UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Implementando la semántica de QVT mediante el mecanismo de transformación de modelos

Autores: Carina Moldes, Silvana Mossi
Director: Dra. Claudia Pons
Carrera: Licenciatura en Sistemas

Resumen

La principal idea compartida por todos los paradigmas englobados dentro del Desarrollo de Software Dirigido por Modelos es la conveniencia de que los programadores empleen lenguajes de más alto nivel de abstracción que los lenguajes de programación, esto es, lenguajes que manejen conceptos más cercanos al dominio de la aplicación. Estos lenguajes que proporcionan mayor nivel de abstracción se denominan lenguajes de modelado o lenguajes específicos del dominio (DSLs). La creación de un DSL abarca lenguajes de metamodelado, sintaxis abstracta, sintaxis concreta, semántica y transformaciones. Pero la sintaxis del DSL no es todo. El siguiente paso consiste en definir su semántica, es decir el significado de cada una de las construcciones sintácticas del DSL. En nuestra tesis estudiamos la aplicabilidad de lenguajes de transformación de modelos para implementar la semántica de lenguajes de dominio específico. En particular nos enfocamos en el lenguaje estándar QVT (Query/Views/Transformations) como DSL y utilizamos el lenguaje de transformación de modelos ATL para implementar su semántica y el framework de xtext para obtener la instancia del metamodelo de QVT utilizada en la transformación. Como resultado final, se implementó una herramienta de software que extiende la plataforma eclipse para obtener la semántica del lenguaje QVT.

Palabras Claves

Desarrollo Dirigido por Modelos, Transformación de Modelos, Metamodelos, QVT, ATL, Semántica, Sintaxis, DSL, Lambda Cálculo, Eclipse.

Conclusiones

Las definiciones de la semántica de los DSLs brindan un importante avance en el DSDM ya que mediante un lenguaje de transformación de modelos resultan definiciones más sencillas, precisas y fáciles de entender. Para establecer la semántica de QVT, utilizamos como dominio semántico un lenguaje para transformaciones ya existente como Lambda Cálculo. De esta manera contamos con la ventaja de que este lenguaje ya tiene su semántica bien definida y provee una maquinaria en funcionamiento para ejecutarlo.

Trabajos Realizados

Investigación del DSDM y herramientas existentes en el mercado para su uso.Investigación de los lenguajes de transformación de modelos, con pruebas de casos concretos de estudios para analizar las diferencias entre cada herramienta. Análisis de Framework para obtener de manera automática instancias de metamodelos. Análisis y comprensión del mapeo de las operaciones de QVT al lenguaje lambda. Investigación para el desarrollo de una herramienta como extensión de la plataforma eclipse.

Trabajos Futuros

Las reglas de transformación implementadas por nuestra herramienta estan basadas en un subconjunto del lenguaje QVT, el cual abarca la parte imperativa del lenguaje.Es necesario extender el conjunto de reglas que conforman la transformación para poder así abarcar todo el espectro del lenguaje QVT.

Con respecto a la herramienta, otros de los objetivos a cumplir, es la generación final del modelo de lambda cálculo con sintaxis de Haskell para poder ser ejecutado por un compilador.

Fecha de la presentación: MARZO 2011

Agradecimientos

Carina

A mis amigos, compañeros de estudio, con los que compartí todos estos años, y los que me acompañaron en esta etapa de mi vida, que sin dudas, fue una etapa de mucho crecimiento como profesional y también como persona.

A Claudia por el apoyo y la buena predisposición, durante el tiempo que nos acompañó como directora.

A Silvana, por el esfuerzo y la dedicación que le entregó a la tesis, y fundamentalmente por ser una gran amiga.

Y finalmente, los más importantes, mis padres, mi hermana y mi abuela, por el apoyo incondicional y la confianza que tuvieron desde el primer día, y porque solo gracias a ellos puedo estar hoy en el lugar que estoy.

Silvana

Quiero agradecerle a Claudia por todo su apoyo, generosidad, por estar dispuesta a escuchar nuestras consultas y brindarnos su tiempo personal.

A Cari, porque me ha acompañado en materias de la facultad y por eso la elegí para transitar este último e importante camino de la facultad. Por su paciencia, responsabilidad y porque pude expresarme libremente con ella en el desarrollo de la tesis.

A mis amigas; Gaby, Vani y Jime que me regaló la facultad y me acompañaron durante muchos años y espero que me sigan acompañando en lo personal como hasta ahora.

Y finalmente a los más importantes, mi familia y Gus. Mi mamá, compañera de toda la vida, siempre ahí para que no afloje, agradecida de todo lo que me dio en estos años de la facultad; mi papá, que lo extraño mucho, pero que me sigue dando fuerzas desde el cielo y Gus; mi gran sustento y mi compañero siempre.

Índice

Capítulo 1

Introducción

En este capítulo se describe la motivación y el contexto en el que se enmarca esta tesis, nuestra propuesta para los problemas a resolver y los objetivos a cumplir. A modo de introducción presentamos los principales aportes del Desarrollo de Software dirigido por Modelos (MDD) para la automatización del desarrollo de software.

Motivación

Los modelos proveen abstracciones de un sistema físico que permiten a los ingenieros analizar el sistema ignorando detalles complejos mientras se enfocan en las partes más relevantes, como puede ser la lógica del negocio. Todas las formas de ingeniería se basan en el uso de modelos para facilitar la comprensión de sistemas complejos, los modelos son utilizados en muchas formas: para predecir la calidad de un sistema, razonar acerca de propiedades específicas cuando ciertos aspectos del sistema cambian, y para comunicar características claves del sistema a todos los involucrados en el desarrollo.
Los modelos pueden ser los precursores de la implementación física del sistema o ser generados de sistemas ya existentes para comprender su funcionamiento.
En el mundo del software, el modelado existe desde hace tiempo, se remonta a los primeros días de la programación. El estado actual de esta práctica emplea el lenguaje de modelado unificado o UML (Unified Modeling Laguage) como el arma principal en la notación del modelado. UML permite a los equipos de desarrollo capturar una variedad de características importantes de un sistema, en modelos. Durante el proceso de desarrollo de software, se realizan transformaciones entre modelos, por ejemplo, el modelo de análisis es transformado en un modelo de diseño y así sucesivamente hasta llegar al código. Sin embargo, la transformación entre estos modelos es primordialmente manual, lo que tiende a ser complejo. Una forma útil de describir la distintas maneras en que es utilizado el modelado en la actualidad, es observar las distintas formas en que el código es sincronizado con el modelo, éstas aproximaciones al modelado van desde no utilizar modelos en ninguna fase del desarrollo hasta proyectos donde sólo se realizan modelos conceptuales sin llegar a nunca a codificar.
Actualmente buena parte de los desarrolladores toma la aproximación de “solo código” y utilizan poco los modelos. Esta aproximación hace difícil la evolución de estos sistemas dado que en muchos de los casos los desarrolladores originales del sistema no se encuentran durante las fases de mantenimiento del mismo.
Una mejora es proveer de visualizaciones de código en alguna notación adecuada, esto es, mientras el desarrollador va generando el código, éste es visualizado para entender de mejor manera su estructura. Las ventajas del modelado pueden ser apreciadas en mayor grado cuando se habla de ingeniería de ida y vuelta (RTE por sus siglas en inglés RoundTrip Engineering) que ofrece un intercambio bidireccional entre el modelo abstracto que describe la arquitectura del sistema y el código. Esta aproximación requiere de mucha disciplina de los participantes ya que puede ocurrir un defasaje entre

los modelos y su implementación si no hay comunicación y procesos bien establecidos. En la aproximación centrada en el modelo, los modelos del sistema tienen un nivel de detalle lo suficientemente completo como para permitir la generación completa de la implementación del sistema basándose solo en los modelos. El proceso de generación del código puede llegar a aplicar una serie de reglas de transformación, los cuales generalmente le permiten al desarrollador la elección entre los distintos patrones que podrán ser aplicados a los modelos para transformarlos en modelos más complejos o código.
La última aproximación está basada sólo en modelos, en ésta, los desarrolladores usan los modelos sólo como ayuda para entender y comprender el negocio o el dominio del problema, o simplemente para analizar la arquitectura de una solución posible

Organización

Para abordar el caso de estudio implementado, se realiza una introducción sobre el Desarrollo Dirigido por Modelos, tratando como principales temas, el objetivo, la arquitectura sobre la que trabaja, y el funcionamiento. Se mencionan los diferentes tipos de modelos que definen DDM, y los métodos específicos de transformación de modelos.
En el siguiente capítulo se describe el lenguaje de transformación QVT Operacional, para el cual se definirá su semántica a través de un lenguaje formal.
Seguidamente, una breve descripción del lenguaje Lambda Calculo, el cual será utilizado para definir la semántica de QVT Operacional
A continuación se incluye un capítulo introductorio sobre la semántica de un lenguaje de programación, dando la definición de los diferentes enfoques que existen para definir una semántica.
El siguiente capitulo, tiene como objetivo comenzar a detallar las herramientas utilizadas en cada etapa del proceso de transformación. Como primer paso se describe la generación del modelo de entrada, y las dos alternativas, junto a las ventajas y desventajas de cada una.
A continuación, se define la semántica de QVT Operacional y la utilización del lenguaje ATL para la implementación de la transformación
Y por último la especificación de implementación de la herramienta que implementa la semántica.

Capítulo 2

Conceptos básicos

2.1 Desarrollo de Software Dirigido por Modelos

El Desarrollo de Software Dirigido por Modelos (MDD) y más concretamente la propuesta MDA (Model Driven Architecture) de OMG constituye una aproximación para el desarrollo de sistemas software, basada en la separación entre la especificación de la funcionalidad esencial del sistema y la implementación de dicha funcionalidad usando plataformas de implementación específicas.

La iniciativa MDA cubre un amplio espectro de áreas de investigación (metamodelos basados en MOF, perfiles UML, transformaciones de modelos, definición de lenguajes de transformación (QVT), construcción de modelos PIM y PSM y transformaciones entre ellos, construcción de herramientas de soporte, aplicación en métodos de desarrollo y en dominios específicos, etc.). Algunos de estos aspectos están bien fundamentados y se están empezando a aplicar con éxito, otros sin embargo están todavía en proceso de definición. En este contexto son necesarios esfuerzos que conviertan MDA y sus conceptos y técnicas relacionados en una aproximación coherente, basada en estándares abiertos, y soportada por técnicas y herramientas maduras.

Con la generación completa del código, es poco probable, o innecesario la inspección del código, sólo se tienen los modelos tal como ocurre en la actualidad con los lenguajes de 3a. generación donde no es necesario inspeccionar el código en ensamblador. Las herramientas y técnicas para hacer esto posible han llegado a un estado de madurez donde se ha vuelto práctico incluso para aplicaciones a gran escala. Como parte de este esfuerzo para hacer más incremental la aceptación del enfoque MDD, la OMG creó una serie de estándares de soporte a MDD a los cuales agrupo en las especificaciones de MDA.

Así MDA es un estándar que promueve a MDD y agrupa a varios lenguajes que pueden ser utilizados para seguir un enfoque dirigido por modelos en una organización, MDA intenta estandarizar MDD, que durante muchos años ha estado a la deriva. MDA no define técnicas, etapas ni artefactos, pero si proporciona una estructura tecnológica y conceptual para poder implementar de manera correcta MDD.

2.2 Objetivo del Desarrollo Dirigido por Modelos

El objetivo principal del DSDM lleva a resolver problemas de tiempo, costes y calidad asociados a la creación de software. En este contexto MDA proporciona un marco de trabajo en que es posible especificar modelos, en diferentes niveles de abstracción, y pasar desde un modelo a otro por medio de transformaciones. Dichas transformaciones

de modelos deben ser expresadas de manera clara y precisa usando un lenguaje definido para ese propósito, como por ejemplo, ATL o QVT.

2.3 Arquitectura Dirigida por Modelos

Model Driven Architecture (MDA): es la estandarización de la OMG como plataforma para soportar MDD.

2.3.1 Conceptos básicos de MDA

Algunos de los conceptos más importantes que forman parte de la especificación MDA son:
Modelo. Es una descripción o especificación mediante un lenguaje visual de un sistema.

Metamodelo. Es la descripción y especificación de los elementos y reglas que se utilizan para crear modelos semánticamente correctos para un dominio en particular. También puede definirse como el modelo de un lenguaje de modelado.

Dirigido por modelos (Model Driven). Se dice que es dirigido (o guiado) por modelos porque provee mecanismos que usan modelos para dirigir el curso del diseño, la construcción, la implementación, la operación, el mantenimiento y la modificación de una aplicación. Es decir, el proceso depende de los modelos.

Arquitectura. La arquitectura de un sistema es la especificación de las partes y conectores del sistema, así como las reglas de interacción.

Vista. Es una representación del sistema desde la perspectiva de un punto de vista determinado.

Plataforma. Es un conjunto de subsistemas y tecnologías que proveen un conjunto coherente de funcionalidad que puede ser usada en cualquier aplicación sin tener en cuenta detalles de cómo la funcionalidad es implementada

Punto de vista. Un punto de vista en un sistema es una técnica de abstracción que utiliza un conjunto selecto de conceptos arquitecturales y reglas de estructuración, de manera que se enfoque la atención sólo en un problema particular del sistema. MDA especifica tres puntos de vista sobre un sistema: el punto de vista independiente de la computación, el punto de vista independiente de la plataforma y el punto de vista específico de la plataforma.
• Punto de vista independiente de la computación. Este se enfoca en el ambiente del sistema y los requerimientos del mismo, es decir, la lógica del negocio. Los detalles de la estructura y el procesamiento del sistema están escondidos o no han sido determinados.
• Punto de vista independiente de la plataforma. Se enfoca en la operación del sistema mientras oculta los detalles específicos para cierta plataforma, es decir, muestra la parte de la implementación que es idéntica de una plataforma a otra.

• Punto de vista específico de una plataforma. Combina el punto de vista independiente de la plataforma con el detalle del uso de una plataforma especifica.

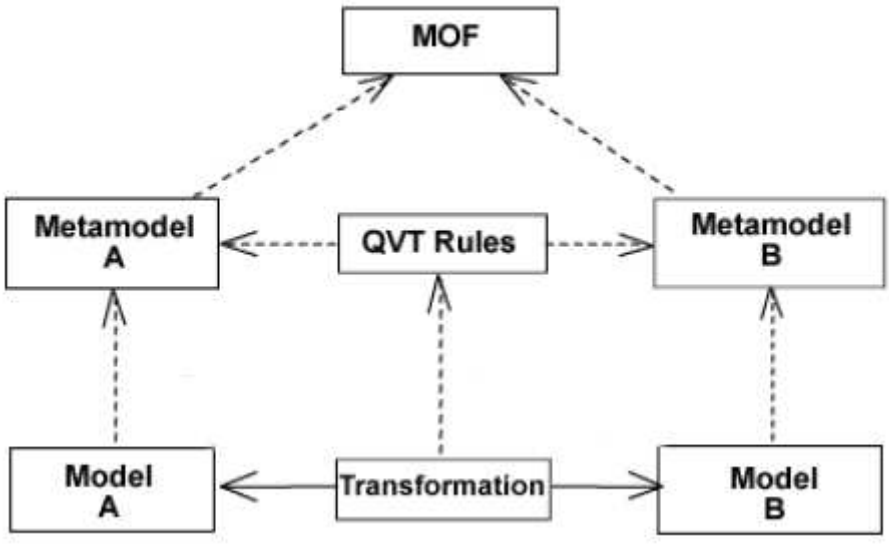
Modelo independiente de la computación (CIM Computer Independent Model). Es una descripción de la lógica del negocio desde una perspectiva independiente de la computación. Es un modelo del dominio

Modelo Independiente de la plataforma (PIM Platform Independent Model). Es una descripción de la funcionalidad del sistema en forma independiente de las características de plataformas de implementación específicas.

Modelo especifico a una plataforma (PSM Platform Specific Model). Es una vista del sistema desde el punto de vista de plataforma específica. Un PSM combina las especificaciones en el PIM con los detalles de como el sistema utiliza un tipo de plataforma en particular. Es una descripción del sistema en términos de una plataforma específica. Por ejemplo, .NET, J2EE, relacional.

Modelo específico de implementación (ISM (Implementation Specific Model) Es una descripción (especificación) del sistema a nivel de código. Por ejemplo, Java, C#.

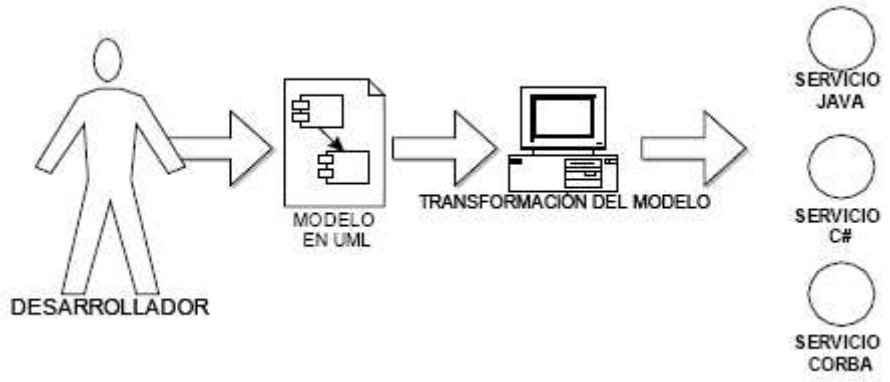
Transformación de modelos. Es el proceso de convertir un modelo a otro modelo del mismo sistema. Generalmente el PIM es combinado con alguna información adicional para producir un PSM.



2.3.2 Funcionamiento de MDA

En la actualidad, existe un amplio conocimiento sobre como traducir, por compilación o interpretación, un lenguaje de alto nivel (Java o SQL) a operaciones que un microprocesador es capaz de ejecutar. De igual manera, en MDA existen "compiladores" capaces de traducir modelos de datos o de la aplicación basados en UML a lenguajes de alto nivel y por lo tanto a las distintas plataformas de los sistemas actuales, pero más importante, a las plataformas del futuro. La idea es que, como en la

actualidad sucede en la industria automotriz, donde mucho del proceso de desarrollo de nuevos vehículos se hace en computadoras y simuladores, para después construir las partes y ensamblar los vehículos de forma automatizada, en la industria de software suceda algo similar: que el proceso de desarrollo de software se base en modelos en una computadora los cuales, inicialmente, serán PIMs y mediante transformaciones hechas por la computadora, poder generar los PSMs para una o varias plataformas, y al final transformar éstos a código que implemente la solución descrita en los modelos. Si nuevas tecnologías surgen, solo hay que transformar los modelos independientes de la plataforma a los modelos específicos de la nueva plataforma y regenerar la aplicación. Si la aplicación requiere integrarse con otras aplicaciones, se modifican los modelos, y después se regeneran las aplicaciones. Así, entre las metas de MDA se tienen: la portabilidad, la interoperabilidad y la reutilización. La figura resume el proceso de desarrollo utilizando MDA. En MDA un desarrollador sólo crea PIMs que son interpretados por una computadora para generar PSMs y posteriormente el código para distintas plataformas. De ésta forma el proceso de desarrollo es acelerado de manera considerable.



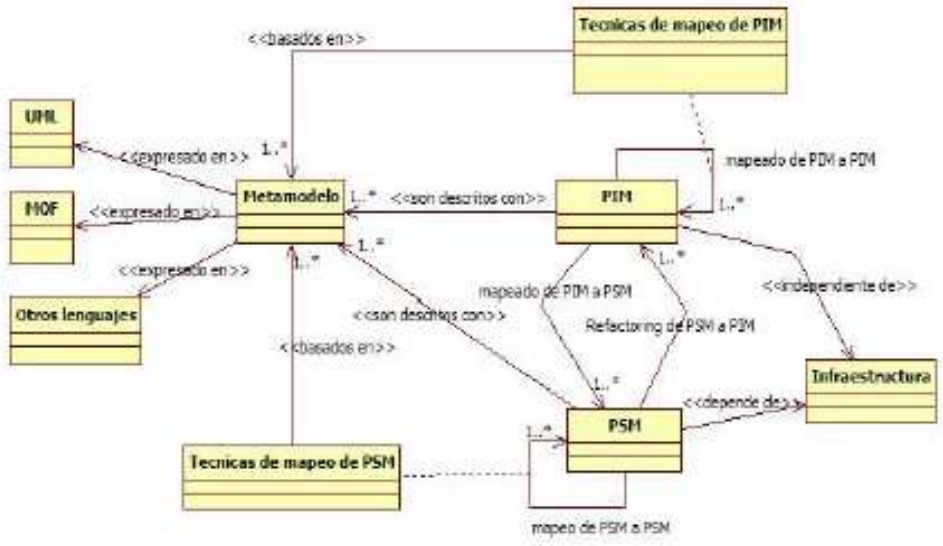
Esta es la premisa principal de una arquitectura manejada por modelos, el permitir la definición de modelos de datos y aplicaciones que puedan ser transformados por una computadora permitiendo la flexibilidad a largo plazo de:

• Implementación. Nueva infraestructura de implementación puede ser integrada o generada de los diseños existentes, para después regenerar el código de la aplicación.
• Integración. Se puede también automatizar la producción de puentes de integración de datos y la conexión entre aplicaciones.
• Mantenimiento. Se da acceso directo a la especificación del sistema, lo que facilita su mantenimiento, se da mantenimiento a los modelos, no al código.
• Pruebas y simulación. Los modelos pueden ser validados contra los requerimientos, probados para varias infraestructuras y usados para, de manera directa, simular el comportamiento del sistema que se está diseñando.

Para utilizar MDA uno de los primeros pasos del proceso es modelar los requerimientos funcionales del sistema en un CIM, el cual describirá los aspectos relacionados con el uso de la aplicación, y ayuda a presentar de manera precisa qué es lo que se espera que el sistema haga, esto será útil no sólo como ayuda para comprender el problema, sino también como una fuente donde se genera un vocabulario compartido que será utilizado en otros modelos.

Mas tarde un PIM es desarrollado, éste describe el funcionamiento del sistema pero no los detalles de su implementación en una plataforma especifica y puede ser creado para implementar una o más arquitecturas. El arquitecto entonces escoge una o varias

plataformas que le permitan implementar el sistema con las características arquitecturales definidas, de aquí el arquitecto genera un PSM .
Finalmente el PSM es traducido a código propio de la plataforma especificada. Una forma de comprender todo el funcionamiento de MDA y sus elementos principales es observando su metamodelo. La figura expresa el metamodelo de la descripción de MDA, el cual resume todos los elementos que se han mencionado. Se puede apreciar que la base de todo en MDA es la definición de un metamodelo, es decir, un lenguaje de modelado, el cual es utilizado para la creación de PIM's, PSM's y CIM's. El metamodelo de MDA también muestra que para hacer el mapeo entre los distintos modelos es necesario establecer reglas o técnicas de mapeo entre los distintos modelos.



De manera más sintética, los pasos a seguir en el proceso MDA de desarrollo de software son los siguientes:

1. Definir un CIM que muestre el sistema dentro del entorno en el que va a operar, este modelo nos ayudará a entender exactamente lo que el sistema va a hacer independientemente de cómo se implementará.

2. Construir un PIM, que describe el sistema, pero no muestra los detalles de su implementación en ninguna plataforma.

3. En este estado del proceso el arquitecto de software ha de elegir una o varias plataformas que permitan la implementación del sistema con las cualidades arquitectónicas deseadas.

4. El arquitecto marca los elementos del PIM para indicar los mappings que han de ser usados para llevar a cabo la transformación de ese PIM en un PSM; o bien, si se utilizan transformaciones de metamodelos, se utilizará una máquina de transformación.

5. Transformar el PIM marcado en un PSM –que puede ser realizado manualmente o con ayuda de una herramienta- ; la entrada de la transformación será el PIM marcado y el mapping; la salida es el PSM y el registro de la transformación.

6. Un PSM puede proporcionar más o menos detalle, dependiendo de su propósito, ya que un PIM puede ser una implementación si proporciona toda la información necesaria para construir un sistema y ponerlo en operación, o bien puede ser el PIM de la siguiente iteración del proceso MDA hasta que se consiga una implementación adecuada del sistema.

2.3.3 Mapeo y transformación de modelos

El mapeo es un conjunto de reglas y técnicas usadas para modificar un modelo de manera que se pueda generar uno nuevo. El mapeo es utilizado para transformar de:
• PIM a PIM. Esta transformación es utilizada cuando los modelos son mejorados, filtrados o especializados durante el ciclo de vida de desarrollo sin necesitar ninguna información dependiente de la plataforma. Una de las formas de mapeo más obvias es entre los modelos de análisis y el diseño.

• PIM a PSM. Esta transformación es utilizada cuando el PIM está lo suficientemente refinado para ser proyectado a una infraestructura de ejecución. La proyección está basada en las características de la plataforma utilizada.

• PSM a PSM. Esta transformación puede requerirse en la implementación y realización de componentes. Por ejemplo, el empaquetado de un componente se realiza seleccionando servicios y configuración. Una vez empaquetado, la entrega del componente puede ser realizada especificando los datos de inicialización, servidores de instalación, generación y configuración del contenedor. Esta transformación está ligada al refinamiento de un modelo PSM a un PSM mejorado y más completo.

• PSM a PIM. Esta transformación puede ser requerida para abstraer modelos de implementaciones existentes en una tecnología específica a una independiente de la plataforma. Este procedimiento es sin duda uno de los más complicados y difícilmente puede ser automatizado.

• PSM a Código. Esta transformación es la última de la cadena y permite generar el código específico para una plataforma en particular utilizando un PSM. Una vez mejorados los PSM o actualizados debido al surgimiento de nuevos requerimientos una transformación de este tipo es necesaria para regenerar el sistema.
Para implementar definir las reglas de transformación de mapeo se requiere conocer los metamodelos de los modelos de entrada y salida. Las reglas de la ejecución de la transformación pueden ser generadas utilizando herramientas UML. Existen diversas formas de transformar PIMs expresados en UML en su correspondientes PSMs:

1. Una persona puede estudiar el PIM y manualmente construir un PSM y quizás construir o refinar el mapeo entre los dos.

2. Una persona puede estudiar el PIM y utilizar patrones de refinamiento conocidos para reducir la carga en la construcción del PSM y la relación entre ambos.

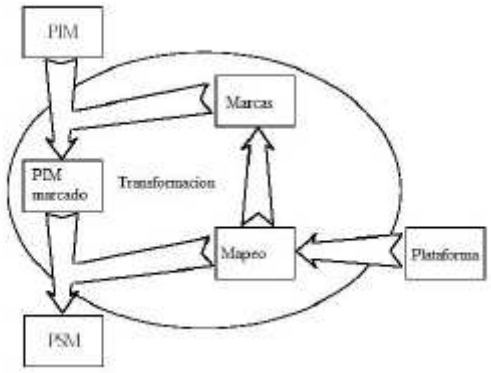
3. Un algoritmo puede ser aplicado al PIM y crear un esqueleto del PSM, el cual sera mejorado de manera manual, quizás utilizando alguno de los patrones de refinamiento de 2.

4. Un algoritmo puede crear un PSM completo de un PIM detallado. Este deberá de manera implícita o explícita grabar los patrones de refinamiento para ser utilizados por otras herramientas automatizadas.

Una consideración adicional es que es más fácil generar código ejecutable de características estructurales de un modelo que de características de comportamiento. La automatización de transformaciones es mas trazable cuando la transformación está parametrizada de alguna manera, como por ejemplo, cuando una persona selecciona opciones de un conjunto predefinido que determinará como será realizada la transformación.
Generalmente se debe de realizar un proceso de marcado del PIM, en el cual se seleccionarán ciertas características no funcionales que se desea tenga el PSM, características que no pueden ser determinadas con la información que ofrece el PIM. Las marcas en un modelo también pueden especificar la calidad de la implementación, estas podrían requerir parámetros, por ejemplo, una marca que indique "soporte de múltiples conexiones" puede requerir un parámetro que indique el límite máximo de conexiones que aceptará, o alguno que indique políticas de limite de tiempo (timeout). Para que las marcas sean utilizadas de manera apropiada es recomendable que sean estructuradas, limitadas y modeladas, por ejemplo, un conjunto de marcas mutuamente exclusivas necesitaran ser agrupadas, de manera que el arquitecto sepa que no más de una de estas marcas puede ser aplicada a la transformación de un mismo elemento. El mapeo también debe incluir plantillas, que son modelos parametrizados que especifican tipos particulares de transformaciones, son como patrones de diseño, pero incluyen especificaciones más detalladas para guiar la transformación. Un conjunto de marcas en un modelo pueden estar asociadas a una plantilla, de manera que estas marcas indiquen que las instancias de un modelo en particular deberán ser transformadas de acuerdo a ésta platilla. Otras marcas pueden ser utilizadas para llenar parámetros requeridos por una plantilla.

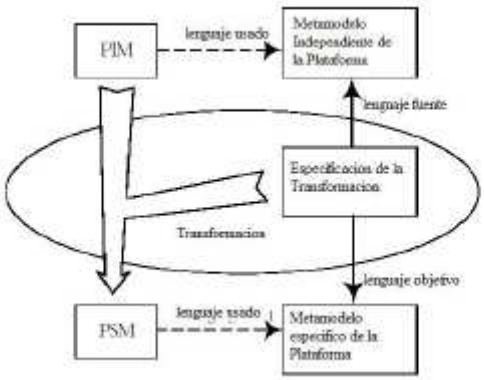
2.3.4 Métodos específicos de transformación de modelos

Transformación por marcado. En un modelo son colocadas una serie de marcas que serán utilizadas para guiar la transformación (figura). Una vez que se tiene el modelo marcado, se aplican reglas de mapeo definidas para una plataforma específica para transformar el PIM en un PSM.

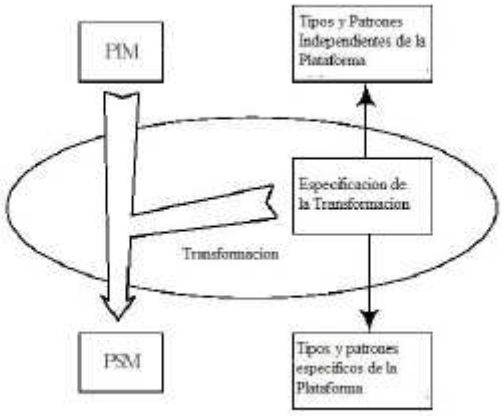


Transformación por metamodelo. Un modelo es construido usando un lenguaje independiente de la plataforma especificado por algún metamodelo. Se hace la

transformación de este modelo a un lenguaje específico de la plataforma especificado en algún otro metamodelo. Es decir se transforma de un lenguaje de modelado a otro (figura). Se tiene un lenguaje de modelado fuente que será transformado en un lenguaje de modelado objetivo.



Transformación por patrones. Patrones pueden ser utilizados en la especificación del mapeado (figura). El mapeado incluirá entonces patrones y marcas correspondientes a elementos de dichos patrones.



Se debe notar que sea cual sea el método de transformación seleccionado para pasar de un modelo a otro se debe determinar un conjunto de reglas de transformación, en las cuales se indica que elementos del modelo nuevo serán generados tomando como base elementos del modelo original. Así el mapeo consiste en determinar estas reglas de transformación, mapear los elementos de un modelo a los de otro, y de esta forma, facilitar el proceso de transformación.

2.3.5 Base tecnológica de MDA

MDA esta basado en tecnologías de la OMG (Object Managment Group) que es una organización no lucrativa encargada de definir estándares en el dominio de la orientación a objetos. Estas se describen brevemente a continuación a manera de introducción.

UML
El lenguaje de modelado unificado (UML) es un lenguaje de modelado estándar para la visualización, especificación y documentación de sistemas de software. Los modelos de MDA pueden ser especificados usando UML. UML resuelve el problema de la arquitectura, objetos e interacciones entre objetos. Los artefactos capturados en UML (en términos de casos de uso, clases, diagramas de actividad, etc.) pueden ser exportados a otras herramientas del proceso de desarrollo usando XMI (XML Metadata Interchange).

XMI
XMI (XML Metadata Interchange) es un mecanismo para estándar de intercambio de modelos de manera textual siguiendo un formato XML entre distintas herramientas, repositorios y middleware. XMI es parte fundamental del mundo del modelado y juega un rol importante en el uso de XML como parte importante de MDA.

OCL (Object Constraint Language)
En el modelado orientado a objetos, un modelo como el de clases no es suficiente para lograr una especificación precisa. Puede ser necesario describir características adicionales sobre los objetos del modelo. Muchas veces estas características se describen en lenguaje natural. La práctica ha revelado que muy frecuentemente esto produce ambigüedades. Para escribir especificaciones correctas se han desarrollado los lenguajes formales.
OCL es un lenguaje formal para expresar restricciones libres de efectos colaterales. Los usuarios de UML y de otros lenguajes visuales pueden usar OCL para especificar restricciones y otras expresiones incluidas en sus modelos. OCL tiene características de un lenguaje de expresión, de un lenguaje de modelado y de un lenguaje formal. Es un lenguaje formal, fácil de leer y escribir. Ha sido desarrollado como un lenguaje de modelado para negocios dentro de la división de seguros de IBM. OCL es un lenguaje de expresión puro. Por lo tanto, garantiza que una expresión OCL no tendrá efectos colaterales; no puede cambiar nada en el modelo. Esto significa que el estado del sistema no cambiará nunca como consecuencia de la evaluación de una expresión OCL. Todos los valores de todos los objetos, incluyendo todos los enlaces, no cambiarán cuando una expresión OCL es evaluada, simplemente devuelve un valor. OCL no es un lenguaje de programación, por lo tanto, no es posible escribir lógica de programa o flujo de control en OCL. No es posible invocar procesos o activar operaciones que no sean consultas en OCL.

