UNIVERSIDAD POLITECNICA DE VALENCIA

facultad de Informática. Valencia

ESTUDIO Y SOPORTE PARA TRANSFORMACIÓN, MANIPULACIÓN Y REGISTRO DE MODELOS EN UNA HERRAMIENTA DE GESTIÓN DE MODELOS

Proyecto final de carrera

Septiembre 2006. Valencia

Realizado por:

Luis Hoyos Cuesta

Dirigido por:

Artur Boronat Moll

Dr. José Á. Carsí Cubel

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

ISSI

MOMENT
A framework for Model Management

A mis padres y hermano

Agradecimientos

A Artur Boronat Moll, por su incansable estímulo y tenaz colaboración desde el inicio de este proyecto y por ser ejemplo de muchas virtudes que debe tener un investigador.

A Abel Gómez Llana por su constante y eficaz colaboración desde el principio en este trabajo. Sin su apoyo y enseñanzas, éste no habría visto la luz.

A Pascual Queralt, por su excelente profesionalidad y colaboración en la depuración del analizador de programas QVT Relations.

A Joaquín Oriente Cantos y Gabriel Merín Cubero por su compañerismo.

Al Prof. Dr. D. Isidro Ramos Salavert y al Prof. Dr. D. José Ángel Carsí Cubel por el apoyo de palabra y obra que desde el principio me han brindado y por sus lecciones de ciencia y honradez.

A la Dra. Dña. Agueda Cuesta García por sus valiosos consejos de edición y presentación del contenido de este trabajo.

A Isabel Gómez Sánchez por su paciencia y apoyo incondicional.

Financiación

Este proyecto ha sido desarrollado con una beca de colaboración a tiempo parcial en el grupo de investigación “Ingeniería del Software y Sistemas de Información” (ISSI) perteneciente al “Departamento de Sistemas Automáticos y Computación” (DSIC) de la Universidad Politécnica de Valencia.

"¿Me podrías indicar, por favor, hacia dónde
tengo que ir desde aquí?"
"Eso depende de a dónde quieras llegar"

L. Carroll
Alicia en el País de las Maravillas

ÍNDICES

Página

Figuras Página

Tablas Página

I. INTRODUCCIÓN

1. INGENIERÍA DIRIGIDA POR MODELOS Y POR ARQUITECTURA

La ingeniería del software ha permitido a los desarrolladores construir sistemas más complejos y fiables a lo largo de los años. El número de líneas de código que forman un sistema se ha incrementado significativamente en los últimos años, pasando de las diez mil líneas, a los diez millones en la actualidad [Jéz03].

Diferentes técnicas han permitido a los desarrolladores tratar la creciente complejidad de los sistemas software. En primer lugar, metodologías como *Rational Unified Process* (RUP), *Structured Analysis and Design Technique* (SADT), Catalysis B o *Extreme Programming*, definen claramente cada paso del proceso de desarrollo. En segundo lugar, mecanismos para elevar el nivel de abstracción, como la programación funcional, orientación a objetos, *middleware*, o aspectos, han permitido a los desarrolladores encapsular mejor la complejidad de los sistemas, y en consecuencia, producir programas más modulares, reusables y extensibles. En tercer lugar, la verificación de software y las pruebas han ayudado a reforzar la calidad de los sistemas finales.

En este sentido, la ingeniería dirigida por modelos (*Model-Driven Engineering*– MDE) intenta organizar los nuevos esfuerzos en estas direcciones proponiendo un marco (1) para definir metodologías claramente, (2) para desarrollar sistemas a cualquier nivel de abstracción, y (3) para organizar y automatizar las actividades de prueba y validación [Fond04].

Además, esta técnica establece que cualquier especificación debe ser expresada con modelos, lo que ofrece la ventaja de que son a su vez comprensibles por los humanos y las máquinas. Los modelos, dependiendo de qué representen, pueden residir en cualquier nivel de abstracción, y pueden ser limitados para dirigirse solo a ciertos aspectos del sistema. Puesto que son comprensibles por la máquina, un gran número de herramientas pueden automatizar (al menos parcialmente) ciertas tareas, (refactorización, refinamientos, generación de código).

Como consecuencia, el proceso de desarrollo de software se torna iterativo, refinando modelos abstractos en otros más concretos, y al final, generando el código completo automáticamente.

Diversas herramientas CASE (*Computer Aided Software Engineering*) se desarrollaron a final de los ochenta con la idea de facilitar los procesos de desarrollo y mantenimiento software. No obstante, a parte de la dificultad de elección de la herramienta idónea, existen diversos problemas añadidos: falta de control de versiones, características de trazabilidad, poca reusabilidad, o imposibilidad de sincronización entre los modelos y sus implementaciones.

Además, otro problema importante es la imposibilidad de hacer que estas herramientas CASE interoperen entre ellas. Al final, solo unas pocas herramientas (que imponen un método muy específico con notaciones bien definidas) serán capaces de acompañar al sistema a lo largo de su tiempo de vida.

No obstante, esto conlleva un problema: el ciclo de vida de un sistema puede ser extremadamente largo (incluso más de treinta años [Fond04]). Depender en este caso de una única herramienta CASE (o un pequeño conjunto) es a menudo inaceptable, ya que pueden dejar de estar soportadas a lo largo del periodo. Una

solución, es desarrollar según una serie de estándares que todas las herramientas CASE deberán usar a lo largo del ciclo completo de vida de los sistemas software.

Para tratar algunos de estos problemas en el contexto de MDE, el grupo *Object Management Group* (OMG) ha lanzado la iniciativa *Model Driven Architecture* (MDA), tratando de reunir y definir todas las especificaciones necesarias para proporcionar la aproximación MDA al desarrollo de software.

Estas especificaciones intentan, precisamente, definir qué lenguaje debe ser utilizado para expresar modelos, cómo especificar las transformaciones de éstos, cómo intercambiar modelos, cómo almacenarlos y hacer que los modelos evolucionen, y, más recientemente, cómo generar código. No obstante, puesto que MDA está aún en sus primeras fases, y que todavía no hay apenas aplicaciones industriales que usen MDA, algunas de estas especificaciones carecen de cierta precisión (otras incluso faltan).

Para superar algunos problemas técnicos, como la interoperabilidad, el control de versiones, o las transformaciones; la estandarización parece ser la única solución correcta. Sin embargo, dada la libertad que MDE proporciona a los usuarios, la estandarización no resuelve problemas como la complejidad y precisión, que son inherentes a la metodología proporcionada y sus notaciones asociadas.

1.1. Aplicación de un proceso MDE

La idea que promueve MDE es usar modelos a diferentes niveles de abstracción para desarrollar los sistemas. De esta manera, la principal actividad de los desarrolladores MDE es diseñar modelos, como los que usaban para desarrollar código, pero ahora guiados por una metodología.

La ventaja de tener un proceso MDE es que éste debe definir claramente cada paso a dar, forzando a los desarrolladores a seguir la metodología definida. Debe especificar la secuencia de modelos a desarrollar, y cómo derivar un modelo a partir de otro del nivel de abstracción inmediatamente superior. Proporcionando a los desarrolladores una metodología como esta, podrán saber en cualquier momento a lo largo del proceso de desarrollo, qué se debe hacer en cada paso de desarrollo y cómo conseguirlo.

La aplicación de un proceso MDE se muestra en la Figura 1. El sistema en desarrollo es descrito en primer lugar por un modelo a un alto nivel de abstracción, esto es, ignorando cualquier tipo de dependencia de la plataforma. Este modelo es llamado *Platform Independent Model* (PIM) en la terminología MDA. Un buen candidato para el PIM son, por ejemplo, los casos de uso. Posteriormente, se deben realizar una serie de refinamientos interactivos con el objetivo de hacer el sistema más específico de la plataforma en cada paso. Por ejemplo, en el siguiente paso, el sistema podría ser expresado de nuevo, pero de forma más precisa. En este caso se podría describir mediante diagramas de clases y diagramas de estados, para mostrar el comportamiento del sistema.

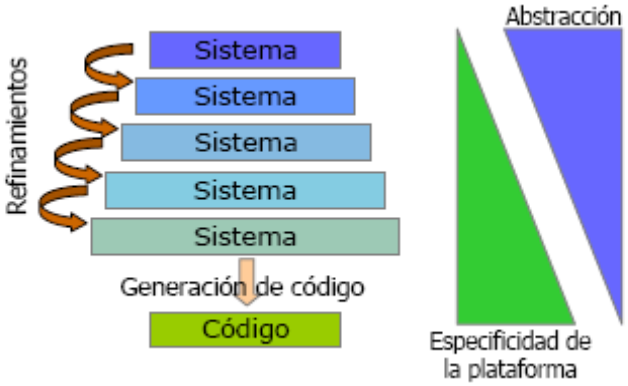


Figura 1. Aplicación de un proceso MDE.

En la terminología introducida por MDA, un modelo refinado se llamará *Platform Specific Model* (PSM), siendo el PIM el modelo del nivel de abstracción inmediatamente superior (que fue el modelo fuente para el paso de refinamiento correspondiente).

En cada paso del proceso MDE la información relacionada con la calidad puede ser integrada también, como verificación, validación y generación de casos de prueba. Una verificación puede ser comprobar que un modelo específico a plataforma no rompe la especificación descrita por su modelo independiente de plataforma, o viceversa, en el caso de ingeniería inversa. Un paso de la validación puede permitir a los desarrolladores del sistema (o incluso los clientes) instanciar prototipos de los modelos intermedios con el objetivo de probar sus funcionalidades antes de que el sistema esté implementado por completo. La generación de casos de prueba de forma automática puede producir resultados para diversos escenarios, esto es, conjuntos de mensajes que serán enviados y recibidos por el sistema en cuestión, permitiendo de esta forma probar su implementación actual.

Una de las ventajas más importantes de usar el proceso MDE es su adaptabilidad a los cambios. Cuando un cambio ocurre, siendo en el mayor nivel de abstracción (por ejemplo, un cambio en los requerimientos del sistema) o en el menor nivel de abstracción (por ejemplo, portándolo a otra plataforma, como moverlo de PostgreSQL a MySQL), su impacto está bien localizado y las partes que no son afectadas por el cambio son inmediatamente reusables. No obstante, los refinamientos, deben ser realizados una vez más para actualizar las partes cambiantes. Esto se vuelve más problemático cuando el lenguaje de modelado cambia puesto que estos re-refinamientos no son posibles directamente.

2. META OBJECT FACILITY

Tal y como se ha comentado, el grupo OMG ha propuesto un marco de trabajo en el ámbito de la ingeniería de modelos denominado MDA. MDA es un proceso de desarrollo de software y pretende establecerse como un estándar *de facto* en este ámbito. Por lo tanto el objetivo es producir sistemas informáticos ejecutables.

La *Meta Object Facilty* (MOF) [MOF] es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo. Mediante MOF se pueden definir nuevos metamodelos, y por lo tanto nuevos vocabularios (de hecho se podría decir lenguajes, pero es conveniente no utilizar el término para evitar confusiones) con las mismas

herramientas con que se definen modelos. Por otra parte, cabe preguntarse si existe un vocabulario de modelos superior que se utiliza para definir metamodelos. La respuesta es que sí, a este metamodelo de metamodelos se le denomina metametamodelo. Pero como también es un modelo, ¿se podría seguir extendiendo esta pirámide de forma infinita?. En la práctica esto no tiene sentido, y los metamodelos y modelos se suelen organizar en una estructura de cuatro capas M3-M0 (ver Figura 2) con la siguiente distribución:

- El nivel inferior, denominado M0, es en el que se sitúan los datos, es decir las instancias del sistema bajo estudio.
- En el nivel M1 se sitúan los modelos, tal y como los hemos introducido aquí, descripciones abstractas de un sistema
- En la capa inmediatamente superior, denominada M2, se sitúan los metamodelos, "vocabularios para definir modelos".
- El nivel M3, que cierra la estructura por arriba, contiene el vocabulario base que permite definir metamodelos. Cabe resaltar que este nivel suele contener un único vocabulario, que caracteriza la aproximación de modelos escogida. Es imperativo que este vocabulario o metametamodelo esté definido utilizando como vocabulario a sí mismo, de ahí que se cierre la estructura.

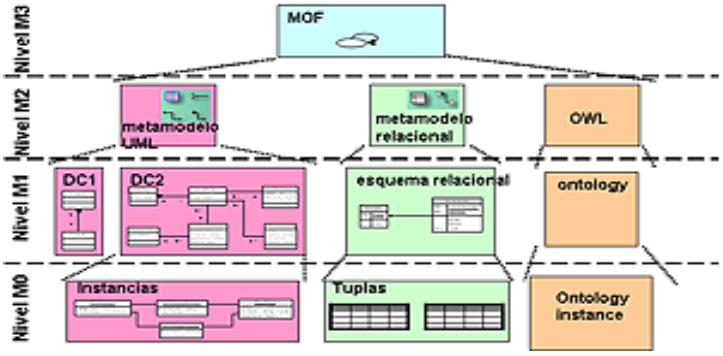


Figura 2. Jerarquía MOF.

Esta estructura de cuatro capas permite conseguir una gran riqueza de vocabularios para describir distintos tipos de sistemas, o bien para proporcionar diversos puntos de vista de un mismo sistema.

Resulta interesante destacar que esta asignación fija de niveles puede resultar confusa en ocasiones. Quizá es más interesante fijar como idea fundamental la relación entre un modelo y su vocabulario, y darse cuenta de que esta relación ocurre en todos los niveles descritos. Esta relación se denomina "reificación". Decimos que un modelo "x" es una reificación de un vocabulario "x+1", al que denominamos metamodelo. El modelo está en el nivel inferior, nivel de reificación; y el metamodelo en el superior, nivel meta. Podemos aplicar esta dualidad al metamodelo "x+1" ya que si ahora lo situamos en el nivel de reificación, vemos también que "x+1" necesariamente está definido por un vocabulario "x+2". Por lo tanto podemos situar un modelo tanto en el nivel meta y decir que tiene reificaciones, como en el nivel de reificación, y decir que proviene de un metamodelo. Cabe resaltar el caso especial del metametamodelo (nivel M3) que se define a sí mismo, por lo tanto se podría decir que es una reificación de sí mismo.

MDA sitúa en la capa M2 diversos metamodelos bien conocidos que están definidos mediante MOF, como por ejemplo:

- UML, que proporciona un vocabulario para describir gran cantidad de sistemas.
- UML se caracteriza por ser un vocabulario independiente de dominio, si bien tiene sus raíces en el modelado a objetos.
- CWM, un vocabulario específico para el dominio de los sistemas relacionados con la minería o explotación de datos.
- QVT, un vocabulario que extiende OCL para expresar relaciones entre modelos.

3. GESTIÓN DE MODELOS

En un proceso de ingeniería MDE se parte de un conjunto de modelos que describen el sistema de interés de manera abstracta. A partir de estos modelos, y mediante una serie de procesos de refinamiento y transformación, se pretende obtener de manera automática el artefacto software ejecutable final. Es en estos procesos donde se centra el trabajo del ingeniero MDE. Mientras que los modelos serán creados por analistas o especialistas de dominio, el ingeniero MDE debe encargarse de establecer los denominados *mappings* o relaciones de transformación que permitirán refinar los modelos originales, produciendo como resultado el sistema informático requerido en la tecnología de implementación deseada.

Con tan sólo sustituir estos *mappings* es posible obtener el sistema en otra tecnología de implementación.

Bien, hasta ahora este es el punto crítico donde muestra señales de flaqueza la aproximación MDE, ya que no existe ningún proceso unívoco que garantice la obtención del sistema software requerido. En realidad se podría decir que MDE se ha visto perjudicada por la falta de un proceso estándar o una metodología aceptada para solventar este paso, ya que las numerosas aproximaciones *ad-hoc*, poco documentadas, sólo han conseguido minar la confianza de los expertos en MDE.

