klagenfurt/Velden (Austria). 16-20 June 2003.
201. Vela, B., Acuña, C. J., & Marcos, E. (2004). *A Model Driven Approach for XML Database Development.* In Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling (Vol. 3288, pp. 780-794): Springer.

202. Vela, B., Fernandez Medina, E., Marcos, E., & Piattini, M. (2006). *Model driven development of secure XML databases.* ACM SIGMOD Record, 35(3), 22-27.
203. Verner, L. BPM: *The Promise and the Challenge.* Queue of ACM, 2(4). pp. 82-91
204. Vignaga, A. *A methodological approach to developing model transformations.* En actas del Model-Driven Engineering Languages and Systems (MoDELS 2007). Nashville (TN), Estados Unidos, Octubre, 2007
205. Vignaga, *A. Metrics for Measuring ATL Model Transformation.* Reporte Técnico: TR_DCC-20090430-006 Recuperado de: http://swp.dcc.uchile.cl/TR/2009/TR_DCC-20090430-006.pdf
206. Vignaga, A. *Transformation Models: An Application and Insights*, Reporte Técnico TR/DCC-2008-4 Recuperado de: http://www.dcc.uchile.cl/TR/2008/TR_DCC-2008-004.pdf. May 2008
207. Vignaga, A. Perovich, D. y Bastarrica, M.C. *Towards Layered Specifications of Model Transformation.* Reporte Técnico TR/DCC-2007-1 Recuperado de: http://www.dcc.uchile.cl/TR/2007/TR_DCC-2007-001.pdf. January 2007
208. Wagelaar, D. (2008). *Composition Techniques for Rule-Based Model Transformation Languages*. Paper presented at the 1st International conference on Theory and Practice of Model Transformations (ICMT 2008), Zurich, Switzerland.
209. W3C. (2004). *XML Extensible Markup Language (XML) 1.0* (Third Edition). W3C Recommendation. Bray, T., Paoli, J, Sperberg-McQu4een, C. M., Maler, E. and Yergeau F. Retrieved from: http://www.w3.org/TR/2004/REC-xml-20040204/, 2004.
210. W3C XML Schema Working Group. (2001). *XML Schema Parts 0-2 [Primer, Structures, Datatypes]*. W3C Recommendation. Retrieved from: http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/ y http://www.w3.org/TR/xmlschema-2/, 2001.
211. W3C. (2004). *RDF/XML Syntax Specification.* Retrieved July 30, 2006, from: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210

Tabla de Acrónimos

Acrónimos	Descripción
AMW	ATLAS Model Weaver
ATL	ATLAS Transformation Language
BD	Base de Datos
CASE	Computer Aided Software Engineering
DSL	Domain Specific Language
DTD	Document Type Definition
EMF	Eclipse Modelling Framework
EMP	Eclipse Modelling Project
GMF	Generic Modeliing Framework
GPL	General Purpose Language
IDE	Integrated Development Environtment
JET	Java Emitter Templates
JMI	Java Metadata Interface
DSDM	Desarrollo de Software Dirigido por Modelos
LHS	Left Hand Side
M2DAT	MIDAS MDA Tool
M2DAT-DB	MIDAS MDA Tool for DataBases
MeTAGeM	A Meta-Tool for Automatic Generation of transformation Model
M-LTH	Meta-modelo para Lenguajes de Transformación que siguen la aproximación Híbrida
M2M	Model to Model
M2T	Model to Text

Acrónimos	Descripción
M-TIP	Meta-modelo de Transformación Independiente de Plataforma
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OR	Object-Relational
ORDB	Object-Relational DataBase
PDM	Platform Dependent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
RHS	Right Hand Side
SQL	Structured Query Language
UML	Unified Modelling Language
QVT	Query/View/Transformation
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XSD	XML Schema Definition