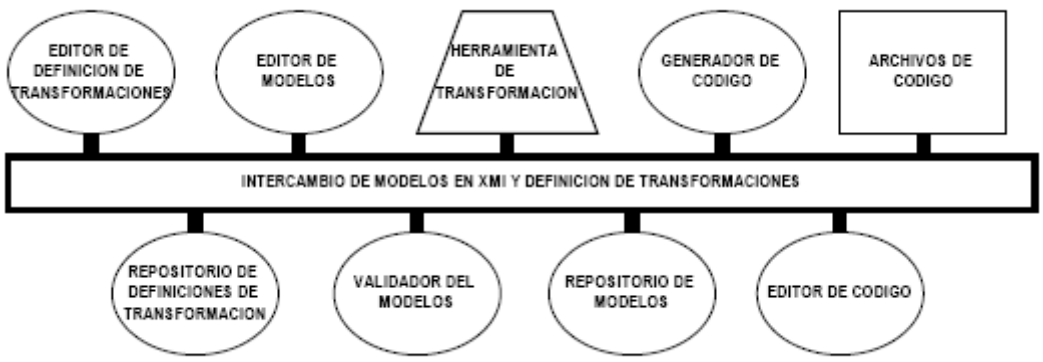
Construcción de PIM's y PSM's en UML
El poder de UML (a diferencia de otros lenguajes) radica en que fue definido basándose en los conceptos más importantes del modelado. Sumado a esto se tiene la ventaja de que los modelos en UML pueden ser representados tanto de forma gráfica como textual utilizando XMI, lo que facilita la transformación entre modelos. Los modelos representados en UML pueden ser muy ricos semánticamente, ya que UML provee elementos para la definición de restricciones y comportamiento tales como:

- Indicar limitaciones sobre un conjunto de atributos.
- Indicar pre y postcondiciones para especificar métodos.
- Indicar si el valor de un parámetro puede ser nulo.
- Indicar si una operación tiene efectos colaterales.
- Indicar patrones de especificaciones y diseño.

Especificar las restricciones en un lenguaje formal (como OCL) en lugar de utilizar lenguaje normal permite reducir la ambigüedad de la especificación y por lo tanto facilita la implementación de importantes aspectos:
Provee al programador de instrucciones más precisas, eliminando la probabilidad de que éste tenga que interpretar las decisiones que tomó el diseñador.
Disminuye la cantidad de trabajo requerido para hacer que diferentes implementaciones de la misma especificación trabajen juntas, o integrar implementaciones de dos especificaciones cuyos modelos estan relacionados.
Provee la base para definir pruebas de conformidad para diferentes implementaciones.
Estandariza la especificación de restricciones de manera que herramientas pueden comunicarse entre si siguiendo este lenguaje. Debido a que UML es un lenguaje diseñado específicamente para ser independiente de la plataforma resulta una solución natural para realizar la construcción de PIM's.

2.3.6 Herramientas MDA

Aunque las herramientas de transformación son el corazón del desarrollo en MDA, no son las únicas herramientas que éste enfoque requiere. Algunas de las herramientas necesarias en un ambiente de desarrollo MDA son:
• Editor de código (IDE). Las funciones que proveen los ambientes de desarrollo integrado (IDE), como la depuración, compilación y edición de código, no deben pasarse por alto.
• Repositorio de modelos. Una base de datos de modelos.
• Editor de modelos (herramienta CASE). Donde los modelos pueden ser construidos y modificados.
• Validador de modelos. Los modelos usados para la generación de otros modelos deben de estar extremadamente bien definidos. Los validadores revisan los modelos contra un conjunto de reglas (predefinidas o definidas por el usuario) para asegurar que el modelo está listo para ser usado en una transformación.
• Editor de definición de transformaciones. Un editor para crear y modificar una definición.
• Repositorio de definiciones de transformación. Un lugar de almacenamiento de las definiciones de transformaciones que son utilizadas para pasar de modelo a otro.
• Ejecución de modelos. Un motor que provee una plataforma virtual donde se pueden ejecutar los modelos.



Una parte fundamental de las herramientas que soportan MDA debe ser la capacidad de dividir los modelos en CIM, PIM y PSMs, así como el marcado de estos y la definición de las transformaciones, para después generar transformaciones entre modelos y generación de código a partir de ellos (y no solo a la generación de código a partir del diagrama de clases como manejan algunas herramientas CASE).
Existen a la vez dos tipos de herramientas MDA, aquellas que generan el código y aquellas que "ejecutan" el modelo. El código generado del primer grupo de herramientas tiene que ser en la mayoría de los casos modificado, ya que no es código terminado que pueda ser compilado y ejecutado (es más bien un 'esqueleto'), eso sin tomar en cuenta que en la mayoría de los casos los programadores tienden a cambiar la estructura del código generado, de tal forma que el modelo queda desactualizado. Algunas herramientas intentan resolver este problema mediante lo que se conoce como ingeniería inversa, con la cual es posible volver a actualizar el modelo a partir del código modificado, sin embargo aun así debe de existir disciplina en los cambios para evitar desincronización entre modelos.
El segundo tipo de herramientas utiliza UML y algunas extensiones del mismo para modelar a detalle una aplicación de tal forma que el modelo pueda ser ejecutado dentro de la herramienta.
Algunas de estas herramientas generan código como Java, C++ o algún lenguaje propio y generan una serie de paquetes que pueden ser ejecutados dentro del servidor de la misma aplicación. El principal problema de estas herramientas es que el generar un modelo UML que pueda ser ejecutado no es sencillo, se debe tener un amplio conocimiento de UML para obtener beneficios reales de una herramienta de este tipo. A continuación se presenta un conjunto de herramientas encontradas durante el proceso de investigación para la escritura de este trabajo.

Capitulo 3

Transformaciones MDA

3.2 Definición de transformación MDA

Las Transformaciones MDA proveen una manera confiable y completa de convertir los elementos del modelo y los fragmentos del modelo de un dominio a otro.
La propuesta MDA (Model Driven Architecture- Arquitectura Dirigida por Modelos) [1] de la OMG (Object Management Group)[2] presenta un proceso de desarrollo de software concebido para dar soporte al desarrollo de sistemas, donde los conceptos más importantes son los modelos y las transformaciones entre ellos que generan a su vez, otros modelos. Estos se convierten, entonces, en los guías del desarrollo de software. En primer lugar, y según es propuesto por MDA, se definen uno o más modelos PIM (Platform-Independent Model) en algún lenguaje específico y que son independientes de cualquier plataforma de desarrollo. Estos PIMs se traducen en uno o más modelos PSM (Platform-Specific Model) que son específicos de la plataforma donde se ejecuten.
Esta "traducción" entre PIMs y PSMs se conoce como "transformación de modelos".
Describir transformaciones de modelos requiere de lenguajes específicos para la definición de las mismas. Actualmente, existen varias propuestas de lenguajes, muchas de ellas basadas en el estándar de la OMG, QVT (Query/View/Transformation) [3].
Entre los lenguajes definidos existen gran variedad de "tipos": icónicos y textuales, declarativos, operacionales y declarativos-operacionales, algunos basados en QVT y no otros no, compatibles con MOF [4], aquellos que soportan OCL [5], los que proveen trazabilidad, los que proveen composición de transformaciones y hasta aquellos a los que se les ha implementado una herramienta CASE o un plugin para Eclipse.

3.3 Clasificación de los lenguajes de transformación

Desde la aparición de la metodología MDA, mucho se ha propuesto y definido en cuanto a lenguajes y herramientas que sirven de soporte y automatizan sus diferentes aspectos. Uno de estos aspectos donde se ha puesto más énfasis es en la definición de lenguajes que permiten traducir un modelo en otro, pasando desde PIMs a PSMs según indica MDA.
Antes de continuar, enunciamos algunas definiciones de conceptos de lenguajes de transformación: una transformación es la generación automática de un modelo de salida o target a partir de un modelo de entrada o source y de acuerdo a la descripción de una transformación. Esta descripción se compone de una o más reglas de transformación que describen cómo un modelo source puede transformarse en un modelo target (cada uno en sus respectivos lenguajes).
Finalmente, una regla de transformación describe cómo un elemento del source puede ser transformado en uno o más elementos del target.

Lenguaje	Transformación

ATL (Atlas Transformation Language) [7]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el Atlas group (INRIA)
BOTL(Basic Object-Oriented Transformation Language) [8]	Propuesta gráfica de la Universidad Técnica de München
GreAT (Graph Rewriting and Transformations) [8]	Basado en la transformación de grafos. Es la propuesta de una organización independiente: ESCHER Research Institute (The Embedded Systems Consortium for Hybrid and Embedded Research)
JMI (Java Metadata Interface) [10]	Propuesta de Sun basado en MOF que permite manipulación de archivos XMI.
Kent o KMTL(Kent Model Transformation Language)[10]	Propuesta realizada por la Universidad de Kent
MTRANS [12]	Proyecto de la Universidad de Nantes. Es un Framework que permite expresar transformaciones de modelos.
Mod-Transf [13]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el Dart team (INRIA)
MOFScript [14]	Lenguaje de transformación modelo a texto (cuya propuesta pertenece a la OMG) y herramienta como plugin para Eclipse
MOLA (Model transformation Language) [15]	Lenguaje gráfico para describir transformaciones propuesto por la Universidad de Letonia.
MT model transformation language [16]	Basado en QVT y desarrollado como DSL (Domain Specific Language) por L. Tratt del King's College de Londres.
MTL (Model Transformation Language) [17]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el Triskell team (INRIA)
QVT (Query/View/ Transformation) [3]	Especificación estándar de OMG. Está basado en MOF (Meta Object Facility) para lenguajes de transformación en MDA.
Stratego [18]	Lenguaje de descripción de transformaciones de programas. La herramienta desarrollada como soporte del lenguaje es Stratego/XT
Tefkat [19]	Lenguaje declarativo basado en MOF y QVT. Es el aporte de la Universidad de Queensland.
UMLX [20]	Lenguaje gráfico que extiende a UML y también a QVT.
xUML (eXecutable UML) [21]	Propuesta basada en UML para la construcción de modelos de dominio

	ejecutables y sus transformaciones.

En la sección anterior se ha podido ver que las propuestas de lenguajes de transformación de modelos son muchas y muy diversas. Es claro que a la hora de describir una transformación aparece un abanico de opciones de las cuales elegir. Y surge, como trabajo adicional, la elección de un lenguaje. Resulta necesario entonces conocer las características de estos lenguajes.
A continuación se listan diferentes criterios a partir de los cuales se realizará el análisis de algunos de los lenguajes presentados. Para dicho análisis se definieron dos metamodelos y transformaciones entre los mismos (compuestas por varias reglas) y se utilizó cada uno de los lenguajes seleccionados para describir la transformación

Definición
La definición de una transformación determina características generales de la transformación. Se enuncian a continuación:
• Tipo: indica si la sintaxis del lenguaje es icónica (gráfica) o textual y si es modelo a modelo (M2M) o modelo a texto (M2Text).
• Estilo: un lenguaje puede ser declarativo, operacional o ambos
• Pre y post condiciones: si es posible especificarlos en la transformación y de qué forma puede hacerse (coloquial, con OCL, otro lenguaje). Lo deseable es que se puedan especificar y en lo posible, en OCL ya que es un lenguaje formal y estándar.
• MOF/QVT compatible: indica si el lenguaje se basa en los estándares definidos por OMG: si los metamodelos (lenguajes de dominio y codominio) que participan en la transformación son instancias de MOF y si el lenguaje de transformación se basa en QVT. Lo esperado es que los lenguajes sean compatibles con ambas

Reglas de Transformación
Las reglas de transformación son unidades más pequeñas que componen una transformación.
Como fue mencionado anteriormente, estas describen cómo un elemento del modelo de entrada puede ser transformado en uno o más elementos del modelo de salida.
Los aspectos a analizar son:
• Dominio: define uno o más modelos "de entrada" u origen o source y uno o más modelos "de salida" o destino o target sobre los cuales operarán las reglas de transformación
    - o Lenguajes del dominio: cada dominio tiene un lenguaje asociado. Se espera que los lenguajes sean instancias de MOF.
    - o Dirección: indica si los metamodelos source y target son in, out, o in/out (en el sentido en que se usan los parámetros en programación). Y si es unidireccional o bidireccional el sentido de la transformación
    - o Relación entre origen y destino de la transformación: Si el source y el destino tienen el mismo o diferente modelo.
• Cuerpo de la transformación: en cuanto al cuerpo de la transformación, podemos decir que consta de:
    - o Declaración de (meta)variables y sus tipos
    - o Patrones de transformaciones
    - o Separación sintáctica: algunos lenguajes separan claramente las partes de una regla de transformación que operan sobre el modelo source (LHS) de las partes que operan sobre el modelo target (RHS). La separación sintáctica hace a la legibilidad del lenguaje.

o Estructuras intermedias: si el lenguaje recurre a alguna estructura adicional para describir la transformación y que no es parte del modelo a transformar.

o Parametrización: el tipo más simple de parametrización es el uso de los parámetros del control que permiten el paso de valores.

• Aplicación de las reglas: como hemos mencionado, una regla se aplica a algún elemento del modelo de entrada. Como puede haber más de una regla que "machee" con un elemento particular, se debe definir alguna estrategia para determinar el orden de aplicación de las reglas.

o Orden: la aplicación de las reglas puede ser determinístico o no determinístico.

o Aplicación condicional: en algunas reglas de transformación se puede tener aplicación condicional de las mismas. En estos casos, existe una condición que debe ser verdadera para que la regla se ejecute.

o Iteración de reglas: si el lenguaje provee estructuras de iteración, o mecanismos de recursión.

• Organización de las reglas: se refiere a la composición y estructuración de múltiples reglas. Se puede dividir en varios aspectos:

o Modularización: en el caso en que se provea modularización de las reglas (agrupar un conjunto de reglas en un módulo, que puede ser llamado desde otros módulos).

o Mecanismos de reuso: la definición de regla/s en base a otra/s.

Trazabilidad

Hace referencia a aquellos mecanismos provistos por los lenguajes que permiten guardar ciertos aspectos de la ejecución de la transformación. Se pueden crear y mantener "conexiones" o links entre elementos del dominio y del codominio que mapean los dominios source y target, cada vez que una regla de transformación es ejecutada.

Composición

La composición indica cómo pueden relacionarse un número de transformaciones para obtener una nueva. Pueden encadenarse dos o más transformaciones consecutivamente; pueden componerse dos o más transformaciones existentes en una nueva transformación con sus nuevas relaciones o pueden combinarse dos transformaciones de forma tal que se obtienen codominios más amplios.

3.4 Análisis de los Lenguajes Seleccionados

ATL (Atlas Transformation Language)

3.4.1 ATL

Es un lenguaje de transformación de modelos híbrido que permite, en su definición de transformaciones, especificar construcciones declarativas y operativas. La propuesta es del ATLAS Group del INRIA & LINA, de la Universidad de Nantes y fue desarrollado como parte de la plataforma AMMA (ATLAS Model Management Architecture).

ATL es un lenguaje modelo a modelo híbrido o mixto, en el sentido que permite construcciones tanto declarativas como imperativas.

Otros detalles de la Definición del lenguaje:

• Es compatible con los estándares de la OMG: es posible describir transformaciones modelo a modelo (y ambos deben ser instancias de MOF). Además, el lenguaje se basa en QVT.
• Permite definir pre y post condiciones en un lenguaje ya conocido: OCL
En cuanto a las reglas de transformación, ATL define un dominio (metamodelo) para el source y otro dominio para el target, siendo ambos instancias de MOF y con direcciones in y out respectivamente. Source y target pueden tener iguales o diferentes dominios (aunque sean iguales, ambos deben ser claramente identificados). Si bien las transformaciones son unidirecciones, ATL permite la definición de transformaciones bidireccionales como la implementación de dos transformaciones, una para cada dirección.
Como características particulares del lenguaje, podemos mencionar las estructuras que define este lenguaje. En primer lugar, la definición de transformaciones forman módulos (modules) que contienen las declaraciones iniciales y un número de helpers y reglas de transformación. Los helpers, son una estructura intermedia dentro de las transformaciones que facilitan la navegación, la modularización y el reuso. Permiten definir operaciones y tuplas OCL Existe también una construcción llamada called rule, y que representa a un procedure. Estas pueden contener argumentos y pueden ser invocadas por su nombre. Resulta sumamente expresivo y de fácil escritura para quienes conocen OCL (no es necesario aprender un nuevo lenguaje).
La aplicación de las reglas se realiza de forma no determinística, por "macheo" de reglas y no se ha provisto ninguna construcción o claúsula que permita aplicar en forma condicional las reglas. Cabe mencionar que la invocación de called rules es determinística. Esta invocación, junto con la utilización de parámetros permite soportar recursión.
ATL provee modularización por sus procedimientos o called rules, además de que sus módulos pueden incluir a otros; y mecanismos de reuso ya que las reglas se pueden heredar.
En cuanto a la trazabilidad, este lenguaje crea un traceability link con la ejecución de cada regla que se guarda en el motor de la transformación. Este link relaciona a tres (3) elementos: la regla, el "macheo" (los elementos del source) y los elementos creados en el target.
ATL permite componer transformaciones mediante la definición de reglas.
Para finalizar, vale la pena mencionar que ATL se ha convertido en un lenguaje tan utilizado que ya se ha definido un repositorio de transformaciones entre diversos lenguajes (http://www.eclipse.org/m2m/atl/atlTransformations/) .Ya existen más de 60 trasformaciones documentadas y con los archivos fuente disponibles.

3.4.2 MOFScript

MOFScript es un lenguaje de transformación de modelo a texto presentado por la OMG. Este lenguaje presta particular atención a la manipulación de strings y de texto y al control e impresión de salida de archivos.
No sólo es un estándar, sino que además se basa en otros estándares de la OMG: es QVT compatible y MOF compatible con el modelo de entrada (el target siempre es texto). Para las reglas de transformación, MOFScript define un metamodelo de entrada para el source y sobre el cual operarán las reglas. El target es generalmente un archivo de texto (o salida de texto por pantalla).
Las transformaciones son siempre unidireccionales y no es posible definir pre y post condiciones para ellas. La separación sintáctica resulta clara por la misma definición de

reglas, lo que hace a la legibilidad del lenguaje. No provee estructuras intermedias, pero sí parametrización necesaria para la invocación de reglas.
En cuanto a la aplicación de reglas, podemos decir que se aplican en forma determinística y en orden secuencial. Se provee aplicación condicional e iteración de reglas. Las condiciones de aplicación se expresan en cláusulas when, como definición de guardas. La iteración se realiza mediante los iteradores for each y while.
MOFScript no organiza las reglas en módulos propiamente dichos. Ahora bien, en la definición de una regla se puede invocar a otras reglas, utilizando incluso parámetros, con lo cual se asemeja bastante a un módulo. Y es posible definir jerarquías de transformaciones.
Finalmente, para traceability, MOFScript define (pero no implementa aún) un conjunto de conceptos para relacionar elementos del source con sus ubicaciones en los archivos de texto generados en el target.
En cuanto a la composición de transformaciones, no es algo que se haya pensado en este lenguaje aún.

3.4.3 MOLA (Model transformation Language)

MOLA es un lenguaje gráfico de transformación de modelos, propuesto por la Universidad de Letonia. La intensión de este lenguaje es combinar la programación estructurada tradicional con reglas basadas en patrones de transformación.
La definición del lenguaje no se basó en QVT y los metamodelos que participan de la transformación pueden o no ser instancias de MOF; lo único restrictivo es que ambos deben ser definidos como dos modelos diferentes (source y target) aunque sean el mismo. El source es el modelo de entrada in/out (sus elementos pueden ser modificados) y el target de salida (sólo out).
Las transformaciones son unidireccionales.
Si bien no se mencionan pre y post condiciones, el lenguaje provee notas donde pueden escribirse cláusulas OCL para la aplicación condicional de reglas. Estas podrían usarse entonces para pre y post condiciones.
En cuanto al cuerpo de la transformación, se ve que no hay separación sintáctica de los elementos que se transforman y de los que se crean. Tampoco provee estructuras intermedias.
El elemento principal del lenguaje es un concepto gráfico del loop, que se utiliza mucho para iterar sobre los elementos de los modelos que participan de la transformación. También para ello, permite definir variables locales a cada regla de transformación.
Sobre la aplicación de reglas, podemos decir que se aplican en forma determinística y en orden secuencial según son definidas. Existe la aplicación condicional mediante la adición de notas con cláusulas OCL. La iteración se realiza mediante dos tipos de loop: un loop de “tipo uno” que se ejecuta una vez para cada instancia válida del source (for each); un loop de “tipo dos” que continúa la ejecución mientras haya al menos una instancia válida en el source (while). El lenguaje utiliza patrones definidos para ser aplicados en las transformaciones.
Si bien gráficamente resulta intuitivo, las iteraciones pueden volverse en poco confusas, sobre todo si hay varias reglas anidadas.
En cuanto a traceability, MOLA permite definir “mapping associations” para tracear instancias entre modelos. Estas se definen en las reglas como notas anexadas entre los elementos del dominio y codominio que participan de la transformación.
La composición de transformaciones no ha sido tenida en cuenta en la definición del lenguaje.

3.4.4 Tefkat

Tefkat es la definición e implementación de un lenguaje para transformación de los modelos. La propuesta fue realizada por de la Universidad de Queensland, Australia y se encuentra en una etapa de desarrollo bastante avanzada con respecto a otros lenguajes.

El lenguaje transforma modelos (M2M) en forma textual y ha adoptado un paradigma declarativo. Tefkat es totalmente compatible con los estándares de la OMG. Entiende que los modelos source y target definen dominios diferentes para cada uno de ellos, siendo ambos instancias de MOF.

Como desventaja, podemos mencionar que no hay declaración de pre ni de post condiciones.

Introduciéndonos en el cuerpo de la transformación, vemos que es posible declarar variables y metavariables, y que la separación sintáctica es provista por el lenguaje. De hecho, le aporta mucha legibilidad y claridad a la escritura de las reglas. Existen estructuras FORALL (algo de source) MAKE (algo del target) y FROM (algo de source) TO (algo del target) más la identación de las sentencias.

Tefkat también provee patrones (para el source) y templates (para el target) que se utilizan para nombrar restricciones que se pueden utilizar en más de una regla. Estas construcciones pueden a su vez parametrizarse, permitiendo la invocación a patrones y la recursión.

Las reglas se aplican en forma no determinística. Existe la aplicación condicional soportada por la cláusula IF-THEN-ELSE que permite la ejecución de reglas cuando la guarda es evaluada como verdadera. Las reglas pueden iterar y ejecutarse con recursión.

Para traceability, Tefkat incluye una cláusula LINKING que representa un zapping (asociación) entre los elementos del source y del target que participan de la transformación y que son almacenados una vez realizada la misma.

En este lenguaje es posible componer transformaciones mediante la definición de reglas, tomando transformaciones existentes y creando una nueva que preserva las relaciones.

3.4.5 UMLX

UMLX es un lenguaje gráfico de transformaciones entre modelos (M2M) y que se basó en extensiones mínimas a UML. Es una propuesta de E. Willink, del GMT Consortium. En un último avance, se ha anunciado que la transformación textual que la herramienta UMLX traduce desde el gráfico, puede también editarse. Como se traduce a lenguaje OCL, sería muy conveniente para aquellos que ya conocen este lenguaje y no se cae en la necesidad de conocer otro lenguaje. La definición de este lenguaje se basó en QVT. Asimismo, los modelos participantes de las transformaciones, deben ser instancias de MOF. Cada uno de estos es tomado como un dominio diferente (uno de entrada o in y otro de salida o out), aunque se trate de los mismos modelos. Para estos no pueden especificarse pre o post condiciones.

En cuanto a las reglas de transformación, UMLX utiliza diferentes íconos gráficos para las transformaciones, para las reglas y para las relaciones de creación, preservación y eliminación de elementos. Cabe destacar que la semántica de los íconos del lenguaje no se ha definido, y aunque es gráficamente intuido, para algunas relaciones, no queda claro cuál es su significado.

Las estructuras que define este lenguaje son bastante simples e intuitivas, pero carecen de una semántica bien definida. No hay módulos, estructuras intermedias ni cláusulas de iteración o condición. Sólo existen variables locales, para indicar que se hace referencia a una instancia en particular. Por ejemplo, si se define una regla para todas las clases, y se define una regla también para cada atributo, al hacer referencia a una instancia de

clase particular (cl, por ejemplo), se pueden mencionar a todos los atributos de cl, notándolo con @cl.
La aplicación de las reglas se realiza de forma no determinística, por pattern matching de reglas y no se ha provisto ninguna construcción o cláusula que permita aplicar en forma condicional las reglas. Tampoco pueden invocarse otras reglas, ni se utilizan parámetros. Ni la trazabilidad ni la composición han sido agregadas al lenguaje.

Conclusiones

De este análisis se desprende que tanto ATL como Tefkat son lenguajes muy completos y en avanzado desarrollo. Ambos adhieren a los estándares de la OMG, utilizan un lenguaje formal como es OCL y proveen mecanismos para traceability y composición de las reglas de transformación. ATL hace uso además de estructuras intermedias que mejoran la legibilidad y el reuso. Dentro de los lenguajes gráficos, MOLA es una buena propuesta: con buena legibilidad, intuitiva, fácil de comprender y de utilizar.

Capítulo 4

QVT: el estándar de OMG para transformaciones

Una regla de transformación de modelos debe definir, evitando cualquier ambigüedad, la relación implícita que existe entre sus partes. MDA no especifica ni prescribe ningún lenguaje para la transformación de modelos. El estándar actualmente establecido por OMG para crear consultas, vistas y transformaciones de modelos es QVT (Query, Views, Transformations). Las transformaciones, en el contexto de QVT se clasifican en relación (relation) y función (mapping); las relaciones especifican transformaciones multidireccionales, no permiten crear o modificar modelos, pero sí chequear la consistencia entre dos o más modelos relacionados. Las funciones, en cambio, implementan la transformación, es decir, transforman elementos de un dominio en elementos de otro. Se han propuesto varios lenguajes de transformación: BOTL; ATL; Tefkat; Kent Model y también el uso de sentencias OCL para especificar las transformaciones. Todos estos lenguajes asumen que los modelos involucrados en la transformación cuentan con una definición formal de su sintaxis, expresada en términos de metamodelos MOF.

"Query-View-Transformations", da respuesta a la necesidad de transformaciones entre modelos cuyos lenguajes están definidos con MOF, con los siguientes elementos:

• **Query** (Consulta): es una expresión evaluada sobre un modelo dado que tiene como resultado 1 ó más instancias de tipos (del modelo fuente o en el lenguaje de consulta):

• **View** (Vista): es un modelo derivado completamente de otro modelo. Son más generales que las Consultas. Suelen ser entidades de sólo lectura.

• **Transformation**: genera un modelo destino desde un modelo fuente. Las relaciones especifican las transformaciones y las correspondencias (mappings) las implementan.

4.1 Objetivos de QVT

• Requisitos obligatorios:

- Lenguaje de Consulta
- Lenguaje de Transformación del lenguaje fuente al lenguaje destino.
- Definido con metamodelos MOF
- Ejecutable / Transportable versus Caja-Negra XForms
- Definición de vistas
- Transformaciones declarativas / incrementales

• Requisitos opcionales:

- Transformaciones bidireccionales

- Registro de las transformaciones
- Reutilización y extensión de transformaciones genéricas
- Las transformaciones con semática de transacciones: commit, rollback.
- Tranformaciones dentro de un lenguaje específico.

4.1.1 Descripción general de QVT

QVT es el estándar de OMG para transformaciones. Comprende tres diferentes lenguajes M2M: dos lenguajes declarativos llamados Relations y Core, y un tercer lenguaje, de naturaleza imperativa, llamado Operational Mappings.

La especificación de QVT define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational como puede observarse en la figura 4.1.1.1. Estos paquetes principales se comunican entre sí y comparten otros paquetes intermedios (Figura 4.1.1.2). El paquete QVTBase define estructuras comunes para transformaciones. El paquete QVTRelation usa expresiones de patrones template definidas en el paquete QVTTemplateExp. El paquete QVTOperational extiende al QVTRelation, dado que usa el mismo framework para trazas. Usa también las expresiones imperativas definidas en el paquete ImperativeOCL. Todos los paquetes QVT dependen del paquete EssentialOCL de OCL 2.0, y a través de él también dependen de EMOF (EssentialMOF).

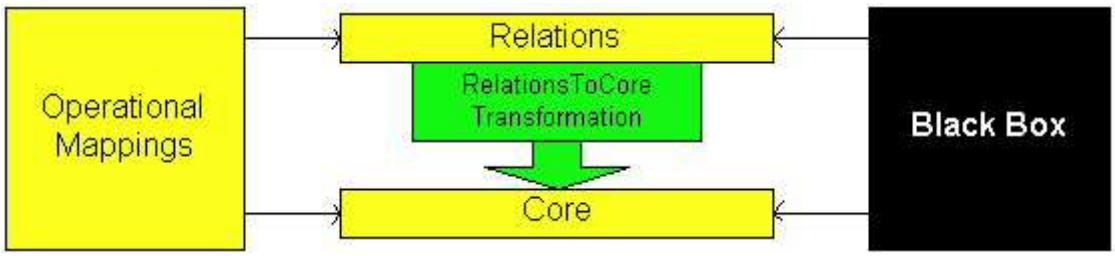


Figura 4.1.1.1: Relación entre los metamodelos de QVT

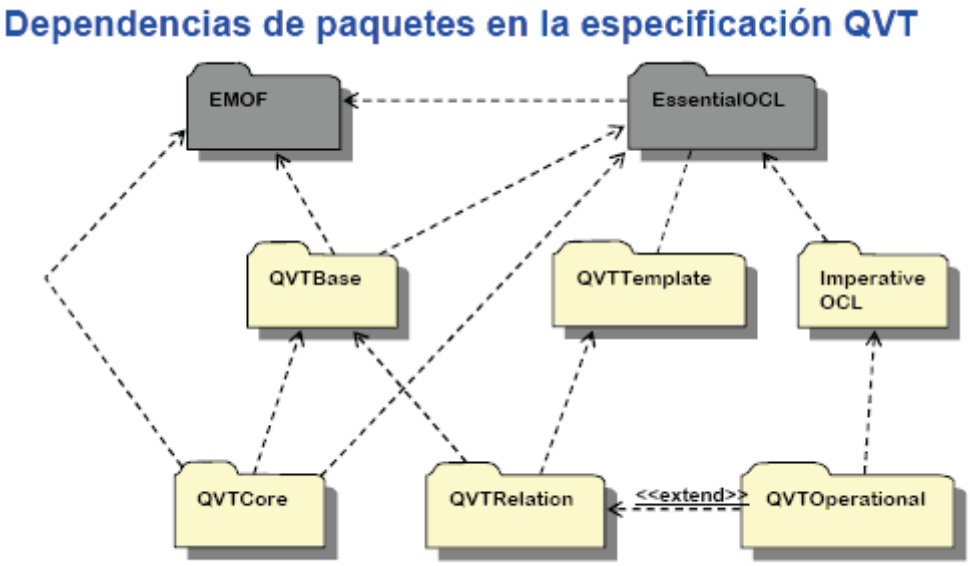


Figura 4.1.1.2: Dependencias de paquetes en la especificación QVT

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

4.1.2 QVT Declarativo

La parte declarativa de QVT está dividida en una arquitectura de dos niveles. Las capas son:

- un metamodelo y lenguaje *Relations* amigable para el usuario, que soporta pattern matching complejo de objetos y creación de template para objetos. Las trazas entre elementos del modelo involucrados en una transformación se crean implícitamente. Soporta propagación de cambios, ya que provee un mecanismo para identificar elementos del modelo destino.

- un metamodelo y lenguaje *Core* definido usando extensiones minimales de EMOF y OCL. Las trazas no son automáticamente generadas, se definen explícitamente como modelos MOF, y pueden crearse y borrarse como cualquier otro objeto. El lenguaje Core no soporta pattern matching para los elementos de modelos. Esta propuesta absolutamente minimal lleva a que el Core sea el "assembler" de los lenguajes de transformación.

Relations

Es una especificación declarativa de las relaciones entre modelos MOF. Las relaciones pueden pedir que otras relaciones también se establezcan entre elementos particulares del modelo, que cumplen con los *pattern matching*. En este lenguaje, una transformación entre modelos se especifica como un conjunto de relaciones que deben establecerse para que la transformación sea exitosa.

Los modelos tienen nombre, y los elementos que contienen deben ser de tipos correspondientes al metamodelo que referencian. Un ejemplo:

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

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

En el ejemplo que sigue, en la relación PackageToSchema, el dominio para el modelo "uml" está marcado como checkonly y el dominio para el modelo "rdbms" está marcado enforce. Estas marcas habilitan la modificación (creación/borrado) de elementos en el modelo "rdbms", pero no en el modelo "uml", el cual puede ser solamente leído pero no modificado.

```
relation PackageToSchema /* transforma cada paquete UML a un esquema
relacional */
{
      checkonly domain uml p:Package {name=pn}
      enforce domain rdbms s:Schema {name=pn}
}
```

Relations top-Level

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

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

Una relación topLevel tiene la palabra top antecediéndola para distinguirla sintácticamente. En el ejemplo de arriba, `PackageToSchema` y `ClassToTable` son relaciones topLevel, mientras que `AttributeToColumn` es una relación no topLevel, que será invocada por alguna de las otras para su ejecución.

4.1.3 QVT Operational

Además de sus lenguajes declarativos, QVT proporciona dos mecanismos para implementaciones de transformaciones: un lenguaje estándar, Operational Mappings, e implementaciones no-estándar o Black-box.

El lenguaje Operational Mappings se especificó como una forma estándar para proveer implementaciones imperativas. Proporciona una extensión del lenguaje OCL mediante el agregado de nuevas construcciones con efectos laterales que permiten un estilo más procedural, y una sintaxis que resulta familiar a los programadores. Este lenguaje puede ser usado en dos formas diferentes:

- Primero, es posible especificar una transformación únicamente en el lenguaje Operational Mappings. Una transformación escrita usando solamente operaciones Mapping es llamada *Transformación Operacional* y es en la cual nos centraremos en las siguientes secciones.
- Alternativamente, es posible trabajar en modo híbrido. El usuario tiene entonces que especificar algunos aspectos de la transformación en el lenguaje Relations (o Core), e implementar reglas individuales en lenguaje imperativo a través de operaciones Mappings.

4.1.4 Operational Transformations y conceptos relacionados

Una transformación Operacional representa la definición de una transformación unidireccional, Tiene una signatura indicando los modelos involucrados en la transformación que se modelan como *Model Parameter* y define una operación entry, llamada "main", la cual representa el código inicial a ser ejecutado para realizar la transformación y modelada como una *Entry Operation*. La signatura es obligatoria, pero no así su implementación. Esto permite implementaciones de caja negra definidas fuera de QVT.

El ejemplo que sigue muestra la signatura y el punto de entrada de una transformación llamada Uml2Rdbms, que transforma diagramas de clase UML en tablas RDBMS.