Uno de los principales problemas encontrados es la falta de infraestructuras y herramientas de soporte que permitan, no sólo crear y trabajar con estos *mappings*, sino manipular modelos en general. En el contexto de los lenguajes de programación estamos acostumbrados a una gran variedad de lenguajes, potentes entornos integrados de programación, herramientas para gestionar versiones del código y otras facilidades que automatizan gran parte del trabajo. En el contexto de la ingeniería de modelos ocurre todo lo contrario, y es que a pesar de la gran variedad de herramientas para trabajar con UML, debe tenerse en cuenta que UML no es más que un metamodelo concreto y que las herramientas disponibles no permiten trabajar con otros metamodelos ni permiten producir soluciones genéricas.

De ahí que la situación más común es utilizar un lenguaje orientado a objetos para representar estos modelos y manipularlos mediante esa representación. Las actividades de manipulación incluyen diseñar correspondencias entre modelos, modificar modelos o *mappings*, generar un modelo a partir de otro basándose en un *mapping* o generar la representación equivalente de un modelo en otro metamodelo.

Evidentemente este esquema de trabajo es muy costoso y poco reutilizable, ya que generalmente las soluciones creadas no son lo suficientemente genéricas para ser aplicables a más de un metamodelo, y la interoperabilidad entre soluciones elaboradas por distintas partes es poco menos que imposible. Estas soluciones *ad-hoc* son costosas de implementar debido a la escasa ayuda proporcionada por los entornos de

desarrollo actuales poco familiarizados con modelos, y costosas de rentabilizar, ya que continuamente aparecen nuevas aproximaciones o soluciones MDE y cuando, inevitablemente, se hace necesario cambiar de tecnología, resulta difícil reutilizar el trabajo realizado anteriormente.

En este contexto ha surgido una nueva disciplina denominada Gestión de Modelos (en inglés, *Model Management*). Esta disciplina, introducida por P. Bernstein en [Ber00], pretende proporcionar una infraestructura específica y modelos de forma genérica y reutilizable.

Se dice "genérica" en el sentido de que las herramientas proporcionadas sean aplicables a cualquier metamodelo, y entre metamodelos. Por otra parte, pretende ser "reutilizable" en el sentido de que un conjunto de procesos definidos para un metamodelo sean aplicables a modelos de otro metamodelo con modificaciones mínimas. De esta forma se proporcionaría una base común para la creación de herramientas de manipulación de modelos, reduciendo los costes y facilitando la interoperabilidad. Además se facilitaría el surgimiento de procesos estandarizados de desarrollo dentro del contexto MDE.

Para conseguirlo, la gestión de modelos considera a los modelos como ciudadanos de primer orden. Se trata de proporcionar operadores y abstracciones que permitan manipular a los modelos de forma directa y genérica. En la literatura se discuten los operadores que permitirían mejorar la productividad [Ber03], algunos ejemplos son:

- El operador *ModelGen* que toma un modelo A y lo proyecta en otro metamodelo, obteniendo un modelo B y un *mapping* entre A y B.
- El operador *Merge*, que toma dos modelos A y B y un *mapping* entre ellos y devuelve la unión de ambos y los *mappings* que relacionan al resultado con A y B.
- El operador *Diff*, que toma un modelo A y un *mapping* entre A y B y devuelve el submodelo de A que no pertenece al *mapping*.
- El operador *Match*, que toma dos modelos y obtiene una correspondencia (*mapping*) entre ellos.
- El operador *Compose*, que toma un *mapping* entre dos modelos A y B y un *mapping* entre dos modelos B y C y obtiene el *mapping* entre A y C.

3.1. Aproximaciones existentes

[RONDO]. Este sistema, desarrollado entre otros por P. Bernstein., representa los modelos en forma de grafos dirigidos, y facilita un conjunto de operadores de alto nivel para manipularlos, similares a los descritos anteriormente. La traducción de instancias de modelos como grafos se hace mediante unos conversores especiales, desarrollados para cada metamodelo en concreto. En RONDO, los operadores de manipulación están implementados de forma imperativa.

[AMMA]. En esta herramienta toda la información sobre los componentes conocidos en una plataforma dada, están almacenados en un modelo específico denominado "megamodel". Un "megamodel" es un tipo de modelo que guarda toda la información relacionada con referencias y meta información sobre cualquier recurso accesible, incluyendo relaciones entre estos recursos. Esto permite construir una

infraestructura mínima y altamente extensible. Por ejemplo, permite extender fácilmente una plataforma local a una plataforma distribuida, como por ejemplo un sistema P2P, sin modificaciones significantes en los mecanismos de interoperabilidad. Además, esta aproximación encaja bien con el esquema conceptual general desarrollado para gestión de modelos.

4. MOMENT: UN MARCO DE TRABAJO PARA GESTIÓN DE MODELOS

Basándose en la experiencia obtenida en la aplicación de formalismos de especificaciones algebraicas a la recuperación de sistemas legados [BoP04][BoC05], se ha propuesto el desarrollo de una herramienta que dé soporte algebraico a los operadores genéricos introducidos en la gestión de modelos. Esta herramienta se denomina MOMENT.

Con esta herramienta se pretende probar la falsedad del los mitos basados en la poca productividad de las herramientas formales y en el aumento del coste del proceso software debido a su uso. Como sistema formal, se ha elegido un eficiente sistema de reescritura de términos, Maude, que ya ha sido empleado en muchos ámbitos formales [Mar02]. Como entorno industrial para desarrollar la plataforma MOMENT, se ha utilizado Eclipse y el Eclipse Modeling Framework (EMF).

4.1. ¿Qué es Maude?

Maude es un lenguaje de programación de alto rendimiento que soporta la especificación y programación lógica tanto ecuacional como de reescritura para un amplio abanico de aplicaciones. Maude ha sido influenciado de una manera muy importante por el lenguaje OBJ3, que puede considerarse como un sublenguaje de Maude para la lógica ecuacional.

La lógica de reescritura es una lógica de cambios concurrentes que puede tratar fácilmente con estados y con computación concurrente. Tiene propiedades deseables tales como un marco general semántico para dar semántica ejecutable a un gran número de lenguajes y modelos de concurrencia. En particular, soporta muy bien la computación concurrente orientada a objetos.

- Maude soporta de una manera sistemática y eficiente la reflexión lógica. Esto permite a Maude ser un lenguaje extremadamente potente y extensible, permitiéndole soportar un álgebra de operaciones de composición de módulos extensible.
- Maude es potente ya que puede modelar casi todo, desde el conjunto de los números naturales hasta un sistema biológico donde se programe el lenguaje Maude a sí mismo. Se dice que cualquier cosa que se pueda escribir, hablar o describir mediante el lenguaje humano, se puede expresar con instrucciones Maude [MaudeMan].
- Maude es simple. Su semántica está basada en los fundamentos de la teoría de clases, lo cual es bastante intuitivo y directo. Comparado con la mayoría de los lenguajes, Maude no tiene apenas sintaxis que memorizar.

- Maude está bien establecido. Aunque, depende de cómo se mire, puede llegar a ser extremadamente abstracto. Su diseño permite tanta flexibilidad que la sintaxis puede parecer más bien abstracta.
- Maude es desafiante. Puede llegar a solucionar lo más difícil ó complejo. Esto desafía al programador a ser astuto, en vez de resolver el problema afrontándolo con una serie de variables globales y funciones que a menudo son un caos de por sí.

Aunque Maude es un intérprete, su rendimiento es tal que puede ser usado en aplicaciones serias con un rendimiento competitivo y muchas ventajas sobre código convencional. Ilustramos esto con un ejemplo. Un componente, que se usaba para probar si una traza de eventos satisfacía cierta fórmula dada en lógica lineal temporal (LTL), fue escrito en Maude para un proyecto de la NASA [MaudeMan]. El componente tenía una prueba trivial de corrección, ocupaba apenas una página y fue desarrollado en unas horas. Este componente reemplazó un componente similar escrito en Java que tenía aproximadamente 5000 líneas y que llevó más de un mes para ser desarrollado por un programador con experiencia. El código Java traducía una formula LTL en un autómata de büchi y era aproximadamente tres veces más lento que el código Maude. La implementación actual de Maude puede ejecutar reescrituras sintácticas con velocidades típicas de medio millón a varios millones de reescrituras por segundo, dependiendo de la aplicación particular y la máquina en la que funcione (la estimación anterior supone un Pentium a 900Mhz). La razón por la cual el intérprete de Maude consigue alto rendimiento es que las reglas de reescritura son cuidadosamente analizadas y semicompiladas mediante algoritmos muy eficientes.

Se llama Core Maude al intérprete de Maude 2.0 implementado en C++ y que provee toda la funcionalidad básica del lenguaje.

Full Maude es una extensión de Maude, escrito en Maude, que dota a Maude de un potente y extensible álgebra de módulo en el cual los módulos de Maude pueden ser combinados conjuntamente para construir módulos más complejos. Los módulos pueden ser parametrizados, y pueden ser instanciados usando los "*views*" (vistas). Los parámetros son teorías especificando los requerimientos semánticos para una correcta instanciación. Las teorías mismas pueden ser parametrizadas. Módulos orientados a objetos (que también pueden ser parametrizados) soportan objetos, mensajes, clases y herencia. También es posible subir y bajar en la torre de reflexión usando comandos de Full Maude.

En cuanto a las capacidades de parametrización que proporciona Full Maude, cabe reseñar que para futuras versiones, se proporcionará soporte para ella directamente en Core Maude. Es destacable apuntar esto, ya que MOMENT se apoya de manera muy importante en este mecanismo, por lo que actualmente se está trabajando en portar el álgebra escrita en Full Maude a Core Maude con soporte para parametrización (actualmente en versión Core Maude alpha86a, que dará origen a Core Maude 2.2) ya que proporciona importantes mejoras en la eficiencia.

4.2. Eclipse y Eclipse Modeling Framework

A pesar de la potencia de cálculo que Maude proporciona, presenta importantes carencias en cuanto a la interacción con el usuario se refiere. Por otra parte, MOMENT pretende integrar el formalismo en un entorno de modelado industrial. En este caso, la plataforma elegida es Eclipse, ya que proporciona un *framework* de modelado como es EMF [EMF].

Esta integración de un método formal en una herramienta industrial de desarrollo de software, combina los esfuerzos que se están realizando sobre ambas herramientas en direcciones divergentes: en Maude sobre aspectos teóricos, y en EMF sobre su aplicación a la Ingeniería del Software. De esta manera, evitamos el aislamiento de Maude de ámbito industrial, debido a la premisa "constrúyelo y ya vendrán", frecuentemente mantenida por los desarrolladores de métodos formales. Este hecho permite utilizar Maude para solucionar problemas reales en la Ingeniería del Software, sin limitarse únicamente la solución de los llamados ejemplos de juguete.

Eclipse es un proyecto de desarrollo software de código abierto, cuyo propósito es proporcionar una plataforma de herramientas altamente integradas. El trabajo en Eclipse consiste en un proyecto central que incluye un *framework* genérico para la integración de herramientas, y un entorno de desarrollo Java construido usando el *framework* anterior. Otros proyectos extienden el *framework* núcleo para soportar tipos de herramientas y entornos de desarrollo específicos, entre los que encontramos EMF. Los proyectos en Eclipse se implementan en Java y se ejecutan en diversos sistemas operativos, incluyendo Windows y Linux.

Eclipse.org es un consorcio de diversas compañías que se han comprometido a proporcionar soporte al proyecto Eclipse en términos de tiempo, experiencia, tecnología o conocimiento. Los proyectos que conforman Eclipse operan bajo un organigrama bien definido que marca los roles y responsabilidades de los diversos participantes, incluyendo el consejo, los usuarios de Eclipse, los desarrolladores y los comités de gestión de proyectos.

4.2.1. Los proyectos

El trabajo de desarrollo en Eclipse está dividido en tres proyectos principales: el Proyecto Eclipse (*Eclipse Project*), el Proyecto de Herramientas (*Tools Project*) y el Proyecto de Tecnología (*Technology Project*). El proyecto Eclipse contiene los componentes básicos necesarios para desarrollar utilizando Eclipse. Sus componentes son esencialmente fijos, y se pueden descargar como una unidad referida como *Eclipse SDK (Software Development Kit)*. Los componentes de los otros dos proyectos se utilizan para propósitos específicos y son generalmente independientes, y se descargan de forma separada.

4.2.1.1. El proyecto Eclipse

El proyecto Eclipse proporciona soporte para el desarrollo de una plataforma, o un *framework*, para la implementación de entornos de desarrollo integrados (IDEs). El *framework* de Eclipse se implementa en Java, pero también se emplea para implementar herramientas para otros lenguajes (por ejemplo, C++ o XML).

A su vez, este proyecto se divide en tres subproyectos. En primer lugar, la plataforma es el componente básico de Eclipse, y se considera a menudo que es Eclipse. Define los *frameworks* y servicios requeridos para proporcionar el soporte a la estructura de *plug-ins* y a la integración de herramientas. En segundo lugar, el JDT es un entorno de desarrollo Java completamente funcional construido usando Eclipse. Por último, el PDE proporciona vistas y editores para facilitar la creación de *plug-ins* para Eclipse. El PDE construye y extiende el JDT proporcionando soporte para las partes no-Java del *plug-in* en la actividad de desarrollo del mismo, como registrar las extensiones del *plug-in* y demás.

4.2.1.2. El proyecto de herramientas

El proyecto de herramientas de Eclipse define y coordina la integración de diferentes conjuntos o categorías de herramientas basadas en la plataforma Eclipse.

El proyecto de herramientas de desarrollo C/C++ (CDT), por ejemplo, comprende el conjunto de herramientas que definen un IDE C++. Los editores gráficos usando el *framework* de edición gráfica (GEF y GMF), y los editores basados en modelos utilizando EMF, representan categorías de las herramientas de Eclipse para las cuales se proporcionan soporte en los subproyectos de herramientas.

4.2.1.3. El proyecto de tecnología

El proyecto de Tecnología de Eclipse proporciona una oportunidad a los investigadores, académicos y educadores para involucrarse en la continua evolución de Eclipse.

4.2.2. La plataforma Eclipse

La plataforma Eclipse es un *framework* para construir IDEs. Se describe como "un entorno de desarrollo integrado para todo y nada en particular" [EclOv03]. Simplemente define la estructura básica de un IDE. Herramientas específicas extienden este *framework*, y se "enchufan" en él para definir un IDE particular colectivamente.

4.2.2.1. Arquitectura de *plug-ins*

La unidad básica de función, o un componente, se denomina *plug-in* en Eclipse. La plataforma Eclipse misma, y las herramientas que la extienden se componen de *plug-ins*. Una sola herramienta puede consistir en un único *plug-in*, pero herramientas más complejas se dividen típicamente en varios.

Desde una perspectiva de empaquetado, un *plug-in* incluye todo lo necesario para ejecutar un componente, como código Java, imágenes, texto traducido, etc. También incluye un archivo de manifiesto, llamado "plugin.xml", que declara las interconexiones con otros *plug-ins*. Indica, entre otras cosas, las siguientes:

- Requiere (*Requires*) - sus dependencias con otros *plug-ins*.
- Exporta (*Exports*) - la visibilidad de sus clases públicas a otros *plug- ins*.
- Puntos de extensión (*Extensión points*) - declaraciones de funcionalidad que hace disponibles a otros *plug-ins*.
- Extensiones (*Extensions*) - su uso de los puntos de extensión de otros *plug-ins*.

Al arrancar, la plataforma Eclipse descubre todos los *plug-ins* disponibles y casa las extensiones con sus correspondientes puntos de extensión.

4.2.2.2. Recursos del espacio de trabajo (*Workspace*)

Las herramientas integradas en Eclipse trabajan con carpetas y archivos ordinarios, pero utilizando una API de más alto nivel, basada en recursos, proyectos y un espacio de trabajo. Un recurso es la representación de Eclipse de un archivo o una carpeta.