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
// el punto de entrada de la ejecucion

main() {
      uml.objectsOfType(Package)->map packageToSchema();
}

..
}
```

La signatura de esta transformación en particular declara que un modelo `rdbms` de tipo `RDBMS` será derivado desde un modelo `uml` de tipo `UML`. En el ejemplo, el cuerpo de la transformación `(main)` especifica que en primer lugar se recupera la lista de objetos de tipo Paquete y luego se aplica la operación de mapeo (mapping operation) llamada `packageToSchema()` sobre cada paquete de dicha lista. Esto último se logra utilizando la operación predefinida `map()` que itera sobre la lista.

Operational Transformation parte del metamodelo

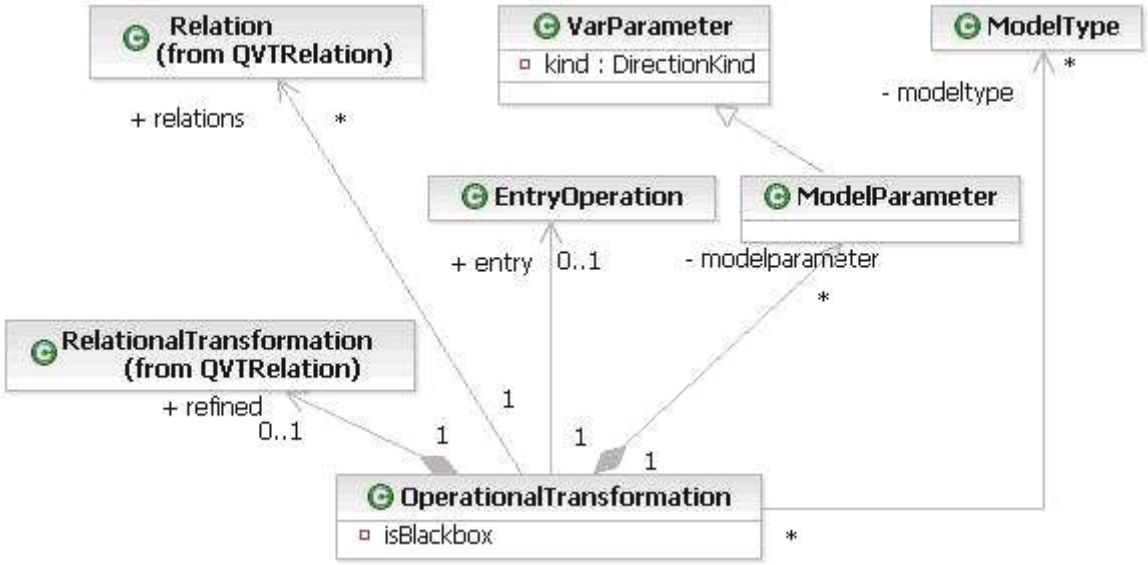


Figura 4.1: metamodelo de Operational Transformation

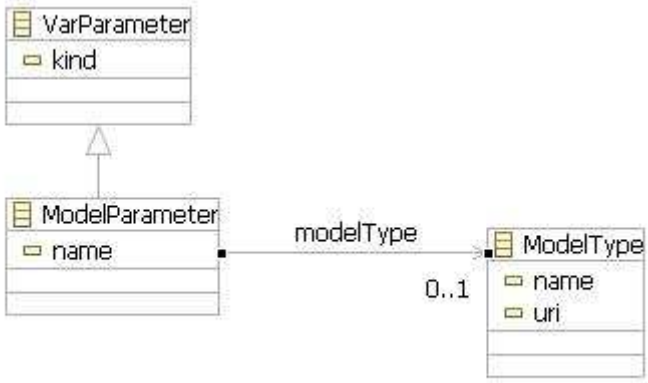
- EntryOperation [0..1]: Una operación actuando como punto de entrada para la ejecución de la transformación operacional.
- ModelParameter [1..*]: Indica la signatura de esta transformación operacional. Un parámetro de modelo indica un tipo de dirección (in/out/inout) y un tipo dado por un tipo de modelo. Cada parámetro tiene un tipo de ModelType.
- ModelType: Cada parámetro de modelo se corresponde a un tipo de modelo que esta definido por la transformación. Un tipo de modelo esta definido por el metamodelo.
- VarParameter: un parámetro variable es un concepto abstracto que fue introducido para permitir referenciar parámetros de la misma manera que las variables, especialmente en las expresiones OCL.
- RelationalTransformation: es una especialización de Transformation y representa una definición de transformación escrita en el lenguaje QVTRelation.

4.1.4.1 Model Parameter

Es un parámetro de una Operational Transformation y conforma la signatura de la misma. Cada parámetro referencia implícitamente a un modelo que participa en la transformación.

Cada parámetro esta precedido de un indicador que indica el modo de ejecución sobre el modelo: `in` significa que no se pueden realizar cambios sobre el modelo, es decir, que seria de solo lectura; `inout` significa que el modelo puede ser actualizado; `out` significa que el modelo va a ser creado. Estos parámetros son accesibles globalmente en toda la transformación.

Cada parámetro tiene un tipo modelado como Model Type que detallaremos a continuación de esta sección.



4.1.4.2 Model Type

Cada parámetro se ajusta a un modelo tipado, es cual es definido o referenciado por la transformación. Un modelo tipado esta definido por un metamodelo, conforme a un tipo y a un conjunto opcional de expresiones de restricción. El metamodelo define el conjunto de clases y propiedades que son alcanzados y esperados por la transformación.

El tipo esta definido de dos maneras: **`strict`** and **`effective`** (este es el default). Cuando el tipo es **`strict`**, los objetos del modelo deben ser necesariamente instancias de las clases del metamodelo asociado. Cuando el tipo es effective cualquier objeto en el modelo que tiene un tipo que referencia a un tipo asociado al metamodelo debe contener propiedades definidas la clase del metamodelo y debe tener tipos compatibles.

Un modelo tipado es definido como una subclase de `Class` entonces es posible definir operations y propiedades de `Class`

Cuando un `modeltype` es explícitamente declarado, la sintaxis es la siguiente:

```
modeltype <modeltypeid> "<conformance>"
uses <packageid>("<uri>") where {<expressions>…};
```

Las condiciones extras pueden usar la variable **`self`** la cual es definida implícitamente en el bloque **`where`** y referencia conceptualmente a un instancia del model type (el modelo).

La siguiente declaración declara que un model type usa para su definición un paquete existente denominado SimpleUml y provee su URI. La segunda declaración solamente da la URI e indica un tipo stric de conformidad.

```
modeltype UML uses SimpleUml("http://omg.qvt-samples.SimpleUml");
modeltype RDBMS "strict" uses "http://omg.qvt-samples.SimpleRdbms";
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS);
```

La declaración de un model type debe tener el mismo nombre que la definición del metamodelo.

```
metamodel SimpleUML { … };
metamodel SimpleRDBMS { … };
transformation Uml2Rdbms(in uml:SimpleUML, out rdbms:SimpleRDBMS);
```

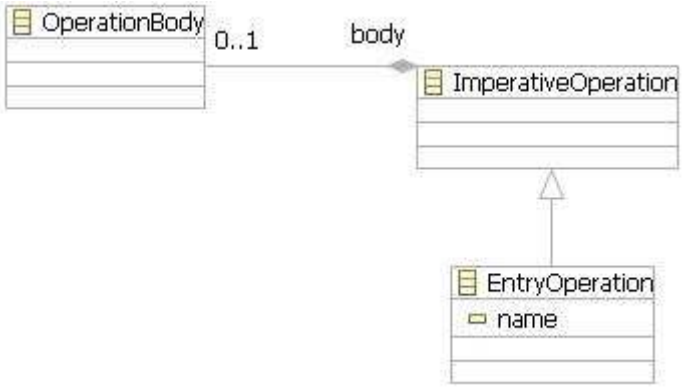
4.1.4.3 Entry Operation

Es el punto de entrada de la ejecución de la transformación. Su cuerpo esta formado por una lista ordenada de expresiones ejecutadas en secuencia y modelado como *Operation Body*. Una transformación debe definir un solo punto de entrada el cual es invocado cuando la ejecución de la transformación comienza.

Un entry operation no tiene parámetros pero tiene acceso a todas las propiedades o parámetros indicados como globales, tales como parámetro modelo.

Como ya mencionamos el nombre de una entry operation es main

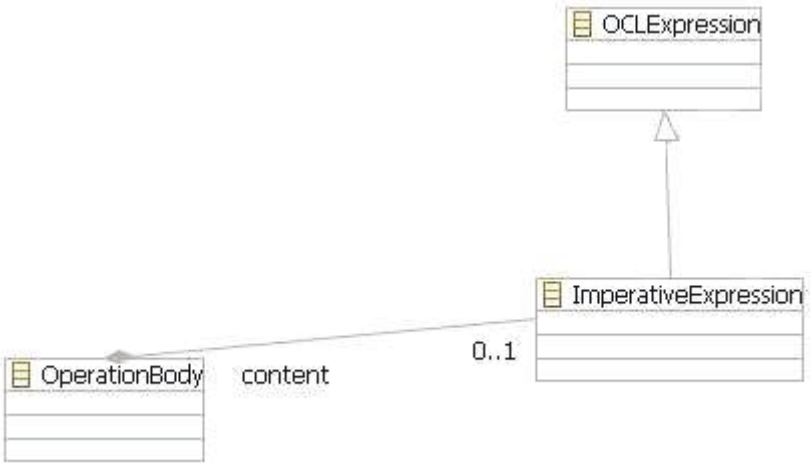
```
transformation UmlCleaning (inout uml:UML);
main() {
      uml->objectsOfType(Package)->map cleanPackage();
}
```



4.1.4.4 Operation body

Una operation body contiene la implementación de una *Imperative Operation* que es una lista ordenada de expresiones que se ejecutaran en secuencia. Una operation body define un alcance que esta contenido en el alcance de la definición de la operación. Las variables y los parámetros que estén definidos fuera del alcance, no serán accesibles. La

asociación content: OclExpression [*] {composes, ordered} es una lista de expresiones del cuerpo de la operación.



4.1.4.5 Mapping operations

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

Una mappingOperation se describe sintácticamente mediante una signatura, una guarda (su cláusula when), el cuerpo del mapping y una poscondición (su cláusula where). La operación puede no incluir un cuerpo (es decir, que oculta su implementación) y en ese caso se trata de una operación de caja negra (black-box).

Una mapping operation siempre refina una relación, donde cada dominio se corresponde con un parámetro del mapping. El cuerpo de una operación mapping se estructura en tres secciones opcionales:

- La sección de inicialización es usada para crear los elementos de salida,
- la intermedia, sirve para asignarle valores a los elementos de salida
- y la de finalización, para definir código que se ejecute antes de salir del cuerpo.

La operación mapping que sigue define cómo un paquete UML se transforma en un esquema RDBMS.

```
mapping Package::packageToSchema() : Schema

when { self.name.startingWith() <> "_"}
{
      name := self.name;
      table := self.ownedElement->map class2table();
}
```

La relación implícita asociada con esta operación mapping tiene la siguiente estructura:

```
relation REL_PackageToSchema {
checkonly domain:uml (self:Package)[]
enforce domain:rdbms (result:Schema)[]
when { self.name.startingWith() <> "_" }
}
```

La relación PackageToSchema indica que el dominio para el modelo “uml” está marcado como checkonly y el dominio para el modelo “rdbms” está marcado enforce. Estas marcas habilitan la modificación (creación/borrado) de elementos en el modelo “rdbms”, pero no en el modelo “uml”, el cual puede ser solamente leído pero no modificado.

Capítulo 5

Lambda Cálculo

5.1 Introducción

El objetivo del paradigma funcional [22] es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitando el concepto de estado del cómputo. La secuencia de computaciones llevadas a cabo por el programa se regiría única y exclusivamente por la reescritura de definiciones más amplias a otras cada vez más concretas y definidas, usando lo que se denominan definiciones dirigidas.

Otro de los objetivos primordiales de dicho paradigma es buscar satisfacer las necesidades del usuario con respecto a operaciones matemáticas y convertirse en un lenguaje más expresivo.

5.2 Paradigma Funcional

5.2.1 Historia

Sus orígenes provienen del Cálculo Lambda (o λ-cαlculo), una teoría matemática elaborada por Alonso Church como apoyo a sus estudios sobre Computabilidad.

5.2.2 Cálculo Lambda

Los orígenes teóricos del modelo funcional se remontan a la década del 30, más precisamente al año 1934, cuando Alonso Church introdujo un modelo matemático de computación llamado lambda calculo.

A pesar de que en esta época las computadoras aun no existían el lambda cálculo se puede considerar como el primer lenguaje funcional de la historia y sus fundamentos fueron la base de toda la teoría de la programación funcional y de los lenguajes funcionales desarrollados posteriormente. Se puede decir que los lenguajes funcionales modernos son versiones de lambda cálculo con numerosas ayudas sintácticas.

5.2.3 Características

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial (el significado de una expresión depende únicamente del significado de sus subexpresiones), y por tanto, la carencia total de efectos laterales.

Otras características propias de estos lenguajes son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas).

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos. La diferencia entre ambos consiste en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes procedimentales, como las secuencias de instrucciones o la asignación de variables.

En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.

Entre los lenguajes funcionales puros, cabe destacar a Haskell y Miranda. Los lenguajes funcionales híbridos más conocidos son Lisp, Scheme, Ocaml y Standard ML (estos dos últimos, descendientes del lenguaje ML).

La programación funcional, es un modelo basado en la evaluación de funciones matemáticas, entendidas como mecanismos para aplicar ciertas operaciones sobre algunos valores o argumentos, para obtener un resultado o valor de la función para tales argumentos.

5.3 Lenguajes Funcionales

Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas.

La desventaja de este modelo es que resulta bastante alejado del modelo de la máquina de von Neumann y, por lo tanto, la eficiencia de ejecución de los intérpretes de lenguajes funcionales no es comparable con la ejecución de los programas imperativos precompilados. Para remediar la deficiencia, se está buscando utilizar arquitecturas paralelas que mejoren el desempeño de los programas funcionales, sin que hasta la fecha estos intentos tengan un impacto real importante**.**

5.4 Definición formal

5.4.1 Sintaxis

En el cálculo lambda, una expresión o término se define recursivamente a través de las siguientes reglas de formación:

1. Toda variable es un término: x, y, z, u, v, w, x1, x2, y9,...

2. Si t es un término y x es una variable, entonces (λx.t) es un término (llamado una abstracción lambda).
3. Si t y s son términos, entonces (ts) es un término (llamado una aplicación lambda).
4. Nada más es un término.

Según estas reglas de formación, las siguientes cadenas de caracteres son términos:

x
(xy)
(((xz)y)x)
(λx.x)
((λx.x)y)
(λz.(λx.y))
((x(λz.z))z)

Por convención se suelen omitir los paréntesis externos, ya que no cumplen ninguna función de desambiguación. Por ejemplo se escribe (λz.z)z en vez de ((λz.z)z), y se escribe x(y(zx)) en lugar de (x(y(zx))). Además se suele adoptar la convención de que la aplicación de funciones es asociativa hacia la izquierda. Esto quiere decir, por ejemplo, que xyzz debe entenderse como (((xy)z)z), y que (λz.z)yzx debe entenderse como ((((λz.z)y)z)x).

Las primeras dos reglas generan funciones, mientras que la última describe la aplicación de una función a un argumento. Una abstracción lambda λx.t representa una función anónima que toma un único argumento, y se dice que el signo λ liga la variable x en el término t. En cambio, una aplicación lambda ts representa la aplicación de un argumento s a una función t. Por ejemplo, λx.x representa la función identidad x → x, y (λx.x)y representa la función identidad aplicada a y. Luego, λx.y representa la función constante x → y, que devuelve y sin importar qué argumento se le dé.

Lo que las hace interesantes a las expresiones son las nociones de equivalencia y reducción que pueden ser definidas sobre ellas.

5.4.2 Variables libres y ligadas

Las apariciones (ocurrencias) de variables en una expresión son de tres tipos:

1. Ocurrencias de ligadura (binders)
2. Ocurrencias ligadas (bound occurrences)
3. Ocurrencias libres (free occurrences)

Las variables de ligadura son aquellas que están entre el λ y el punto. Por ejemplo, siendo E una expresión lambda:

(λ *x y z*. E) Las ocurrencias de ligadura son *x*,*y* y *z*.

La ligadura de ocurrencias de una variable está definida recursivamente sobre la estructura de las expresiones lambda, de esta manera:

1. En expresiones de la forma *V*, donde *V* es una variable, *V* es una ocurrencia libre.
2. En expresiones de la forma λ *V*. *E*, las ocurrencias son libres en *E* salvo aquellas de *V*. En este caso las *V* en *E* se dicen ligadas por el λ antes *V*.
3. En expresiones de la forma (*E E′*), las ocurrencias libres son aquellas ocurrencias de *E* y *E′*.

Expresiones lambda tales como λ *x*. (*x y*) no definen funciones porque las ocurrencias de *y* están libres. Si la expresión no tiene variables libres, se dice que es cerrada.

Como se permite la repetición del identificador de variables, cada ligadura tiene una zona de alcance asociada. Un ejemplo típico es: (λx.x(λx.x))x, donde el alcance de la ligadura más a la derecha afecta sólo a la *x* que tiene ahí, la situación de la otra ligadura es análoga, pero no incluye el alcance de la primera. Por último la *x* más a la derecha está libre. Por lo tanto, esa expresión puede reexpresarse así (λy.y(λz.z))x

5.4.3 α-conversión

La regla de alfa-conversión fue pensada para expresar la siguiente idea: los nombres de las variables ligadas no son importantes. Por ejemplo λ*x*.*x* y λ*y*.*y* son la misma función. Sin embargo, esta regla no es tan simple como parece a primera vista. Hay algunas restricciones que hay que cumplir antes de cambiar el nombre de una variable por otra. Por ejemplo, si reemplazamos *x* por *y* en λ*x*.λ*y*.*x*, obtenemos λ*y*.λ*y*.*y*, que claramente, no es la misma función. Este fenómemo se conoce como **captura de variables**.

La regla de alfa-conversión establece que si *V* y *W* son variables, *E* es una expresión lambda, y

E[*V* := *W*]

representa la expresión *E* con todas las ocurrencias libres de *V* en *E* reemplazadas con *W*, entonces

λ *V*. *E* == λ *W*. *E*[*V* := *W*]

si *W* no está libre en *E* y *W* no está ligada a un λ donde se haya reemplazado a V. Esta regla nos dice, por ejemplo, que λ *x*. (λ *x*. *x*) *x* es equivalente a λ *y*. (λ *x*. *x*) *y*.

En un ejemplo de otro tipo, se ve que

```
for (int i = 0; i < max; i++) { proc (i); }
```

es equivalente a

```
for (int j = 0; j < max; j++) { proc (j); }
```

5.4.4 β-reducción

La regla de beta reducción expresa la idea de la aplicación funcional. Enuncia que

$$((\lambda\ V.\ E)\ E') == E[V := E']$$

si todas las ocurrencias de E' están libres en $E[V := E']$.

Una expresión de la forma $((\lambda\ V.\ E)\ E')$ es llamada un **beta redex**. Una lambda expresión que no admite ninguna beta reducción se dice que está en su forma normal. No toda expresión lambda tiene forma normal, pero si existe, es única. Más aún, existe un algoritmo para computar la formal normal: la reducción de orden normal. La ejecución de este algoritmo termina si y sólo si la expresión lambda tiene forma normal. El teorema de Church-Rosser nos dice que dos expresiones reducen a una misma si y sólo si son equivalentes (salvo por el nombre de sus variables ligadas).

5.4.5 η-conversión

Es la tercera regla, eta conversión, que podría ser añadida a las dos anteriores para formar una nueva relación de equivalencia. La eta conversión expresa la idea de extensionalidad, que en este contexto implica que dos funciones son la misma si y solo si dan el mismo resultado para cualquier argumento. La eta conversión convierte entre $\lambda\ x.\ f\ x$ y f siempre que x no aparezca sola en f. Esto puede ser entendido como equivalente a la extensionalidad así:

Si f y g son extensionalmente equivalentes, es decir, si $f\ a == g\ a$ para cualquier expresión lambda a entonces, en particular tomando a como una variable x que no aparece sola en f ni en g, tenemos que $f\ x == g\ x$ y por tanto $\lambda\ x.\ f\ x == \lambda\ x.\ g\ x$, y así por eta conversión $f == g$. Entonces, si aceptamos la eta conversión como válida, resulta que la extensionalidad es válida.

Inversamente, si aceptamos la extensionalidad como válida entonces, dado que por beta reducción de todo y tenemos que $(\lambda\ x.\ f\ x)\ y == f\ y$, resulta que $\lambda\ x.\ f\ x == f$; es decir, descubrimos que la eta conversión es válida.

5.5 Programación funcional

El componente básico de los lenguajes funcionales es la noción de función y su estructura de control esencial la aplicación de una función. Entre los lenguajes funcionales se encuentran ISWIM, ML, LISP y todos sus derivados, como Scheme. Las características fundamentales de los lenguajes funcionales de programación son:

1. El valor de una expresión depende sólo de los valores de sus subexpresiones si las tiene. La programación funcional pura es una programación sin asignaciones. En realidad, la mayoría de los lenguajes funcionales son impuros, ya que permiten asignaciones. Sin embargo, su estilo de programación es diferente al de los lenguajes de programación imperativa.

2. Almacenamiento implícito. El programador no debe preocuparse en manejar el almacenamiento de datos. Una consecuencia de esto es que la implementación del lenguaje debe realizar una "recolección de basura" para recuperar la memoria que se ha usado y no se volverá a utilizar.

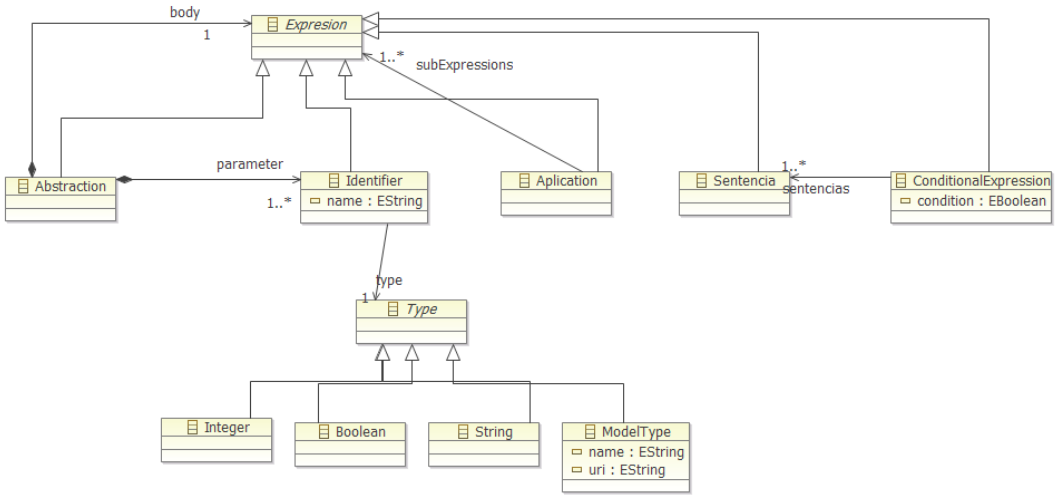
3. Las funciones representan el principal elemento del lenguaje. Una función puede ser el valor de una expresión, pasarse como argumento o colocarse en una estructura de datos. Esto permite potentes operaciones.

En términos generales, al desarrollar software en el paradigma de programación, en contraste con el paradigma de programación imperativo, se tiende a enfatizar más la evaluación de expresiones que la ejecución secuencial de comandos.

5.6 Metamodelo de Lambda Cálculo

Para realizar la transformación, fueron necesarios los metamodelos de entrada (QVT) y salida (Lambda Cálculo). El metamodelo de salida, es la base sobre la cual ATL trabaja utilizando las reglas definidas, para transformar un archivo en lenguaje QVT en el correspondiente en lenguaje Lambda Cálculo.

El metamodelo se realizó teniendo en cuenta los elementos que conforman un lenguaje funcional y en particular tomando como base la sintaxis del lenguaje Haskell [42].



Como se puede ver en la figura, el metamodelo contiene los elementos básicos de todo lenguaje funcional: Abstracción, Aplicación, Constantes y Variables. Para cumplir con las reglas de la transformación especificadas en el documento de la Dra. Giandini, y teniendo en cuenta la sintaxis de Haskell , incorporamos al modelos otros elementos que escapan al lenguaje funcional puro. Estos elementos son: las sentencias condicionales, las cuales permiten implementar sentencias if-then-else y while , definimos tipos de datos, los cuales nos permiten trabajar con un lenguaje tipado, y en particular, definimos el tipo ModelType, el cual es un tipo abstracto que nos permite darle una definición al tipo Model Type de QVT.

5.6.1 Implementación

El metamodelo fue construido desde eclipse con el editor de metamodelos de EMF. El editor permite dibujar el metamodelo, y a partir del mismo se genera el archivo ecore con el cual trabajará ATL para generar el modelo de salida.

5.7 Lenguaje Funcional Haskell

Debido a que Haskell es un lenguaje funcional puro, todos los cómputos vienen descritos a través de la evaluación de expresiones (términos sintácticos) para producir valores (entidades abstractas que son vistas como respuestas). Todo valor tiene asociado un tipo.

Ejemplos de expresiones son los valores atómicos tales como el entero 5, o el carácter 'a', o la función \x -> x+1, y los valores estructurados como la lista [1,2,3] y el par ('b',4).

Ya que las expresiones denotan valores, las expresiones de tipo son términos sintácticos que denotan tipos. Ejemplos de expresiones de tipo son los tipos atómicos Integer (enteros con precisión ilimitada), Char (caracteres), Integer->Integer (funciones que aplican Integer sobre Integer), así como los tipos estructurados [Integer] (lista homogénea de enteros) y (Char,Integer) (par formado por un carácter y un entero).

Todos los valores de Haskell son de primera categoría ("first-class") ---pueden ser argumentos o resultados de funciones, o pueden ser ubicados en estructuras de datos, etc. Por otro lado, los tipos de Haskell no son de primera categoría. En cierto sentido, los tipos describen valores, y la asociación de un valor con su tipo se llama un tipificado (typing). Utilizando los ejemplos anteriores, se pueden escribir tipos como los siguientes:

```
    5  :: Integer
   'a' :: Char
   inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

El sistema de tipificación estático de Haskell define formalmente la relación entre tipos y valores. Esta tipificación estática asegura que un programa Haskell está bien tipificado (type safe); es decir, que el programador no puede evaluar expresiones con tipos erróneos. Por ejemplo, no podemos sumar dos caracteres, ya que la expresión 'a'+'b' está mal tipificada. La ventaja principal de la tipificación estática es bien conocida: todos los errores de tipificado son detectados durante la compilación. No todos los errores son debidos al sistema de tipos; una expresión tal como 1/0 es tipificable pero su evaluación provoca un error en tiempo de ejecución. No obstante, el sistema de tipos puede encontrar errores durante la compilación, lo que proporciona al programador una ayuda para razonar sobre los programas, y también permite al compilador generar un código más eficiente (por ejemplo, no se requiere ninguna información de tipo o pruebas durante la ejecución).

El sistema de tipos también asegura que los tipos que el usuario proporciona para las funciones son correctos. De hecho, el sistema de tipos de Haskell es lo suficientemente potente como para describir cualquier tipo de función en cuyo caso diremos que el sistema de tipos infiere tipos correctos. No obstante, son aconsejables las oportunas declaraciones de tipos para las funciones, ya que el tipificado de funciones es una forma eficiente de documentar y ayudar al programador a detectar errores.

Capitulo 6

Semántica de un lenguaje de programación

6.1 Importancia de la definición de una semántica

La especificación formal de los lenguajes de programación tiene una singular importancia para la comunidad de investigadores de la Ciencia de la Computación. A través de la misma se expone con rigurosidad una serie de detalles que, para el programador común, son transparentes o carecen de significado. Una definición caracteriza la instrumentación de cualquier rasgo del lenguaje, de forma tal que pueda ser usada en la prueba de propiedades de programas como la corrección, terminación, eficiencia y equivalencias entre ellos. La definición de cualquier lenguaje debe reunir ciertas cualidades, como son:

- Debe ser total o completa, en el sentido de que los aspectos esenciales del lenguaje estén rigurosamente definidos.
- Debe ser natural, clara y legible.
- Debe emplear una notación adecuada, comprensible y factible de ser implementada.

A partir de la década del 60 esto ha sido tarea primordial de los científicos de esta ciencia. En cuanto a la definición de la sintaxis se cuenta con métodos de definición ampliamente aceptados (ej. La notación BNF), para los cuales existen algoritmos cuyas implementaciones son muy eficientes. En cuanto a la semántica, sin embargo, no es hasta la década del 70 que adquiere un impulso trascendente.

La formalización de la semántica de los lenguajes de programación ha sido estudiada y elaborada desde posiciones diferentes. Fundamentalmente se distinguen tres enfoques: el operacional, el axiomático y el denotacional.

6.1.1 Enfoque operacional

En el enfoque operacional, el más cercano a la construcción de compiladores, la semántica del lenguaje se define en términos de una máquina abstracta. Es decir, a cada instrucción del lenguaje se le asocia un conjunto de operaciones de máquina.

Las operaciones describen cambios en el estado de la máquina caracterizado por el conjunto de valores de las variables y en el acto de control. Mediante el control se define el flujo de ejecución del programa, incluyendo la terminación o interrupción de su trabajo. A partir del estado donde termina el programa se determina el resultado de su ejecución. En la medida de cuan satisfactoria sea la abstracción del hardware, será su valor como guía para la estructuración general e instrumentación de compiladores.

En la actualidad existen dos tendencias principales de la semántica operacional:

- La aproximación Máquina Abstracta. Se especifica una vía de implementación del lenguaje sobre alguna computadora (idealizada) de bajo nivel, o en su defecto la traducción a un lenguaje de bajo nivel para el cual se tiene una semántica operacional. La ventaja de este enfoque es su aproximación a la implementación. Su gran desventaja es que lleva implícito una gran cantidad de detalles irrelevantes.

- La Semántica Operacional Estructural. Fue introducida por Plotkin a principios de la década de los ochenta. Parte de la construcción de reglas de generación, para definir todas las relaciones relevantes. Estas reglas simbólicas trabajan directamente con la sintaxis del lenguaje. Las mismas, especifican las transiciones elementales de un programa, por inducción sobre su estructura. La Semántica Operacional Estructurada es un poco más abstracta y clara que la aproximación Máquina Abstracta.

6.1.2 Enfoque Axiomático

La semántica axiomática descansa en una teoría formal cuyas fórmulas son ciertas aserciones o propiedades de los operadores e instrucciones del lenguaje. Los axiomas y reglas de deducción se utilizan para la demostración de propiedades de programas, por lo que constituyen el basamento fundamental de la construcción de verificadores.

Un conjunto de axiomas, junto con una descripción formal del programa, pueden permitir la deducción del resultado de la ejecución del programa en cualquier ambiente dado, así como inferir las propiedades más generales, tales como la corrección, terminación y equivalencia entre programas.

Este enfoque, por estar orientado a las necesidades de demostración de propiedades de programas, no constituye un camino natural ni directo a algún método de implementación.

6.1.3 Enfoque denotacional

El enfoque denotacional, que constituye el eslabón de enlace entre los enfoques anteriores, opera asignándole un significado (entidad matemática) al programa. A través de esta entidad se describe la dependencia funcional entre el resultado de la ejecución de un programa y sus datos iniciales. En particular se utiliza fuertemente la teoría de conjuntos, donde los programas se expresan como funciones donde la máquina no está presente. Este enfoque, también llamado matemático o funcional, es libre de los detalles de implementación.

La semántica denotacional, a diferencia de otros métodos, parte de considerar la interpretación como un conjunto de funciones totales incluyendo funciones que están bien definidas sobre toda función admisible como argumento (una clase de funcionales) y que aun pueden aplicarse a ellas mismas.

Uno de los mayores retos, para los estudiosos del enfoque denotacional, es buscar variantes constructivas que permitan la implementación directa de prototipos de los lenguajes de programación.

6.2 Definición de la semántica utilizando un lenguaje formal

Para realizar la definición de la semántica de un lenguaje de programación imperativo, como es QVT Operacional, se optó por un lenguaje de programación funcional por las ventajas que este presenta:

- Permite definiciones simples.
- Facilita el estudio de aspectos computacionales.
- Su carácter formal facilita la demostración de propiedades.

Este tipo de lenguajes resulta útil en las siguientes aplicaciones:

- Compilación de lenguajes funcionales.
- Especificar semántica a lenguajes imperativos.
- Formalismo para definir otras teorías.

Capítulo 7

Generación de modelo: EMF y Xtext

Con el auge del desarrollo software dirigido por modelos aumenta la necesidad de crear nuevos lenguajes de modelado. Ello lleva a la aparición de una nueva disciplina, que es la ingeniería de lenguajes de modelado [23]. El proceso de ingeniería de un nuevo lenguaje de modelado está compuesto de diversas fases:

1. El primer paso es crear las reglas de construcción o sintaxis del lenguaje de modelado. Esto se realiza mediante la creación de un metamodelo, o modelo del lenguaje que describe usando conceptos similares a los de los diagramas de clases, la sintaxis abstracta o reglas para la construcción de modelos del lenguaje. Dicho metamodelo se construye usando un lenguaje de metamodelado.
Existen diversos lenguajes de metamodelado, tales como KM3 [24] o MOF [3], aunque Ecore está considerado como el estándar de facto dentro de la comunidad de modelado.
Normalmente no es posible especificar cualquier tipo de restricción entre los elementos de un lenguaje de modelado usando exclusivamente los elementos del lenguaje de metamodelado. Por ello, tales lenguajes de metamodelado se acompañan con un lenguaje de especificación de restricciones. OCL [25] es el lenguaje de restricciones más usado para desarrollar esta tarea.
2. Como se ha comentado en el punto anterior, un metamodelo establece la sintaxis abstracta del lenguaje de modelado, pero no especifica la sintaxis concreta o notación del lenguaje de modelado. Por tanto el siguiente paso en la ingeniería de un nuevo lenguaje de modelado, es la definición de una notación o sintaxis concreta para el lenguaje. Esta notación puede ser tanto textual como gráfica. Para la definición de notaciones gráficas, GMF es la herramienta más popular cuando se trabaja con Ecore.
3. Como último paso, se necesita definir la semántica del lenguaje de modelado. Existen un amplio rango de técnicas para desarrollar esta tarea, tales como: definir la semántica de cada elemento de manera informal; especificar la semántica de cada elemento usando un lenguaje formal, tales como máquinas de estado abstractas; o crear generadores que transformen los elementos de modelado en elementos de otro lenguaje con un significado bien definido.