Un proyecto es un tipo especial de carpeta que corresponde con una carpeta especificada por el usuario en el sistema de archivos subyacente. Las subcarpetas del proyecto son las mismas que en el sistema de archivos físico, pero los proyectos son las carpetas de más alto nivel en el contenedor virtual llamado espacio de trabajo (*workspace*).

4.2.2.3. El *framework* UI

El *framework* UI de Eclipse consiste en dos conjuntos de herramientas de propósito general. Como SWT, o JFace. SWT (*Standard Widget Toolkit*) es un conjunto de librerías gráficas independientes del sistema operativo, implementadas utilizando componentes nativos siempre que sea posible. Por otra parte, *JFace* (implementado utilizando SWT), proporciona un sistema de clases de visores que proporcionan una conexión de más alto nivel con los datos que muestran. Otra parte notable es el *framework* de acciones, que se utiliza para agregar comandos a los menús y a las barras de herramientas.

4.3. Espacios tecnológicos

Un espacio tecnológico (ET) es un contexto de trabajo en el que se dispone de un conjunto de conceptos bien definidos, una base de conocimiento, herramientas, y una serie de posibilidades de aplicación específicas [Ber03]. Un espacio tecnológico además suele ir asociado a una comunidad de usuarios/investigadores bien reconocida, un soporte educacional, una literatura común, terminología y saber hacer. Ejemplos de espacios tecnológicos son el ET XML, el ET DBMS, el ET de las sintaxis abstractas, el ET de las ontologías, el ET de MOF/MDA, en el que se enmarca UML, y el ET de EMF, que guarda un gran parecido con el anterior.

4.3.1. Puentes tecnológicos

Cada espacio tecnológico tiene unas características que le hacen especialmente apropiado para resolver un tipo de problemas. Sin embargo, muchas veces, lo más apropiado es trabajar con varios ETs a la vez. Para ello existen o es posible definir enlaces o puentes entre espacios. Por ejemplo son bien conocidos los puentes de MDA al ET de sintaxis abstractas, o de UML al ET XML a través de XMI.

Un puente entre espacios puede ser bidireccional, como en los ejemplos comentados, o unidireccional, cuando no es posible reconstruir el artefacto origen.

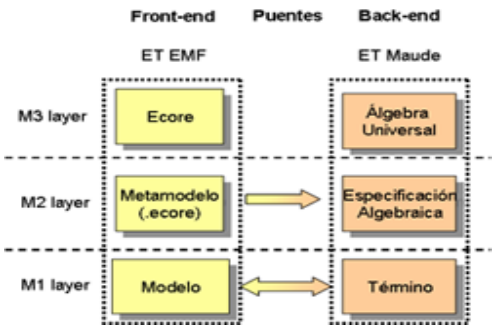


Figura 3. Puentes tecnológicos en MOMENT.

En MOMENT los operadores de gestión de modelos han sido especificados algebraicamente utilizando el formalismo Maude. El ET de Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET también puede ser visto como un paradigma de modelado, considerando el álgebra universal de Maude como el lenguaje de definición de metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

MOMENT representa un modelo como una estructura de términos algebraicos, caracterizados por una especificación algebraica que proviene del metamodelo. Para poder utilizar MOMENT desde la ingeniería de modelos será necesario disponer de unos puentes tecnológicos entre ambos espacios tecnológicos. El primer problema que este proyecto resuelve es la definición y construcción de estos puentes tecnológicos entre el ET de Maude y el ET de EMF.

Los puentes creados como parte de este proyecto permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF. Se ha escogido EMF dentro del campo MDE por su interoperabilidad. Se espera que EMF sea una puerta de entrada a otros entornos MDE, y que de esta manera el trabajo realizado sirva para habilitar de manera lo más completa posible la interoperabilidad de MOMENT con MDE.

4.4. Operadores de MOMENT

MOMENT proporciona un conjunto de operadores genéricos para manipular modelos. Estos operadores se clasifican en dos grupos dependiendo de si los modelos a manipular son de trazabilidad o no.

4.4.1. Operadores comunes

Denominamos operadores comunes a aquellos que son aplicables a cualquier tipo de modelo y poseen una semántica basada en teoría de conjuntos. Los operadores comunes de MOMENT son los siguientes:

- El operador *Merge*, que toma como parámetros de entrada un modelo “A” y un modelo “B”, y devuelve como resultado un modelo “C”, que es la unión de los elementos de lo modelos “A” y “B”, y dos modelos de trazabilidad “mapAC” y “map BC”, uno por cada modelo de entrada.
- El operador *Diff*, que toma como parámetros de entrada un modelo “A” y un modelo “B”, y devuelve como resultado un modelo “C”, que es la diferencia o resta de elementos entre el modelo “A” y el modelo “B”, y un solo modelo de trazabilidad “mapAC”.
- El operador *Cross*, que toma como parámetros de entrada un modelo “A” y un modelo “B”, y devuelve como resultado un modelo “C”, que es la intersección de los elementos de los modelos de “A” y “B”, y un dos modelos de trazabilidad “mapAB” y “mapBC”, uno por cada modelo de entrada.
- El operador *ModelGen*, que toma como modelo de entrada un modelo “A” y lo proyecta en otro metamodelo, obteniendo un modelo “B” y un modelo de trazabilidad entre “A” y “B”.

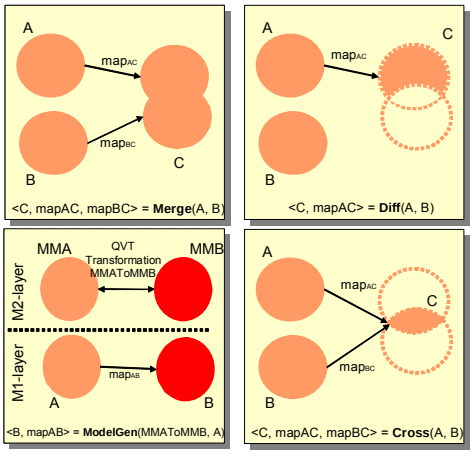


Figura 4. Operadores comunes de MOMENT.

4.4.2. Operadores de soporte para la navegación

Este tipo de operadores son más específicos que los anteriores, y se han diseñado para tratar con modelos de trazabilidad (modelos de enlaces que permiten relacionar elementos entre dos modelos distintos). Los operadores de trazabilidad considerados en MOMENT son:

- *Range*. Este operador obtiene, dados un modelo de trazabilidad (mapAB), un modelo dominio (A'), y un modelo rango (B), un submodelo del modelo rango B (B') cuyos elementos se relacionan con los de A'.
- *RestrictDomain.* Recibiendo un modelo dominio (A') y un modelo de trazabilidad (mapAB), el operador devuelve un modelo con los enlaces de trazabilidad de "mapAB" que tienen elementos del modelo A' como elementos dominio.
- *Compose*. Este operador realiza la composición de dos modelos de trazabilidad. Dados tres modelos A, B y C, y los modelos de trazabilidad "mapAB" y "mapBC" que relacionan respectivamente A con B y B con C; el operador *Compose* obtiene el modelo "mapAC" haciendo explícita la relación entre los elementos de A y C por la propiedad transitiva.

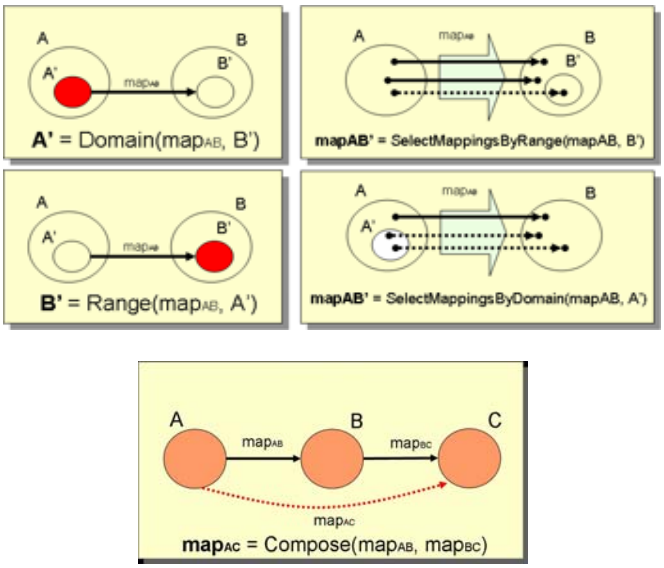


Figura 5. Operadores de trazabilidad en MOMENT.

II. OBJETIVOS

El objetivo principal de este proyecto es implementar en la herramienta de gestión de modelos MOMENT el soporte necesario para transformar, manipular y registrar modelos. Para ello se integrarán diferentes lenguajes específicos de dominio (DSLs) que permitan soportar estas acciones. Concretamente, estos DSLs serán dos: un lenguaje de especificación de transformaciones entre modelos, QVT-Relations, y un lenguaje específico de dominio diseñado en MOMENT, que permite la especificación de operadores complejos destinados a manipular modelos.

1. SOPORTE PARA TRANSFORMACIONES ENTRE MODELOS

Los objetivos perseguidos para dar soporte a la transformación de modelos, se resumen en integrar en la plataforma MOMENT el lenguaje QVT-Relations, de manera que a partir de una especificación textual QVT-Relations, se construya un modelo QVT-Relations que sea utilizado por MOMENT para ejecutar la transformación. Para ello, es necesario realizar las siguientes tareas:

- Desarrollar un analizador (*parser*) de programas QVT-Relations.
- Construir modelos QVT-Relations a partir de programas QVT-Relations analizados.

2. SOPORTE PARA MANIPULACIÓN DE MODELOS

Los objetivos perseguidos para dar soporte a la manipulación de modelos en MOMENT, se basan en proporcionar una serie de utilidades y herramientas que permitan dar soporte al DSL de definición de operadores complejos de MOMENT. Para ello, se realizarán las siguientes tareas:

- Desarrollar un editor textual de especificaciones textuales de operadores complejos.
- Desarrollar un *parser* de especificaciones textuales de operadores complejos.
- Construir el modelo de operador complejo asociado a una especificación textual de un operador complejo
- Desarrollar un menú emergente que permita el análisis de un fichero que contiene una especificación textual de operador complejo

3. SOPORTE PARA REGISTRO DE MODELOS

Los objetivos perseguidos para dar soporte al registro de modelos se resumen en integrar en la arquitectura de MOMENT un componente (MOMENT *Registry*), que permita el registro de los diferentes modelos que tienen cabida en MOMENT y que además pueda suministrar estos modelos registrados a aquellos componentes que los necesiten. Para ello, es necesario llevar a cabo las siguiente tareas:

- Desarrollar un repositorio de los artefactos metamodelos, transformaciones, operadores y relaciones de equivalencia vistos como modelos

- Proporcionar una capa intermedia entre la persistencia física de los artefactos software y la capa lógica (de utilización)
- Construir una interfaz gráfica que permita interactuar al usuario con el repositorio
- Implementar los métodos necesarios en el repositorio para el manejo de artefactos y la interacción con otros componentes o herramientas de MOMENT.

III. LENGUAJES ESPECÍFICOS DE DOMINIO

1. INTRODUCCIÓN A LOS DSLs

Tradicionalmente se han usado lenguajes de propósito general (GPL) para resolver cualquier tipo de problema software. En los últimos tiempos, ha surgido un estilo de desarrollo software con la finalidad de describir sistemas software utilizando lenguajes específicos de dominio (*Domain Specific Language* – DSL). La idea básica de un lenguaje específico de dominio (DSL) es un lenguaje destinado para solucionar un tipo de problema concreto. Como se desarrollará en los puntos 1.1, 1.2 y 1.3, un DSL puede ser visto desde tres perspectivas diferentes: lenguaje de programación, lenguaje de especificación y arquitectura software.

1.1. DSL – Lenguaje de programación

Un lenguaje específico de dominio puede verse como un lenguaje de programación dedicado a resolver un problema concreto. Un DSL proporciona construcciones abstractas y notaciones apropiadas, constituyendo un lenguaje pequeño, más declarativo que imperativo y menos expresivo que un lenguaje de propósito general.

Por ejemplo, los *shells* de Unix pueden considerarse como DSLs cuyas abstracciones y notaciones de dominio incluyen *streams* (como entrada estándar o salida estándar) y operaciones sobre *streams* (como redirecciones y tuberías). Los *shells* también ofrecen una interfaz sencilla para ejecutar y controlar procesos, y mecanismos de control de flujo y manipulación de cadenas de caracteres.

Comúnmente se han utilizado términos como micro lenguajes, lenguajes de aplicación o lenguajes de muy alto nivel, para referirse a los lenguajes específicos de dominio.

1.2. DSL – Lenguaje de especificación

Puesto que los lenguajes específicos de dominio pueden ser altamente declarativos y ocultar muchos detalles de implementación, algunos DSLs pueden considerarse más como lenguajes de especificación que como lenguajes de programación. Frecuentemente, estos DSLs son ejecutables. Sus puntos fuertes siguen siendo abstracciones y notaciones específicas, así como una potente expresividad restringida al dominio del problema.

Por ejemplo, considerando el comando *make* de Unix, que permite mantener programas, determina automáticamente qué partes de un programa necesitan ser recompiladas, y da los comandos necesarios para realizarla. El lenguaje de los *makefiles* es pequeño y principalmente declarativo, aunque también contiene algunas construcciones imperativas. Su poder expresivo se limita a actualizar las dependencias de la tarea; las acciones de recompilación se delegan al *shell*. Oculta detalles de implementación como la fecha de última actualización del archivo y proporciona abstracciones de dominios tales como sufijos de fichero y reglas de compilación implícitas. Como resultado, el usuario puede expresar de manera concisa y precisa dependencias de actualización.

1.3. DSL – Arquitectura *software*

Las arquitecturas software expresan cómo los sistemas deberían construirse a partir de una serie de componentes y cómo estos componentes deberían interactuar entre ellos. Desde la perspectiva de una arquitectura software, un DSL puede verse tanto como un mecanismo de parametrización, como un modelo de interfaz. Estas dos distinciones tienen un impacto en la estructura del software, de hecho, el rango de adaptabilidad del software está definido por el DSL.

1.3.1. Mecanismo de parametrización

Un programa o una librería pueden ser más o menos genéricos dependiendo del objetivo del problema a resolver. Por ejemplo, una librería científica puede ser altamente genérica considerando la gran variedad de problemas para los cuáles puede aplicarse. Partiendo de la idea de genericidad, se llega a parámetros complejos que pueden verse como lenguajes específicos de dominio. Por ejemplo, el formato de una cadena de caracteres argumento de una función *printf*, puede considerarse tanto un parámetro complejo, como un DSL muy simple. Considerar un programa DSL como un argumento complejo sumamente parametrizado puede sonar inventado, pero es realmente el paso final de una cadena con cada vez mayor potencia expresiva en la parametrización. Esta situación se ilustra en los comandos Unix *grep*, *sort*, *find*, *sed*, *make*, *awk*, etc., y en la progresión de parámetros de líneas de comandos simples a ficheros de programa. En consecuencia, el parámetro termina siendo un programa que ha de ser procesado, aumentando así la potencia de la parametrización.

1.3.2. Interfaz a una biblioteca

Como una biblioteca puede llegar a ser muy grande y genérica, su usabilidad decrece debido a los múltiples puntos de entrada, parámetros y opciones ofrecidas. Como resultado, la biblioteca podría ser ignorada por los programadores debido a que es demasiado compleja de utilizar. En esta situación, un DSL puede ofrecer una interfaz específica de dominio a una biblioteca, de manera que los programadores no necesiten manipular directamente las numerosas construcciones de bloques altamente parametrizadas; la complejidad está ocultada. Otra situación común, es cuando algunos patrones de llamadas a bibliotecas ocurren frecuentemente. Por ejemplo, los *shells* de Unix son interfaces estándar de las bibliotecas Unix. Análogamente, SQL oculta las consultas de bajo nivel a una base de datos. Esta idea es compartida por los lenguajes de *script* que aglutinan un conjunto de componentes escritos en lenguajes de programación tradicionales.