7.1 Desarrollo de metamodelos con Ecore

EMF4 [26] es un marco de trabajo de Eclipse que unifica Java, XML y UML, permitiendo a los desarrolladores construir rápidamente aplicaciones robustas basadas en modelos simples.
El metamodelo usado para representar modelos en EMF se denomina Ecore. Ecore es en sí mismo, un metamodelo EMF, y así es su propio metamodelo, es decir, un metametamodelo.
Podemos observar en la figura 7.1.1 un ejemplo básico de Ecore en el que se ven sus partes más importantes. El metamodelo define grafos, estos están compuestos de un número indeterminado de nodos y de relaciones. Las relaciones tienen un origen y un destino, ambos referencian a un nodo, el cual tiene como atributos “nombre" (de tipo EString, una subclase de String), y “tipo" (el cual tomaría como valor uno de los literales que están definidos en el tipo especial enumerado definido por el usuario TipoNodo).

En la figura 7.1.2 podemos apreciar el núcleo del metamodelo Ecore con los elementos principales para definir un metamodelo. Esencialmente este modelo define cuatro tipos de objetos.

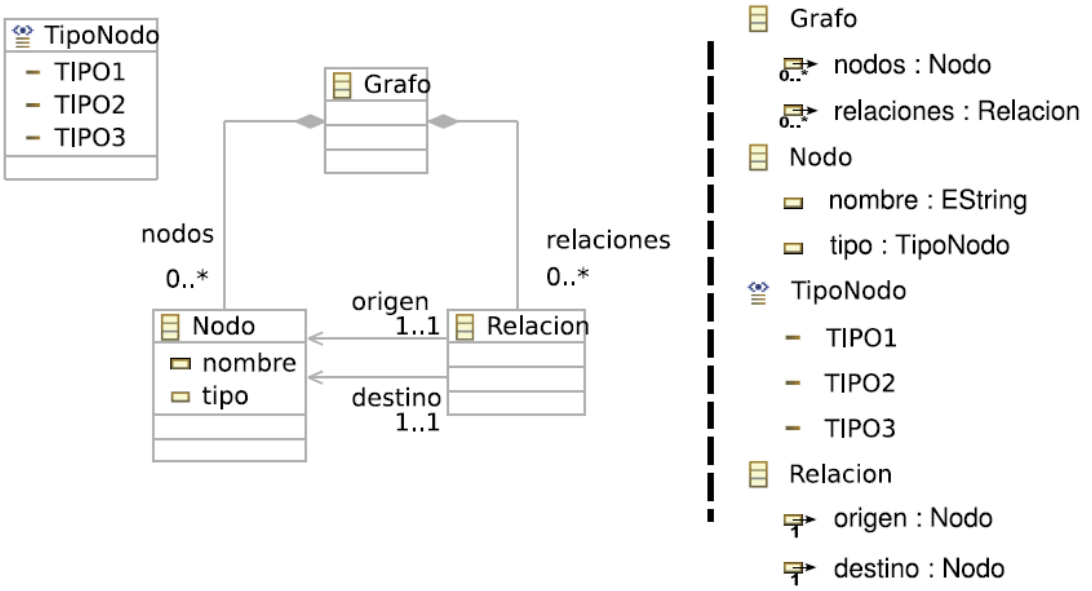


Figura 7.1.1: Ejemplo de metamodelo básico en Ecore

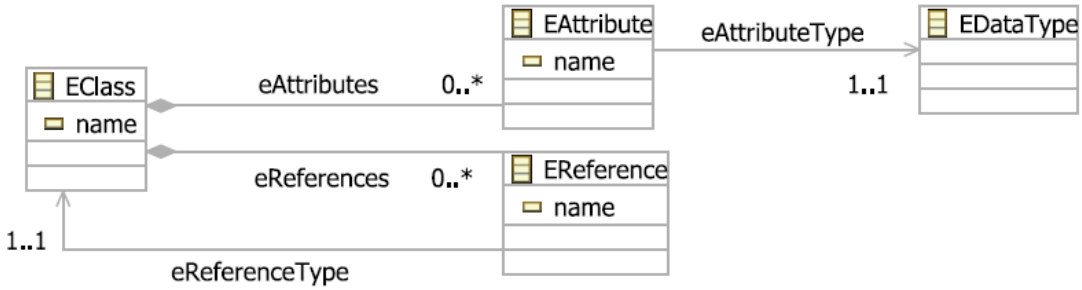


Figura 7.1.2: Subconjunto simplificado del metamodelo Ecore

1. EClass define clases concretas. Las clases están definidas por el nombre y pueden contener un número de atributos y referencias. Para dar soporte a la herencia, una clase puede referir a otras clases como supertipos.
2. EAttribute modela atributos, el componente de los datos de un objeto. Está identificado por un nombre, y tiene un tipo.
3. EDataType modela los tipos de los atributos, representando tipos de datos primitivos y objetos definidos en Java, pero no en EMF. También están identificados por un nombre.
4. EReference es utilizado como asociación entre clases; modela el final de una asociación. Al igual que con los atributos, las referencias están identificadas por un nombre y tienen un tipo. En cualquier caso, este tipo debe de ser EClass. Si la asociación es navegable en la otra dirección, debe de existir una referencia correspondiente. En una referencia se especifican los límites superiores e inferiores de su multiplicidad.

Finalmente, una referencia puede ser usada para representar una asociación fuerte, llamada “composición", la referencia específica si hay que realizar la semántica de composición.

7.2 Desarrollo de editores de modelado textuales con XTEXT

El marco de edición textual XText permite crear editores de texto para lenguajes basados en EMF. XText provee un lenguaje de definición de sintaxis basado en una gramática (EBNF). En este lenguaje, el ingeniero puede describir una notación textual dado un metamodelo Ecore. Para esta descripción, el marco puede generar automáticamente un editor XText. Los editores XText son editores de texto decorados mediante Eclipse. Esto significa que tienen las mismas facilidades que por ejemplo, un editor Java. Algunas de sus características son: realzado de sintaxis, asistencia a la auto-conclusión de código, navegación inteligente, información contextual o visionado de errores.
XText es en sí mismo un plugin eclipse, y cada editor XText creado sería otro plugin eclipse.
Para crear un editor XText es necesario crear primeramente un metamodelo que se adapte a las necesidades de nuestro problema. Luego se define la gramática, donde además se asignan los elementos de la gramática con los elementos del metamodelo. El resto del trabajo es automático y se generaría el código necesario para crear un editor textual para el problema. Este código puede ser modificado para adaptar ciertos comportamientos que no se pueden definir en la gramática.

ATL
ATL, es una herramienta que a partir de dos metamodelos, en este caso, el metamodelo de entrada QVT Operacional y metamodelo de salida Lambda Cálculo, una instancia del metamodelo de entrada y una transformación que indica como mapear cada elemento del dominio en el codominio, genera un modelo de salida. Dichos modelos son instancias de cada metamodelo que deben estar en formato XMI para ser manipulados por la herramienta. (XMI o XML Metadata Interchange es una especificación para el Intercambio de Diagramas la cual fue escrita para proveer una manera de compartir modelos UML entre diferentes herramientas de modelado).

La extracción de modelos requiere establecer un puente entre diferentes espacios tecnológicos, en particular entre la tecnología del DSDM (modelware) y las tecnologías empleadas para describir el texto de los archivos origen, normalmente gramáticas (grammarware) y esquemas XML. En un escenario de migración de plataformas, este puente es unidireccional y consiste en la creación de un analizador sintáctico que construye los modelos a partir de los artefactos fuente.

En esta instancia de nuestro trabajo, nos vimos en la necesidad de encontrar la forma de obtener de un archivo escrito en lenguaje QVT Operacional, el correspondiente en XMI. Nos encontramos con dos alternativas posibles, la generación en forma manual que provee la herramienta EMF, o una generación automática a través de una herramienta externa.

7.3 Generación manual de instancias de los metamodelos con herramienta EMF

EMF provee una manera simple de instanciar los metamodelos desde la herramienta. Funciona de la siguiente manera, una vez que se tienen los metamodelos, se posiciona sobre la clase raíz del metamodelo del que se quiere obtener una instancia, y desde el menú contextual con la opción *Create Dynamic Instance* se crea el archivo con la instancia de la clase seleccionada como se observa en la figura 7.3.1. Es decir, se crea el objeto raíz y a partir del mismo se crean los hijos y se editan las propiedades.

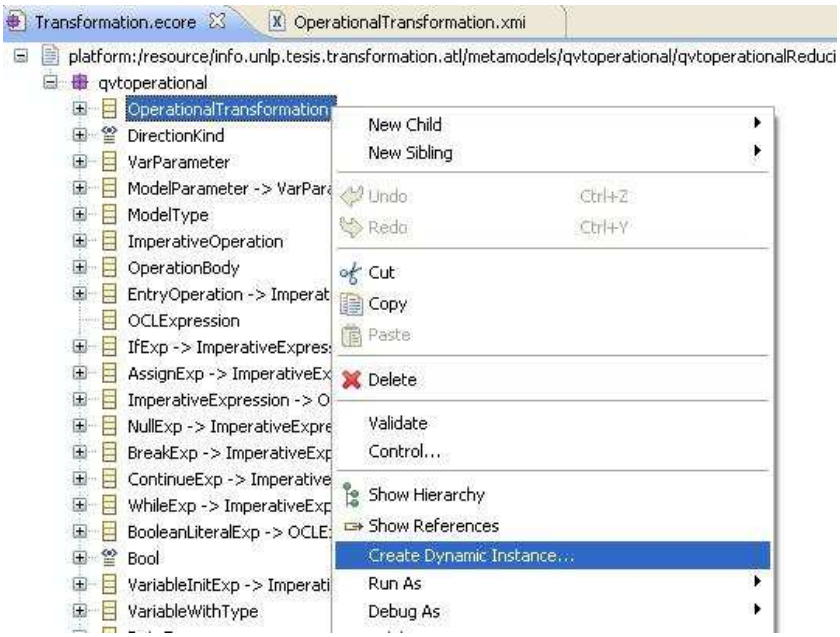


Figura 7.3.1: Instanciación dinámica del metamodelo

Para crear los hijos, es decir, continuar con la instanciación del metamodel se debe abrir el archivo generado con el formulario de edición de EMF. Para ello seleccionamos el archivo generado, que generalmente tienen extensión .xmi y con el menú contextual seleccionamos *Open With → Generic EMF Form Editor* como puede observarse en la figura 7.3.2a y 7.3.2b.

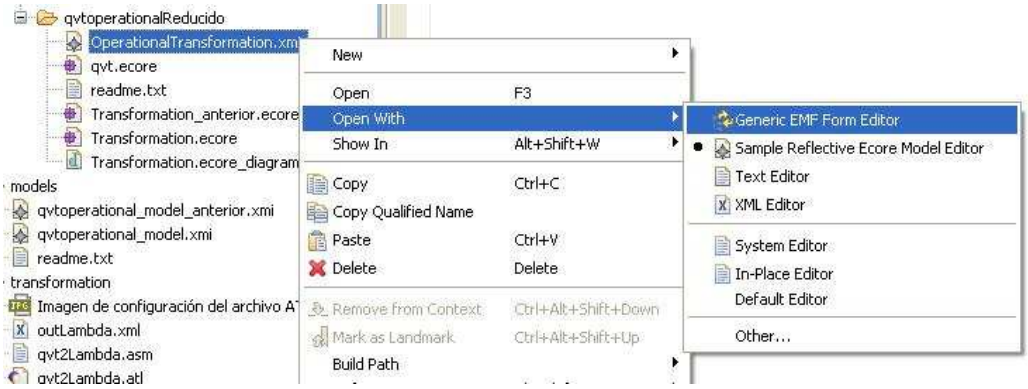


Figura 7.3.2a: Continuando la configuración de la instancia.

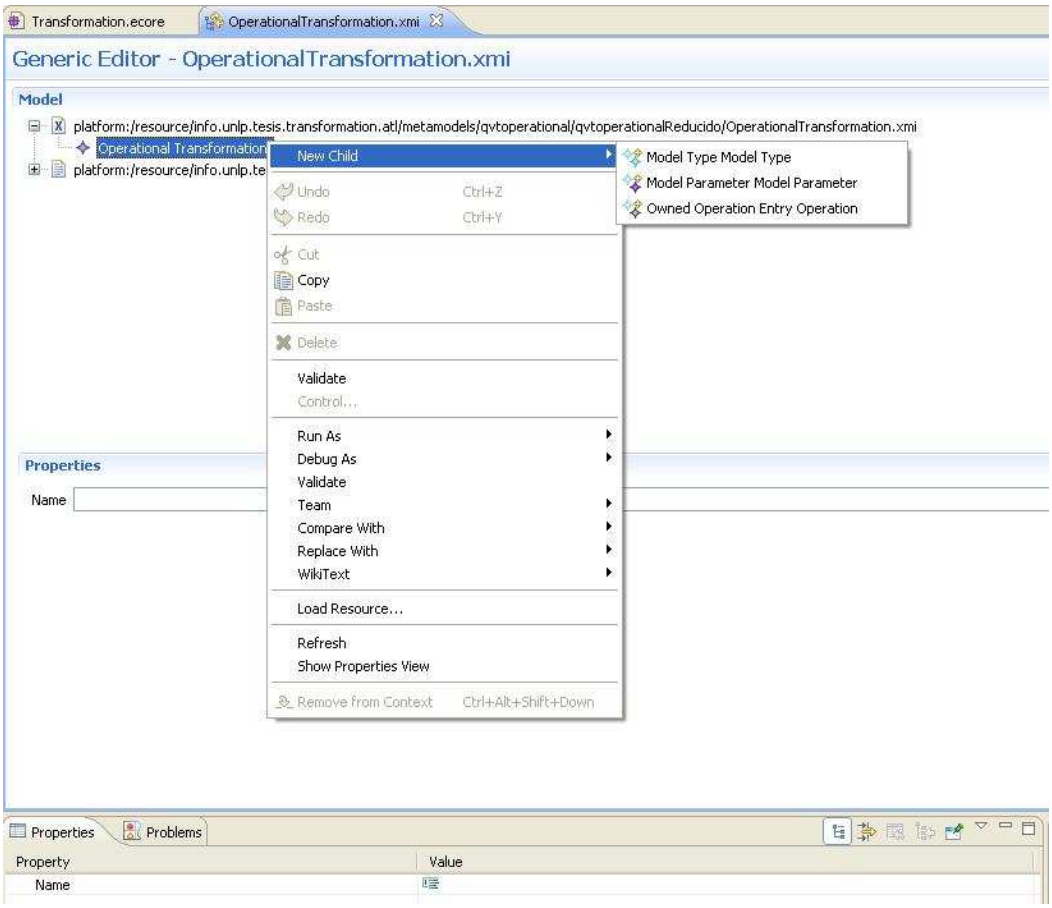


Figura 7.3.2b: Continuando la configuración de la instancia.

7.3.1 Ventajas:

Esta manera de extraer modelos, presenta la ventaja de que solo se necesita del plugin EMF para obtener los modelos.
Dado que la obtención de modelos es sólo un paso previo a la transformación, es decir no es el punto central del trabajo, es una alternativa viable por la practicidad que desde ese punto de vista tiene, en especial cuando el modelo no es muy complejo.
Además el hecho de no interactuar con otra herramienta descarta el problema de incompatibilidad que se podría llegar a generar por el hecho de realizar una parte del proceso con una herramienta y la otra parte con otra diferente.

7.3.2 Desventajas:

La desventaja que presenta obtener los metamodelos de esta manera, se centra en el hecho de que la instancia se va construyendo de a pasos, primero el objeto raíz, luego los hijos y las propiedades y que además el usuario tiene que configurarlo, es decir, no se realiza de una manera automática. Esta metodología se puede tornar muy fastidiosa si el modelo a construir es relativamente grande.
Por otra parte una vez salvado el modelo, no es posible modificarlo, excepto que se edite el archivo xmi que se generó lo cual no es una tarea sencilla de realizar.

7.3.3 Conclusión:

En el caso particular de la generación de modelos para la trasformación QVT-Lambda, concluimos que para la muestra de nuestro trabajo era imprescindible obtener los

modelos en una manera más directa y automática, por lo cual se descartó el uso de EMF para la generación de las instancias de los modelos.

7.4 Generación automática de las instancias de los metamodelos mediante herramienta XTEXT

Xtext es un Framework que permite crear un lenguaje específico del dominio con todo lo que ello implica, es decir, compilador e intérpretes y editores. Pero también pudimos investigar que mediante sus componentes podíamos obtener las instancias de un metamodelo.

Un proyecto xtext esta formado por tres proyectos:

- info.unlp.tesis.transformation: Define el EBNF para el lenguaje de dominio específico. Establece las reglas sintácticas y semánticas requeridas para la creación del modelo de una transformación de QVT.
- info.unlp.tesis.transformation.***generator***: Crea el modelo conforme con el lenguaje, en sintaxis XMI.
- info.unlp.tesis.transformation.***ui:*** Como su nombre lo indica, define el editor para el nuevo lenguaje, adicionando características como syntax coloring y auto completion.

7.4.1 Gramática

Como mencionamos anteriormente es necesario contar con una gramática EBNF que describa el metamodelo y a partir de la gramática y de un archivo que respete la misma, se generará el correspondiente xmi. En nuestro caso tenemos una gramática que describe QVT de forma reducida ya que no tomamos todo el metamodelo del mismo y un archivo de transformación de qvt. La gramática definida para QVT es la siguiente:

```
grammar info.unlp.tesis.Transformation with
org.eclipse.xtext.common.Terminals

generate qvtoperational "http://qvtoperational"

OperationalTransformation:
  (modelType+=ModelType)+
  'transformation' name=ID '(' (modelParameter+=ModelParameter) (','
(modelParameter+=ModelParameter))* ')' (';')?
  (ownedOperation=EntryOperation)?;

enum DirectionKind : in = 'in' | inout = 'inout' | out = 'out' ;

VarParameter:
  kind=DirectionKind|ModelParameter;

ModelParameter:
  kind=DirectionKind name=ID ':' modelType=[ModelType];

ModelType:
```

```
  'modelType' name=ID 'strict uses' uri=STRING ';';

ImperativeOperation:
  body=OperationBody|EntryOperation;

OperationBody:
  content+=OCLExpression (';' (content+=OCLExpression))*;

EntryOperation:
  name='main' '{' body=OperationBody '}';

OCLExpression:
  BooleanLiteralExp|ImperativeExpression|StringLiteralExp|
  IntegerLiteralExp|OperationCallExp;

IfExp:
  'if' '(' condition=OCLExpression ')' 'then' '{'
thenExpression=OCLExpression '}' 'else'
  '{' elseExpression=OCLExpression '}';

AssignExp:
  left=Variable ':=' value=OCLExpression;

ImperativeExpression:

BreakExp|ContinueExp|WhileExp|VariableInitExp|Variable|IfExp|AssignExp
;

BreakExp:
  break ='break';

ContinueExp:
 continue = 'continue';

WhileExp:
  'while' '(' condition=OCLExpression (';' condition=OCLExpression)*
')' '{' body+=OCLExpression (';'(body+=OCLExpression))* '}';

BooleanLiteralExp:
  booleanSymbol=Bool;

enum Bool : trueValue = 'true' | falseValue = 'false' ;

VariableInitExp:
  'var' referredVariable=VariableWithType;

VariableWithType:
  name=ID ':' type=DataType (':=' value=OCLExpression)?;

DataType:
  String='string'|Integer='integer'|Boolean='boolean';

Variable:
  name=ID;

StringLiteralExp:
  stringSymbol=STRING;

IntegerLiteralExp:
  integerSymbol=INT;
```

```
OperationCallExp:
  referresOperation=Operation argument=OCLExpression |'('argument1=
  OCLExpression referresOperation=Operation
argument2=OCLExpression')';

enum Operation:
      and = 'and' | or = 'or' | not = 'not' |menor = '<' | mayor = '>'
| igual = '=' | distinto = '<>' | suma = '+' | resta = '-' ;
```

Esta gramática tiene dos propósitos:

- Definir la sintaxis
- Contener la información necesaria para que el analizador pueda crear el modelo en el proceso de parseo.

En Xtext cada regla que define la gramática tiene un nombre único y al igual que las clases en Java debe reflejar la ubicación de la clase dentro de una ruta. En el caso del archivo de la definición de la gramática esta ubicado en `info.unlp.tesis.Transformation` por lo tanto el nombre de la gramática es `info.unlp.tesis.Transformation`.

La segunda parte de la sentencia define una librería la cual es usada en la definición de la gramática y en el caso particular de esta gramática `org.eclipse.xtext.common.Terminals` define las reglas de los terminales más comunes, sales como ID, STRING e INT.

La segunda sentencia declara un paquete EMF Ecore, que será derivado de la gramática:

```
generate qvtoperational "http://qvtoperational"
```

Los paquetes de Ecore son un conjunto de clases las cuales son usadas para representar el modelo en memoria del archivo de texto.

El resto de las sentencias definen las reglas del parser. La regla principal o la que deriva al resto de las reglas la denominamos `OperationalTransformation`

```
OperationalTransformation:
  (modelType+=ModelType)+
  'transformation' name=ID '(' (modelParameter+=ModelParameter)
(',' (modelParameter+=ModelParameter))* ')' (';')?
  (ownedOperation=EntryOperation)?;
```

Esta regla define que una OperationalTransformation consiste de uno o más ModelType que serán definidos después. Como podemos observar se puede definir la cantidad de entidades que tendrá el modelo definiendo una cardinalidad '+'. En Xtext existen cuatro tipos de cardinalidades permitidas:

Sin indicar nada	Exactamente una
?	Cero o una
*	Cero o más
+	Una o más

Para conectar diferentes tipos de objetos se tiene que asignar los elementos retornados de una regla a alguna propiedad de la regla que se esta evaluando. Esto se hace mediante el uso de las asignaciones de los llamados.

```
modelType+=ModelType
```

La asignación con el operador "+=" agrega cada objeto retornado por la regla ModelType a modelType. Los diferentes tipos de operadores para la asignación son los siguientes:

lista+=…	Aplicables a atributos lista y corresponde a *getList().add(...)*
Feature=…	Corresponde a *setFeature()*
condición? =	Corresponde a *setCondition(true)*

Continuando con la sintaxis vemos que cada OperationlTransformation contiene la palabra clave `'transformation'` y seguido el nombre de la transformación y a continuación paréntesis de apertura los cuales contienen la regla ModelParameter y al final el paréntesis de cierre.

```
'transformation' name=ID '(' (modelParameter+=ModelParameter)
(',' (modelParameter+=ModelParameter))* ')'
```

Las palabras claves son simplemente declaradas como literales dentro de las reglas. El nombre usa la regla ID desde la librería `org.eclipse.xtext.common.Terminals` y con el operador de asignación "=".

Dentro de una gramática en Xtext no solo podemos definir un tipo de datos sino dos tipos de datos mediante una regla de tipo *Feature*. Como ejemplo tenemos la siguiente regla:

```
OCLExpression:
  BooleanLiteralExp|ImperativeExpression|StringLiteralExp|
  IntegerLiteralExp|OperationCallExp;
```

En la cual diferentes tipos de datos heredan del mismo tipo, es decir, tienen un tipo en común. De esta manera tenemos que una OCLExpression puede ser una BooleanLiteralExp o una ImperativeExpression o StringLiteralExp o IntegerLiteralExp o OperationCallExp. La regla OCLExpression delega en cualquiera de ellos usando el operador "|" y de esa manera los subtipos son referenciados por una simple referencia cruzada.

El siguiente paso es definir la sintaxis de una Feature

```
Feature:
      name=ID ':' type=TypeRef;
```

En muchas plataformas existen diferencias entre la noción de un atributo simple a la de una referencia.
En las bases de datos relacionales, por ejemplo, una tabla contiene valores y atributos directamente pero las referencias son modeladas con el significado de foreign key. En las tecnologías de persistencia relacional de objetos como JPA o Hibernate las referencias pueden ser ciclos de vida semánticos adicionales en los cuales se define a que objetos referenciar en ciertas circunstancias.

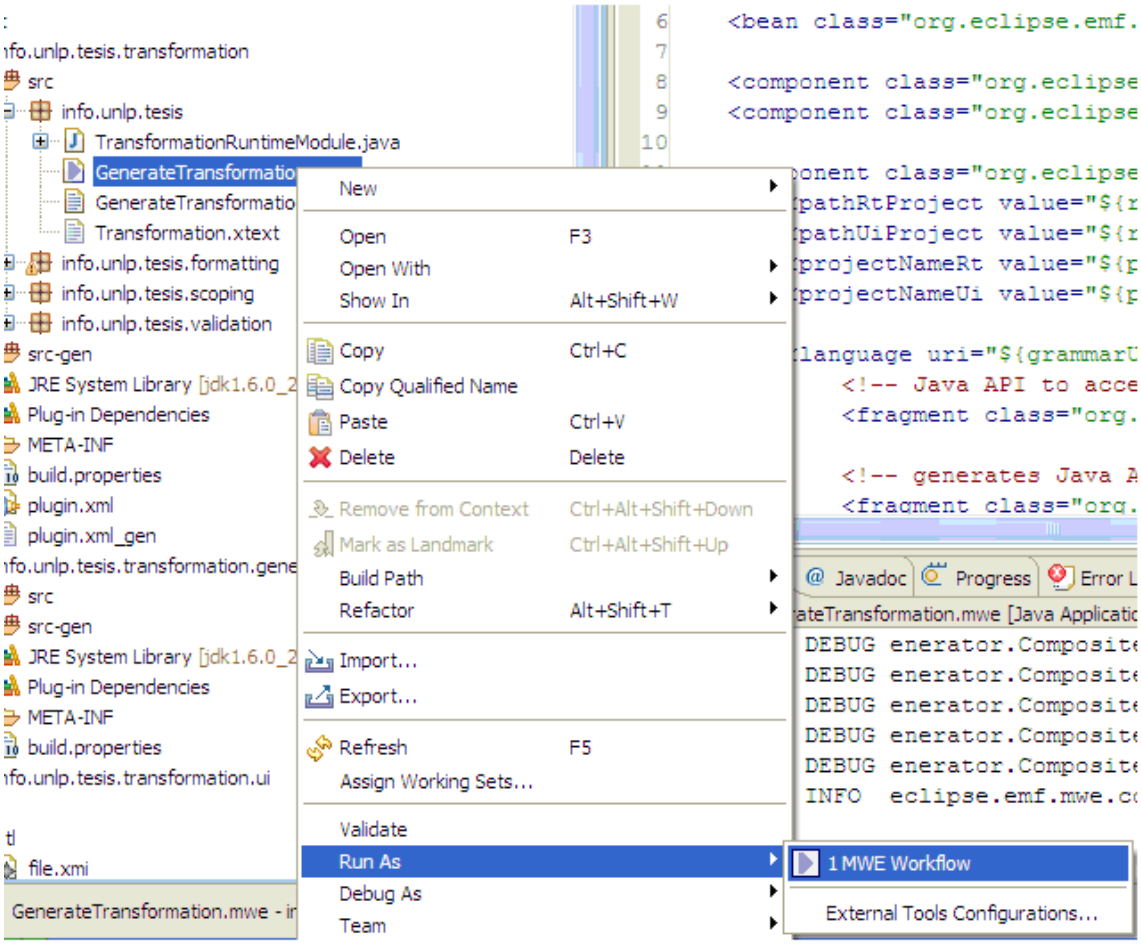
Los tipos referenciados tiene una propiedad adicional para definir la multiplicidad (mucho a uno). Veremos la regla de parseo definida de la siguiente manera:

```
TypeRef :
referenced=[Type] (multi?='*')?;
```

La presencia del postfijo "*" indica un flag booleano "true" que indica que es una referencia a un valor múltiple. Este es el propósito del operador de asignación "?=".

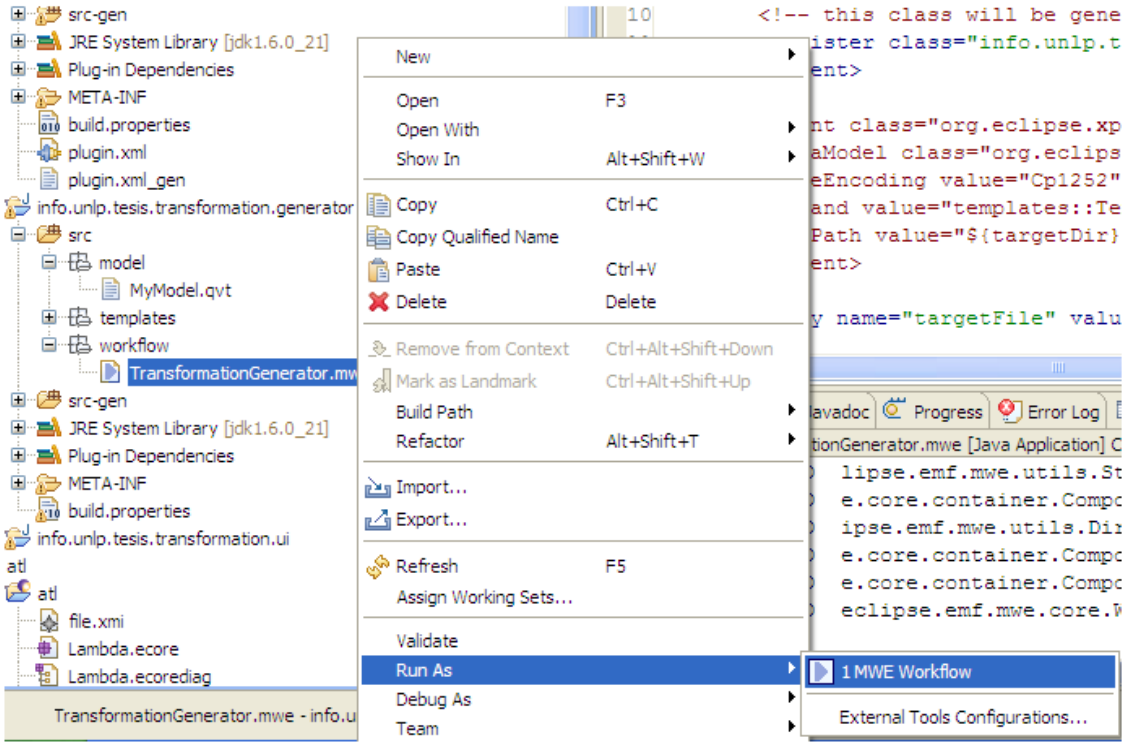
7.4.2 Proceso - Etapas

Xtext trabaja en dos etapas, en la primera toma como entrada una gramática EBNF que describe el metamodelo y genera un archivo ecore para ser usado luego en la instanciación junto con clases necesarias para el editor como así también para leer el modelo.



1. Generación del metamodelo (ecore) a partir de una gramática EBNF

En una segunda etapa, tomando como entrada el ecore generado y el modelo escrito utilizando la gramática, genera el modelo en formato xmi para ser utilizado luego como elemento de entrada para la transformación ejecutada con ATL.



2. Generación de modelo a partir del metamodelo generado en 1.

7.4.3 Proceso – Workflows

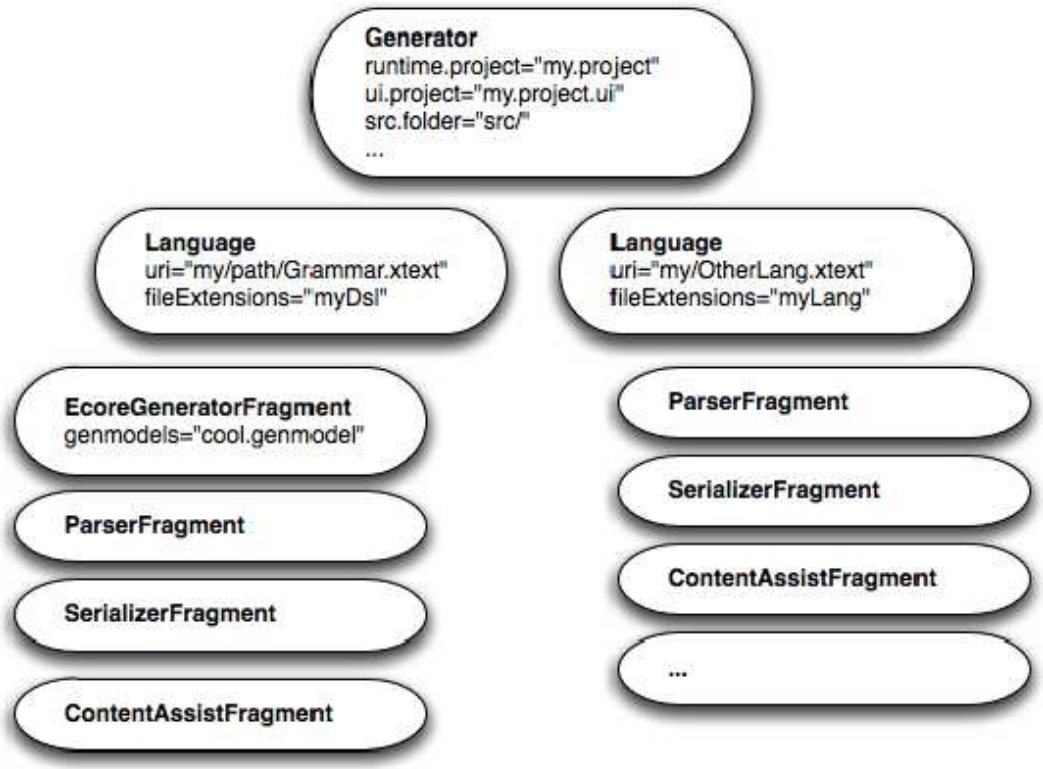
XText no solo proporciona gran cantidad de implementaciones genéricas para la infraestructura de los lenguajes, sino también genera código para crear algunos de sus componentes. Los componentes generados son, por ejemplo, el analizador, el serializador, el modelo Ecore y un par de clases de base conveniente para ayudar a los contenidos, etc

El proceso de generación parte del proyecto EMFT (Eclipse Modeling Framework Technology) [43], el cual permite agrupar nuevas tecnologías que permiten extender o complementar a EMF. Dentro de este grupo se encuentra MWE (Modeling Workflow Engine)

7.4.3.1 Introducción a MWE

Es un motor generador declarativo y configurable. Los usuarios pueden describir composiciones arbitrarias de objeto por medio de una sintaxis sencilla, concisa, que permite declarar instancias de objetos, valores de los atributos y referencias. Un caso de uso es la definición de workflows. Este workflow consiste generalmente en un número de *componentes* que interactúan entre sí y que pueden ser configurables a partir de un archivo de propiedades. Existen componentes para leer los recursos de EMF, para realizar operaciones (transformaciones) sobre ellos y modificarlos o para generar cualquier número de otros artefactos.
Los workflows suelen ser ejecutados en JVM. Sin embargo no hay restricciones para para proporcionar los componentes que generan múltiples hilos o procesos nuevos.

El componente principal es el *Generator* el cual contiene información de las carpetas de código, información del proyecto e información del lenguaje. Por cada lenguaje se define la URI al archivo que contiene la definición de la gramática y los distintos fragmentos necesarios.



Cada fragmento trabaja sobre el modelo EMF pasado como un parámetro de entrada. Un fragmento puede generar código en una ubicación específica y contribuir para generar artefactos compartidos. En la siguiente tabla se listan los fragmentos estándares más comunes:

Class	Generated Artifacts	Related Documentation
EcoreGeneratorFragment	EMF code for generated models	Model inference
XtextAntlrGeneratorFragment	ANTLR grammar, parser, lexer and related services	
GrammarAccessFragment	Access to the grammar	
ResourceFactoryFragment	EMF resource factory	
ParseTreeConstructorFragment	Model-to-text serialization	Serialization
JavaScopingFragment	Java-based scoping	Java-based scoping
JavaValidatorFragment	Java-based model validation	Java-based validation
CheckFragment	Xpand/Check-based model validation	Check-based validation
FormatterFragment	Code formatter	Declarative formatter
LabelProviderFragment	Label provider	Label provider
OutlineNodeAdapterFactoryFragment	Outline view configuration	Outline
TransformerFragment	Outline view configuration	Outline
JavaBasedContentAssistFragment	Java-based content assist	Content assist
XtextAntlrUiGeneratorFragment	Content assist helper based on ANTLR	Content assist
SimpleProjectWizardFragment	New project wizard	Project wizard

7.4.3.2 Configuración del primer workflow

Como ya se ha explicado el uso de MWE de EMFT con el fin de crear una instancia, configurar y ejecutar esta estructura de componentes. A continuación vemos la configuración del generador de XText escrito en MWE:

```
<workflow>
	<property file="info/unlp/tesis/GenerateTransformation.properties"/>

	<property name="runtimeProject" value="../${projectName}"/>

	<bean class="org.eclipse.emf.mwe.utils.StandaloneSetup"
platformUri="${runtimeProject}/.."/>

	<component class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
directory="${runtimeProject}/src-gen"/>
	<component class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
directory="${runtimeProject}.ui/src-gen"/>

	<component class="org.eclipse.xtext.generator.Generator">
		<pathRtProject value="${runtimeProject}"/>
		<pathUiProject value="${runtimeProject}.ui"/>
		<projectNameRt value="${projectName}"/>
		<projectNameUi value="${projectName}.ui"/>

		<language uri="${grammarURI}"
fileExtensions="${file.extensions}">
			<fragment
class="org.eclipse.xtext.generator.grammarAccess.GrammarAccessFragment"/>

			<fragment
class="org.eclipse.xtext.generator.ecore.EcoreGeneratorFragment"/>

			<fragment
class="org.eclipse.xtext.generator.parseTreeConstructor.ParseTreeConstructorFr
agment"/>
```

```
                    <fragment
class="org.eclipse.xtext.generator.resourceFactory.ResourceFactoryFragment"
                            fileExtensions="${file.extensions}"/>

                    <fragment
class="org.eclipse.xtext.generator.AntlrDelegatingFragment" />

            </language>
      </component>
</workflow>
```

Como se puede observar el elemento raíz es `workflow` el cual acepta beans y componentes. El elemento `property` es el primer concepto en el lenguaje de configuración de MWE y básicamente actua como preprocesador, el cual reemplaza todas las ocurrencia de `${propertyName}` con el correspondiente valor. La declaración de las propiedades pueden ser definidas en el mismo workflow (`<property name="runtimeProject" value="info.unlp.tesis.transformation"/>`) u obtener su valor de un archivo de propiedades el cual es importado en la configuración del MWE como es el caso de nuestra implementación en el cual primero se importa el archivo de propiedades y luego de ello se declara la propiedad de la cual su valor es obtenido del archivo de propiedades.

```
<property file="info/unlp/tesis/GenerateTransformation.properties"/>

<property name="runtimeProject" value="../${projectName}"/>
```

Archivo de propiedades:

```
grammarURI=classpath:/info/unlp/tesis/Transformation.xtext
file.extensions=qvt
projectName=info.unlp.tesis.transformation
```

El método *Workflow.setBean()* proporciona un medio para aplicar efectos secundarios , el cual desafortunadamente es requerido para el proyecto. En este caso se deja la configuración por defecto.

Siguiendo al elemento `bean` hay tres elementos `component`. El método *Workflow.addComponent()* crea instancias de *IWorkflowComponent*, el cual es el concepto principal del modelado de workflow en MWE.
Los dos primeros componentes como sus nombres lo indican permiten limpiar o vaciar las carpetas de los proyectos que contienen código generado. Y el tercer componente es el *Generator* de Xtext; instancia de IWorkflowComponent y puede ser usado dentro de los workflows de MWE, componente principal para la generación y el cual contiene fragmentos.

Cuando se realiza la creación del MWE por defecto la herramienta agrega dentro de este componente todos los fragmentos estándares. Para nuestra implementación fueron solamente necesarios los siguientes:

- GrammarAccessFragment: Necesario para acceder a la gramática y requerido para otros fragmentos.
- EcoreGeneratorFragment: Crea y almacena modelos de Ecore. Necesario para que nos generara el ecore para la transformación en ATL.

- ParseTreeConstructorFragment: Componente de serialización.
- ResourceFactoryFragment: Factory de recursos estándares para usar con EMF.
- AntlrDelefatingFragment: Necesario para el generador AntlLR

7.4.3.3 Configuración del segundo workflow

En este segundo workflow necesitamos que lea nuestro modelo y la gramática y que nos genere el correspondiente en formato xmi. Para ello la configuración es la siguiente:

```
<workflow>
      <property name="modelFile"
value="classpath:/model/MyModel.qvt"/>
      <property name="targetDir" value="src-gen"/>

      <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup"
platformUri=".."/>

      <component class="org.eclipse.emf.mwe.utils.DirectoryCleaner"
directory="${targetDir}"/>

      <component class="org.eclipse.xtext.MweReader"
uri="${modelFile}">
            <register
class="info.unlp.tesis.TransformationStandaloneSetup"/>
      </component>

      <property name="targetFile" value="src-
gen/qvtoperational_model.xmi"/>

    <component class="org.eclipse.emf.mwe.utils.Writer">
        <cloneSlotContents value="true"/>
        <uri value="${targetFile}"/>
        <modelSlot value="model" />
    </component>

</workflow>
```

De esta manera tenemos definido con la propiedad `modelFile` la ubicación de nuestro modelo, es decir, la transformación en qvt y con la propiedad `targetFile` la ubicación y nombre del archivo al cual queremos generar. Luego contamos con los componentes necesarios:

- MweReader: componente que nos permite leer el modelo
- Writer: componente para generar el modelo en xmi

7.4.4 Ventajas:

La extracción de modelos se realiza en solo dos pasos, como muestran las dos figuras anteriores, se escribe en un archivo el modelo, y se ejecutan los pasos necesarios para la generación.

Es más intuitivo para trabajar debido a que el modelo, es escrito en el lenguaje con el que se desea trabajar (en nuestro caso, escribimos un archivo en lenguaje QVT), y la herramienta se encarga de la conversión a xmi, por lo cual para el desarrollador es transparente el uso del formato xmi.

En el caso de un error en el modelo, sólo se debe corregir el mismo, y ejecutar nuevamente el segundo workflow, cabe destacar que encontrar un error en un archivo escrito en un lenguaje de programación es más sencillo que encontrarlo en un archivo en formato xmi.

7.4.5 Desventajas:

La principal desventaja se encuentra en el hecho de trabajar con una herramienta externa a ATL. Es decir, en conocer con cierta profundidad el funcionamiento de XText, como por ejemplo las reglas que existen para escribir la sintaxis EBNF.

Es una herramienta que hoy en día se está desarrollando, por lo cual se actualiza constantemente, lo cual implica una revisión permanente de las versiones que salen al mercado.

7.4.6 Conclusión:

Se optó por el uso de Xtext, dado que nos permitió trabajar de una manera más cómoda por la flexibilidad que presenta.

Además nos pareció la manera más prolija para presentar nuestro trabajo.

Capítulo 8

Transformación: herramientas

8.1 Análisis de herramientas para transformación de modelos

ATL es un framework para administrar transformaciones basadas en modelos. Es un lenguaje mixto, es decir, es una mezcla de construcciones imperativas y declarativas. Un modelo fuente se transforma en un modelo destino mediante una definición de transformación escrita en ATL, que también es un modelo. Los modelos fuente, destino y la definición de la transformación, responden a sus metamodelos respectivos y, a su vez, todos los metamodelos se ajustan a MOF. La parte básica de ATL incluye todos los componentes requeridos para configurar y ejecutar transformaciones, en particular, el EMF (Eclipse Modelling framework) y MDR (Meta Data repository) que permiten, respectivamente, manejar modelos definidos de acuerdo a la semántica Ecore y MOF.

Para que ATL pueda ejecutar las transformaciones necesita los metamodelos fuente y destinos en un formato tal que permita dar soporte al metamodelado, dicho formato es el *ecore* (metamodelo de EMF).

Para obtener los modelos en formato de ecore, se analizaron diferentes herramientas con el fin de generar a partir de un archivo de texto, los metamodelos de QVT y Lambda cálculo.

8.1.1 **AMMA (Atlas Model Management Architecture)**

Es una plataforma de gestión de modelos diseñada y desarrollada por INRIA. Esta plataforma es un plugin de Eclipse. Está basada en el estándar de la OMG, MOF, y utiliza Ecore como lenguaje de modelado. MOF propone el modelo MOF como lenguaje abstracto para definir todo tipo de modelos y "arquitectura de nivel cuatro" que está enfocada a MDA. AMMA proporciona ATL (Atlas Transformation Language), un lenguaje declarativo e imperativo (lenguaje híbrido) para la transformación de modelos. A pesar de que no está basado en el estándar QVT, se utiliza actualmente en numerosos grupos y proyectos de investigación, e incluso hay proyectos que han utilizado ATL, como el proyecto ModelWare.

8.1.2 **XMI XSLT**

Es un marco de trabajo tecnológico que no está basado en QVT y proporciona soporte para el manejo de modelos. El uso de esta herramienta permite la transformación de modelos definiendo los modelos como documentos XMI y utilizando XSLT para transformar de un documento a otro. El problema que tiene este marco de trabajo es que la transformación está basada en sintaxis y no en semántica.

8.1.3 **Borland Together**

Es un producto que integra Java IDE, que originalmente nace de JBuilder con una herramienta de modelado UML. Tiene soporte para Eclipse y los diagramas pueden crearse de forma importada. Genera soporte para UML, Java 6, C++ y CORBA.

8.1.4 **SmartQVT**

Es una implementación completa en java del lenguaje operacional QVT. La herramienta se distribuye como un complemento a Eclipse de libre distribución y se caracteriza por estar implementada de forma imperativa.
Por ser la herramienta que presenta mayor nivel de madurez en la actualidad, se investigó con mayor profundidad la posibilidad de usar esta herramienta como generadora de los metamodelos que luego serían el punto de partida de las transformaciones ejecutadas con ATL.

8.1.4.1 **Problemas encontrados**

SmartQVT, es un plugin de eclipse que permite, a partir de un archivo escrito en lenguaje QVT, generar los metamodelos y ejecutar las transformaciones escritas entre los mismos.
Por la madurez y la posibilidad de generar los metamodelos que presenta la herramienta, se decidió realizar una investigación más profunda acompañada de pruebas con ejemplos concretos para ver el funcionamiento de la misma.
La herramienta trabaja en dos etapas, en una primera etapa, genera a partir de un archivo escrito en lenguaje QVT (en el cual se escribió únicamente la estructura de ambos metamodelos) un árbol sintáctico que representa dichos modelos. En una segunda etapa, genera a partir del árbol sintáctico antes generado, las clases java correspondientes a cada elemento del metamodelo. El hecho de que esta generación automática se realizara en dos etapas, nos permitió pensar que podríamos usar la herramienta solo en su primera etapa, es decir generar solo el árbol sintáctico que luego urtilizaríamos en ATL. El inconveniente es que el archivo obtenido, es generado en un formato interno de la herramienta, es decir, un formato que sólo es útil dentro de la herramienta. Investigamos diferentes maneras de serializar dicho archivo para convertirlo a un xml o un ecore, pero no tuvimos éxito en las pruebas y por tal motivo se descartó el uso de SmartQVT.

8.1.5 **openArchitectureWare**

Es un generador de entornos de trabajo modular implementado en Java. Esta herramienta ofrece soporte a la trasformación de un modelo a otro, de texto a modelo, y de modelo a texto. Está basado en la plataforma Eclipse y soporta modelos basados en EMF pero puede trabajar con otros modelos.

8.1.6 **MediniQVT**

Es una herramienta de transformación de modelos. MediniQVT está implementado para realizar las transformaciones que se instauraron desde la OMG, es decir, siempre basando sus transformaciones de modelos en QVT. La aplicación incluye herramientas para el desarrollo de transformaciones, así como un depurador gráfico y un editor. Estos componentes son de libre distribución pero sólo para uso no comercial. MediniQVT efectúa las transformaciones QVT expresadas en la sintaxis concreta del lenguaje de relaciones de QVT (QVT relations). MediniQVT está integrado en Eclipse. La aplicación posee un editor con asistente de código que permite especificar las relaciones modelo origen modelo destino, ya sea el elemento o sus propiedades. MediniQVT proporciona también un depurador para efectuar las transformaciones y de esta forma evitar errores en el modelo destino. Un hecho a destacar es que permite las transformaciones bidireccionales.

8.1.7 **GME (Generic Modelling Environment)**

Es un conjunto de herramientas configurables para crear modelos de diseño específico. Para crear uno de dichos modelos es importante tener en cuenta que el peso de la correcta definición del dominio recae sobre el metamodelo, es decir, en la configuración del metamodelo se debe modelar el dominio correcto de la aplicación. El metamodelo de entrada debe contener toda la sintaxis, semántica e información del dominio; conceptos que serán utilizados para construir modelos, ver qué relaciones pueden existir entre ellos, cómo pueden estar organizados y cómo los verá el ingeniero de modelos. El metamodelo también debe contener las reglas que gobiernan la construcción de modelos.

El lenguaje de metamodelado está basado en los diagramas de clases UML y en las restricciones OCL. Lo metamodelos que especifican el modelo se utilizan para generar automáticamente un entorno de dominio específico. Este entorno se utiliza después para construir modelos específicos de dominio que se almacenan en una base de datos de modelos o en formato XML. Estos modelos se utilizan para generar automáticamente las aplicaciones.

GME tiene una arquitectura modular y extensible. GME es fácilmente extensible; los componentes definidos fuera del ámbito de la herramienta y utilizados para la extensión de la misma pueden escribirse en cualquier lenguaje que soporte COM (C++, Visual Basic, C#, Python etc.). GME tiene muchas características avanzadas y dispone de un elemento ya integrado que impone todas las restricciones de dominio durante la construcción de los modelos, denominado “director”. GME soporta múltiples aspectos del modelado como son la combinación de lenguajes de modelado y el soporte de bibliotecas de modelos para su reutilización.

8.1.8 **GMF (Graphical Modelling Framework)**

Proporciona un generador de modelos. GMF está embebido en Eclipse, plataforma de desarrollo que permite la ejecución de los editores gráficos generados por GMF a partir de un modelo. Eclipse está construido y trabaja en Java. GMF está basado en EMF (Eclipse Modelling Framework) y en GEF (Graphical Eclipse Framework), ambos proporcionados por la plataforma Eclipse. En GMF se ha adoptado el término “toolsmith” para referirse a los desarrolladores que utilizan la herramienta para construir extensiones que posteriormente se integran como parte de la misma. Estas extensiones reciben el nombre de plugins. Otro término, “practitioner”, se utiliza para referirse a aquellos que utilizan los plugins como medio para el desarrollo. Durante la utilización de GMF para la generación de un modelo, la descripción del modelo se realiza una sola vez, al comienzo. Una vez realizada la especificación del dominio, la herramienta se encarga de interpretar las correspondencias con el modelo durante el resto de proceso de generación del editor.

Capítulo 9

ATL en detalle

9.1 Transformación de modelo.

En el ámbito de una ingeniería basada en modelos una transformación de modelos pretende ofrecer una manera de producir modelos "***target***" desde modelos "***source***". Para este propósito se debe permitir a los desarrolladores definir los elementos del modelo origen que se correspondan y naveguen para inicializar los elementos de un modelo destino.

Una transformación de modelos es la creación automática de un modelo destino desde un modelo origen.

Formalmente una transformación de modelo simple tiene que definir la manera de generar un modelo Mb, conforme a un *metamodelo* MMb desde un modelo Ma conforme a un metamodelo MMa. Y a su vez cada metamodelo debe ajustarse a su *metametamodelo* correspondiente.

La siguiente figura 9.1.1 resume todos los procesos de transformación de modelos. Un modelo *Ma*, conforme a su metamodelo *MMa*, se transformo en un modelo Mb conforme a un metamodelo *MMb*. A su vez la transformación esta definida por un modelo *Mt* el cual es conforme a un metamodelo *MMt*. Este último metamodelo junto con los metamodelos *MMa* y *MMb* son conformes al metametamodelo *MMM* (como MOF o Ecore)

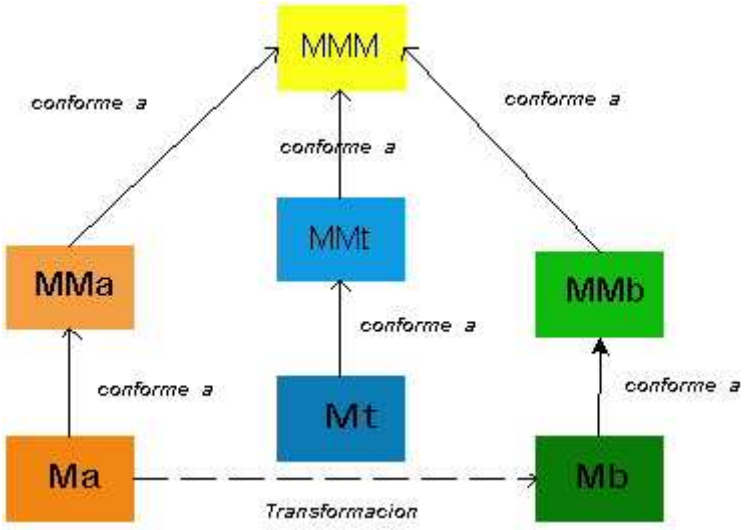


Figura 9.1.1: Transformación de modelos en ATL

ATL es un lenguaje de transformación de modelos que permite especificar como uno o más modelos destinos pueden ser creados a partir de un conjunto de modelos orígenes. En otras palabras, ATL introduce un conjunto de conceptos que permite la transformación de modelos.

La figura 9.1.2 presenta un ejemplo de transformación (Autor2Persona) que permite generar el modelo Persona conforme al metamodelo MMPersona, desde un modelo Autor conforme a un metamodelo MMAutor. A través de una transformación ATL conforme a un metamodelo ATL. Los tres metamodelos son expresados usando la semántica del metametamodelo Ecore.

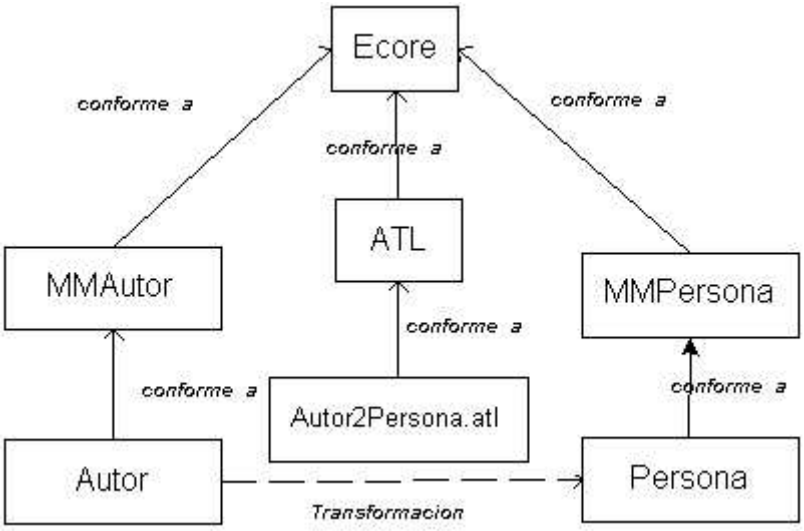


Figura 9.1.2: Transformación Autor2Persona en ATL

9.2 Visión General de ATL

El lenguaje ATL ofrece a los desarrolladores diseñar diferentes tipos de unidades de ATL. Una unidad de ATL, cualquiera sea el tipo esta definido en distintos archivos ATL. Los archivos ATL son caracterizados por la extensión *.atl*.

Como una respuesta a la OMG MOF/QVT y RFP, ATL puso mayor énfasis a las transformaciones de modelo a modelo. Tales operaciones sobre los modelos pueden estar especificadas en los denominados *modules* ATL. Además de los modelos, el lenguaje ATL permite crear programas que obtienen un valor simple de un modelo. Esas unidades son llamadas *queries*. Estos valores simples son los denominados valores primitivos, tales como un string o un integer.

Finalmente, el lenguaje ofrece la posibilidad de desarrollar *libraries* independientes que pueden ser importadas desde diferentes tipos de unidades ATL, incluyendo librerías propias de ATL. Estas librerías permiten una manera de poder refactorizar el código que es usado en múltiples unidades ATL. Cabe aclarar que los tres tipos de unidades de ATL comparten la misma extensión. La diferencia entre ellas está detallada en la siguiente sub-sección.

Esta sección explica cual es el uso de cada tipo de unidad y provee una introducción al contenido de las diferentes unidades.

9.2.1 ATL module

Un modulo ATL corresponde a una transformación modelo a modelo. Este tipo de unidad de ATL permitir especificar la manera de producir un conjunto de modelos destinos desde un conjunto de modelos orígenes. Ambos modelos deben tener como tipo asociado su respectivo metamodelo. Además un modulo ATL acepta un numero fijo de modelos como entrada y retorna un numero fijo de modelos de salida. Como consecuencia, un modulo ATL no puede generar un numero desconocido de modelos destinos.

La sección 9.2.1.1 detalla la estructura de un modulo ATL. La sección 9.2.1.2 presenta los dos modos de ejecución de los módulos. Finalmente, las semánticas de ejecución de un modulo de ATL son presentadas en la sección 9.2.1.3.

9.2.1.1 Estructura de un modulo ATL

Un modulo esta compuesto por los siguientes elementos:

- La sección cabecera que define algunos atributos que son relativos al modulo de transformación.
- Una sección opcional de librerías que permite importar algunas librerías existentes de ATL.
- Un conjunto de helpers que pueden ser vistos en ATL como un equivalente en Java a métodos.
- Un conjunto de reglas que define la manera que los modelos de salida son generados desde un modelo de origen.

Los helpers y las reglas no pertenecen a sectores específicos en una transformación ATL. Pueden ser declaradas en cualquier orden con respecto a ciertas condiciones.
El resto de los elementos que se mencionaron serán detallados en las siguientes subsecciones.

9.2.1.1.1 *Sección Cabecera*

La sección cabecera define el nombre del modulo de transformación y el nombre de las variables correspondientes a los modelos origen y destino. También se indica el modo de ejecución del modulo. La sintaxis de la sección se define como sigue:

```
module nombre_modelo;
create modelo_output [from|refines] modelo_input;
```

A continuación de la palabra clave **`module`** se indica el nombre del modulo. Notar que el nombre del archivo ATL contiene el mismo nombre que el modulo.
La declaración del modelo destino es indicado a continuación de la palabra clave **`create`**, mientras que los modelos orígenes son introducidos luego de la palabra clave **`from`** (en el caso de un modo normal) o de la palabra clave **`refines`** (en el caso de una transformación refinada).

La declaración de un modelo ya sea de entrada o de salida deben respetar el esquema *`nombre_modelo:nombre_metamodelo`*. De esta manera se puede declarar más de un

modelo de entrada o un modelo de salida separando las declaraciones por coma. Los nombres que se usan en la declaración deben ser usados para identificarlos. Como consecuencia, cada nombre de modelo declarado tiene que ser único en el conjunto de modelos declarados.

El siguiente es un ejemplo que representa la cabecera de un archivo Book2Publication.atl que representa la transformación de un Libro en una Publicacion [40].

```
module Book2Publication;
create OUT : Publication from IN : Book;
```

9.2.1.1.2 *Sección Librerías*
La sección import permite declarar las librerías ATL que serán importadas para realizar la transformación. La declaración de una librería en ATL se realiza de la siguiente manera:

```
uses nombre_libreria;
```

Por ejemplo para importar una librería de strings, se debe escribir:

```
uses strings;
```

Notar que es posible definir distintos usos de librerías usando sucesivamente instrucciones **uses**.

9.2.1.1.3 *Sección Helpers*
Los helpers de ATL pueden ser vistos como los métodos en Java. A través de ellos es posible definir la refactorización de código que puede ser invocado desde diferentes puntos de una transformación.

Un helper es definido por los siguientes elementos:

- Un nombre (el cual corresponde al nombre del método)
- Un tipo de contexto. El tipo de contexto define el contexto en el cual este atributo es definido (de la misma manera un método es definido en el contexto de una clase)
- Un tipo de valor de retorno. Notar que en ATL cada helper debe tener un tipo de valor de retorno.
- Una expresión ATL que representa el código del helper.
- Un conjunto opcional de parámetros, los cuales son identificados por el par (nombre del parámetro, tipo del parámetro).

A modo de ejemplo, tenemos un helper que retorna el máximo entre dos valores que son pasados como parámetros. La declaración de tal helper debe ser la siguiente:

```
helper context Integer def : max(x : Integer) : Integer = ...;
```

También es posible declarar un helper que no tiene parámetros como el siguiente:

```
helper context Integer def : double() : Integer = self * 2;
```

En algunos casos puede ser necesario declarar un helper sin importar el contexto. Esto no es posible en ATL pero el lenguaje permite declarar los helpers con el contexto por default (el cual corresponde al modulo ATL). Esto es posible omitiendo la parte del contexto en la definición del contexto.

```
helper def : max(x1 : Integer, x2 : Integer) : Integer = ...;
```

Como se puede observar varios helpers tienen el mismo nombre dentro de la misma transformación. Esto es posible ya que tienen distinta signatura lo que lo hacen distinguibles por el procesador ATL.

El lenguaje ATL también hace posible definir atributos. Un atributo es un tipo específico de helper el cual no tiene parámetros y que tampoco define un contexto.

En el resto del documento, el término atributo será específicamente utilizado para referirse a este tipo de helper, considerando que el término genérico de helper hará referencia a un helper funcional. Así, el atributo del helper `double` definido anteriormente será declarado como sigue:

```
helper context Integer def : double : Integer = self * 2;
```

La declaración de un helper sin parámetros y la de un atributo parecen ser equivalentes. Son equivalentes desde el punto de vista funcional pero existe una diferencia significantes entre ellos cuando consideramos la ejecución semántica. De hecho, comparado con el resultado de un helper funcional que se calcula cada vez que el helper se llama, el valor de retorno de un atributo ATL se calcula sólo una vez cuando el valor es requerido para la primera vez. Como consecuencia, declarando un atributo es mas eficiente que definiendo un helper que debe ser ejecutado tantas veces como es llamado.

Notar que los atributos ATL que se definen en el contexto del módulo ATL son inicializados en el orden en que han sido declarados en el archivo ATL. Esto implica que el orden de declaración de este tipo de atributo es de cierta importancia: un atributo definido en el contexto del módulo ATL tiene que ser declarado en el módulo ATL tras otro atributo que depende de su inicialización. Un orden equivocado en la declaración de los atributos en el ATL módulo planteará un error durante la fase de inicialización del programa ATL.

9.2.1.1.4 *Sección reglas*

En ATL existen dos tipos diferentes de regla que corresponden a dos modos de programación diferentes proporcionado por ATL: las reglas de macheo (programación declarativa) y las reglas que son llamadas explícitamente (programación imperativa).

Reglas Declarativas. Las reglas declarativas son el corazón de una transformación declarativa y que permiten especificar:

1) Por cada tipo de modelo origen los elementos que deben ser generados
2) La manera en que los elementos generados deben ser inicializados

Una regla declarativa es identificada por su nombre. Coincide con un determinado tipo de elemento del modelo origen, y genera uno o más tipos de elementos del modelo destino. La regla especifica la manera de generar los elementos del modelo destino y como deben ser inicializados con cada macheo del elemento del modelo origen.

Una regla declarativa es declarada utilizando la palabra clave *rule*. Esta compuesta por dos secciones obligatorias (los patrones de origen y de destino) y dos opcionales (variables locales e imperativas). En la definición, la sección de variables es introducida por la palabra clave *using*. Esto permite que se localice, se inicialice y se declare un numero de variables locales que solo son visibles en el alcance de la regla.

El patrón origen de una regla declarativa es definido después de la palabra clave *from*. El patrón origen está formado por:

- Un conjunto etiquetado de elementos de los metamodelos origen
- Una guarda que es una expresión booleana a modo de filtro.

Se producirá un matching si se encuentran elementos en los modelos origen que sean de los tipos especificados en el patrón y satisfagan la guarda.

El patrón destino de una regla declarativa es definido después de la palabra clave *to*. El patrón destino está compuesto de:

- Elementos etiquetados del metamodelo destino
- Para cada elemento una inicialización de sus propiedades

Para cada coincidencia se aplica el patrón destino: Se crean los elementos destino, se inicializan sus propiedades: primero, evaluando el valor de las expresiones; después, asignando dicho valor a la propiedad indicada.

Finalmente, la sección opcional imperativa la cual es introducida por la palabra clave *do*. Permite especificar código imperativo que debe ser ejecutado después de la inicialización de los elementos destino generados por la regla.

Reglas Imperativas. Las reglas imperativas pueden ser vistas como un tipo particular de los helpers: ellos deben ser llamados explícitamente y pueden aceptar parámetros. Sin embargo, opuestos a ellos las reglas imperativas pueden generar elementos del modelo destino como las reglas declarativas. Una regla imperativa tiene que ser llamada explícitamente desde la sección imperativa, de cualquiera tipo de regla, ya sea declarativa o cualquiera otra.

Como una regla declarativa, una regla imperativa es declarada utilizando la palabra clave *rule*. Como las reglas declarativas, una regla imperativa puede incluir una sección opcional de declaración de variables. Sin embargo, ya que no tienen que coincidir elementos del modelo origen, una regla imperativa no incluye un patrón origen. Además, su patrón destino, lo que hace posible generar elementos del modelo destino, también es opcional. Notar que, desde la regla imperativa no coincide con cualquier elemento del modelo origen, la inicialización de los elementos del metamodelo que son generados por el patrón destino tiene que estar basada en una combinación de variables locales, los parámetros y atributos del módulo. El patrón destino de una regla imperativa

se define de la misma forma que el patrón destino de una regla coincidente. También es introducido por la palabra clave *to*.

9.2.1.2 Modos de ejecución del módulo

El motor de ejecución de ATL define dos modos diferentes de ejecución del modulo. Con el valor por defecto modo de ejecución, el desarrollador ha de especificar explícitamente la manera de que elementos del modelo destino deben ser generados a partir de elementos del modelo origen. En este ámbito, el diseño de una transformación que apunta a copiar su modelo origen con sólo unas pocas modificaciones puede resultar muy agotador. Diseñar esta transformación en el modo de ejecución por defecto por lo tanto requiere del desarrollador para especificar las reglas que generará elementos del modelo modificado, pero también todas las reglas que sólo debe copiar sin modificar nada en los elementos del modelo origen.

El modo de ejecución refinado ha sido diseñado para este tipo de situación: permite a los desarrolladores sólo especificar las modificaciones que tiene que realizarse entre la transformación origen y los modelos destino.

Estos dos modos de ejecución son descriptos en las siguientes subsecciones.

9.2.1.2.1 Modo normal de ejecución

Este modo es el modo por default de la transformación ATL Esta asociada con la palabra clave *from* en la sección cabecera del modulo.

El desarrollador tiene que especificar cual de las dos tipos de reglas hace el camino a transformar cada elemento del modelo destino. Este modo de ejecución satisface a la mayoría de las transformaciones donde los modelos destino difieren del modelo origen.

9.2.1.2.1 Modo refinado de ejecución

El modo refinado de ejecución ha sido introducido para facilitar la programación de transformaciones similares entre modelos origen y destino. Con el modo refinado, los desarrolladores puedan concentrarse en el código ATL dedicada a la generación de elementos del modelo destino, es decir a la modificación. Otros elementos del modelo (por ejemplo, aquellos que permanecen sin cambios entre el modelo origen y el destino) están implícitamente copiados del modelo fuente al modelo origen por el motor ATL.

El modo refinado está asociado con la palabra clave *refines* en la sección de cabecera del módulo. La granularidad del modo se define en el nivel del elemento del modelo. Esto significa que el desarrollador tendrá que especificar cómo generar un elemento del modelo tan pronto como la transformación modifica una de sus características (ya sea un atributo o una referencia). Por otra parte, no es necesario el desarrollador para especificar el código ATL que corresponde a la copia del los elementos del modelo

cambiados. Esta característica puede dar como resultado importante un ahorro de código ATL, que, al final, hace que la programación sea simple y fácil.

El modo refinado sólo puede utilizarse para transformar un solo modelo origen en un único modelo destino. Ambos modelos origen y destino deben ajustarse al mismo metamodelo.

Notar que, debido a las semánticas de ejecución del modo refinado, algunas precauciones específicas todavía tienen que ser adoptadas por los desarrolladores. De hecho, con implementaciones actuales del motor, que se transformó en un elemento del modelo origen, un elemento del modelo destino tiene que coincidir con una de las siguientes condiciones:

- Se transformó en una regla explícitamente señalados por el desarrollador.
- Se refiere (directa o indirectamente) a un elemento del modelo origen transformado.

Esto significa que el elemento del modelo origen no debe ser copiado dentro del elemento del modelo destino correspondiente si:

- Ningún elemento del modelo destino es generado por la transformación;
- No explícitamente, el elemento del modelo destino transformado refiere, directa o indirectamente, al elemento del modelo origen.