2. ¿POR QUÉ USAR UN DSL?

Los lenguajes específicos de dominios son más atractivos que los lenguajes de propósito general para una gran variedad de aplicaciones.

- **Programación más fácil.** Debido a las abstracciones, notaciones y fórmulas declarativas, un programa DSL es más conciso y legible que un lenguaje de propósito general. Por lo tanto, el tiempo de desarrollo se acorta y se mejora el

mantenimiento. Como la programación se centra en qué computar y no en cómo computarlo, el usuario no se desvía del dominio de la solución del problema.

- **Reutilización sistemática.** La mayoría de los entornos de lenguajes de propósito general incluyen la habilidad de agrupar operaciones comunes en librerías. Aunque algunas son librerías estándar, su reutilización se deja en manos del programador. Por otro lado, un DSL ofrece guías y construcciones que fuerzan la reutilización.
- **Verificación más sencilla.** Los DSLs permiten comprobar muchas propiedades de programas. Al contrario que en los lenguajes de propósito general, la semántica de un DSL puede restringirse a hacer decidibles algunas propiedades que son críticas en un dominio. Por ejemplo, el comando *make* de Unix previene de la existencia de ciclos, lo que previene la no terminación.

3. VENTAJAS E INCONVENIENTES DE UTILIZAR DSLs

Para analizar las ventajas e inconvenientes de la utilización de lenguajes específicos de dominio, es necesario dividirlos en dos categorías diferentes: DSLs externos y DSLs internos. Los externos son escritos utilizando un lenguaje diferente al lenguaje de la aplicación y son transformados en el lenguaje de la aplicación mediante algún tipo de compilador o intérprete. Por ejemplo, pequeños lenguajes Unix, modelos de datos y ficheros de configuración XML caen en esta categoría. Los DSLs internos expresan el DSL directamente en el propio lenguaje, la tradición “Lisp” es el mejor ejemplo de esta categoría.

Una vez introducidas las dos categorías de DSLs, se examinará cada una de ellas por separado, analizando las ventajas e inconvenientes que aportan respectivamente.

3.1. DSL externo

Se ha definido un DSL externo como aquél que es escrito en un lenguaje diferente al lenguaje de la aplicación.

El principal punto fuerte de un DSL externo es la libertad que ofrece para expresar la solución del problema de la manera que se quiera. Como resultado la sencillez del DSL dependerá de la habilidad para expresar el dominio del problema de la forma más fácil posible. El formato estará limitado a la capacidad de construir un traductor que permita analizar y procesar el fichero de configuración, y producir “algo” ejecutable en el lenguaje de la aplicación.

Obviamente, esto supone una desventaja, es necesario construir un traductor. Para lenguajes simples no resulta difícil hacerlo, pero para lenguajes complejos puede convertirse en una tarea más complicada. No obstante, existen generadores automáticos de analizadores y compiladores que facilitan notablemente la tarea de construir el traductor.

La gran desventaja de los DSLs externos es que carecen de integración simbólica, es decir, el DSL realmente no está enlazado al lenguaje de la aplicación.

Ahora que los entornos de programación son cada vez más sofisticados, esto se está convirtiendo en un problema cada vez mayor.

Una objeción especialmente común en los DSLs externos es el problema de la cacofonía de lenguajes. Este problema se traduce en la dificultad de aprender los lenguajes, ya que utilizar muchos lenguajes resulta mucho más complicado que utilizar uno solo. Este hecho puede inducir a confusión sobre la utilización o no de DSLs, ya que a menudo deriva en una perspectiva de múltiples lenguajes de propósito general que realmente podrían tener como resultado la cacofonía. Sin embargo, los DSLs por su cercanía al dominio del problema tienden a ser limitados y sencillos, haciéndolos más fáciles de aprender. Además, no se parecen a los lenguajes de programación corrientes.

Fundamentalmente para cualquier tamaño razonable de programa, se trata con un conjunto de abstracciones que han de ser manipuladas. Comúnmente, estas abstracciones se manipulan utilizando objetos y métodos, lo que resulta factible, pero proporciona una gramática limitada para expresar lo que se quiere decir. La utilización de DSLs externos permite tener una gramática más fácil de manipular. La pregunta es si la comodidad añadida de utilizar el DSL externo es mayor que el coste de aprender el DSL.

Otro hecho importante es la dificultad de diseñar DSLs; el diseño de un lenguaje es difícil, por lo que el diseño de múltiples DSLs será difícil para la mayoría de proyectos de desarrollo software. Otra vez, esta objeción alza el pensamiento de lenguajes de propósito general antes que los lenguajes específicos de dominio. En este punto, la clave fundamental es conseguir buenas abstracciones en el diseño de DSLs, que derivarán en la simplicidad del lenguaje, potencia expresiva y facilidad de aprendizaje.

3.2. DSL interno

El DSL interno voltea los pros y los contras del DSL externo. Se elimina la barrera simbólica con el lenguaje base o lenguaje de la aplicación, y se tiene disponibilidad total del lenguaje de la aplicación junto con todas las herramientas existentes para ese lenguaje.

Uno de los problemas a discutir es que hay una gran diferencia entre lenguajes de programación convencionales (C, C + +, Java, C#) y esos lenguajes como Lisp que son concertados especialmente a DSLs internos. El estilo de DSL interno es mucho más alcanzable en Lisp o SmallTalk que en Java o C#.

Los DSLs internos están limitados por la sintaxis y la estructura del lenguaje base. Aunque se disponga de herramientas para el lenguaje base, este lenguaje no sabe qué es lo que se va a hacer con el DSL, por lo que estas herramientas no dan un soporte completo para el DSL.

Tener disponibilidad total del lenguaje base en el DSL es una ventaja a medias. Si se está familiarizado con el lenguaje base, no hay ningún problema. Sin embargo, uno de las ventajas de un DSL es que permite programar sin conocer completamente el lenguaje base, lo que facilita a los programadores introducir información específica de dominio directamente en el sistema. No obstante, un DSL interno puede hacer esta tarea complicada porque hay muchos lugares donde un usuario puede confundirse si no está familiarizado con el lenguaje base.

Una manera de pensar sobre los lenguajes de propósito general es que proporcionan muchas herramientas, mientras que un DSL solo usa unas pocas de estas herramientas. Tener más herramientas de las necesarias normalmente hace las cosas más difíciles porque es necesario aprender qué son todas estas herramientas, antes de poder averiguar cuáles se van a usar para el DSL. Es posible establecer una analogía con las herramientas que proporciona una aplicación de procesador de textos. Mucha gente se queja de que son difíciles de usar porque tienen muchísimas herramientas, muchas más de lo que una persona suele necesitar. Pero como todas estas herramientas pueden ser utilizadas por alguna persona, una aplicación satisface las necesidades de todo el mundo basándose en una aplicación muy grande con soporte a todas las herramientas posibles. Una alternativa podría ser tener múltiples aplicaciones, cada una de las cuáles centrada en una funcionalidad determinada. De esta forma, cada una de estas aplicaciones sería más fácil de aprender y usar. El problema es el encarecimiento de construir todas estas aplicaciones de propósitos concretos. Esto es una comparativa muy similar a lo que ocurre entre los lenguajes de propósito general (con DSLs internos) y los DSLs externos.

Puesto que los DSLs internos están cerrados al lenguaje de programación base, esto puede presentar dificultades cuando se quiere expresar algo que no se corresponde correctamente con el lenguaje de programación base.

4. INFRAESTRUCTURA TECNOLÓGICA PARA LA INTEGRACIÓN DE DSLs EN MOMENT

Para integrar un DSL en la plataforma MOMENT hay que tener en cuenta diversos aspectos relacionados con la representación de los artefactos software y su ejecución. En MOMENT todos los artefactos se representan mediante modelos en el espacio tecnológico de EMF. Los puentes tecnológicos definidos entre el espacio tecnológico de EMF y el de Maude, permiten interoperar al *front end* de la herramienta, EMF, con el *back end*, MAUDE. De esta manera, es posible generar el código Maude correspondiente a un modelo EMF, ejecutarlo en Maude y devolver los resultados obtenidos a EMF.

Un DSL tiene una sintaxis, que puede ser gráfica o textual, por lo que para poder integrar un DSL en MOMENT, es necesario facilitar un editor que permita codificar especificaciones de ese lenguaje. Con lo visto anteriormente, esta especificación gráfica o textual tiene que ser transformada en un modelo EMF para que pueda ser utilizado en MOMENT.

Para permitir ejecuciones de especificaciones de un cierto DSL en MOMENT, es necesario “proyectar” debidamente a Maude el modelo EMF correspondiente a una especificación gráfica o textual. Así, se generará el código Maude asociado al modelo, cuya ejecución en Maude tendrá los efectos deseados por una ejecución de una especificación de ese DSL. Esta “proyección” se realiza mediante la implementación de los denominados “proyectores”, que no son más que plantillas de generación automática de código Maude.

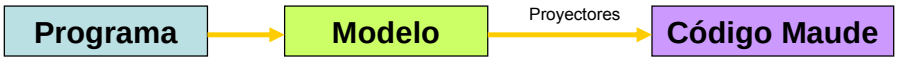


Figura 6. Pasos para integrar un DSL en MOMENT.

4.1. Creación de editores

4.1.1. Editor gráfico

La creación de editores gráficos o visuales puede realizarse mediante los *plug-ins* GEF [GEF] Y GMF [GMF] de Eclipse. Éstos permiten la definición de editores visuales personalizados de forma rápida y sencilla, a través de cinco pasos:

- **Definición del modelo de dominio**. En este paso se indica el metamodelo *ecore* del DSL.
- **Definición gráfica**. En esta parte se definen las propiedades y formas de todos los elementos gráficos que aparecerán en el editor: cajas de información, conectores, etiquetas, etc.
- **Definición de herramientas**. En esta parte se definen todos los elementos que formarán parte de la paleta de herramientas del editor.
- **Definición de correspondencias**. En esta parte se establecen todas las correspondencias entre los elementos del metamodelo, los elementos gráficos definidos y los elementos de la paleta de herramientas creados.
- **Generación de código**. Como último paso, se validan las correspondencias establecidas en el paso anterior, y se genera automáticamente el código del editor.

4.1.2. Editor textual

La creación de un editor textual se realiza de forma automática aprovechando el mecanismo de extensión que ofrece el *workbench* de Eclipse. De esta manera se obtiene de forma automática un *plug-in* que contiene todas las clases Java necesarias para implementar un editor textual en Eclipse. Una vez obtenido el *plug-in*, basta con modificar a mano las clases pertinentes para personalizar el editor según las necesidades deseadas. Para ver más detalladamente el proceso de creación de un editor textual, se puede consultar el artículo de Edwin Ho [Ho03].

4.2. Generación de un modelo EMF a partir de especificación textual

4.2.1. Diseño

La generación de un modelo EMF a partir de su especificación textual, ha sido diseñada como un caso especial de compilador/traductor. Se trata entonces de "compilar" una especificación textual del DSL, que se encontrará codificado en un fichero de texto con una extensión determinada, y “traducirla” en una representación más adecuada en forma de modelo. Para poder generar el modelo, es necesario tener definido el metamodelo *ecore* del DSL. El lenguaje con el que ha de trabajar el compilador es la sintaxis determinada por el DSL.

En la Figura 7, se presenta un esquema del funcionamiento del proceso compilación/traducción de un programa textual de un DSL determinado. En el proceso intervienen tanto elementos activos, representados por cajas cerradas, como flujos de

datos, representados por flechas. Si entre los elementos activos hay una flecha llamada "A", esto quiere decir que el elemento origen produce el flujo de datos "A" que es usado por el elemento destino. A continuación se analiza brevemente cada elemento:

- **Análisis léxico**. Este elemento activo trabaja al nivel más bajo de la sintaxis: el vocabulario de símbolos. El proceso de análisis léxico descompone el texto de su flujo de entrada en caracteres y los agrupa en *tokens*. Los tokens son los símbolos léxicos del lenguaje, también denominados lexemas. Se asemejan en cierta manera a las palabras en el lenguaje natural. Una vez identificados los tokens, son transmitidos al siguiente nivel de análisis. El programa que permite realizar este análisis es el analizador léxico, o simplemente *lexer* (o *scanner*).
- **Análisis sintáctico**. En esta fase se aplican las reglas sintácticas del lenguaje analizado con el fin de comprobar que el texto origen valida la sintaxis del lenguaje que se esté analizando, y si es así construir una estructura de datos que sea manipulable por un sistema informático. La estructura utilizada suele ser un Árbol de Sintaxis Abstracta (AST), que no es más que una estructura en forma de árbol que representa los diferentes patrones sintácticos presentes en la gramática. Se denominan abstractos porque se elimina toda la información que no es de interés, como los espacios en blanco, signos de puntuación o paréntesis. El programa que permite realizar este análisis se llama analizador sintáctico, o en inglés *parser*.
- **Traductor**. Esta etapa del proceso recorre el AST identificando todos los elementos, propiedades y relaciones entre elementos que han de estar presentes en el modelo que se ha de generar. Según avanza el recorrido sobre el AST se crean instancias de objetos según los tipos de objetos definidos en el metamodelo del DSL. Esta fase termina con la serialización en XMI del modelo creado.

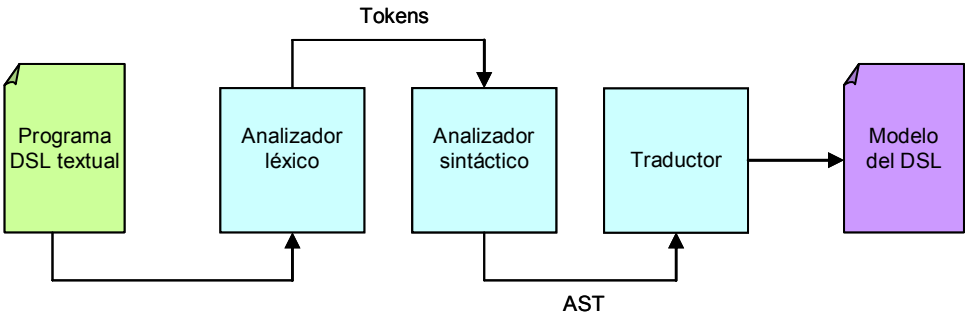


Figura 7. Proceso de compilación/traducción.

4.2.2. Implementación

Para generar los analizadores léxico y sintáctico, y el traductor requeridos se ha empleado el generador de *parsers* ANTLR, cuyo funcionamiento es similar a los conocidos Flex y Bison.

ANTLR es un generador de intérpretes de última generación capaz de generar el analizador léxico, el sintáctico y además también el semántico, dando cobertura de esta forma a todo el proceso de compilación. ANTLR toma como entrada una gramática definida mediante un lenguaje propio cuya sintaxis está basada en EBNF que permite definir los tres tipos de analizadores, es decir, léxico, sintáctico y

semánticos. A partir de esta gramática, ANTLR genera automáticamente el código Java que implementa el analizador correspondiente. Hay que destacar que el analizador semántico, además de hacer la función de analizar semánticamente el programa de entrada, se encarga de realizar la traducción de la especificación textual al modelo correspondiente.

Para obtener más información de ANTLR se recomienda acudir a la página web [ANTLR] o ya en castellano, a la guía escrita en [GarT03].

4.3. Implementación de proyectores

Los proyectores son plantillas de generación de código automático que permiten recorrer los diferentes elementos que forman un modelo y generar el código Maude correspondiente que permita una ejecución adecuada.

La implementación de los proyectores en MOMENT se realiza utilizando el motor de plantillas Velocity [VELOCITY].