En consecuencia, puede ser útil, cuando se diseña un módulo ATL en modo refinado, especificar otras normas explícitas a fin de asegurarse de que todos los elementos del modelo origen se transforman en su correspondiente elemento del modelo destino.

9.2.2 ATL query

Un ATL query consiste en una transformación de un tipo primitivo a un modelo. Las consultas ATL pueden ser vistas como una operación que computa valores primitivos de un conjunto de modelos orígenes. El uso más común es para la generación de un texto desde un conjunto de modelos orígenes. De todas maneras, las consultas no están limitadas al procesamiento de valores como strings sino que también pueden retornar un número o un valor boolean.

Las siguientes subsecciones describen la estructura y la semántica de ejecución de las consultas en ATL

9.2.2.1 Estructura de una consulta en ATL

Después de la sección opcional de importación de librerías, puede definirse la instanciación de una consulta ATL. Esta instanciación es introducida por la palabra clave *query* seguida de un nombre y especifica la manera en que el resultado es obtenido, es decir la expresión ATL para su procesamiento.

```
query nombre_consulta = expresión_atl;
```

Junto a la instanciación, un ATL query puede incluir un número de definiciones de helpers o atributo. Notar que, aunque una consulta ATL no es estrictamente un módulo, este define su propio tipo de contexto default. Por tanto, es posible, para los desarrolladores, declarar helpers y atributos definidos en el contexto del módulo y en el alcance de una consulta ATL.

9.2.2.1 Semántica de ejecución de las consultas

Como un modulo de ATL, la ejecución de una consulta en ATL esta organizada en sucesivas fases.

- La primer fase es la de inicialización que corresponde a la fase de inicialización del modulo ATL y es dedicada a la inicialización de los atributos que son definidos en el contexto del modulo ATL.
- La segunda fase es la de cálculo. Durante esta fase, se calcula el valor que retornara la consulta ejecutando el código declarativo de la consulta. Notar que los helpers que pueden ser definidos dentro de la consulta y pueden ser llamados en ambas fases, en la de inicialización y en la de cálculo.

9.2.3 ATL library

El último tipo de unidad de ATL son las librerías. Desarrollar una librería permite definir un conjunto de helpers que pueden ser llamados desde diferentes unidades ATLs.

Como los demás tipos de unidades de ATL, una librería puede ser incluida como una sección opcional dentro del modulo. Excepto que la sección import como las librerías,define un número de helpers que se distribuirán en las unidades ATL que importara la librería.

Comparado con un módulo ATL, no existe ningún elemento del módulo por defecto para las librerías. Como consecuencia, es imposible, en las librerías, declarar helpers que se definen en el contexto del módulo. Esto significa que todos los helpers definidos dentro de una librería deben ser explícitamente asociados con un determinado contexto.

Comparado con módulos y queries, una librería ATL no puede ejecutarse independientemente. Esto significa que una librería no está asociada con cualquier fase de inicialización en tiempo de ejecución. Debido a esta falta de fase de inicialización, atributos, helpers no pueden ser definidos dentro de una librería ATL.

9.3 Ejemplo de transformación ATL: Book → Publication

9.3.1 Descripción

Este ejemplo describe una transformación simple. En el metamodelo Book la clase Book contiene una lista ordenada de Chapters. Estos capítulos contienen la información

del número de páginas del capitulo. El metamodelo Publication es simple; la clase Publication contiene el titulo y el número total de páginas.

La transformación consiste en crear para cada Libro una Publicación cuyo atributo nbPages sea la suma de las páginas de los capítulos del libro. Es decir para todos los libros se debe visitar todos los capítulos para calcular el total de páginas.

9.3.2 Metamodelos

El metamodelo origen **Book** como puede verse en la figura 9.3.2.1 consiste en una clase Book la cual contiene un conjunto de Chapters. Cada Book tiene un titulo y cada Chapter un titulo. La instancia de Chapter contiene información del número de páginas.

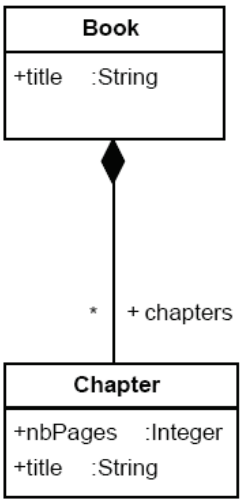


Figura 9.3.2.1 Book

El metamodelo destino **Publication** como puede verse en la figura 9.3.1.1.2 consiste en una clase Publication la cual contiene el titulo y el número de paginas

Figura 9.3.2.2 Publication

9.3.3 Especificación de reglas

La definición de las reglas son las siguientes:

- Por cada instancia de libro se debe crear una instancia de publicación. Los atributos de la publicación deben ser los siguientes:
  - ✓ El titulo de la publicación debe ser el titulo del libro.

- ✓ El número total de páginas de la publicación debe ser la suma de las páginas de los capítulos del libro.

9.3.4 Implementación del ejemplo Book → Publication

La implementación de la transformación consiste en una regla Book2Publication. Esta regla usa un helper en el cual todas las páginas de todos los Chapters de un Book son recorridas y sumadas.

```
module Book2Publication;
create OUT : Publication from IN : Book;


helper context Book!Book def : getSumPages() : Integer =
            self.chapters->collect(f|f.nbPages).sum()
;


rule Book2Publication {
      from
            b : Book!Book
      to
            out : Publication!Publication (
                  title <- b.title,
                  nbPages <- b.getSumPages()
            )
}
```

9.4 Lenguaje ATL

Esta sección esta dedicada a la descripción del lenguaje ATL. Como fue introducido en la sección anterior, el lenguaje permite definir tres tipos de unidades: modulo, query y librerías. Acorde a su tipo, estos diferentes tipos de unidades pueden formar una combinación de helpers, atributos y reglas declarativas o imperativas. Esta sección apunta a detallar la sintaxis de los diferentes elementos de ATL. Para este propósito el lenguaje ATL esta basado en la regla de OMG OCL [5] para los tipos de datos y las expresiones declarativas.

Existen algunas pocas diferencias entre la definición de OCL y la implementación de ATL. Estas serán especificadas y remarcadas en esta sección.

9.4.1 Sistema de tipos

El sistema de tipos de ATL es muy cerrado pero no tanto como el definido por OCL. La figura 9.4.1.1 provee una visión general de la estructura del sistema de tipos considerado en ATL. Los diferentes tipos de datos presentados en el esquema representan instancias de la clase *OclType*.

La clase raíz es la clase abstracta *OclAny* de la cual los demás tipos la extienden. ATL considera seis tipos principales de tipos de datos: los tipos primitivos, tipos de datos colecciones, tipo tupla, tipo map, tipo enumeración y el tipo elemento de un modelo.
La clase OclType puede ser considerada como la definición de los tipos. Los diferentes elementos que aparecen en la figura representan instancias de tipos que están definidos por OCL (excepto el mapa y el tipo de dato modulo de ATL) e implementados por el motor ATL.

Los tipos primitivos de OCL corresponden a los tipos de datos básicos para un lenguaje (el string, el boolean y los tipos numéricos). El tipo de dato colecciones introducido por OCL provee a ATL con diferentes semánticas para el manejo de los elementos de la colección. Los tipos de datos adicionales incluidos son las enumeraciones, la tupla, el tipo de datos mapa y el tipo de datos elemento de un modelo. Este último corresponde a un tipo de entidad que puede ser declarado dentro de la transformación ATL. Por ultimo, el tipo de dato modulo de ATL el cual es específico del lenguaje ATL y esta asociado con la ejecución de las unidades de ATL (ya sean módulos o consultas).

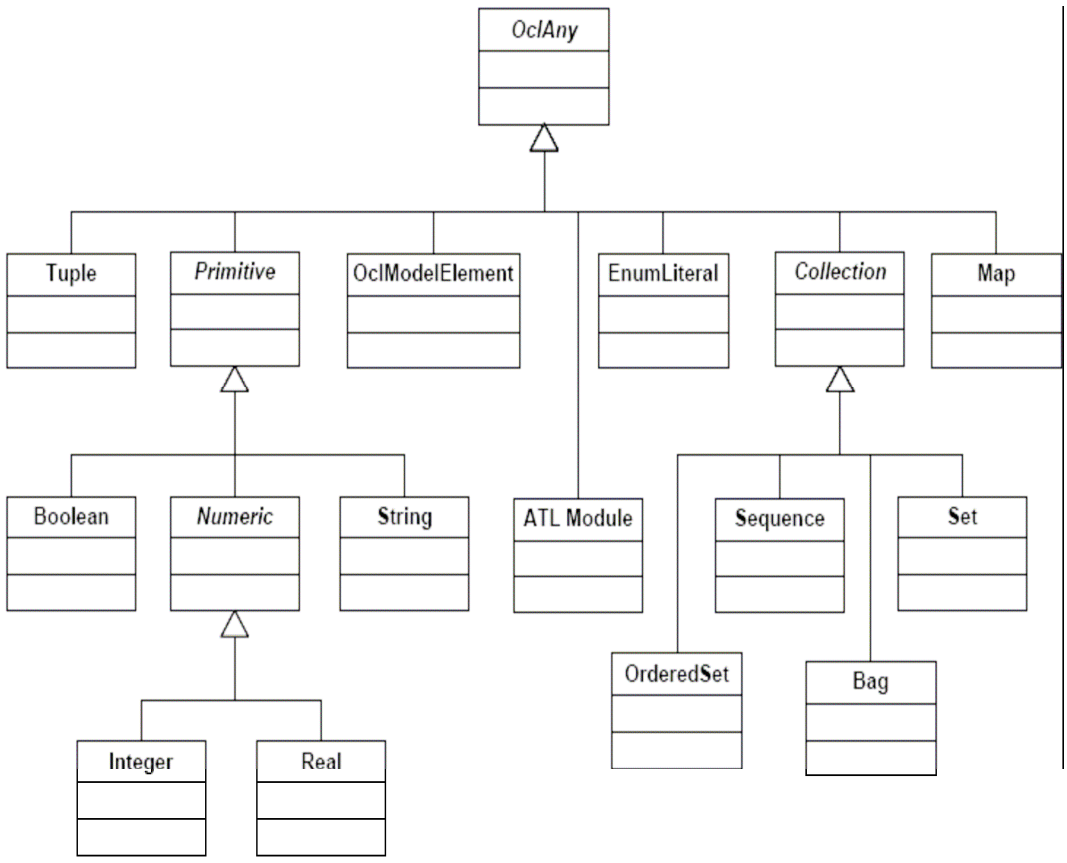


Figura 9.4.1.1: Metamodelo del sistema de tipos de ATL

9.4.1.1 Operaciones de OclType

Las instancias de la clase OclType corresponden a la definición de los distintos tipos especificados por OCL. Esta asociada a la operación OCL: *allInstances()*. Esta operación que no acepta parámetros retorna un conjunto de todas las instancias del tipo que la invoca.

La implementación de ATL provee una operación adicional que permite obtener todas las instancias de un tipo que corresponden a un metamodelo indicado como parámetro. Esta operación es la siguiente: *allInstancesFrom(metamodelo : String)*.

9.4.1.2 Operaciones de OclAny

Esta sección describe un conjunto de operaciones que son comunes a todos los tipos de datos. La sintaxis usada para invocar a una operación desde una variable en ATL son las siguientes:

```
self.nombre_operacion(parametros)
```

ATL provee soporte para las siguientes operaciones:

- Operaciones de comparación: = , <>
- *oclIsUndefined():* Indica sí *self* no está definido (análogo al null)
- *oclIsKindOf(t: oclType):* Indica si *self* es instancia directa o indirecta del tipo *t*
- *oclIsTypeOf(t:oclType):* Indica si *self* es instancia directa del tipo *t*

Las operaciones oclIsNew() y oclAsType() definidas por OCL no son soportadas por el motor ATL. Sin embargo, ATL, implementa un número adicional de operaciones:

- *toString():* Versión textual del objeto
- *oclType():* Tipo del objeto
- *asSequence(), asSet(), asBag():* Convierte el objeto en secuencia, conjunto y bolsa
- *refImmediateComposite()*: Obtiene el objeto bajo el cual *self* está compuesto.
- *output(s : String)*: escribe el string en la consola del Eclipse. Esta operación no retorna ningún valor y solo debe usarse en bloques imperativos de ATL.
- *debug(s : String)* : retorna el valor contenido en *self* y escribe el string "s : valor_self" a la consola Eclipse.

9.4.1.3 ATL module

El tipo de dato module es específico del lenguaje ATL. Este tipo de dato interno permite representar la unidad de ATL que es la que se esta corriendo actualmente en el motor. Tiene una sola instancia y se accede usando la variable *thisModule.* Esta variable permite acceder a los helpers y a los atributos que fueron declarados en el contexto del modulo ATL.

Este tipo de dato provee la operación *resolveTemp(var:oclAny, target:String).* Esta operación permite determinar que patrón target se aplicará a un objeto origen *var*. Útil cuando no se desea aplicar el patrón por defecto.

Notar que esta definida en el alcance de un módulo ATL, esta operación debe ser llamada desde la variable *thisModule*. La operación *resolveTemp ()* no debe ser llamada antes de la fase de matcheo. Esto significa que la operación puede ser llamada desde:

- el patrón origen y la sección do de cualquier regla declarativa;
- el patrón origen y la sección do de cualquier regla declarativa, de manera que la operación sea ejecutada después de la fase de matcheo.

9.4.1.4 Tipos de datos primitivos

OCL define cuatro tipos de datos primitivos junto a sus operaciones:

- `Boolean` con sus posibles valores `true` o `false;`
  - ✓ Operadores logicos: and, or, xor, not;
  - ✓ *implies(b : Boolean):* retorna false si *self* es true y *b* es falso, y retorna true en los demás casos.

- Numéricos como `Integer` que esta asociado a los valores enteros (1, -5, 2, 34, ….) `y Real` que esta asociado a los valores flotantes (1.5, 3.14, …)
  - ✓ Comparación: <, >, =>, >=
  - ✓ Operadores: *, +, -, /, div(x), max(x), min(x), abs()
  - ✓ Particulares: mod(x) para Integer y floor y round para Real
  - ✓ Aritméticas: cos, sin, toDegrees, toRadians, exp, log, sqrt…

- `String.` Un string esta definido entre `''`. Se indexa comenzando desde el 1.
  - ✓ *size():* retorna el numero de catacteres contenidos en el string self.
  - ✓ *concat(s : String):* retorna un string en el cual es concatenado el string *s* el final del string *self*.
  - ✓ *substring(lower : Integer, upper : Integer):* retorna un substring de *self* comenzando desde el carácter *lower* hasta el carácter *upper*.
  - ✓ De conversión: *toInteger(), toReal(),toSequence(), toUpper(), toLower()*
  - ✓ De salida: *writeTo(file:String):* escribe la cadena a un fichero. Las rutas relativas se consideran desde el directorio de eclipse; *println():* escribe en la salida estándar.

9.4.1.5 Tipo de datos y operaciones de Collections

OCL define una cantidad de tipos de datos de colecciones que provee diferentes maneras de manejar los elementos de ellas. Los tipos que provee son Set, OrderedSet, Bag y Sequence. Collection es una superclase abstracta de los diferentes tipos de datos.

Estas 4 subclases tienen las siguientes características:

- Set es una colección sin duplicados y sin orden.

- OrderedSet es una colección sin duplicados y con orden.
- Bag es una colección con duplicados y sin orden.
- Sequence es una colección con duplicados y con orden.

La declaración de una colección tiene que incluir el tipo de los elementos que la componen. La definición de una colección es la siguiente:

```
tipo_de_coleccion(tipo_dato_elemento)
```

Los tipos de datos son `Set`, `OrderedSet`, `Sequence` y `Bag`. El tipo de dato de los elementos puede ser cualquier tipo oclType incluyendo otros tipos de colecciones.

La instanciación de una colección es de la siguiente manera:

```
tipo_coleccion{elementos}
```

OCL define una cantidad de operaciones que son comunes a los diferentes tipos de colecciones. Estas operaciones se describen a continuación.
La invocación de las mismas se realizan utilizando -> , es decir:

```
self->nombre_operacion(parametros)
```

Las siguientes son las operaciones comunes:

- *size():* retorna la cantidad de elementos en la colección self.
- *includes(x:oclAny):* Devuelve *true* si *x* pertenece a la colección.
- *excludes(x:oclAny):* Devuelve *true* si *x no* pertenece a la colección.
- *count(x:oclAny):* Numero de veces que aparece x en la colección
- *includesAll(c: Collection):* Devuelve true si todos los objetos de c pertenecen a la colección.
- *excludesAll(c: Collection):* Devuelve *true* si ninguno de los objetos de c pertenecen a la colección.
- *isEmpty(), notEmpty()*
- *sum():* Aplica el operador + a todos los elementos de la colección
- *asBag(), asSequence(), asSet()*

9.4.1.6 Tipo de datos Enumeration

Un enumerativo es un OclType. Los enumerativos tienen que estar definidos dentro de los metamodelos origen y destino de una transformación.

Como ejemplo consideremos un enumerativo llamado Género que define dos tipos de valores posibles, Femenino y masculino. Para acceder al valor femenino del enumerativo en OCL sería de la siguiente manera: Genero::Femenino.

Pero en la implementación de ATL es diferente la manera de acceder al valor. Para acceder a los literales de una enumeración se utiliza el carácter #. Continuando con nuestro ejemplo seria `#female.`

Este tipo de dato no tiene operaciones para realizarse sobre el.

9.4.1.7 Tipo de datos Tuple

El tipo de dato tupla permite defnir valores compuestos. Un tupla consiste en una serie de piezas que pueden tener cada uno un tipo distinto.

Cada parte de este tipo de dato esta asociado con un OclType y es identificado con un nombre único. La declaración debe formarse con la siguiente sintaxis:

```
TupleType(nombre_var1 : tipo_var1, ..., nombre_var n : tipo_var n)
```

El orden en el cual son declaradas las variables no es significante. Como un ejemplo, es posible considerar la declaración de una tupla asociando un elemento del modelo Author desde el metamodelo MMAuthor con un par que contiene el titulo del libro y el nombre del editor del libro.

```
TupleType(a : MMAuthor!Author, title : String, editor : String)
```

La instanciación de la tupla debe respetar la siguiente sintaxis:

```
Tuple{ nombre_var 1 [:tipo_var 1]? = init_exp1, ..., nombre_var n
[:tipo_var n]? = init_expn}
```

Cuando declaramos una instancia de una tupla, los tipos de las variables de las tuplas pueden ser omitidos. Como consecuencia, las dos siguientes instanciaciones correspondientes a un tupla son equivalentes:

```
Tuple{editor : String = 'ATL Eds.', title : String = 'ATL
Manual', a : MMAuthor!Author = anAuthor}
```

```
Tuple{title = 'ATL Manual', a = anAuthor, editor = 'ATL Eds.'}
```

Como en la declaración de la tupla, en la instanciación las partes de una tupla pueden estar en cualquier orden.

Las diferentes partes de una tupla pueden ser accedidas usando la misma notación punto que es usada para la invocación de las operaciones o en los accesos de los atributos de los elementos del modelo. De esta manera:

```
Tuple{title = 'ATL Manual', a = anAuthor, editor = 'ATL
Eds.'}.title
```

Proporciona acceso a la parte del titulo de la tupla.

Además del conjunto de operaciones comunes, la implementación de ATL define una operación de casteo adicional en el contexto del tipo de dato: *asMap()* es una operación

que retorna una variable Map en la cual las partes de una tupla son asociadas con sus respectivos valores.

9.4.1.8 Tipo de datos Map

El tipo de dato mapa no pertenece a la especificación OCL. Este tipo de dato permite manejar una estructura en la cual cada valor esta asociado a una única clave que permite acceder al valor.

La declaración de un mapa conforma la siguiente sintaxis:

```
Map(tipo_clave, tipo_valor)
```

La siguiente declaración asocia algún elemento del modelo Author con una integer como clave:

```
Map(Integer, MMAuthor!Author)
```

La instanciación de un mapa debe respetar la siguiente sintaxis:

```
Map{(key_1, value_1), ..., (key_n, value_n) }
```

A modo de ejemplo, la siguiente espresion instancia un mapa con dos entidades:

```
Map{(0, anAuthor1), (1, anAuthor2)}
```

Además del conjunto de operaciones comunes, la implementación de ATL define las siguientes operaciones:

- *get(key:oclAny):* retorna el valor asociado a la clave *key* dentro del mapa *self* (o OclUndefined, si la calve *key* no forma parte del conjunto de claves de *self*);
- *including(key : oclAny, val : oclAny)* retorna una copia de *self* incluyendo el valor indicado si no contenía la clave *key*;
- *union(m : Map)* retorna un nuevo mapa conteniendo todos los elementos de self e incluyendo todos los elementos de m donde la clave no este incluida en self;
- *getKeys(), getValues()* : Colección con las claves o los valores

9.4.1.9 Tipo de datos Model Element

El último tipo de dato introducido por la especificación de OCL corresponde a los elementos de un modelo. Estos son definidos dentro de los metamodelos origen y destino de una transformación ATL. Los metamodelos usualmente definen diferentes elementos del modelo.

En ATL, las variables de elementos del modelo son referenciadas por la notación `metamodelo!clase`

Un elemento del modelo tiene una cantidad de atributos o referencias. Ambos pueden ser accedidos a través de la notación punto *`self.atributo`*. De esta manera en el contexto del metamodelo MMAuthor la expresión `anAuthor.name` permite acceder al atributo *`name`* de la instancia *`anAuthor`* de la clase Author.

En ATL, los model elements solamente pueden ser generados en las reglas, ya sean declarativas o imperativas. La inicialización consiste en inicializar cada atributo del elemento del modelo. Tales asignaciones son operadas con el significado de binding con los elementos del patrón origen.

Las operaciones permitidas a un elemento del modelo son:

- *oclIsUndefined():* para atributos con cardinalidad 0..1
- *isEmpty():* para atributos con cardinalidad *

9.4.2 Helpers ATL

Como fue introducido en secciones anteriores, en el contexto ATL se pueden definir helpers que serian como métodos en el lenguaje Java. Ellos permiten factorizar código y pueden ser llamados desde diferentes puntos del programa ATL.

Hay dos tipos de helpers en los cuales su sintaxis es muy similar: helper funcional y atributos helper. Ambos tipos deben definir el contexto a los cuales aplican. Sin embargo, podemos comparar diciendo que un atributo helper es mas comúnmente llamado atributo y helper funcional como helper y el cual puede aceptar parámetros. Esta diferencia implica algunas diferencias en la semántica de la ejecución de ambos tipos de helpers como se describió en secciones anteriores.

9.4.2.1 Helpers

Un helper ATL es definido de la siguiente manera:

```
helper [context context_type]? def : nombre_helper (parameters)
: return_type = exp;
```

Cada helper es caracterizado por su contexto, su nombre, su conjunto de parámetros y su tipo de retorno. El contexto de un helper es introducido por la palabra clave context. Este define el tipo de elementos al cual el helper aplica, es decir, el tipo de elemento desde el cual es posible invocarlo. El contexto puede ser omitido en la definición del helper. En ese caso, el helper esta asociado al contexto global del modulo ATL. Esto significa que el alcance de tal helper, se limita al modulo que se esta ejecutando.

El nombre del helper es introducido después de la palabra clave def. Como su contexto, esta parte es la signatura del helper. Un helper acepta un conjunto de parámetros que son especificados entre paréntesis después del nombre del helper. La definición del

parámetro incluye el nombre del parámetro y el tipo del parámetro como es especificado de la siguiente manera:

```
parameter_name : parameter_type
```

Varios parámetros pueden ser declarados separándolos por la coma (“,”). El nombre del parámetro es una variable identificada dentro del helper. Esto significa que dentro de la definición del helper cada nombre de parámetro debe ser único.

El cuerpo o contenido de un helper es especificado como una expresión OCL. Esta expresión puede ser uno de los tantos tipos de expresiones soportados. A modo de ejemplo, presentamos el siguiente helper:

```
helper def : averageLowerThan(s : Sequence(Integer), value :
Real) : Boolean =
let avg : Real = s->sum()/s->size() in avg < value;
```

Este helper denominado averageLowerThan esta definido en el contexto del modulo ATL ya que no tiene especificado el contexto particular. Este helper retorna un valor boolean en el cual determina si el promedio de los valores contenidos en una secuencia de integer (parámetro `s`) es estrictamente menor que un valor real pasado como parámetro. El cuerpo del helper consiste en una expression “`let`” definida e inicializada con la variable `avg`. Esta variable es entonces comparada con el parámetro `value`

9.4.2.2 Limitaciones

La implementación actual sufre tres limitaciones en el dominio de los helpers y atributos.

1. La primera se refiere a la definición de la signatura de los helpers. Los helpers son identificados a través de su signatura la cual incluye el nombre, el contexto y sus parámetros. Sin embargo, solamente considera la composición entre el nombre y el contexto en la signatura: los parámetros de los helpers no hacen posible la discriminación de los helpers que tiene el mismo nombre y el mismo contexto. Esto implica que todos los helpers definidos dentro del mismo contexto en un programa ATL deben tener distinto nombre. Esta restricción solo concierna a helpers que son definidos dentro de una librería que es importada en cualquiera consulta o modulo.
2. La segunda limitación concierna a la definición de los helpers en el contexto de un tipo colecciones. Tales definiciones son actualmente no soportados por el motor ATL.
3. Finalmente, la última limitación concierne a helpers que están relacionados con una librería. Actualmente la implementación no soporta la definición de atributos dentro de una librería ATL.

9.4.2 Reglas ATL

En el alcance del lenguaje ATL la generación de los elementos del modelo origen se realiza a través de reglas. ATL define dos tipos de reglas: declarativas e imperativas.

9.4.2.1 Código imperativo.

ATL permite especificar código imperativo dentro bloques de un regla imperativa o declarativa. Un bloque imperativo esta compuesto de una secuencia de sentencias imperativas. Como en lo lenguajes Java o C++, cada sentencia debe terminar con un punto y coma (“;”).

Actualmente la implementación de ATL provee tres tipos de sentencias: la sentencia de asignación, las sentencias if y for. A diferencia de las expresiones OCL, estas no retornar ningún valor. Como consecuencia ellas pueden ser usadas en el alcance del código declarativo.

9.4.2.2 Limitaciones

No es posible declarar variables dentro de bloques imperativos. Las variables pueden ser usadas en el alcance de los siguientes bloques:

- Elementos del modelo origen y destino declarados en una regla declarativa local;
- Elementos del modelo destino declarados en el llamado de una regla declarativa local;
- Variables declaradas localmente;
- Atributos declarados en el contexto del modelo ATL.

Actualmente no se permite modificar las variables definidas localmente desde sentencias de asignación. Esto significa que tanto los elementos del modelo de origen, como los elementos del modelo destino las variables locales que pueden ser modificadas son los atributos que fueron definidos en el contexto del modulo ATL.

9.4.3 Queries ATL

Las unidades consultas aceptar una cantidad de modelos y produce un único valor de retorno de cualquier dato primitivo. Están compuestas por un elemento query con una cantidad de helpers y atributos que pueden ser definidos en el contexto de cualquier modulo ATL o de cualquier elemento del modelo definido dentro del modelo de origen de la consulta. Una consulta debe comenzar con la declaración de su elemento query. La especificación del elemento tiene la siguiente sintaxis:

```
query query_name = exp;
```

No hay ninguna restricción en el nombre de la consulta. Sin embargo, la consulta debe tener el mismo nombre que el archivo en donde esta definida. El cuerpo del elemento query es una expresión OCL de cualquier tipo primitivo de datos: string, boolean, integer o real. Los helpers y atributos definidos en el archivo de consulta pueden ser llamados en el alcance del cuerpo de un elemento query.

El resultado de una consulta se puede contener en un archivo separado de una manera muy fácil con la operación *writeTo()*. Como un ejemplo, mostramos la siguiente consulta:

```
query PersonNb =
MMPerson!Person.allInstances()-
>size().toString().writeTo('result.txt');
```

Esta consulta es ejecutada sobre un modelo MMPerson el cual contiene entidades Person. La consulta primero obtiene todas las clases Person del modelo y luego obtiene la cantidad. Esta cantidad es casteada a un string y luego escrita dentro del archivo result.txt. A pesar de que el resultado es escrito dentro del archivo, la consulta retorna el resultado.

9.5 Creando un archivo ATL.

La IDE de ATL IDE provee un wizard específico para la creación de archivos ATL. A continuación explicaremos la creación del mismo.

9.5.1 Wizard de creación de un archivo ATL.

El wizard se inicia desde la vista de navegación luego de seleccionar New→ATL File del menú contextual de un proyecto ATL como se muestra en la figura 9.5.1.1. Este comando es también visible desde el comando File de la barra de menú del Eclipse y acciona la aparición de la ventana del wizard.

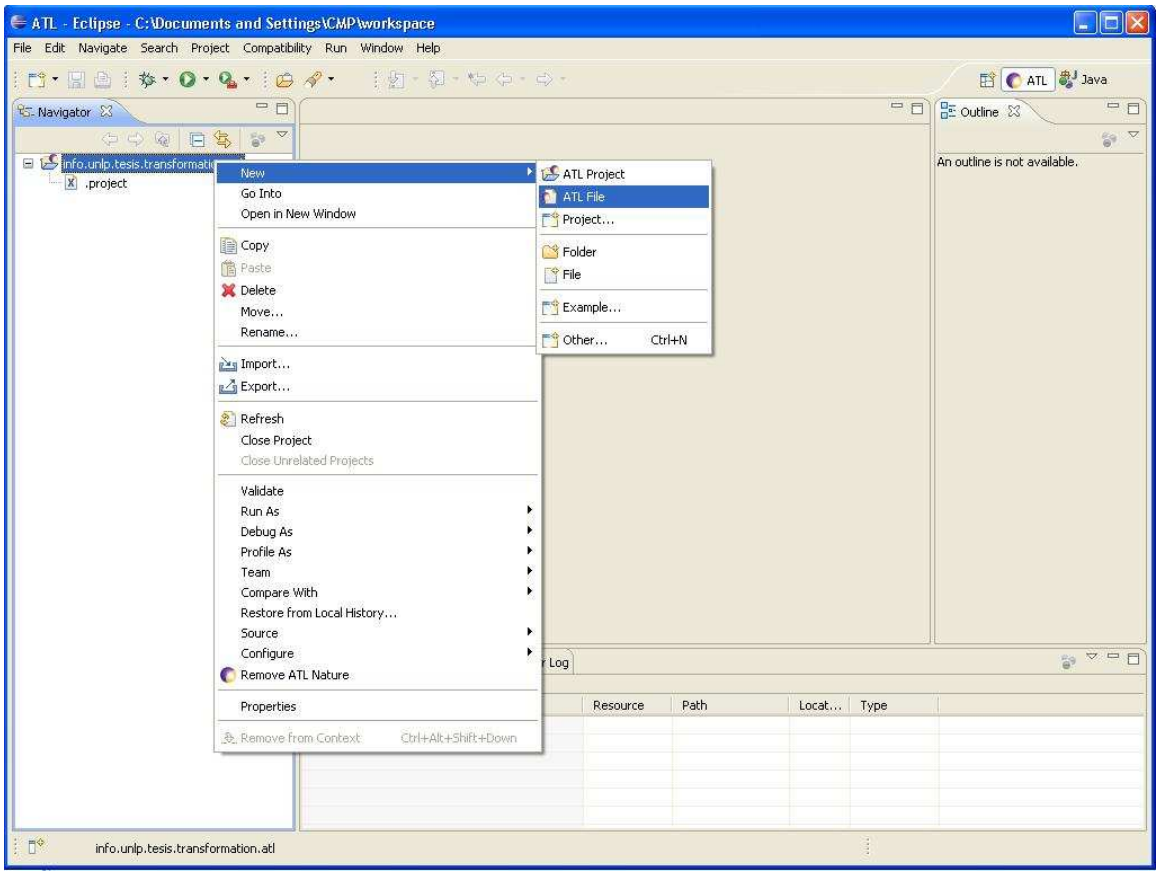


Figura 9.5.1.1. Activación del wizard

El wizard del archivo ATL permite especificar el nombre del archivo que se va a crear y como segundo paso la configuración del header de la transformación, es decir, el nombre del modulo y el tipo de unidad del archivo ATL (un modulo ATL, query o library), el nombre de las variables que contendrán los modelos de origen y destino como así también los metamodelos y las librerías que serán necesarias para le ejecución del programa ATL. A partir de estos datos, el wizard genera el archivo ATL con la sección header correspondiente a la información que se ingreso en la construcción del wizard.

Figura 9.5.1.2. Wizard del archivo ATL. Configuración del nombre del archivo.

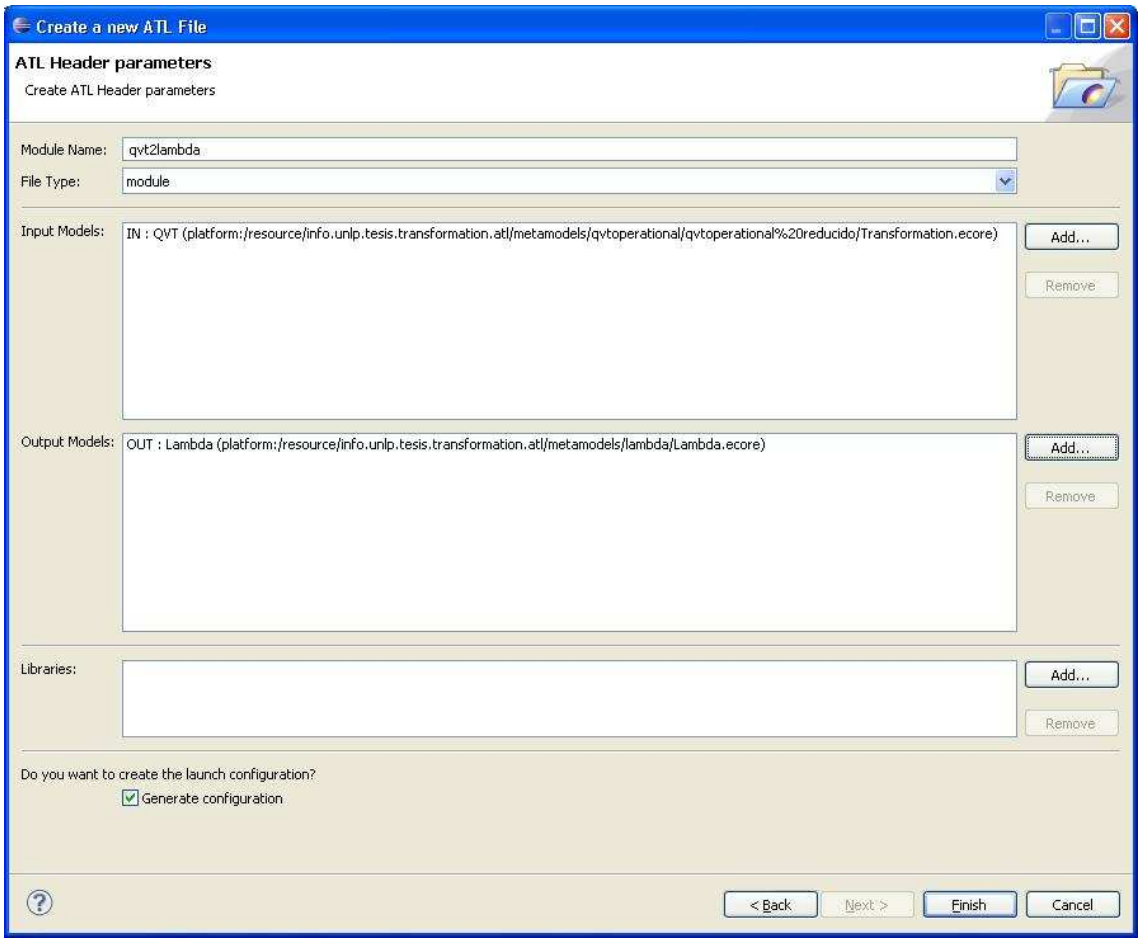


Figura 9.5.1.3. Wizard del archivo ATL.

Como se puede observar en la figura 9.5.1.2 la primera página del wizard nos permite indicar el nombre que llevara nuestro archivo ATL como así también la ubicación del mismo dentro del proyecto ATL. A continuación en la figura 9.5.1.3, siendo la segunda pagina del wizard es una pagina organizada en cuatro secciones: *HEAD, IN, OUT* y *LIB*.

- La sección HEAD tiene como objetivo especificar el nombre del modulo ATL y el tipo de archivo que será. El tipo de archivo se corresponde con el tipo de unidad ATL. Estos tipos de unidades pueden ser seleccionables del combo con las opciones: *module, refining module, query* o *library*. Notar que dependiendo de que tipo de archivo se seleccione dentro de la página estará parcialmente o completamente habilitados los componentes necesarios para su creación.

  Es una buena convención del nombres del modulo con el nombre del metamodelo de origen seguido del carácter "2" y seguido del nombre del metamodelo de destino. Por ejemplo ***qvt2Lambda***.
- La sección IN y OUT de la página permite especificar el nombre de las variables asociadas con el modelo y metamodelo de origen y destino respectivamente. Estas variables se declaran con el botón Add. Es posible declarar múltiples variables con modelos de origen y destino pero se tiene que tener cuidado con el nombre de las mismas ya que deben ser únicos.
- Finalmente, la sección LIB permite especificar el nombre de las librerías que son requeridas para la ejecución del programa ATL.

El modulo template generado por la configuración del wizard ATL es presentado en la siguiente figura.

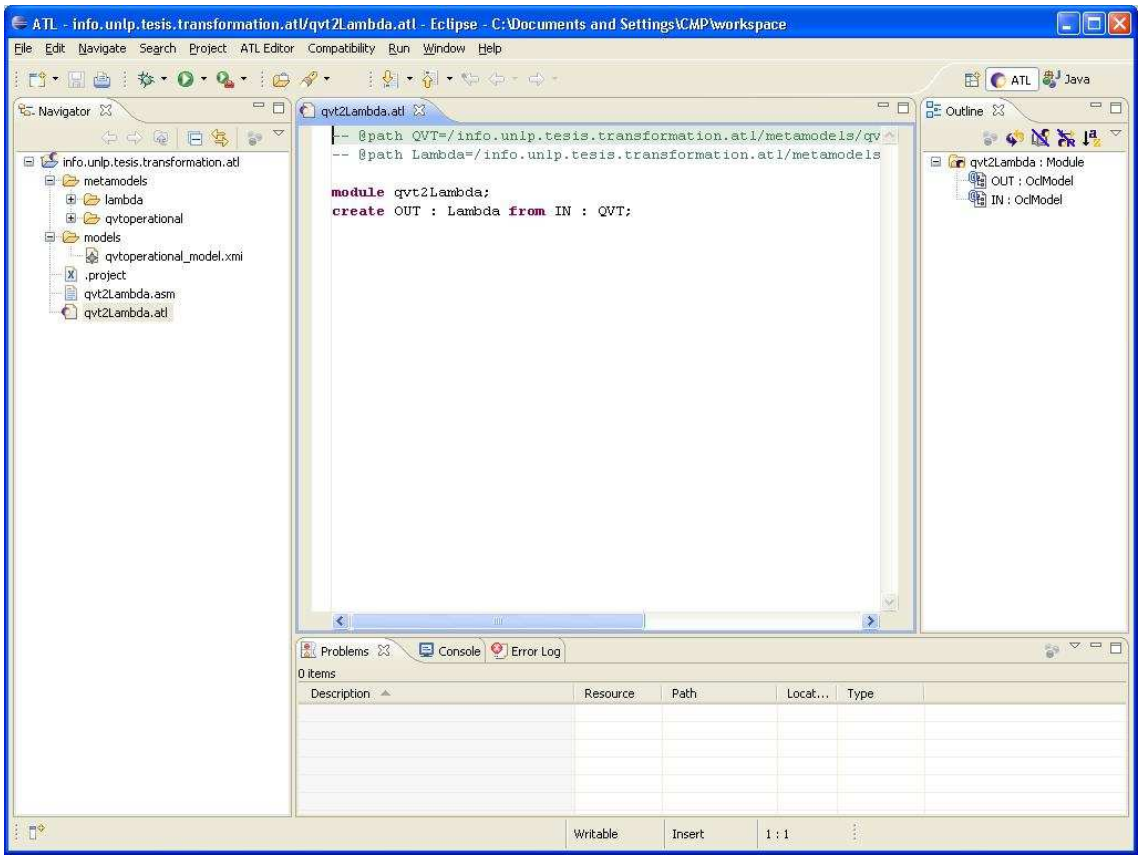


Figura 9.5.1.4. Template modulo ATL.

Además de crear el archivo de transformación, el wizard ATL crea un archivo adicional: archivo de transformación ASM (el cual esta asociado con la extensión .asm). El archivo ASM contiene el bytecode ATL correspondiente al archivo generado de transformación. Este bytecode esta codificado dentro de un lenguaje XML y es actualizado cuando se cambia la transformación (cuando el archivo es guardado).

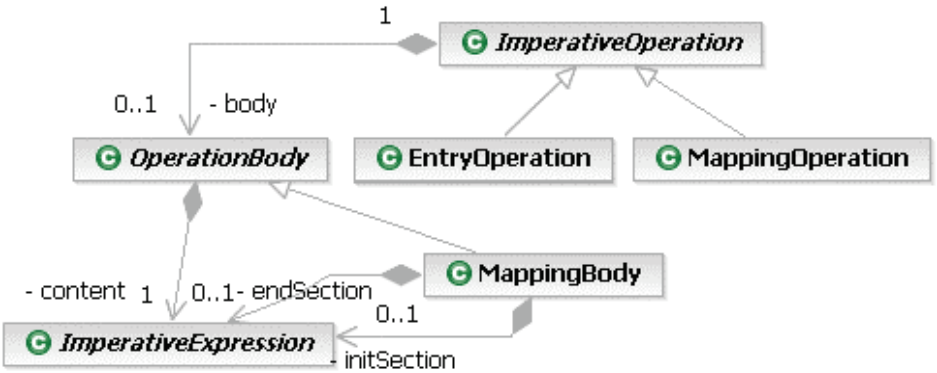
Como pudimos ver la creación del archivo de transformación es simple y clara.

Capítulo 10

Semántica de QVT

10.1 Definición de la semántica de QVT Operacional.

En esta sección definimos la semántica del lenguaje QVT Operacional para describir las transformaciones, cuya sintaxis abstracta está especificada mediante metamodelos y las cuales fueron abordadas en la tesis de la Dra. Roxana Giandini [6][33].
Los metamodelos son denominados: Operational Transformation (ver Figura 4.1) Imperative Operation e Imperative Expression en la siguiente figura.



En QVT Operacional, las Imperative Expression constituyen una extensión de OCL ya que son subclases de OCLExpression pero producen cambios en el ambiente y en el almacenamiento.

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

En los párrafos siguientes, definimos los dominios semánticos y funciones necesarias para definir la función semántica $\boldsymbol{\upsilon}$ para transformaciones de modelos.

10.1.1 Dominios semánticos

Consideramos los siguientes dominios:

Bvalue, dominio primitivo para valores básicos (por ejemplo: bool, nat)

Oids, dominio primitivo para identificadores de valores almacenables
Vars, dominio primitivo para nombres de variables

Evalue = Bvalue + Oids, dominio para valores expresables (es decir valores básicos e identificadores)

Svalue, dominio para valores almacenables (objetos)

Env = [Vars -> Evalue] dominio para el ambiente de variables. Un ambiente es una función que liga nombres de variables con valores expresables.

Store = [Oids -> Svalue] dominio para el almacenamiento (memoria) de objetos. Una memoria es una función inyectiva que liga identificadores de objetos con objetos.

$\sum$ = ***(Env X Store)*** dominio para estados que representan la asignación de valores a variables. Cada estado está representado por el ambiente de ejecución (***Env***) y la memoria (***Store***).

10.1.2 Funciones semánticas

Para la definición de esta semántica, se utilizó notación funcional, que complementa el uso del lenguaje de especificación OCL [5], por ser un lenguaje estándar ampliamente conocido por la comunidad dedicada al área del modelado; es además suficientemente expresivo y de uso amigable.

Función semántica I para expresiones OCL
Utilizaremos la función semántica I definida en el "Appendix A – Semantics" de [5] para evaluar expresiones OCL originales, libres de efectos laterales. Esta función no es aplicable a la sub-jerarquía de las OclExpression con raíz en ImperativeExpression, ya que como mencionamos anteriormente producen cambios de estado.

La función se define:

$$I : OclExpression \rightarrow (Store\ X\ Env) \rightarrow Evalue$$

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

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

La función υ está definida sobre la estructura jerárquica del metamodelo:

υ : OperationalTransformation $\rightarrow$ *(Env X Store)* $\rightarrow$ *(Env X Store)*
υ : ImperativeOperation $\rightarrow$ *(Env X Store)* $\rightarrow$ *(Env X Store)*

υ : OperationBody → *(Env X Store)* → *(Env X Store)*
υ : ImperativeExpression → *(Env X Store)* → *(Env X Store)*

Sean δ: ***Env***, σ: ***Store:***

υ **(**ot: OperationalTransformation) = λ (in:T) **. (second(** υ (ot:entry) (δ^o,σ^o))) (lout)

La semántica de la construcción OperationalTransformation es el resultado de aplicar υ a la operación "main" (de clase EntryOperation) de la transformación comenzando con un environment y un store inicial. Acorde al metamodelo de la figura 4.1, una entry operation es accesible a través del rol con nombre *entry*. Entonces el resultado final se obtiene del segundo componente del par y colocándolo en una ubicación inicial denominada *lout.*

υ **(**entry: EntryOperation) = λ (δ,σ). υ **(**entry**.** body) (δ,σ)

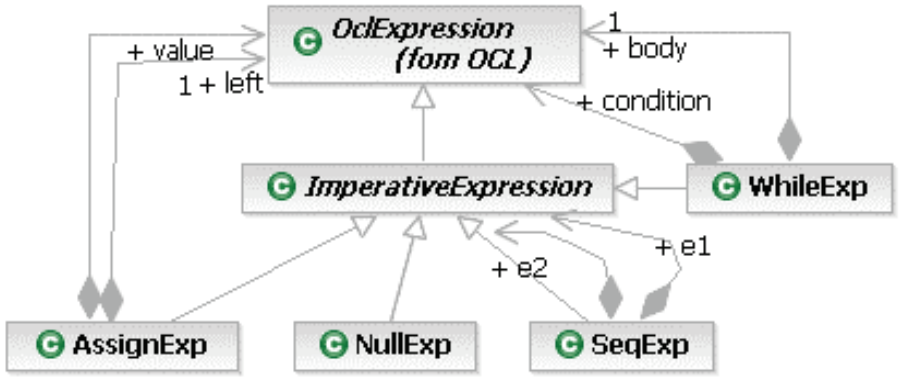
La semántica de una operación EntryOperation es el resultado de aplicar υ al cuerpo de dicha operación, obtenida por la asociación *body*.

υ (body: OperationBody) = λ (δ,σ). υ (body.content) (δ,σ)

La semántica del cuerpo de una operación es el resultado de aplicar υ al contenido del cuerpo de dicha operación, obtenida por la asociación *content*. El contenido de un body se corresponde con una instancia de la jerarquía de ImperativeExpression. Consecuentemente, a partir de este punto, u es la función semántica aplicada a las ImperativeExpression.

En general, la ejecución de las ImperativeExpression produce cambios en δ, no en σ, salvo cuando se crean objetos, por ejemplo con la expresión ObjectExp.

Entonces, la función se aplica a cada Imperative Expression de la siguiente manera:



$$\boldsymbol{\upsilon}\ (\text{e: SeqExp}) = \lambda\ (\delta,\sigma).\ \boldsymbol{\upsilon}\ (\text{e.e2})\ (\boldsymbol{\upsilon}\ (\text{e.e1})\ (\delta,\ \sigma))$$

La semántica de la expresión SeqExp es el resultado de aplicar $\boldsymbol{\upsilon}$, en primer instancia a la primer expresión componente de e, es decir *e1*, y luego, en ese estado modificado, aplicar $\boldsymbol{\upsilon}$ a la segunda componente de e, *e2*.

$$\boldsymbol{\upsilon}\ (\text{e: NullExp}) = \lambda\ (\delta,\sigma).\ (\delta,\ \sigma)$$

La semántica de la expresión NullExp es una función que no produce modificaciones ni en el Env, ni en el Store.

$$\boldsymbol{\upsilon}\ (\text{e:AssigExp}) = \lambda\ (\delta,\sigma).\ (\ \delta\ [\boldsymbol{I}\ (\text{e.value})(\sigma,\ \delta)\ /\ \text{e.left}\ ],\ \sigma)$$

La semántica de la expresión AssigExp es una función que solamente modifica el Env, ligando la parte izquierda de e con el resultado de evaluar, a través de la función $\boldsymbol{I}$, la expresión dada por *e.value*.

$$\begin{aligned}\boldsymbol{\upsilon}\ (\text{e:IfExp}) = \lambda\ (\delta,\sigma).\ &\text{if}\ \boldsymbol{I}\ (\text{e.condition})\ (\sigma,\ \delta) = \text{tt}\\ &\text{then}\ \boldsymbol{\upsilon}\ (\text{e.thenExpression}))\ (\delta,\ \sigma)\\ &\text{else}\ \boldsymbol{\upsilon}\ (\text{e.elseExpression}))\ (\delta,\ \sigma)\end{aligned}$$

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

$$\boldsymbol{\upsilon}\ (\text{e:WhileExp}) = \lambda\,(\delta,\sigma),\ \text{if}\ \boldsymbol{I}\ (\text{e.condition})\ (\delta,\sigma) = \text{ff}$$
$$\text{then}\ (\delta,\sigma)$$
$$\text{else}\ \boldsymbol{\upsilon}\,(\text{e})\ (\boldsymbol{\upsilon}\,(\text{e.body})\ (\delta,\sigma))$$

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

10.2 Implementación de la semántica de QVT Operacional

Como ya hemos definido anteriormente la implementación de la semántica fue realizada con ATL y la cual detallaremos a continuación.

Como se menciono en el capítulo anterior esta implementación se realiza sobre un archivo caracterizado con la extensión *.atl* y definido con diferentes secciones:

- Cabecera
- Importación de librerías
- Helpers y Attributes
- Reglas

10.2.1 Cabecera

En la cabecera especificamos el nombre del módulo ya que es un archivo de este tipo, los modelos, metamodelos implicados y el tipo de transformación.

```
-- @path
QVT=/info.unlp.tesis.transformation.atl/metamodels/qvtoperational/qvto
perationalReducido/Transformation.ecore
-- @path
Lambda=/info.unlp.tesis.transformation.atl/metamodels/lambda/Lambda.ec
ore

module qvt2Lambda;
create OUT : Lambda from IN : QVT;
```

10.2.2 Helpers

Se definieron helpers que nos permitieron extender el metamodelo con funciones y atributos derivados e invocados y definidos en OCL. Son semejantes a los métodos en Java.

Helpers en los cuales se determinaban el contexto sobre el cual el atributo es definido; de la misma manera se define un método en el contexto de determinada clase de objetos de programación.

```
helper context QVT!OCLExpression def:isAssignExp() : Boolean =
       self.oclIsTypeOf(QVT!AssignExp);

helper context QVT!OCLExpression def:isIntegerLiteralExp() : Boolean =
       self.oclIsTypeOf(QVT!IntegerLiteralExp);
```

Otros que nos permitieron machear operadores de QVT con el correspondiente operador en Lambda.

```
helper def: operatorEnumMap : Map(QVT!Operation, String) =
Map {(#"and", '&&'), (#"or", '||'), (#"not", 'not'),
      (#menor, '<'), (#mayor, '>'), (#suma, '+'), (#resta, '-'),
(#igual, '=='), (#distinto, '/='), (#"div", '/'), (#mult, '*')};
```

Y algunos con más lógica en los cuales trabajan con un parámetro y delegan su ejecución en reglas o helpers del módulo declarados como globales.

```
--Crea una Expresion lambda a partir de una OclExpression
helper context QVT!OCLExpression
def:getExpression(par:Lambda!ParSimple): Lambda!Expresion =
      if (self.oclIsTypeOf(QVT!BooleanLiteralExp))then
            thisModule.booleanLiteralExp2Boolean(self)
      else
            if(self.oclIsTypeOf(QVT!StringLiteralExp))then
                  thisModule.stringLiteralExp2StringType(self)
            else
                  if(self.oclIsTypeOf(QVT!IntegerLiteralExp))then
                        thisModule.IntegerLiteralExp2IntegerType(self)
                  else
                        if (self.isBreakExp() or self.isContinueExp()
                              or self.isWhileExp() or
                              self.isVariableInitExp() or
                              self.isAssignExp() or self.isVariable()
                              or self.oclIsTypeOf(QVT!IfExp)) then
                        self.getExpressionFromImperativeExpression(par)
                        else
                        --si es una operationCallExp tenemos que saber
                              si es una infix o prefix operation
                              if (self.referresOperation = #"not") then
thisModule.OperationCallExpNot2OperatorApplication(self,par)
                              else
thisModule.OperationCallExp2OperatorApplication(self)
                              endif
                         endif
                   endif
             endif
       endif;
```

10.2.3 Reglas

Las reglas que se declararon en su mayoría fueron *reglas declarativas lazy* en donde se especificaban el patrón origen que se buscaba en los modelos fuentes y el patrón destino que se creaba en el destino por cada regla que haga matching.

Son una variante de las reglas declarativas y siempre fueron llamadas explícitamente y no generaban elementos nuevos ya que se definieron como *unique*.

```
-- transforma un string de QVT en un string de Lambda
unique lazy rule stringLiteralExp2StringType{
   from stringQVT:QVT!StringLiteralExp
   to stringLambda:Lambda!StringType(
                        value<- stringQVT.stringSymbol.toString()
                    )
}
```

Se implementaron también reglas con guardas que nos permitieron definir una restricción del patrón origen.

```
-- Regla que toma solo las OperationCallExp que sabemos que requieren
de un argumento
unique lazy rule OperationCallExpNot2OperatorApplication {
from operationNot : QVT!OperationCallExp ,par:Lambda!ParSimple
      (operationNot.oclIsTypeOf(QVT!OperationCallExp) and
      operationNot.referresOperation = #"not" )
to operatorApp: Lambda!OperatorApplication(
                  operator <-
thisModule.operatorEnumMap.get(operationNot.referresOperation),
                  leftExp <- OclUndefined,
                  rightExp <-
thisModule.getExpressionForOperationCallExp(operationNot.argument1)
            )
}
```

Capítulo 11

La herramienta de implementación de la semántica

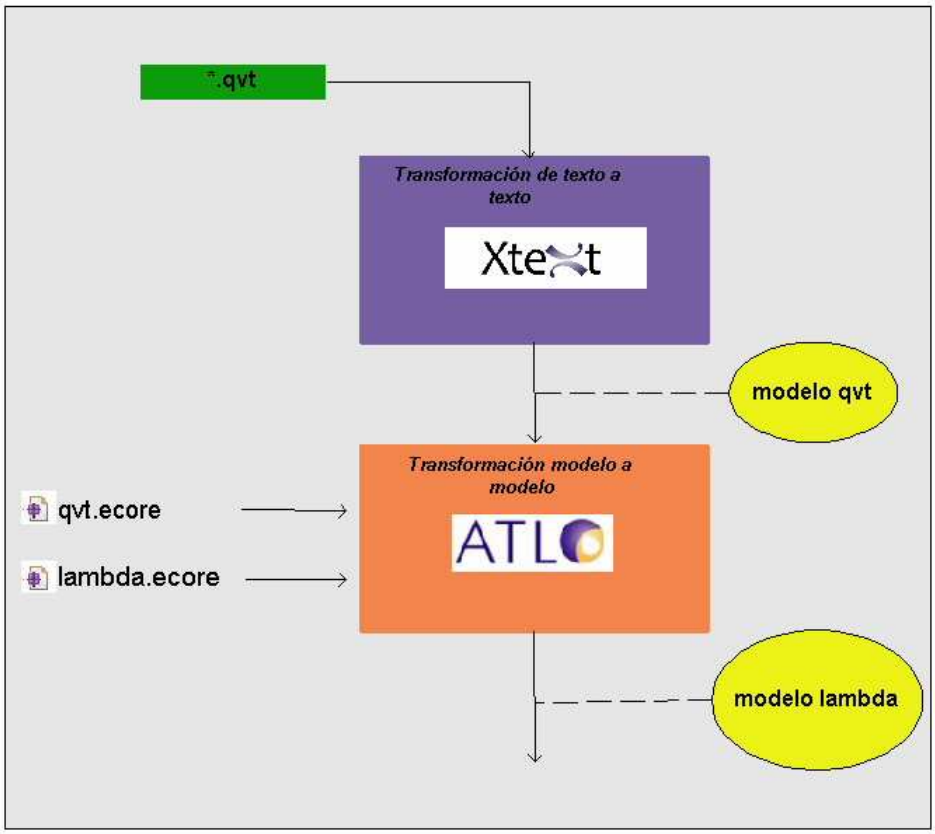
11.1 Aporte de la herramienta.

Hasta ahora existen herramientas que permiten definir DSLs, tales como GME y Microsoft DSL Tools y las cuales cubren satisfactoriamente la definición de la *sintaxis* del lenguaje, aunque presentan ciertas falencias a la hora de definir su semántica. Por ejemplo, la falencia más importante consiste en que la semántica queda hardcodeada resultando muy difícil de entender y mantener.

Por este motivo recientemente (ver [27]) se ha propuesto utilizar lenguajes de transformación de modelos para definir la semántica de los DSLs. En particular nuestra herramienta se enfoca en la semántica del lenguaje estándar QVT (Query/Views/Transformations) como DSL con la definición de la semántica para OperationalTransformation, EntryOperation, OperationBody, SeqExp, NullExo, WhileExp, IfExp y AssignExp.

11.2 Diseño del proceso de la herramienta

Se diseño para la aplicación una serie de procesos. En primer lugar procesar el archivo *.qvt para obtener el correspondiente modelo de qvt con formato xmi y en segundo lugar ejecutar la transformación en ATL.



11.3 Descripción de la herramienta.

La implementación de nuestra herramienta de software permite la definición de la semántica de QVT a través de transformaciones de modelos construidos en el contexto de un proceso de desarrollo de software dirigido por modelos [1] [2].

El fundamento teórico de nuestro trabajo fue definido por la Dra. Roxana Giandini en [6][33].

La herramienta fue planteada como un plug-in para la plataforma de Eclipse [35], la cual es invocada desde la barra de herramientas del mismo.

Al seleccionar el icono correspondiente a la herramienta nos permitirá seleccionar de nuestro workspace el archivo con la definición de una transformación QVT. Luego se presiona Siguiente y nos llevará a una página en donde presionando el botón Generar XMI ejecutará la transformación de texto a texto y luego la transformación de modelo a modelo mediante las definiciones desarrolladas en [6] y [33]. Por último nos mostrará el modelo de lambda generado correspondiente al modelo de qvt.

11.3.1 Introducción técnica a la plataforma Eclipse

La Plataforma Eclipse es una infraestructura abierta, diseñada para construir ambientes integrados de desarrollo (IDEs por su sigla en inglés *Integrated Development Environments*) que pueden ser usados para crear aplicaciones tan diversas como sitios web, programas Java™, programas C++, EJB™s, Servicios Web, etc. Sus creadores lo definen como *"un IDE para todo y nada en particular"*.

Consta de una vista vista de navegación (navigator) que muestra los archivos sobre los que estamos trabajando; el editor de texto muestra el contenido de un archivo; la vista de tareas (tasks) muestra la lista de tareas pendientes así como los errores; la visita de esquema (outline) muestra la estructura del archivo que estamos editando.
A pesar de que la funcionalidad de Eclipse es mucha , gran parte de la funcionalidad es muy genérica. Permite que nuevos componentes puedan utilizar nuevos tipos de contenido, para realizar nuevas tareas con contenidos existentes.

La Plataforma Eclipse permite descubrir, e invocar funcionalidad implementada en componentes llamados plugins. Un fabricante proporciona una herramienta independiente como un plug-in que permite llevar a cabo una determinada actividad. Cuando la Plataforma se inicializa, se le mostrarán al usuario además del entorno de Eclipse todos los plug-ins que tengamos instalados en el entorno.

La calidad de la experiencia del usuario depende de cómo se integren los diferentes plug-ins con la Plataforma y cómo aquéllos puedan comunicarse entre sí.

La Plataforma Eclipse está diseñada para afrontar las siguientes necesidades:

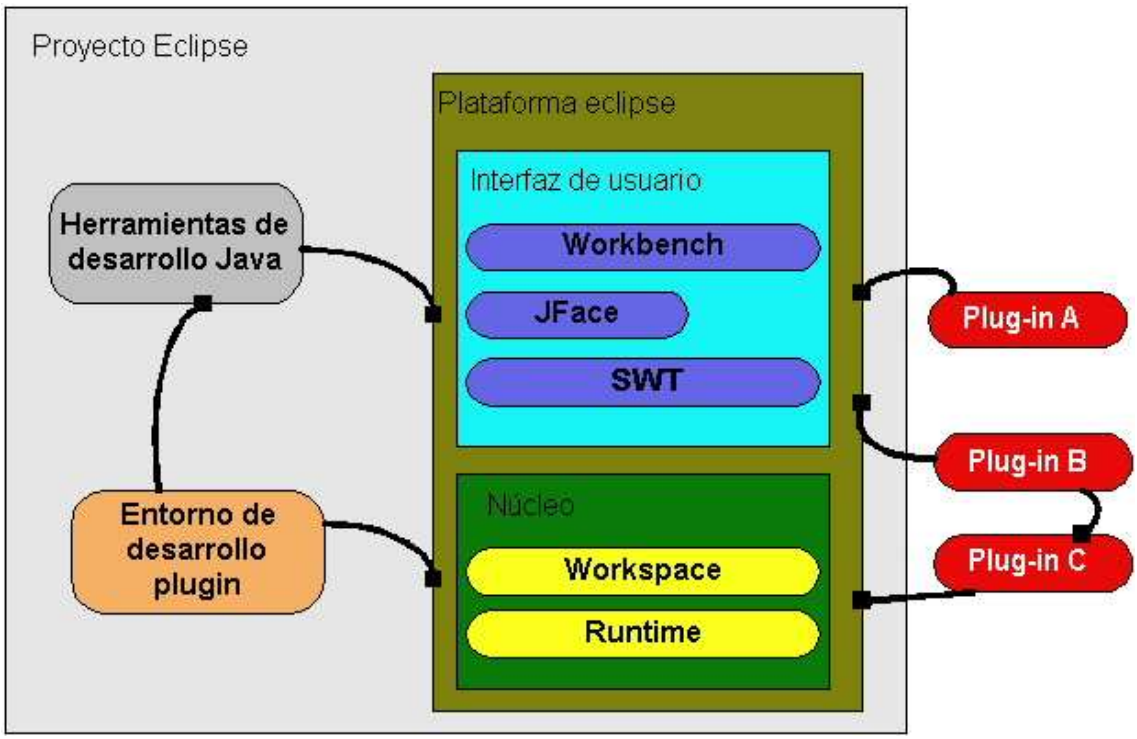
- Soportar la construcción de gran variedad de herramientas de desarrollo.
- Soportar las herramientas proporcionadas por diferentes fabricantes de software independientes (ISV's)
- Soportar herramientas que permitan manipular diferentes contenidos (por ejemplo, HTML, Java, C, JSP, EJB, XML, y GIF).
- Facilitar una integración transparente entre todas las herramientas y tipos de contenidos sin tener en cuenta al proveedor.
- Proporcionar entornos de desarrollo gráfico (GUI) o no gráficos.
- Ejecutarse en una gran variedad de sistemas operativos, incluyendo Windows® y Linux™.
- Hacer hincapié en que el lenguaje de programación sea Java para la construcción de nuevos plug-ins.

El principal objetivo de la Plataforma Eclipse es proporcionar mecanismos, reglas que puedan ser, seguidas por los fabricantes para integrar de manera transparente sus herramientas. Mediante APIs interfaces, clases y métodos, se exponen estos mecanismos. La Plataforma también nos posibilita la construcción nuevas herramientas que extenderán la funcionalidad de la Plataforma.

11.3.2 Arquitectura Eclipse para el desarrollo de plug-ins

En la siguiente figura puede verse la arquitectura de Eclipse. Permite entre otras cosas enriquecerlo con herramientas que pueden ser útiles para el desarrollo de software. Para ello, la plataforma está estructurada como un conjunto de subsistemas, los cuales son implementados en uno o más plug-ins que corren sobre un runtime engine (motor en

tiempo de ejecución). Dichos subsistemas definen puntos de extensión para facilitar la extensión de la plataforma.



Un plug-in es la unidad mínima de funcionalidad de Eclipse que puede ser distribuida de manera separada. Herramientas pequeñas se escriben como un único plug-in, mientras que en las complejas la funcionalidad está en varios plug-ins. Excepto un pequeño núcleo de la plataforma Eclipse, el resto de la funcionalidad de la plataforma Eclipse está implementada como plug-ins.

Los plug-ins están escritos en Java. Un plug-in está formado por un JAR de código Java, ficheros de lectura y otros recursos como imágenes, catálogos de mensajes, librerías de código nativo, etc.
Algunos plug-ins no contienen nada de código. Ejemplo: el plug-in que nos proporciona ayuda en forma de páginas HTML.
Todos los recursos que componen el plug-in se encuentran en un directorio del sistema de archivos del sistema operativo, o en una URL de un servidor. Existe un mecanismo que permite que un plug-in pueda ser sintetizado a partir de distintos fragmentos, cada uno en su propio directorio o en su propia URL. Este mecanismo es utilizado para distribuir diferentes paquetes de idiomas para un plug-in que soporte diversos idiomas (internacionalización).

Cada plug-in tiene un fichero denominado de manifiesto (manifest) en el cual se declaran sus interconexiones con otros plug-ins. La interconexión sigue un modelo muy simple: un plug-in declara un número de los denominados puntos de extensión, y un número de extensiones para uno o más puntos de extensión de otros plug-ins.

Los puntos de extensión de un plug-in pueden ser extendidos por otros plug-ins. Por ejemplo, el plug-in del banco de trabajo (workbench) declara un punto de extensión para las preferencias de usuario. Cualquier otro plug-in puede contribuir con sus propias

extensiones a las preferencias de usuario, mediante la definición de extensiones a este punto de extensión.

Un punto de extensión puede tener un interfaz API. Otros plug-ins contribuyen dando implementaciones a este API. Cualquier plug-in es libre de definir nuevos puntos de extensión así como de proporcionar nuevos APIs para que otros plug-ins puedan utilizarlos.

Al iniciar la Plataforma de Ejecución se descubren de manera dinámica el conjunto de plug-ins disponibles, se leen sus archivos de manifiesto, y se construye en memoria un registro de plug-ins. La Plataforma enlaza cada extensión por el nombre con sus declaraciones de puntos de extensión. Cualquier problema, como extensiones sin sus correspondientes puntos de extensión, se dectectan y se registran. El registro de plug-ins que se ha generado está disponible a través del API de la Plataforma. Nuevos plugins no pueden ser añadidos después del inicio.

Los archivos de manifiesto de los plug-ins contienen XML. Un punto de extensión puede declarar tipos de elementos XML adicionales para ser utilizados en las extensiones. Esto permite el paso de datos entre plug-ins. Además la información del archivo de manifiesto está disponible desde el registro de plug-ins sin haber activado los plug-ins o haber cargado algo de su código. Esto es fundamental para soportar un gran numero de plug-ins, a pesar de que sólo un número muy pequeño de ellos sean utilizados por el usuario. Hasta que el código del plug-in no es cargado, el efecto que tiene sobre el entorno de ejecución es nulo. La utilización de archivos XML permite la utilización de herramientas de desarrollo para la construcción de los mismos. El Entorno de desarrollo de Plug-ins (PDE) incluida en La Plataforma Eclipse es una de estas herramientas.

Un plug-in es activado cuando su código realmente necesita ser ejecutado. Una vez activado, un plug-in utiliza el registro de plug-ins para descubrir y acceder a las extensiones que contribuyen a sus puntos de extensión. Por ejemplo, el plug-in que declara el punto de acceso de preferencias de usuario puede descubrir todas las preferencias de usuario contribuidoras y mostrarlas para construir un cuadro de diálogo con todas ellas. Esto puede ser realizado sin necesidad de cargar el código de esos plug-ins simplemente consultados en registro de plug-ins.

El plug-in contribuidor será activado solo cuando el usuario seleccione la preferencia de la lista. La activación de este modo no sucede de manera automática. Existen APIs para la activación explícita de los plug-ins. Una vez activado, un plug-in permanece activo hasta que la Plataforma finaliza. Cada plug-in tiene su propio subdirectorio donde poder almacenar datos específicos del plug-in; lo que permite mantener datos de sesión entre ejecuciones.