5. DSL PARA LA DEFINICIÓN DE OPERADORES COMPLEJOS EN MOMENT

Tal como se reflejó en la introducción de este trabajo, el entorno de modelado industrial que se ha elegido para integrar MOMENT ha sido Eclipse Modeling Framework (EMF)

La integración de estas tecnologías permite aprovechar también las capacidades de modelado de EMF, y la creación de interfaces de usuario amigables, obteniendo interfaces sencillas de emplear ocultando las peculiaridades de Maude al usuario.

Para ocultar estas peculiaridades en la declaración de operadores complejos en Maude, se ha diseñado en MOMENT un lenguaje específico de dominio (DSL) [GoB06] que permite la definición de operadores complejos de una forma más intuitiva. En la definición de este lenguaje específico de dominio se ha seguido la propia filosofía de desarrollo software dirigido por modelos, de forma que la declaración de un operador se representa mediante un modelo EMF. El modelo correspondiente a una operación compleja de Gestión de Modelos puede construirse mediante las interfaces gráficas que proporciona Eclipse o mediante una representación textual que ha sido definida. La obtención de la especificación final del operador en Maude se realiza de forma automática mediante técnicas de generación automática de código.

En este aparado se muestra cómo se ha resuelto e implementado en MOMENT la definición de operadores complejos con un lenguaje específico de dominio para la Gestión de Modelos mediante la aplicación a un caso de estudio no trivial. La estructura de este apartado es la siguiente: el apartado 5.1 proporciona un ejemplo que ilustra la utilidad de la definición de operadores complejos. El apartado 5.2 describe la arquitectura de la herramienta; y el apartado 5.3 describe la solución propuesta para el problema expuesto en el apartado 5.1.

5.1. Ejemplo de motivación: propagación de cambios

Como ejemplo de motivación, utilizaremos un escenario de propagación de cambios que se asemeja al introducido en [MelR03], donde en primer lugar se define un diagrama de clases UML que consta de dos clases (*Item* y *PurhaseOrder*) que modela un sistema de información para una aplicación de gestión de órdenes de compra.

Para construir la aplicación que almacene la información en una base de datos relacional, se reutiliza la metainformación que describe el diagrama UML. Aplicando un mecanismo de transformación (paso 1), obtenemos el nuevo esquema relacional (Rdb). El mecanismo de transformación también genera un conjunto de enlaces entre el nuevo esquema relacional generado (Rdb) y el diagrama de clases origen (Uml) para proporcionar soporte a la trazabilidad (MapUml2Rdb).

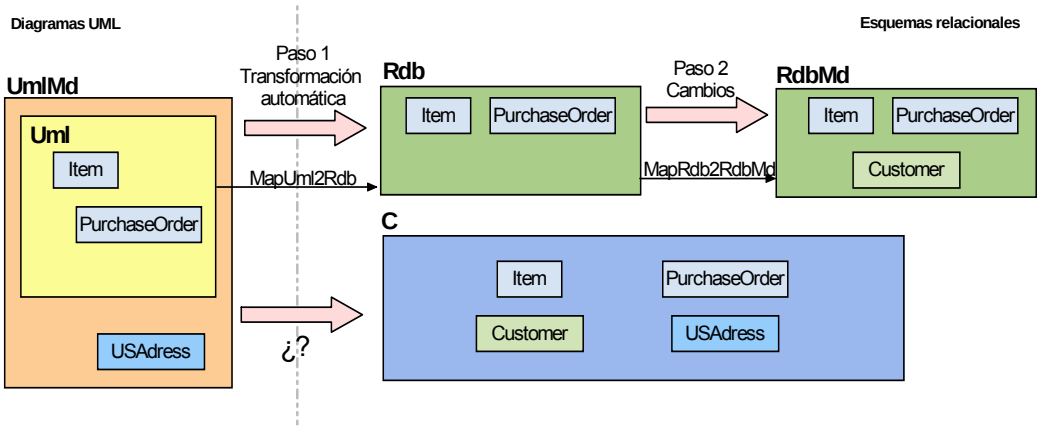


Figura 8. Ejemplo de propagación de cambios.

Tras obtener un esquema relacional semánticamente equivalente al diagrama UML original, se continúa con el desarrollo del nuevo sistema. Esto puede implicar cambios en la aplicación y en el esquema de la base de datos (paso 2), obteniendo el esquema relacional (RdbMd). Estos cambios son trazados y almacenados por la herramienta que gestiona la manipulación del modelo, o directamente por el usuario (MapRdb2RdbMd).

Una vez se ha desarrollado el nuevo sistema, pueden producirse cambios en los requisitos del sistema, requiriendo modificaciones sobre el modelo de la orden de compra. Es más sencillo modificar el diagrama UML que modificar el esquema de la base de datos RdbMd de forma que se mantenga la consistencia entre el diseño de la aplicación y la capa de persistencia. En este punto, la aplicación de nuevo del mecanismo de transformación aplicado en el paso 1 descartaría los cambios aplicados de Rdb a RdbMd.

Una solución a este ejemplo de propagación de cambios puede ser realizado usando operadores de gestión de modelos.

5.2. Ejecución de operadores complejos en MOMENT

El desarrollo de MOMENT ha sido realizado siguiendo la propia filosofía de desarrollo de software dirigido por modelos. Se ha desarrollado un modelo de la aplicación empleando el lenguaje Ecore describiendo todos los elementos que intervienen en la ejecución de operadores. Posteriormente, mediante las capacidades

de generación de código que proporciona EMF se ha obtenido el código Java de la aplicación.

5.2.1. Soporte para definición de operadores complejos

Esta filosofía además, de incrementar la mantenibilidad del código, nos permite manipular todos estos elementos representados en el modelo de la aplicación a un mayor nivel de abstracción; y además proporciona un mecanismo automático de persistencia y recuperación de datos (por defecto el formato de persistencia es XMI, lo que proporciona interoperabilidad con otras aplicaciones).

El diagrama de clases de la Figura 9 muestra la parte del modelo de la herramienta que permite la declaración y ejecución de operadores de gestión de modelos. La clase *Operator* captura la información de la declaración de un operador. Esta clase se especializa en operadores simples *(SimpleOperator)* y operadores complejos *(ComplexOperator)*. La declaración del operador (de la misma manera que la declaración de una función Java, o C), incluye la declaración de sus parámetros formales. Los parámetros formales tanto de entrada *(InputFormalParameter)* como de salida *(OutputFormalParameter)* son especializaciones de la clase *Variable*; disponiendo todos ellos tanto de un nombre como de un tipo *GenericType*. Los tipos en la declaración de un operador se denominan genéricos puesto que la declaración de un operador se realiza de forma genérica independiente del metamodelo concreto.

La clase *OperationInvocation* captura la información sobre la ejecución de un determinado operador (referenciado mediante el rol *calledOperator*) con unos datos concretos, que en la figura se representan mediante la clase *ActualParameter* (parámetro actual). Cada parámetro actual representa los datos concretos con los que se invoca un operador e instancia un parámetro formal de la declaración de un operador (rol *instantiatesFormalParameter*). De igual forma que un parámetro actual, también dispone de un tipo *(ConcreteType)*. Por ejemplo, un parámetro actual podría ser un diagrama de clases UML concreto (por ejemplo el modelo *Uml* del caso de estudio), y su tipo concreto sería el metamodelo UML.

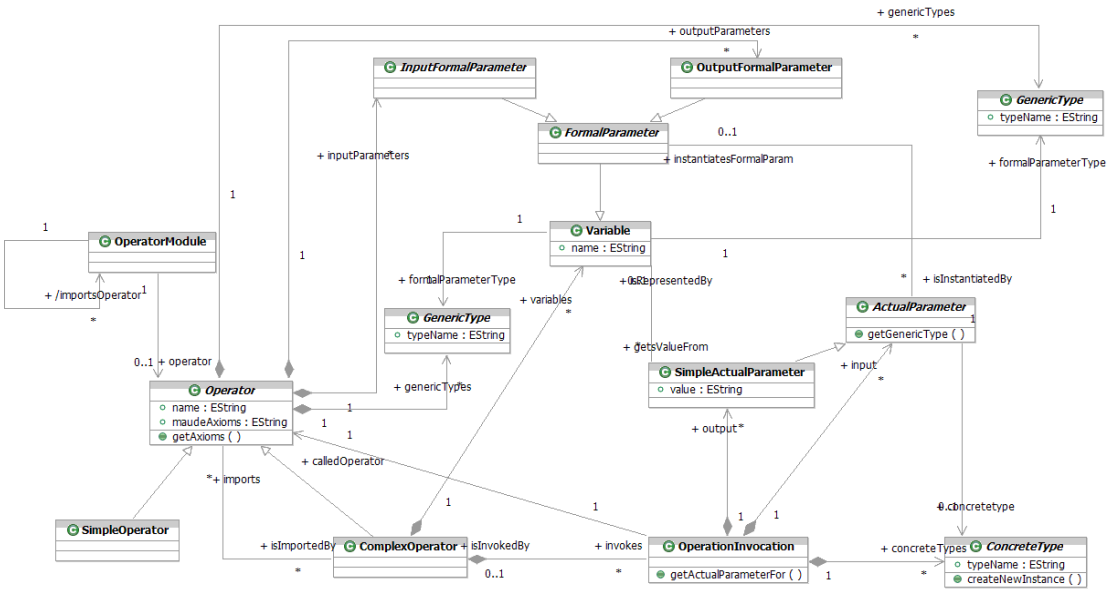


Figura 9. Modelo simplificado para la especificación de operadores complejos en MOMENT.

Esta representación de los operadores complejos como modelos nos permite tratar con ellos a un mayor nivel de abstracción, puesto que las manipulaciones y consultas sobre un operador se realizan directamente sobre los conceptos modelados en la Figura 9, y no sobre un árbol de sintaxis abstracta construido, por ejemplo, en tiempo de compilación a partir de una gramática.

Finalmente, dada esta representación para un operador complejo y mediante técnicas de generación de código, este modelo de un operador determinado será proyectado a código Maude en la forma de un módulo paramétrico independiente de metamodelo.

5.2.2. Arquitectura de la herramienta

MOMENT es un *framework* que hace uso del lenguaje de especificaciones algebraicas Maude para la implementación de todos estos operadores. Para poder emplear un proceso Maude desde un programa Java se ha hecho uso de *Maude Development Tools*, un conjunto de herramientas que extienden Eclipse y proporcionan una API para el uso de Maude de forma programática.

El diagrama de componentes UML de la Figura 10 muestra los elementos de MOMENT relacionados con la ejecución de operaciones de Gestión de Modelos. Los principales módulos son los denominados "Lanzador de operaciones" *(Operator Launcher)* y "Cargador de módulos" *(Module Loader).*

El primero de ellos es el que proporciona la interfaz de MOMENT al usuario. Permite, dada la declaración de un operador (sea simple o complejo), especificar sus parámetros actuales así como dónde se salvarán los resultados devueltos.

El segundo componente, el cargador de módulos, se encarga de controlar de forma transparente al usuario el proceso Maude sobre el que se ejecutarán las transformaciones mediante *Maude Development Tools*.

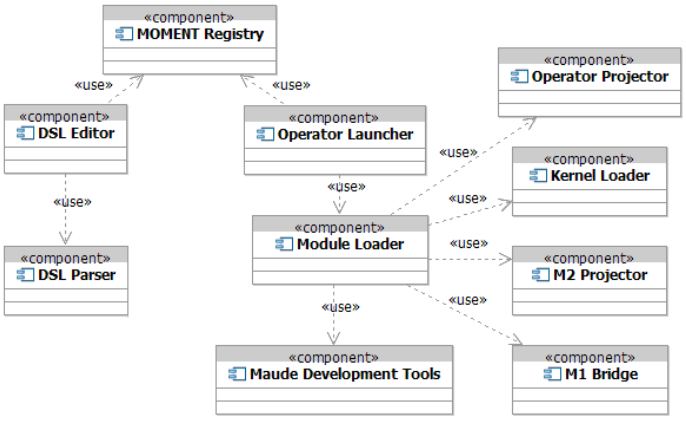


Figura 10. Componentes de MOMENT relacionados con la ejecución de operaciones de gestión de Modelos.

MOMENT se sustenta sobre un conjunto básico de módulos Maude (que denominaremos *kernel*) que proporcionan la funcionalidad básica del *framework*. Entre estos módulos encontramos aquellos que implementan los operadores simples comentados en el apartado 4.4 de la introducción. De esta forma, el *Module Loader*, se ocupa de la preparación del contexto de ejecución (carga de los módulos del *kernel* por parte del *Kernel Loader*) y de la proyección a código Maude todos los elementos

que intervendrán en una ejecución. Esto incluye la especificación algebraica de los metamodelos implicados *(M2 Projector)*, el código del operador a ejecutar *(Operator Projector)*, y los términos correspondientes a los parámetros de entrada en el momento de la invocación de la operación *(M1 Bridge)*. Finalmente, el puente a nivel M1 *(M1 Bridge)* es también el componente encargado de procesar los términos resultantes de una ejecución y recuperarlos en el espacio tecnológico de EMF.

Es de destacar en este punto la expresividad de Maude puesto que a diferencia de otros lenguajes (como Java), la composicionalidad funcional de operaciones es inherente al formalismo de la lógica ecuacional de pertenencia subyacente.

El código generado para definir un operador complejo en Maude presenta una correspondencia directa con la representación de nuestro lenguaje de definición de operadores complejos.

5.2.3. Definición textual de operadores complejos

La representación de los operadores complejos como modelos permite trabajar con éstos desde un mayor nivel de abstracción con todas las ventajas que esto aporta. No obstante, con las interfaces gráficas que proporciona inicialmente Eclipse para la edición de modelos, ésta aproximación puede resultar poco intuitiva la definición de operadores nuevos.

Para la definición de un lenguaje de estas características la opción más natural y sencilla para el usuario sería probablemente proporcionar una representación textual conforme a una gramática bien definida. En este caso, en MOMENT se ha definido la gramática que se incluye en el Anexo I para dar soporte textual a la definición de operadores complejos.

Por ello, y tal como se observa en la Figura 10, existen en MOMENT dos componentes adicionales *DSL Editor* y *DSL Parser*. El primero corresponde a un editor de texto con coloreado de sintaxis que permite la definición de operadores complejos según la gramática definida. Mediante el *DSL Parser*, se obtiene automáticamente el modelo del operador complejo equivalente a su declaración textual. Una vez obtenido el modelo, el *DSL Editor* puede incluir la declaración del operador en el repositorio de operadores que se proporcionan al usuario para su ejecución (denominado *MOMENT Registry*). El *Operator Launcher*, recupera los operadores de este repositorio para ser ejecutados sobre Maude.

5.2.4. El DSL de MOMENT

El DSL especificado en MOMENT para la definición de operadores complejos viene descrito por la gramática del Anexo I.

Toda definición de operador complejo sigue el siguiente esquema:

```
//Importación de operadores simples                                    (1)
#import "ruta_operador"            //importación por ruta
#import <nombre_operador>          //importación por nombre
//Declaración de metamodelos                                           (2)
metamodel metamodelo1, metamodel2, …, metamodelon;
```

```
//Declaración del operador
operator nombre_operador (parámetros_de_entrada)                    (3)
: <tipos_resultado>                                                  (4)
{
//Sentencias
<modelos_salida> = operador_simple_importado (modelos_de_entrada);(5)
return <modelos_resultado>;                                          (6)
}
```

A continuación, se detallan los diferentes pasos que conforman la definición de un operador complejo:

1. La importación de un operador simple puede realizarse de dos formas: por ruta del operador o por nombre del operador. Para poder importar un operador por su nombre, es necesario que éste se encuentre registrado en el MOMENT *Registry* (ver sección VII).
2. La declaración de metamodelos sirve para crear variables de tipo "metamodelo" que permiten especificar los diferentes tipos de metamodelos que intervienen en el operador. De esta manera, es posible diferenciar los modelos de entrada que recibirá el operador.
3. Los parámetros de entrada son todos aquellos elementos que recibe el operador. Éstos se declaran de la forma $tipo_1$ $param_1$, $tipo_2$ $param_2$, …, $tipo_n$ $param_n$. Cuando el operador recibe modelos que conforman a metamodelos diferentes, es necesario declarar diferentes variables de tipo "metamodelo". Así, por ejemplo, si el operador recibe dos modelos de entrada que conforman a metamodelos diferentes, crearemos dos variables de tipo metamodelo y pasaremos los modelos como parámetros de entrada al operador, indicando como tipo la variable "metamodelo" que se ha creado respectivamente.