La Plataforma se ejecuta en una única invocación de la máquina virtual de Java (JVM). A cada plug-in se le asigna su propia invocación cargados de clases (que únicamente se usa para cargar las clases del plug-in así como los recursos del mismo).

11.3.2.1 Workspaces

Las diferentes herramientas que son instaladas en la Plataforma Eclipse actúan sobre ficheros regulares que se almacenan en lo que se denominan espacios de trabajo (workspace), que es específico para el usuario. El espacio de trabajo de un usuario contiene varios directorios a nivel más alto (que se denominan proyectos).

Cada proyecto contiene los archivos que son creados y manipulados por el usuario. Todos los archivos en el espacio de trabajo son directamente accesibles por programas standard y herramientas del sistema operativo.

Para minimizar el riesgo de perder archivos, existe un mecanismo de historial mantenido a nivel de cada uno de los proyectos. El usuario puede controlar como el historial se mantiene a través de las preferencias: tamaño, número de días que se mantiene.

11.3.2.2 Workbench y Toolkits UI

La Plataforma Eclipse tiene interfaz gráfica, construida en base a un workbench (banco de trabajo) que proporciona toda la estructura y presenta un interfaz de usuario (UI) extensible al usuario. Existe también un API de este workbench y su implementación se lleva a cabo a través de lo que se denominan toolkits, que pueden ser de dos tipos:

SWT.- un conjunto de utilidades y librerías gráficas integradas con el sistema nativos de ventanas pero con un API independiente del sistema operativo.
JFace.- Un interfaz gráfico implementado sobre SWT que simplifica las tareas de programación

11.3.3 Interfaz de la herramienta

Se debe contar con un proyecto de diseño de transformaciones QVT y luego presionando el icono dentro de la barra de herramientas con la descripción "*Generar semántica de QVT Operacional*" como nos muestra la figura 11.3.3.1 nos abrirá un wizard que nos guiará para realizar la transformación correspondiente.

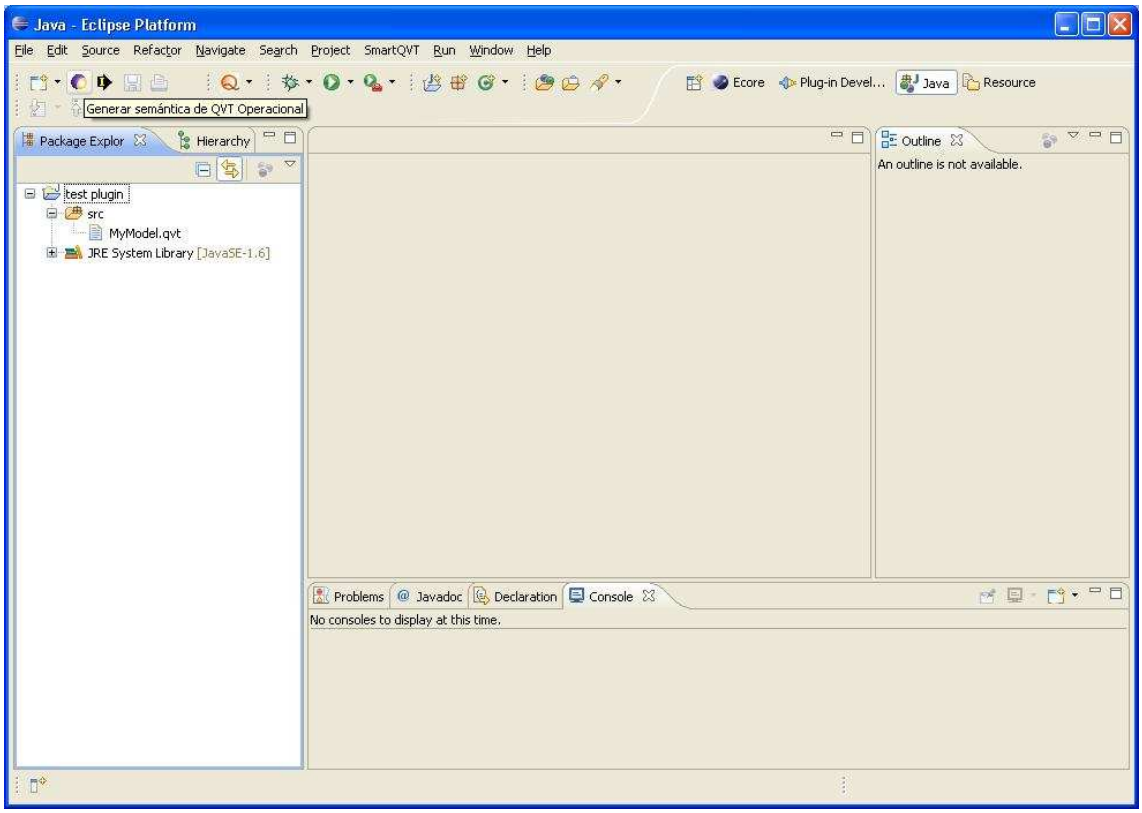
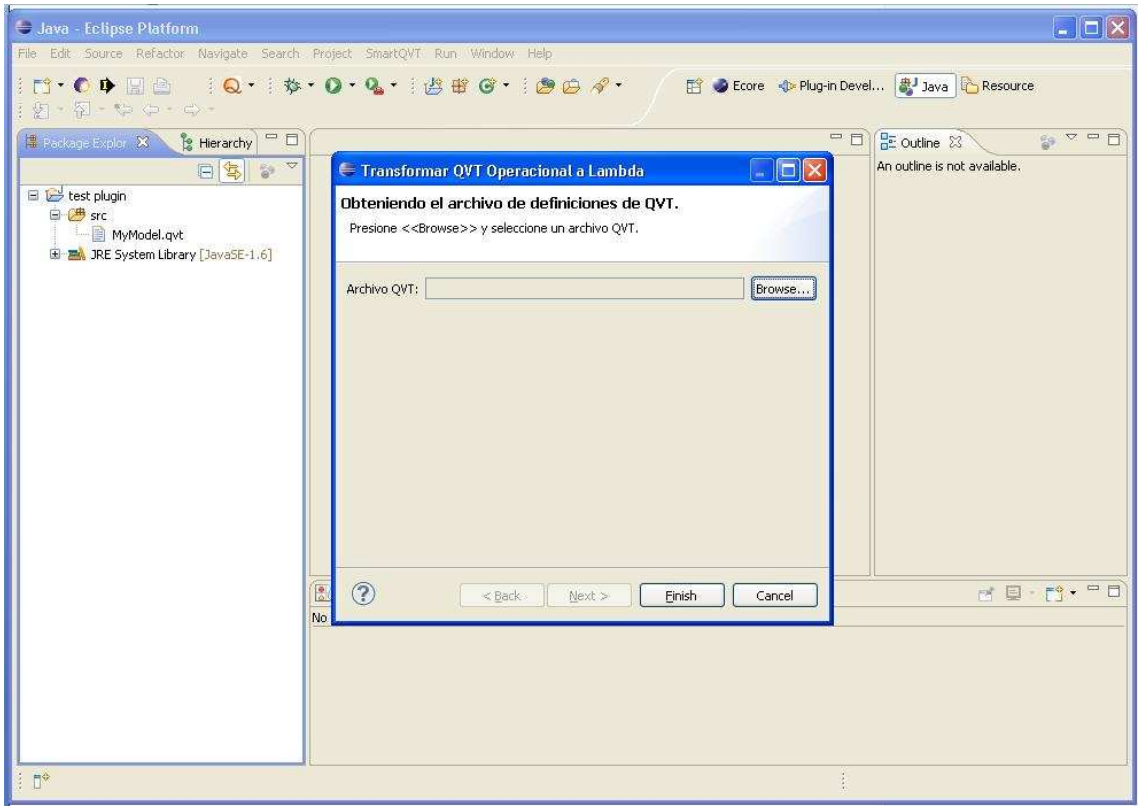


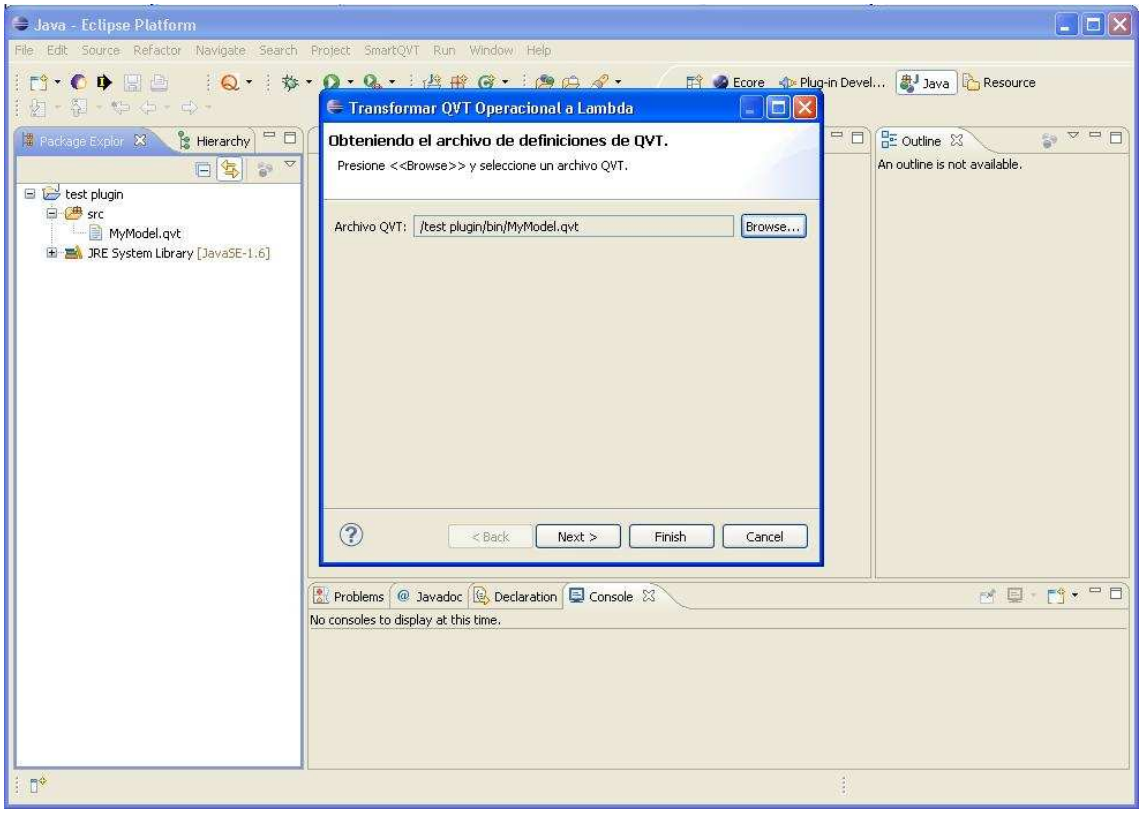
Figura 11.3.3.1: Acceso a la herramienta

Como primer paso debemos seleccionar el archivo *.qvt que tenemos creado en el proyecto a través de la herramienta presionando el botón *Browse*.



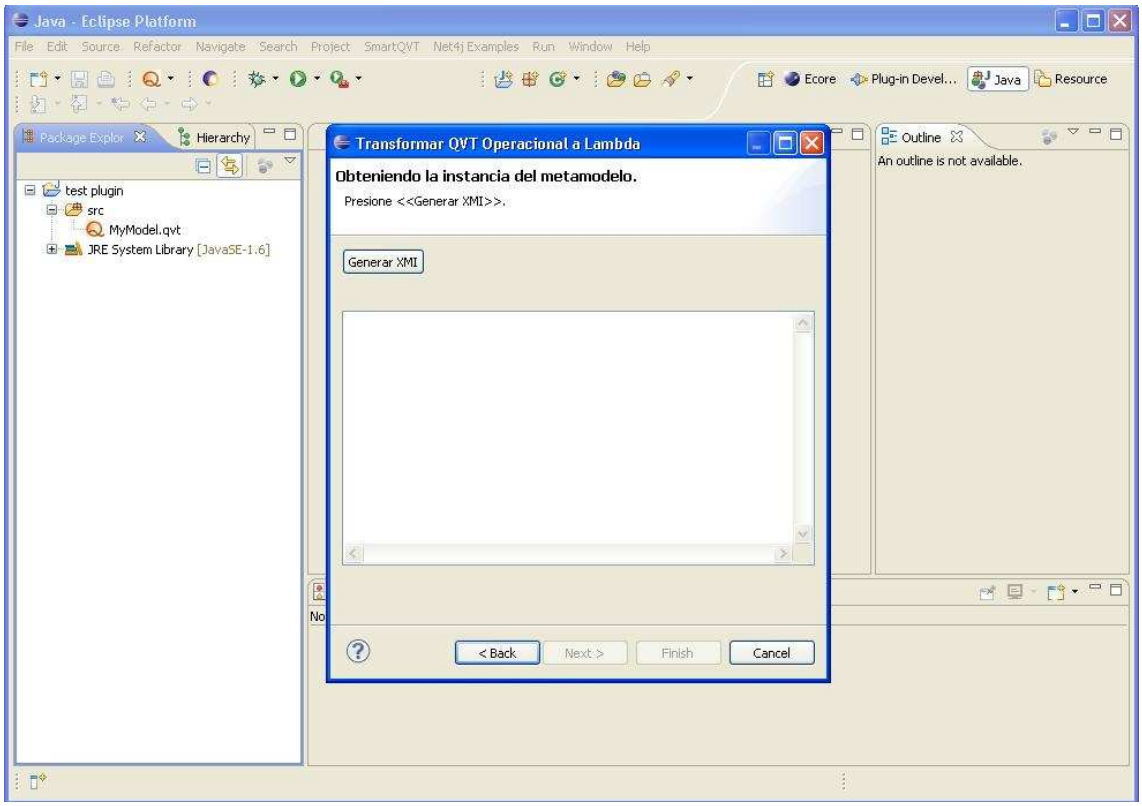
Este botón nos abre un diálogo en el cual nos permite navegar por el proyecto para seleccionar el archivo *.qvt. En el caso de seleccionar cualquier otro archivo por error la herramienta nos informa de esto y nos permite volver a seleccionar el archivo correspondiente.

De esta manera ya se ha indicado cual será el modelo de entrada para la transformación y el camino del archivo queda reflejado en la herramienta en el input *Archivo QVT*.

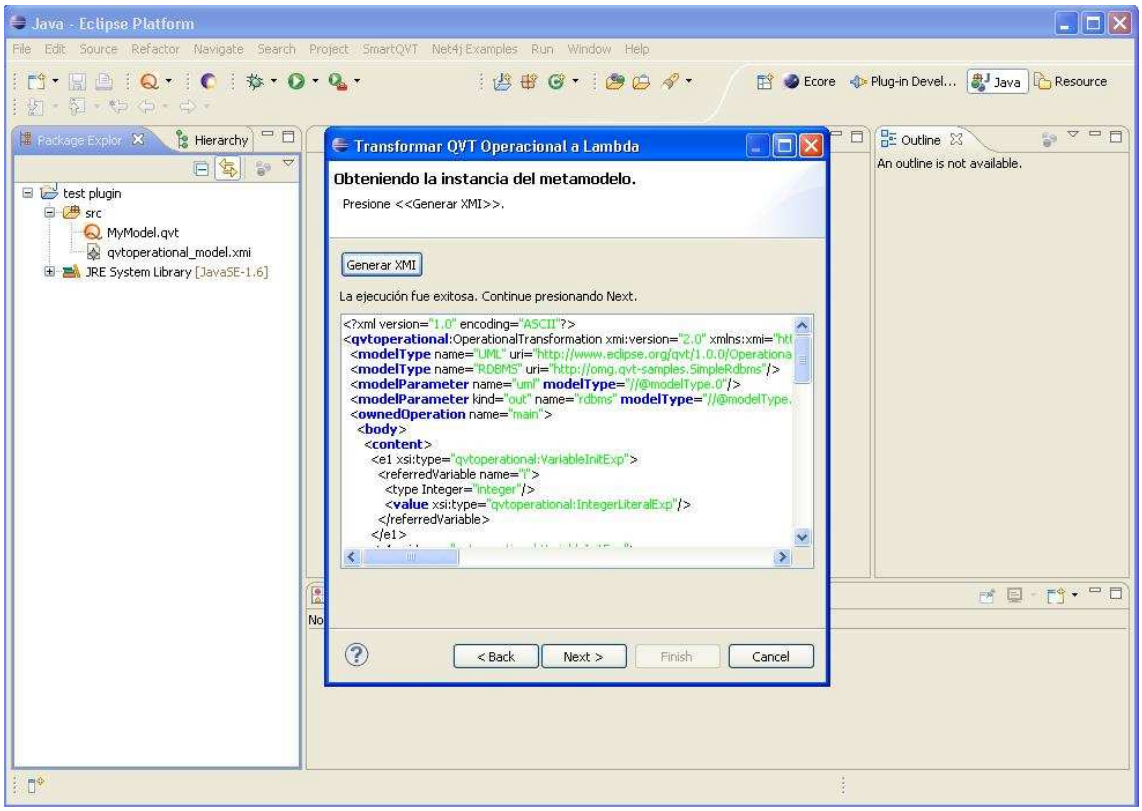


A continuación se debe presionar Next para continuar con el siguiente paso.

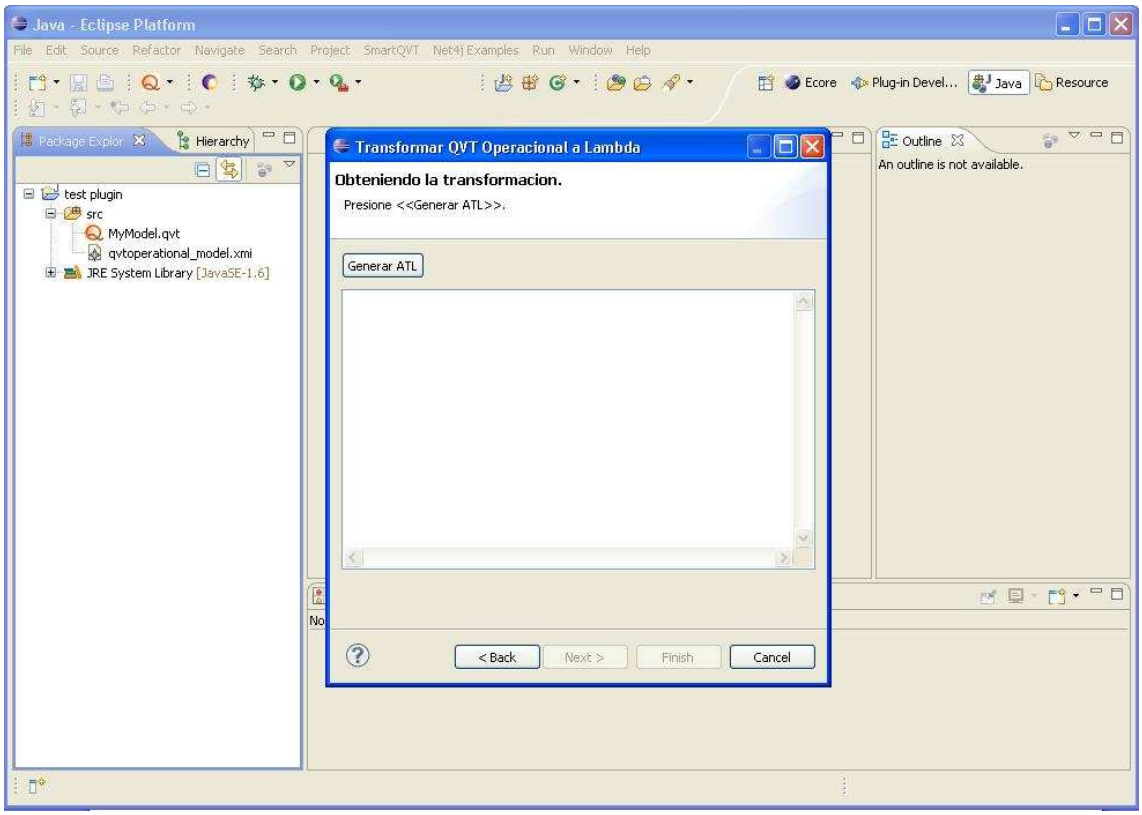
Aquí tenemos un botón que nos permitirá transformar el archivo *.qvt en uno *.xmi para que de esta manera podamos realizar la transformación.



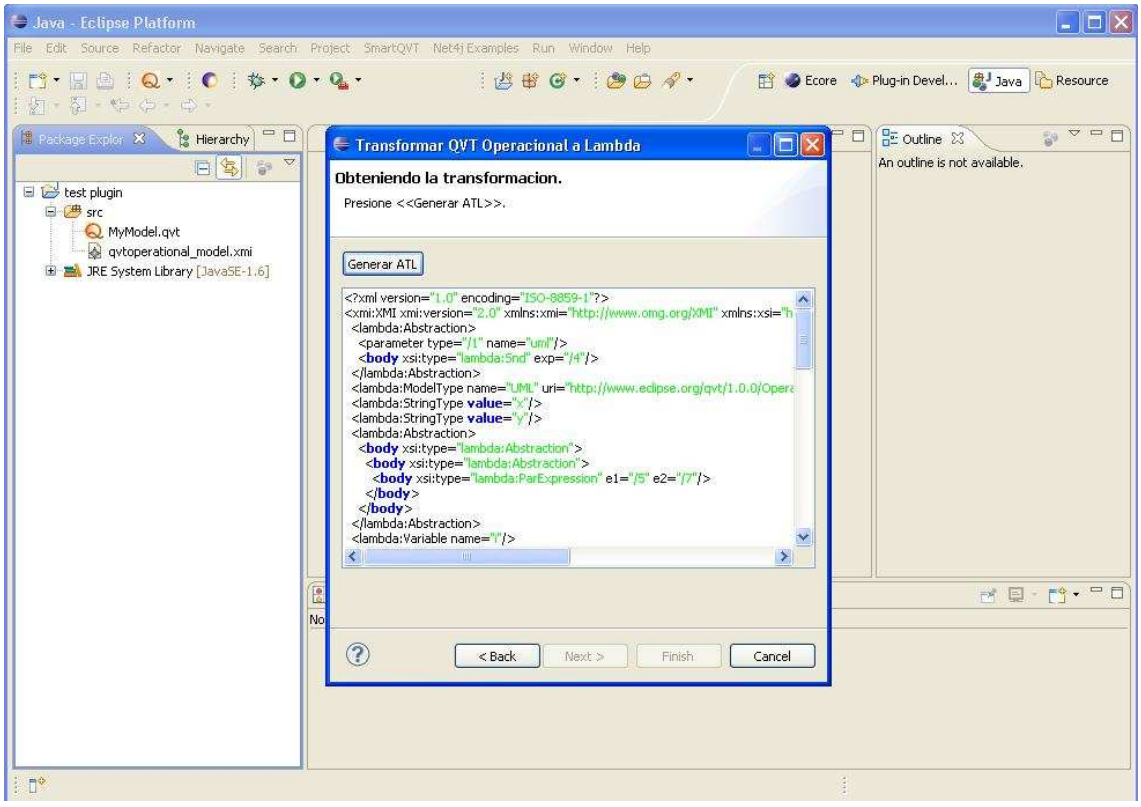
Al presionar el botón si la ejecución fue exitosa nos generará el correspondiente archivo con extensión XMI en el workspace que estamos usando la aplicación. Nos mostrará el contenido del mismo en la pantalla y nos habilitará el botón Next para continuar con el siguiente paso.



Como último paso de esta aplicación se debe presionar Next para culminar la transformación. Se presentará la siguiente pantalla en la cual nos permite ejecutar la transformación en ATL para obtener el correspondiente modelo de Lambda.



Al presionar el botón Generar ATL nos mostrará el resultado de la ejecución en la misma página de la siguiente manera:



De esta manera termina el proceso de la aplicación partiendo de un modelo de QVT operacional y terminando en un modelo Lambda.

Capítulo 12

Conclusiones

Las definiciones de la semántica de los DSLs brindan un importante avance en el Desarrollo de Software Dirigido por Modelos ya que mediante un lenguaje de transformación de modelos resultan definiciones más sencillas, precisas y fáciles de entender.

En nuestro caso de estudio particular, nos enfocamos en el lenguaje estándar QVT Operacional. El mismo está planteado como un lenguaje imperativo donde cada relación es una función o procedimiento con algunas restricciones. Tiene un punto de arranque "main" que es igual al cuerpo del programa principal que existe en la mayoria de los lenguajes imperativos. Cuenta con definición de parámetros formales y con mecanismos de reutilización de librerías.
El flujo de control de una transformación operacional comienza en el cuerpo de la transformación y luego se delega en las invocaciones a los mappings de manera secuencial.

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

Para formalizar la semántica de un lenguaje, es necesario contar con una función $\boldsymbol{v}$ que al aplicarla a un elemento de dicho lenguaje, permita obtener el elemento correspondiente en el dominio semántico.
En particular, para establecer la semántica de QVT, utilizamos como dominio semántico un lenguaje para transformaciones ya existente como Lambda Cálculo. De esta manera contamos con la ventaja de que este lenguaje ya tiene su semántica bien definida y provee una maquinaria en funcionamiento para ejecutarlo. Hemos incluido también el desarrollo de un simple ejemplo de uso de lenguaje QVT, mostrando la aplicación de las operaciones.

Otro punto relevante en esta tesis fue la integración de xtext a la herramienta de transformación. Esto permitió automatizar la generación de un modelo a partir de un metamodelo, proporcionando así una mejora a la utilización del framework EMF, una generación directa y transparente al usuario.

Con la integración de xtext y atl se logró la construcción de una herramienta, la cual a partir de un modelo escrito en lenguaje qvt, obtenga en primer instancia el modelo en formato necesario para atl, y en segunda instancia utilizando dicho modelo y una transformación genere el respectivo modelo en lambda cálculo.

Dicha transformación consiste en la construcción de un mapeo de elementos del lenguaje QVT a elementos de lenguaje Lambda Cálculo acordes a las posibilidades y restricciones que el lenguaje ATL impone.

Trabajo Futuro

Las reglas de transformación implementadas por nuestra herramienta estan basadas en un subconjunto del lenguaje QVT, el cual abarca la parte imperativa del lenguaje. Por esta razón, está limitada a modelos que utilicen elementos del QVT operacional.
Es necesario extender el conjunto de reglas que conforman la transformación para poder así abarcar todo el espectro del lenguaje QVT.

Con respecto a la herramienta, otros de los objetivos a cumplir, es la generación final del modelo de lambda cálculo con sintaxis de Haskell para poder ser ejecutado por un compilador. En la actualidad, la herramienta genera un archivo xmi, el cual posteriormente deberá ser escrito en un lenguaje funcional. Es deseable lograr un paso más en la herramienta que tome el archivo xmi generado por atl, y realice la transformación.

Trabajos relacionados

Con respecto a los trabajos relacionados a nuestro enfoque, la semántica de QVT está formalmente semi definida mediante una combinación del lenguaje natural y la notación matemática o implícitamente definido por su traducción a un lenguaje formal, por ejemplo:

- La propuesta presentada en [44] desarrolla una semántica algebraica para el framework de metamodelado MOF, pero la nocion de formalización aún no esta clara en el standard de MOF: el metamodelo, el modelo y la conformidad del modelo de su metamodelo. Al utilizar el lenguaje Maule, esta semántica formal es más ejecutable y puede ser utilizado para realizar análisis formales útiles.

- En [45] los autores presentan un mecanismo de transformación de modelos que se manifiesta por el operador ModelGen. ModelGen ha sido algebraicamente especificado en Maude permitiendo a los desarrolladores utilizar las herramientas formales para razonar acerca de las características de una transformación, tales como la terminación y confluencia.

- Por otro lado, la implementación de los motores de ejecución de QVT, como SmartQVT[46] para transformaciones operacionales y ModelMorf[47] para transformaciones relacionales pueden ser consideradas como definiciones operacionales de la semántica de QVT.

Nuestro trabajo ha explorado el uso de Lenguajes de Transformación para la definición de la semántica de DSL, en particular hemos desarrollado la semántica del DSL QVT mediante el lenguaje de Transformación ATL. Las ventajas de nuestra propuesta respecto a las mencionadas arriba residen en su claridad, simplicidad, modularidad, posibilidad de extensión y modificación y finalmente en el hecho de aprovechar las

herramientas disponibles en la comunidad MDD para la definición de la semántica de un lenguaje.

Bibliografía

[1] MDA Guide, v1. 0. 1, omg/03-06-01, June 2003. http://www. omg. org.

[2] OMG (Object Management Group) http://www. omg. org

[3] MOF 2. 0 Query/View/Transformations (QVT) - OMG Adopted Specification. March 2005.

[4] Meta Object Facility (MOF) 2. 0. OMG Adopted Specification. 2003. http://www. omg. org.

[5] OCL. The Object Constraint Language Specification – Version 2. 0, for UML 2. 0, revised by the OMG, http://www.omg.org, April 2004.

[6] Giandini R., Pons C., Pérez, G.. A two-level formal semantics for the QVT language. Memorias de la XII Conferencia Iberoamericana en "Software Engineering" (CIbSE). Medellín, Colombia, April 2009. ISBN: 978-958-44-5028-9, Pag: 73-86

[7] Jouault F., Kurtev I. Transforming Models with ATL Workshop in Model Transformation in Practice at the MoDELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005

[8] Marschall, F., Braun, P.: BOTL - The Bidirectional Object Oriented Transformation Language. Instituto de Informática, Universidad Técnica de Munich. Munich (2003)

[9] Agrawal, A., Kalmar, Z., Karsai, G., Shi, F., Vizhanyo, A.: GReAT User Manual.Nashville: Institute for Software-Integrated Systems, Vanderbilt University (2003)

[10] Sun Developer Network: Java Metadata Interface (JMI). SUN (2002) http://java.sun.com/products/jmi/

[11] Akehurst, D.H., Howells, W.G., McDonald-Maier K.D.: Kent Model Transformation Language. En: MoDELS 2005 Conference. Montego Bay, Jamaica (2005)

[12] M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework based on XSLT for model transformations. In WTUML'01, Proceedings of the Workshop on Transformations in UML, Genova, Italy, April 2001

[13] Model transformation- Inria. Mod-Transf ('04). http://modelware.inria.fr/rubrique15.htlm

[14] Eclipse org & Modelware. MOFScript (2005). http://www.eclipse.org/gmt/mofscript/

[15] Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDAFA 2004, University of Linkoping, Sweden, 2004, pp.14-28.

[16] Tratt, L. The MT model transformation language. In MT 2006, Proceedings of the 2006 ACM symposium on Applied computing, pages 1296 - 1303

[17] Akehurst D , Howells W. , McDonald-Maier K. Model Transformation Language. Workshop in Model Transformation in Practice - MoDELS 2005 Conference, Jamaica, Oct 3, 2005

[18] Program-Transformation.Org. Stratego: Strategies for Program Transformation. Program-Transformation (2004). http://www.strategolanguage.org/Stratego/WebHome

[19] Lawley M. , Steel J. Practical Declarative Model Transformation with TefKat. Workshop in Model Transformation in Practice - MoDELS 2005 Conference. Jamaica, Oct 3, 2005

[20] Willink, E. UMLX - A graphical transformation language for MDA. En: OOPSLA 2003 Conference. Anaheim, California (2003)

[21] eXecutable UML.OMG proposal (2005). http://www.omg.org/cgi-bin/doc?ad/2005-4-2

[22] http://www.monografias.com/trabajos30/paradigma-funcional/paradigma-funcional.shtml

[23] Anneke Kleppe. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional, 2008.

[24] Frederic Jouault and Jean B¶ezivin. KM3: a DSL for metamodel specification. In IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037, pages 171-185. Springer, 2006.

[25] Object Management Group. Meta object facility (MOF) 2.0 core ¯nal adopted specification. Technical report, Object Management Group, 2004.

[26] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. Eclipse Modeling Framework. Addison Wesley Professional, 2003.

[27] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, Patrick Valduriez: Model-based DSL frameworks. OOPSLA Companion 2006: 602-616, 2006.

[28] Kleppe, Anneke G. and Warmer Jos, and Bast, Wim. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[29] Stahl, T. and Völter, M. Model-Driven Software Development. John Wiley & Sons, Ltd. (2006)

[30] Domain Specific Modeling: Enabling Full Code Generation. Kelly,Steven; Tolvanen, Juha-Pekka. 2008.

[31] Czarnecki, Helsen. Feature-based survey of model transformation approaches. IBM System Journal, V.45, N3, 2006.

[32] Thompson Simon. Type Theory & Functional Programming. Computing Laboratory, University of Kent March 1999.

[33] Giandini, R. Un Marco Formal para Transformaciones en la Ingeniería de Software Conducida por Modelos. Facultad de Informática. Tesis de Doctorado. UNLP (2008).

[34] Pons C., Giandini R. y Pérez G. – Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica – Octubre 2008

[35] Eclipse http://www.eclipse.org.

[36] Xtext http://www.eclipse.org/Xtext

[37] Xtext User Guide - Copyright 2009 – 2010

[38] ATL http://www.eclipse.org/atl/

[39] ATL User Manual - 2006 by ATLAS group, LINA & INRIA Nantes

[40] The ATL Book to Publication transformation. Available at http://www.eclipse.org/gmt/atl/atlTransformations/.

[41] Lamda Calculo http://wapedia.mobi/es/C%C3%A1lculo_lambda

[42] Haskell *www.haskell.org/*

[43] EMFT - http://www.eclipse.org/modeling/emft/

[44] Boronat, A. and Meseguer, J.: An Algebraic Semantics for MOF. In: Fiadeiro J. and Inverardi P. (Eds.): FASE and ETAPS 2008, Budapest, Hungary. LNCS, vol. 4961, pp.377–391Springer, April 2008.

[45] Boronat, A, Cars´ý J., Ramos, I.: Algebraic specification of a model transformation engine. In: Baresi, Heckel,(eds.) FASE 2006 and ETAPS 2006. LNCS, vol. 3922, pp. 262–277. Springer, Heidelberg (2006)

[46] smartQVT. An open source model transformation tool implementing the MOF 2.0 QVTOperational language. http://smartqvt.elibel.tm.fr/. (2008)

[47] ModelMorf. A Model Transformer for the MOF 2.0 QVT-Relations language. http://www.tcs-trddc.com/ModelMorf/. (2008)