   ```
   metamodel MM1, MM2;
   operator nombre_operador (MM1 modelo1, MM2 modelo2,…) : <…> {…}
   ```

4. En esta parte se declaran los tipos de los modelos que se obtendrán como resultado.
5. Las sentencias constituyen el cuerpo de la declaración del operador y de ellas se obtienen los modelos resultados. Para ello, se realizan invocaciones a los operadores simples importados.
6. Mediante **return** se devuelven los modelos obtenidos como resultado de la ejecución del operador.

5.3. Aplicación de MOMENT al caso de estudio

El problema mostrado en el caso de estudio (apartado 5.1) puede ser simplificado como se muestra en la Figura 11, donde el modelo *MapUml2RdbMd* puede ser fácilmente obtenido de los modelos *MapUml2Rdb* y *MapRdb2RdbMd* mediante el operador *Compose*. Por lo tanto, el problema puede enunciarse de la siguiente manera:

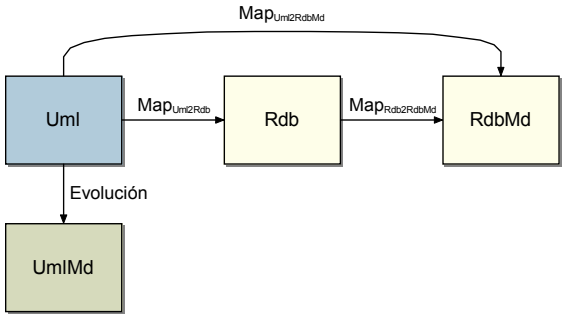


Figura 11. Esquematización del problema del caso de estudio.

"Dados los siguiente modelos: un diagrama de clases UML original (Uml); un diagrama de clases UML (UmlMd), que ha sido evolucionado de UML; un esquema de base de datos relacional RdbMd, que ha sido generado a partir del diagrama de clases UML y modificada posteriormente; y un modelo de trazabilidad entre UML y RDB' (MapUml2RdbMd); deberemos obtener el esquema relacional del diagrama de clases UmlMd que conserve los cambios realizados en RdbMd."

Éste problema puede ser resuelto por el siguiente operador complejo:

```
// Declaración de importaciones de operadores simples
 // Declaración de metamodelos

 operator PropagateChanges(MM1 Uml, MM1 UmlMd, MM2 RdbMd,
                TraceabilityMetamodel MapUmlIni2RdbMd, Transformation Uml2Rdbms)
        : <MM2, TraceabilityMetamodel>
 {

    // Obtención del modelo resultado
    <UmlUnmd, MapUml2UmlUnmd, MapUmlMd2UmlUnmd> = Cross(Uml, UmlMd);                (1)
    <RdbMdMd> = Range(mapUml2RdbMd , UmlUnmd, RdbMd);                               (2)
    <UmlNew, MapUmlMd2UmlNew, MapUmlUnmd2UmlNew> = Diff(UmlMd, UmlUnmd);            (3)
    <RdbNew, MapUmlNew2RdbNew> = ModelGen(Uml2Rdbms, UmlNew);                       (4)
    <Result, MapRdbUnmd2Result, MapRdbNew2Result> = Merge(RdbUnmd, RdbNew);         (5)

    // Generación del modelo de trazabilidad
    <MapUmlUnmd2RdbUnmd> = RestrictDomain(UmlUnmd , MapUml2RdbMd);                  (6)
    <MapUmlUnmd2Result> = Compose(MapUmlUnmd2RdbUnmd, MapRdbUnmd2Result);           (7)
    <MapUmlNew2Result> = Compose(MapUmlNew2RdbNew, MapRdbNew2Result);               (8)
    <MapUmlMd2Result, Map1, Map2> = Merge(MapUmlUnmd2Result, MapUmlNew2Result);     (9)

    return <Result, MapUmlMd2Result>;
 }
```

Este operador está construido a partir de operadores simples del álgebra de MOMENT y los pasos seguidos en el *script* se representan en la Figura 12. Estos pasos son los siguientes:

1. "Unmd" es la parte del modelo UML que permanece sin modificar en el modelo UmlMd.
2. "RdbUnmd" es el submodelo de RdbMd que corresponde a la parte no modificada de UmlMd.
3. "UmlNew" es la parte de UmlMd que ha sido añadida al modelo Uml.
4. "RdbNew" es el esquema relacional obtenido de la traducción de UmlNew al metamodelo relacional.
5. "Result" es el modelo final obtenido de la integración de las bases de datos obtenidas en los pasos 2 y 4.

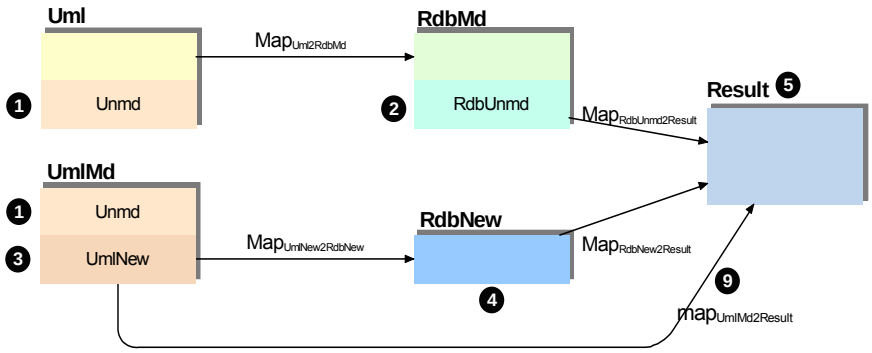


Figura 12. Solución al problema del caso de estudio.

6. "MapUmlUnmd2RdbUnmd" es el modelo de trazabilidad que enlaza los elementos de Uml relacionados elementos de RdbMd que han permanecido sin cambios.
7. "MapUmlUnmd2Result" es el modelo de trazabilidad que relaciona la parte no modificada de Uml (UmlUnmd) y Result combinando MapUmlUnmd2RdbUnmd y MapRdbUnmd2Result.
8. Obtiene el modelo de trazabilidad entre la parte nueva de UmlMd y Result.
9. Une los modelos obtenidos en 7 y 8 mediante el operador Merge, del mismo modo que otro par cualquiera de modelos pertenecientes al mismo metamodelo, obteniendo mapUmlMd2Result.

El operador complejo que se ha mostrado se ha especificado según la gramática del Anexo I. Éste resuelve el problema de la propagación de cambios del caso de estudio de forma independiente de los metamodelos implicados, por lo que puede ser aplicado a cualquier combinación de metamodelos (en lugar de usar los metamodelos UML y relacional). Tras obtenerse su representación como modelo EMF mediante el *parser* implementado, se obtiene de forma automática el código Maude que se adjunta en el Anexo II.

6. HERRAMIENTAS DESARROLLADAS

Para dar soporte en MOMENT a la manipulación de modelos mediante la definición de operadores complejos por parte del usuario, se han desarrollado tres *plug-ins* en Eclipse atendiendo a las indicaciones descritas en el apartado 4 de la sección III. Éstos se desarrollarán detalladamente en los siguientes apartados. En el apartado 6.1 se describe el *plug-in* que contribuye con un editor textual de operadores complejos, en el apartado 6.2 se hace lo propio con el *parser* del lenguaje específico de dominio especificado para la definición de operadores complejos, y por último, en el apartado 6.3 se trata la interfaz gráfica para invocar la compilación/traducción de una definición textual de un operador complejo.

6.1. Soporte para la definición textual de operadores complejo

6.1.1. Descripción

En este apartado se presenta una descripción a alto nivel del *plug-in* que contribuye con el editor textual de operadores complejos. El nombre de este *plug-in* es "es.upv.dsic.issi.moment.dsl.ui.editor", pero por simplicidad nos referiremos a él como *DSL Editor*.

A continuación se presentarán las necesidades a las que el sistema debe dar soporte, las funciones que debe realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo.

6.1.1.1. Funciones del *plug-in*

Este *plug-in* proporciona un editor textual con coloreado de sintaxis para el lenguaje específico de dominio especificado según la gramática EBNF del Anexo I.

Los ficheros que especifican operadores complejos tienen la extensión ".moptext", por tanto, el editor se asociará a este tipo de ficheros de manera que el editor se iniciará automáticamente con la apertura del fichero en Eclipse. A lo largo de este trabajo nos referiremos a este tipo de ficheros como ficheros "moptext".

6.1.1.2. Dependencias

Para que el *plug-in* funcione correctamente se deberá disponer de una distribución de Eclipse con el *framework* EMF instalado.

Puesto que el *plug-in* se genera de manera semiautomática utilizando el sistema de extensiones del *workbench* de Eclipse, automáticamente se incluyen las dependencias internas a aquellos *plug-ins* de Eclipse que son necesarios.

6.1.2. Diseño

Para desarrollar este *plug-in* se utiliza el mecanismo de extensiones que ofrece el *workbench* de Eclipse para crear un editor textual según lo explicado en el apartado 4.1.2 de la sección III de este documento.

6.1.3. Implementación

En el presente apartado se describirán las clases principales que conforman el editor y aquellas modificaciones realizadas para la adaptación del editor al lenguaje específico de dominio que permite la definición de operadores complejos en MOMENT.

6.1.3.1. MomentOpEditor

Esta es la clase principal del editor. Hereda de *org.eclipse.ui.editors.text.Text-Editor* y el constructor por defecto de la clase es **public MomentOpEditor()**. En él se ejecuta el constructor de la clase padre, y se configura el editor para que incluya las funcionalidades adicionales implementadas como el coloreado de sintaxis.

6.1.3.2. MomentOpRuleScanner

En la clase *MomentOpRuleScanner*, se establecen las reglas para el formateado del texto del editor. Esto es lo que permite establecer diferentes colores y formatos para palabras clave, comentarios, etc.

Hereda de la clase *org.eclipse.jface.text.rules.RuleBasedScanner*.

En esta clase se han introducido las palabras reservadas del lenguaje de definición de operadores complejos y se han establecido los diferentes colores para los comentarios, palabras reservadas y código de usuario:

```
public static String[] keyWords =
  {
      "import", "#import", "metamodel", "operator" , "string",
      "float", "rat", "int", "qid", "bool",
      "traceabilitymetamodel",
      "transformation"
  };

private static Color KEY_WORDS_COLOR =
               new Color(Display.getCurrent(), new RGB(128, 0, 64));
private static Color STRING_COLOR =
               new Color(Display.getCurrent(), new RGB(0, 0, 255));
private static Color COMMENT_COLOR =
            new Color (Display.getCurrent(), new RGB (192,192,192));
```

El constructor, public MomentOpRuleScanner(), es el único método de que consta esta clase, y en él se crean todas las reglas y se definen todos los formatos para el coloreado del código de definición de operadores complejos. Para cada *token* especificado en las diferentes reglas se le puede asociar un determinado formato.

En este método se asocian los colores a las palabras reservadas, a los cometarios y al resto de caracteres que conforman el código de definición de operadores complejos:

```
IToken keyWordsToken =
      new Token(new TextAttribute(KEY_WORDS_COLOR,null,SWT.BOLD));
IToken stringToken = new Token(new TextAttribute(STRING_COLOR));
IToken commentToken = new Token (new TextAttribute (COMMENT_COLOR));
```

Una vez creadas todas las reglas, se añaden mediante el uso del método *setRules(IRule[] rules)* de la clase padre.

Para este editor, se han añadido las reglas necesarias para soportar los diferentes tipos de comentarios: comentarios de línea y comentarios multilínea:

```
setRules(new IRule[] {
            // Add rule for processing instructions
            keyWordsRule,
            new SingleLineRule("\"", "\"", stringToken),
            new SingleLineRule("'", "'", stringToken),
            new EndOfLineRule ("//",commentToken),
            new MultiLineRule("/*", "*/", commentToken,'\n',true),
            new WhitespaceRule(new WhitespaceDetector())
      });
```

6.1.3.3. MomentOpSourceViewerConfig

Esta clase permite establecer todas las configuraciones del editor, esto es, establecer los objetos que implementan el coloreado de sintaxis, o el completado de texto.

El constructor por defecto es **public MomentOpSourceViewerConfig()**.

El método **public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)** devuelve un asistente de contenido. Lo crea, y establece algunos parámetros de configuración como la activación automática, o el retardo de aparición.

El método **public IRepresentationReconciler getPresentationReconciler (ISourceViewer sourceViewer)** devuelve el "reconciliador" de la presentación. Cada vez que el usuario cambia el documento, el "reconciliador" determina qué región de la presentación debe ser invalidada y como deber ser reparada. Un daño es el texto que debe ser redibujado, y la reparación es el método utilizado para redibujar el área dañada. El proceso de mantener la presentación visual de un documento a medida que se realizan los cambios se reconoce como "reconciliado".

El método **protected MomentOpRuleScanner getTagScanner()** devuelve el *scanner* que define los atributos de texto según las reglas de coloreado. En caso de que no se haya creado todavía ninguna instancia de *MomentOpRuleScanner*, la crea en este momento.

6.1.4. Archivos resultantes

Como resultado de generación semiautomática del editor, se han obtenido varios ficheros que han sido agrupados en dos paquetes. Además, se ha creado una carpeta en el *plug-in* que contiene la imagen del icono que identificará los ficheros "moptext".

- Paquete "es.upv.dsic.issi.moment.dsl.ui.editor"

  Este paquete contiene el fichero MomentOpEditorPlugin.java que contiene todos los métodos de activación del *plug-in*.

- Paquete "es.upv.dsic.issi.moment.dsl.ui.editor,editors"

  Este paquete contiene cinco ficheros que contiene los métodos que permiten especificar las propiedades del editor, como el coloreado de sintaxis, palabras reservadas, comentarios, etc

  - MomentOpEditor.java
  - MomentOpEditorContributor.java

- MomentOpRuleScanner.java
- MomentOpSourceViewerConfig.java
- WhitespaceDetector.java

6.2. Soporte para la compilación de operadores complejos definidos textualmente

6.2.1. Interfaz gráfica

6.2.1.1. Descripción

En este apartado se presenta una descripción a alto nivel del *plug-in*. Se presentarán las necesidades a las que el sistema debe dar soporte, las funciones que debe realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo. El nombre del *plug-in* es "es.upv.dsic.issi.moment.dsl.parser.ui". Por simplicidad, nos referiremos a este *plug-in* como *DSL Parser* o *parser* del DSL.

- *Funciones del plug-in*

Este *plug-in* deberá proporcionar un menú emergente, activable haciendo clic con el botón derecho sobre un fichero de extensión "moptext", que de la opción de compilar/traducir (analizar) el fichero seleccionado.

- *Dependencias*

Para que el *plug-in* funcione correctamente se deberá disponer de una distribución de Eclipse con el *framework* EMF instalado.

Puesto que el *plug-in* se genera de manera semiautomática utilizando el sistema de extensiones del *workbench* de Eclipse, automáticamente se incluyen las dependencias internas a aquellos *plug-ins* de Eclipse que son necesarios.

Además, es necesario incluir la dependencia al *plug-in* que implementa el análisis de un fichero "moptext". Este *plug-in* es es.upv.dsic.issi.moment.dsl.parser.

6.2.1.2. Diseño

Para desarrollar este *plug-in* se utiliza el mecanismo de extensiones que ofrece el *workbench* de Eclipse. De esta manera se obtiene de forma automática un *plug-in* que contiene todas las clases necesarias para implementar un menú emergente (*pop-up menu*) en Eclipse. Una vez obtenido el *plug-in*, basta con modificar a mano las clases pertinentes para personalizar el menú a las necesidades deseadas.

Por defecto, el *workbench* de Eclipse ofrece un menú emergente activable tras hacer clic con el botón derecho sobre cualquier fichero del entorno de trabajo. Por tanto, el objetivo del *plug-in* es introducir una entrada en este menú que de la opción de analizar el fichero "moptext" seleccionado.

6.2.1.3. Implementación

La implementación de este *plug-in* se reduce a completar las clases generadas por el mecanismo de extensiones. A continuación se describen las clases generadas y aquellas modificaciones realizadas para alcanzar la funcionalidad deseada.

- *DSLParserUIPlugin*

La clase *DSLParserUIPlugin* hereda de *AbstractUIPlugin* y proporciona los métodos necesarios para iniciar y detener la ejecución del *plug-in*. Esta clase no ha sufrido ninguna modificación.

- *ParseDSLTextualProgram*

La clase *ParseDSLTextualProgram* es la que se encarga de capturar el fichero seleccionado y ejecutar el análisis del mismo. Esta clase implementa la interfaz *IObjectActionDelegate* y proporciona el método *void run(IAction action)*. Este método contiene la acción o acciones que se ejecutarán en el momento de seleccionar la opción de menú. En este caso, se trata de analizar un fichero seleccionado, por tanto dentro del método se invoca la clase que realiza esta acción.

```
void run(IAction action {
  DSLParserPlugin.getDefault().createModelFromTextSpec(file);
}
```

Para capturar el fichero seleccionado en el momento de la invocación del menú emergente, se ha añadido el siguiente método a la clase:

```
public void selectionChanged(IAction action, ISelection selection) {
  file = null;
  if(selection instanceof IStructuredSelection) {
     IStructuredSelection sel = (IStructuredSelection)selection;
     Object selElem = sel.getFirstElement();
     if(selElem instanceof IFile)
        file = (IFile)selElem;
  }
}
```

La variable *file* empleada en el método, se define como un atributo de la clase de la forma *"private IFile file;".* Así, el método *selectionChanged* captura el fichero que se encuentra seleccionado en el momento de invocar el menú emergente con el botón derecho del ratón y si en este menú se selecciona la opción de *"parser to DSL"*, entonces en la llamada *DSLParserPlugin.getDefault().createModelFromTextSpec(file)* del método *run* se pasa como argumento el fichero que se ha de analizar.

6.2.1.4. Archivos resultantes

Únicamente se obtienen dos ficheros como resultado de la implementación del menú emergente. Cada fichero se encuentra en un paquete diferente e implementa la clase que da nombre al fichero:

- Paquete “es.upv.dsic.issi.moment.dsl.parser.ui”
    - DSLPaserUIPlugin.java
- Paquete “es.upv.dsic.issi.moment.dsl.parser.ui.popup.actions”
    - ParseDSLTextualProgram.java

6.2.2. Compilador/traductor de operadores complejos definidos textualmente

6.2.2.1. Descripción

En este apartado se presenta una descripción a alto nivel del *plug-in*. Se presentarán las necesidades a las que el sistema debe dar soporte, las funciones que debe realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo. El nombre del *plug-in* es “es.upv.dsic.issi.moment.dsl.parser”.

- *Funciones del plug-in*

Este *plug-in* deberá proporcionar la siguiente funcionalidad:

- Analizar léxica y sintácticamente un programa textual “moptext” que contiene la definición de un operador complejo e informar debidamente al usuario de los posibles errores contenidos en el programa.
- Construir un árbol de sintaxis abstracta (AST) como resultado del análisis sintáctico de la definición textual de un operador complejo de entrada.
- Crear de forma automática el modelo de operador complejo de MOMENT correspondiente a su definición textual.
- Persistir como fichero XMI el modelo de operador complejo creado.

- *Dependencias*

Para que el *plug-in* funcione correctamente se deberá disponer de una distribución de Eclipse con el *framework* EMF instalado.

Además, de manera interna este *plug-in* depende de la librería de “antlr” y de los siguientes *plug-ins* de MOMENT:

- es.upv.dsic.issi.moment.engine.core

  Este *plug-in* contiene todas las clases Java que componen el metamodelo de operador complejo de MOMENT. La dependencia de este *plug-in* es necesaria para poder crear modelos de operadores complejos.

- es.upv.dsic.issi.moment.ui.console

  Este *plug-in* proporciona el soporte para la consola de MOMENT, a través de la cual se mostrarán al usuario aquellos mensajes de error, depuración o información.

- es.upv.dsic.issi.moment.registry

  Este *plug-in* permite acceder al repositorio "MOMENT *Registry*" para cargar aquellos operadores registrados en MOMENT que pueden utilizarse en la definición de un operador complejo.

  MOMENT *Registry* se describirá detalladamente en la sección VII de este documento.

6.2.2.2. Diseño

El *DSL Parser* ha sido diseñado como un caso especial de compilador/traductor siguiendo la analogía presentada en el apartado 4.2 de la sección III de este documento.

En este caso, se trata de "compilar" la definición textual de un operador complejo que se encontrará codificado en un fichero "moptext", y "traducirlo" en una representación más adecuada en forma de modelo de operador complejo de MOMENT. El lenguaje con el que ha de trabajar el compilador es el lenguaje específico de dominio derivado de la gramática EBNF del Anexo I.

6.2.2.3. Implementación

- *Analizador léxico*

La gramática definida para el analizador léxico se ha obtenido a partir de la sintaxis abstracta del lenguaje específico de dominio para la definición de operadores complejos en MOMENT mostrada en el Anexo I.

En esta gramática se han definido todas las palabras reservadas y símbolos del lenguaje, y aquellos elementos que no deben pasarse al analizador sintáctico como son los comentarios, espacios en blanco y caracteres de retorno de carro.

Respecto a los comentarios se ha seguido la sintaxis del lenguaje C. Así, los comentarios de línea pueden definirse como:

- // comentario
- /* comentario */

Los comentarios multilínea vendrán delimitados por los caracteres /*, que delimitan el comienzo, y los caracteres */, que marcan el final del comentario multilínea.

Para evitar ambigüedades en la gramática ANTLR que especifica el analizador léxico, se ha variado el valor de "k". Este valor, indica a ANTLR el número de símbolos de anticipación antes de decidir la elección de una regla.

- *Analizador sintáctico*

Para construir el analizador sintáctico, se ha seguido una implementación de la sintaxis abstracta mostrada en el Anexo I. En esta implementación surgieron problemas debidos a ambigüedades que fueron resueltas modificando el valor de k.

En esta implementación también se han etiquetado los diferentes nodos que formarán el árbol de sintaxis abstracta, de manera que puedan ser reconocidos y procesados adecuadamente por el analizador semántico/traductor.

- *Analizador semántico/traductor*

En la implementación del analizador semántico/traductor se recorre todo el AST de manera ordenada mediante un mecanismo de *pattern matching* que proporciona ANTLR para los nodos del árbol. Recorriendo los elementos del árbol se evalúan ciertas reglas semánticas y se instancian los pertinentes objetos elementos del modelo de operador complejo así como se establecen las relaciones y dependencias entre estos objetos.

Las reglas semánticas introducidas tienen como objetivo detectar las posibles incorrecciones cometidas por el usuario en la utilización de operadores existentes para la definición de nuevos operadores complejos. Las reglas semánticas implementadas son:

- Parámetros formales de entrada en una invocación de operación. Se comprueba que el número de parámetros formales de entrada utilizados en la invocación de una operación coincida con el número de los parámetros de entrada especificados en la declaración del operador invocado.
- Parámetros formales de salida en una invocación de operación. Se comprueba que el número de parámetros formales de salida utilizados en la invocación de una operación coincida con el número de parámetros devueltos por el operador invocado.
- Parámetros de salida de la definición del operador complejo. Se comprueba que efectivamente el operador complejo especificado devuelve el mismo número de parámetros de salida que han sido declarados.

6.2.2.4. Archivos resultantes

Como resultado de la implementación se han creado dos ficheros DSLparser.g y DSLtreewalker.g:

- DSLparser.g

  Este fichero contiene dos gramáticas ANTLR que implementan el analizador léxico y el sintáctico respectivamente.

- DSLtreewalker.g

  Este fichero contiene una gramática ANTLR que recorre el AST de manera apropiada para comprobar las reglas semánticas y crear el modelo del operador complejo correspondiente.

Ambos ficheros se encuentran ubicados en la carpeta “grammar” del *plug-in* “es.upv.dsic.issi.moment.dsl.parser”.

La compilación ANTLR de ambos ficheros produce los ficheros Java que implementan los analizadores especificados por las gramáticas. Estos ficheros se encuentran en la ruta del *plug-in* “/src/es.upv.dsic.issi.moment.dsl.parser.generated”.

7. TRABAJOS RELACIONADOS: EL DSL DE RONDO

Como se comentó en la introducción de este trabajo, son muy pocas las herramientas de gestión de modelos existentes en el panorama de la gestión de modelos. Una de las herramientas existentes más similar a MOMENT es RONDO.

RONDO fue introducido en el apartado 3.1 de la sección I, y al igual que MOMENT, permite definir operadores complejos haciendo uso de operadores existentes utilizando un lenguaje específico de dominio. En este apartado se presentará de manera detallada este lenguaje así como su utilización para resolver un caso de estudio determinado, similar al resuelto a través del DSL para definición de operadores complejos de MOMENT.

7.1. Escenario de motivación

Para motivar la definición de operadores en RONDO, se usará el escenario ilustrado en la Figura 13. Considérese una tienda virtual de internet que necesita proporcionar las órdenes de pedido a su socio comercial. Los datos están almacenados en una base de datos relacional de acuerdo con el esquema s1.

Para el propósito de intercambio de información, ambas compañías acuerdan usar esquemas XML d1. (Las correspondencias entre los elementos del esquema s1 y d1 están representadas por líneas grises). El esquema d1 difiere del s1 en términos de estructura y nombrado.

El esquema relacional sufre cambios periódicos según los requisitos dinámicos del negocio. Asúmase que s2 es una nueva versión de s1, en la cual las columnas “Brand” y “Discount” han sido eliminadas, las columnas “ShipDate”, “FreightCharge”, y “Rebate” han sido añadidas, y la columna “UnitPrice” ha sido renombrada a “Price”. Estos cambios (destacados en negrita en la Figura 13) necesitan propagarse al esquema XML, por lo tanto d1 se convierte d2.

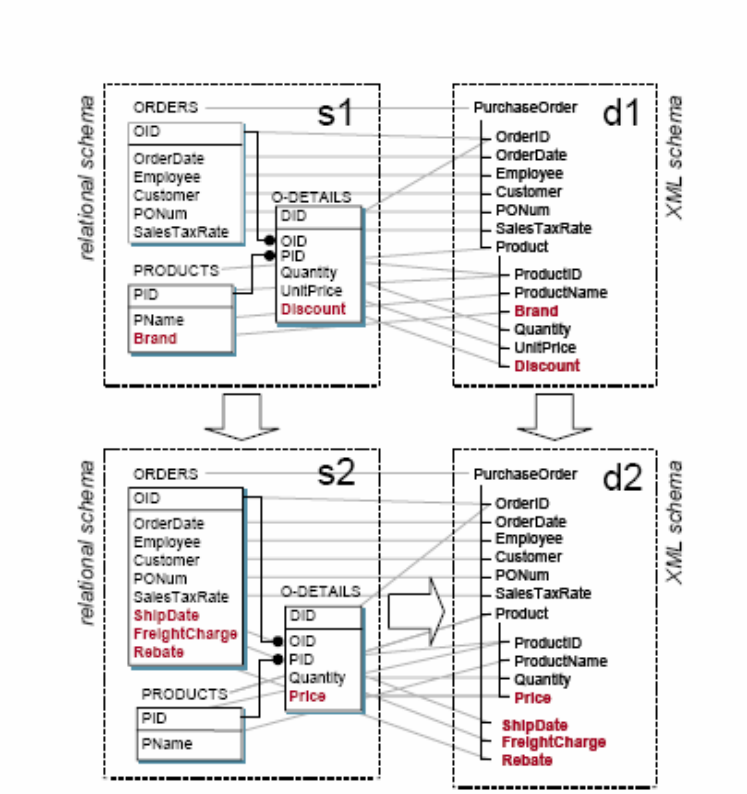


Figura 13. Escenario de ilustración.

La propagación de cambios descrita puede realizarse como se comenta a continuación. Primero, los cambios introducidos por s2 necesitan ser detectados. A continuación, las imágenes d1 de los elementos eliminados en s1 necesitan eliminarse de d1. Por último, las partes del esquema XML de las columnas agregadas y renombradas en s1 necesitan ser unidas en d1 para obtener d2. Durante estos pasos, se requiere la intervención de un ingeniero, por ejemplo, para decidir cuándo debería añadirse la nueva columna “Rebate” al esquema de intercambio o cuándo debería omitirse si no forma parte de los datos de intercambio. Aún así, la mayor parte del trabajo es mecánica y puede automatizarse.

Destacar que el procedimiento indicado podría aplicarse en el caso inverso, cuando el esquema XML d1 es el que ha sido modificado y los cambios han de propagarse al esquema relacional s1. Una idea clave de la gestión genérica de modelos es resolver tales tareas en un alto nivel de abstracción usando un *script* genérico y conciso.

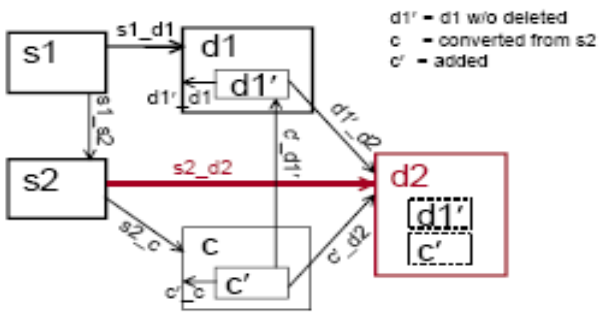


Figura 14. Representación esquemática de una solución de propagación de cambios para el escenario de la Figura 13.

Abajo se presenta un *script* de gestión de modelos que implementa la solución para el escenario descrito de propagación de cambios, y que es directamente ejecutable por RONDO. Para explicar los pasos individuales del *script*, se usará la representación esquemática de la solución mostrada en la Figura 14. Los rectángulos s1, s2, d1 y d2 representan los cuatro esquemas de la Figura 13. Los arcos entre los rectángulos denotan los *mappings* entre los esquemas. Por ejemplo, las correspondencias entre los esquemas s1 y d1 en la Figura 13 se muestran como un arco simple desde el rectángulo s1 al d1 en la Figura 14.

En la parte de debajo de la Figura 14, hay un esquema c, el cual no aparece en la Figura 13. Para ver por qué es necesario, es importante recordar que s1 y d1 están expresados usando dos lenguajes de esquema diferentes. Los elementos del nuevo esquema añadidos a s1, a diferencia de s2, no tienen partes homólogas en el esquema d1. Por ejemplo, el atributo "ShipDate" añadido a la relación "ORDERS", necesitar ser convertido en un subelemento del tipo complejo "PurchaseOrder" en el esquema XML. Este paso, con frecuencia es citado por la literatura como una traducción de esquemas. En esta aproximación, se asume que dicha traducción está disponible como un operador llamado "SQL2XSD", que toma como entrada el esquema relacional y produce como salida un esquema XML y un *mapping* entre los elementos originales y resultantes. De esta manera, el esquema c y el *mapping* s2_c entre s2 y c mostrados en la Figura 14 se obtienen como <c,s2_c> = SQL2XSD(s2). Nótese que el esquema c no es todavía el resultado deseado d2; por ejemplo, c puede contener un tipo complejo innecesario (O-DETAILS), y puede diferir estructuralmente de d1.

Así, la aproximación de RONDO para el escenario de propagación de cambios puede expresarse con el siguiente *script*:

```
s1_s2 = Match(s1, s2);                                                          (1)
<d1′, d1′_d1>=Delete(d1, Traverse(All(s1) − Domain(s1_s2), s1_d1));             (2)
<c′, c′_c>= Extract(c, Traverse(All(s2) − Range(s1_s2), s2_c));                 (3)
c′_d1′ = c′_c ∗ Invert(s2_c) ∗ Invert(s1_s2) ∗ s1_d1 ∗ Invert(d1′_d1);          (4)
<d2, c′_d2, d1′_d2> = Merge(c′, d1′, c′_d1′);                                   (5)
s2_d2 = s2_c ∗ Invert(c′_c) ∗ c′_d2 + Invert(s1_s2) ∗ s1_d1 ∗ Invert(d1′_d1) ∗  d1′_d2;   (6)
return <d2, s2_d2>;                                                             (7)
```

El *script* define un operador genérico *"PropagateChanges"*, que toma seis parámetros como entrada (incluyendo el esquema c), y produce dos valores resultado <d2,s2_d2> como salida. A continuación, se explica el *script* línea por línea:

1. En la línea 1, los esquemas s1 y s2 son "combinados" para detectar cambios. El resultado es un *mapping* s1_s2, mostrado esquemáticamente en la Figura 14. De manera informal, el *mapping* conecta los elementos equivalentes de s1 y s2. Los nuevos elementos de s2 (por ejemplo, "ShipDate") y los elementos eliminados de s1 (por ejemplo, "Brand") no tienen combinaciones de partes homólogas, por lo que permanecen sin conectar.
2. La línea 2 ilustra cómo se pueden combinar los operadores. Primero, los elementos eliminados de s1 son identificados usando la expresión All(s1) – Domain(s1_s2), por ejemplo, todos los elementos de s1 sin elementos combinados. A continuación, estos elementos son usados para recorrer el *mapping* s1_d1. Por ejemplo, el atributo relacional eliminado "Brand" atraviesa s1_d1 y da como resultado el elemento de esquema XML "Brand" de d1. Finalmente, estas imágenes de d1 de elementos eliminados son eliminadas de d1 usando el operador *"Delete"*. El resultado es un nuevo esquema d1' (un

subesquema de d1), y un *mapping* d1'_d1, que describe cómo d1' se relaciona con d1.

3. La línea 3 es bastante similar con la línea 2. Los nuevos elementos de s2, por ejemplo, aquellos fuera del rango de s1_s2, atraviesan s2_c en el modelo convertido c. Por ejemplo, la imagen del atributo relacional "ShipDate" es un elemento esquema XML "ShipDate" obtenido por conversión. Un subesquema c', que contiene las imágenes de los nuevos elementos, es extraído de c usando el operador *"Extract"*, que también devuelve el *mapping* c'_c. Además de los elementos obtenidos por atravesar *mappings* como "ShipDate", c' contiene elementos extra de c, como el tipo complejo que encierra "ShipDate", para hacer a c' un esquema XML bien formado. Estos elementos extras se llaman elementos de soporte.
4. En este punto, d1' es un subesquema de d1 sin elementos eliminados, y c' contiene los elementos añadidos y sus elementos de apoyo. Los esquemas d1' y c' necesita unirse para obtener el resultado final d2 (línea 5). El *mapping* entre d1' y c' se muestra en la Figura 14 como un arco que conecta los dos rectángulos que los contiene. Este *mapping* puede obtenerse componiendo los *mappings* existentes entre c', c, s1, s2, d1 y d1' como c'_c * Invert(s2_c) * Invert(s1_s2) * s1_d1 * Invert(d1'_d1). Para conseguir la composición correcta, los *mappings* s2_c, s1_s2 y d1'_d1 necesitan ser invertidos.
5. El resultado final de la propagación de cambios, el esquema d2, es computado por el operador *"Merge"*. Además, el operador devuelve dos *mappings*, c'_d2 y d1'_d2, que describen como d2 se relaciona con las entradas del *Merge* c' y d1'.
6. Como último paso, se computa s2_d2, una nueva versión del mapping s1_d1 dado como parte de la entrada. Es necesario s2_d2 para garantizar que el *script* de propagación de cambios puede ser aplicado nuevamente si el esquema origen cambia nuevamente. Puesto que d2 se obtiene uniendo d1' y c', el mapping s2_d2 es básicamente una unión de dos mappings, uno entre s2 y la parte d1' de d2, y otro entre s2 y la parte c' de d2. Estos dos mappings pueden obtenerse por composición como s2_c * Invert(c'_c) * c'_d2 and Invert(s1_s2) * s1_d1 * Invert(d1'_d1) * d1'_d2.

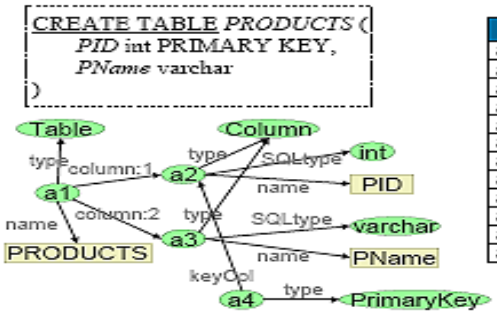
El *script* especificado anteriormente no se limita a la propagación de cambios de un esquema relacional a un esquema XML. De hecho, el problema de la propagación inversa puede resolverse usando el mismo *script* asignando los esquemas XML de partida y modificados a s1 y s2 respectivamente, y el esquema relacional a d1. Por supuesto, los parámetros de entrada c y s2_c han de obtenerse usando un convertidor diferente, por ejemplo, <c,s2_c> = XSD2SQL(s2).

En esta aproximación, cada resultado intermedio del *script* puede ser examinado y ajustado por un ingeniero usando una herramienta gráfica. Específicamente, el resultado del *"Match"* en la línea 1 puede ser post procesado para eliminar combinaciones incorrectas y/o perdidas. De manera similar, la unión en la línea 5 es en general un proceso semiautomático, que requiere de intervención humana. Finalmente, ajustando los resultados intermedios de la composición de operadores en las líneas 2 y 3, el ingeniero puede decidir qué elementos añadidos o eliminados no se deberían propagar.

7.2. Arquitectura de RONDO

Las aplicaciones de gestión de modelos tratan con un amplio abanico de artefactos los cuales no incluyen solamente esquemas, como el relacional y el XML utilizados en el ejemplo, sino también definición de vistas, especificaciones de interfaces, etc. En RONDO, los modelos se representan como grafos etiquetados dirigidos. Esta representación de grafo es muy flexible y puede acomodarse virtualmente a cualquier tipo de modelo.

Además, también se introducen dos estructuras adicionales denominadas "morfismos" y "selectores". Los "morfismos" son relaciones binarias que establecen correspondencias n:m entre los elementos de dos modelos (nodos de dos grafos). Por ejemplo, en el escenario utilizado, los morfismos se utilizan para conservar las trazas de las partes XML de los elementos del esquema relacional. En este caso, dos morfismos, uno entre s1 y d1, y otro entre s2 y d2, como se muestra en la Figura 13 usando líneas grises. El otro concepto, "selector", es un conjunto de elementos usados en modelos. Un beneficio mayor de utilizar selectores es que varias operaciones, que producirían típicamente modelos no-bien formados, si son utilizadas directamente, pueden ser aplicadas a selectores de manera segura.



S	P	O	N
a1	type	Table	
a1	column	a2	1
a1	column	a3	2
a1	name	"PRODUCTS"	
a2	type	Column	
a2	SQLtype	int	
a2	name	"PID"	
a3	type	Column	
a3	SQLType	varchar	
a3	name	"PName"	
a4	type	PrimaryKey	
a4	keyCol	a2	

Figura 15. Ejemplo de modelo como grafo y 4-tupla.

7.2.1. Modelos

En RONDO, los modelos se representan como grafos etiquetados dirigidos. Los nodos de estos grafos denotan elementos del modelo, como por ejemplo relaciones y atributos en esquemas relaciones, definiciones de tipos en esquemas XML, etc. Se asume que cada elemento se identifica de manera única mediante un identificador de objeto (OID). Un grafo dirigido es un conjunto de etiquetas <s, p, o> donde "s" es el nodo origen, "p" es el nombre de la etiqueta, y "o" es el nodo destino. Para un origen "s" dado y una etiqueta "p", los nodos destino tienen que estar ordenados secuencialmente. Su orden puede capturarse mediante una propiedad en la etiqueta. Así, conceptualmente un grafo puede verse como una relación M con cuatro atributos, M(S: OID, P: OID, O: OID U Literal, N: integer), donde N es un atributo opcional usado para ordenación y S, P, O forman una clave única. Los identificadores de nodos y los nombres de etiquetas se trazan desde el conjunto de OIDs, que pueden implementarse como enteros, punteros, reales, URIs o cualquier otro tipo de datos. Los literales incluyen cadenas de caracteres, enteros, reales y cualquier otro tipo de datos. El tipo del atributo O se define como una unión del tipo de los OIDs y los literales.

Considérese el ejemplo de la Figura 15. Ésta ilustra como una tabla relacional "PRODUCTS" definida en SQL DDL (arriba a la izquierda) está representada como un grafo (abajo a la izquierda) y un correspondiente conjunto de 4-tuplas (a la derecha). Los óvalos en el grafo denotan OIDs, y los rectángulos denotan literales. Los nodos a1, a2, a3 representan la tabla "PRODUCTS" y sus columnas "PID" y "PName" respectivamente. El nodo a4 representa la restricción de clave primaria sobre PID. Por legibilidad, los identificadores como "Table" o "Column" están explicados como nombres más que como IDs opácos. El orden de las columnas identificadas por los nodos a2 y a3 está determinado por los valores 1 y 2 del atributo N (cuarto atributo de la tabla con 4-tuplas).

8. CONCLUSIÓN

En los puntos 5 y 7 se han presentado dos DSLs diferentes, MOMENT y RONDO, para la definición de operadores. Ambos DSLs, tienen la misma finalidad, la especificación de operadores que permitan manipular modelos, sin embargo, difieren en la forma de hacerlo y ejecutarlos.

El DSL de MOMENT ofrece una sintaxis más intuitiva y sencilla que la de RONDO, ya que basándonos en los ejemplos del operador de propagación de cambios utilizados en cada unos de ellos, es posible apreciar como en el DSL de MOMENT queda mucho más claro cuáles son los modelos de entrada y los modelos de salida. Además, la utilización de operadores existentes en la definición del operador resulta más sencilla en MOMENT que en RONDO.

Puesto que en MOMENT a partir de la especificación textual de un operador, se obtiene un modelo de operador correspondiente, la definición de operadores toma un cariz de entidad propia que puede se reutilizada por otros operadores, pudiendo de esta manera soportar la composición de operadores. Por tanto, podemos concluir que en MOMENT los operadores son modelos, mientras que en RONDO son especificaciones textuales o *scripts*.

Respecto a la arquitectura de cada herramienta existe una diferencia notable en la forma en la que los modelos son representados, repercutiendo así en las estrategias de ejecución utilizadas. En MOMENT, los modelos son representados como conjuntos de elementos, mientras que en RONDO, se representan utilizando grafos dirigidos. La representación a través de grafos está más extendida dentro del ámbito de MDE y existe mucha literatura al respecto.

IV. FUNDAMENTOS DE TRANSFORMACIÓN DE MODELOS

1. INTRODUCCIÓN

Uno de los mejores métodos para combatir la complejidad del desarrollo software es el uso de la abstracción y la descomposición del problema. La práctica del modelado de software se ha convertido en la vía principal de implementar estos principios. Las aproximaciones de ingeniería dirigida por modelos persiguen trasladar los lenguajes de programación de tercera generación (3GL) a modelos (en particular modelos expresados en UML). El objetivo del desarrollo dirigido por modelos es incrementar la productividad y reducir el tiempo de puesta en el mercado de los productos, permitiendo desarrollar en un nivel de abstracción más elevado y usando conceptos más cercanos al dominio del problema. El aspecto más importante del desarrollo dirigido por modelos consiste en transformar estos modelos de alto nivel en los llamados modelos específicos de plataforma (PSM) que pueden ser utilizados para la generación de código.

En los últimos años, la industria del desarrollo software se ha centrado en la estandarización de notaciones visuales de modelado. Como resultado de este esfuerzo surgió el Lenguaje Unificado de Modelado (UML), que ha sido bastante aceptado por la industria del software, y finalmente convertido en estándar de la OMG. La mayoría de las técnicas de modelado software usan UML, por lo que se ha visto asociado con la iniciativa del desarrollo dirigido por modelos.

UML ofrece numerosas opciones a los desarrolladores para especificar sistemas. Un modelo UML puede representar gráficamente la estructura y/o comportamiento de un sistema de estudio bajo un determinado punto de vista y un cierto nivel de abstracción. De esta manera, es posible manejar la complejidad de un sistema a través de varios modelos, cada uno de los cuales capta un aspecto diferente de la solución. Así, los modelos pueden utilizarse no solo de una manera horizontal (para describir diferentes aspectos del sistema), sino de una manera vertical (para refinar desde un nivel de abstracción elevado a uno más concreto).

Trabajar con múltiples modelos interrelacionados que describen un sistema software, requiere un notable esfuerzo para garantizar la consistencia global. Por tanto, sería muy importante automatizar las tareas de comprobación de la consistencia y de la sincronización, ya que aumentaría la productividad de los desarrolladores y la calidad de los modelos. Además de la sincronización vertical y horizontal de modelos, el grueso de otras actividades, como las listadas abajo, pueden reducirse significativamente mediante la automatización:

- **Síntesis** de un nivel de abstracción elevado (por ejemplo, un análisis o un modelo de diseño) en otro más concreto (por ejemplo, un modelo de un programa Java). Un ejemplo típico de síntesis es la generación de código fuente a partir de un modelo de diseño.
- **Ingeniería inversa**. Es lo contrario de la síntesis y consiste en obtener un nivel de abstracción elevado a partir de uno concreto.
- **Migración** de un programa escrito en un lenguaje a otro lenguaje distinto, pero conservando el mismo nivel de abstracción.
- **Optimización**, el objetivo de una trasformación es mejorar ciertas cualidades operacionales conservando la semántica del software.
- **Refactorización**, cambiar la estructura interna del software para mejorar ciertas características como modificabilidad, reusabilidad, modularidad, adaptabilidad o entendibilidad, sin modificar su comportamiento.

- **Refinamiento**, añadir pasos al proceso de desarrollo software que van desde la toma de requisitos a la finalización del producto en desarrollo.
- **Simplificación** y **normalización**, utilizado para reducir la complejidad sintáctica.
- **Adaptación de componentes**, para modificar y adaptar el código de componentes software existentes de forma dinámica o estática.

Muchas de estas actividades pueden mejorarse como procesos automáticos, que toman uno o varios modelos origen como entrada y producen uno o varios modelos destino como salida, siguiendo un conjunto de reglas de transformación. Nos referimos a este proceso como **transformación de modelos**.

Para que la ingeniería dirigida por modelos sea una realidad, las herramientas deben ser capaces de proporcionar la automatización de las transformaciones de modelos. El desarrollo de estas herramientas no debería ofrecer solamente la posibilidad de aplicar transformaciones de modelos predefinidas sino también proporcionar un lenguaje que permita a los usuarios definirse sus propias transformaciones y poder ejecutarlas posteriormente.

2. TAXONOMÍA DE TRANSFORMACIÓN DE MODELOS

2.1. ¿Qué necesita ser transformado en qué?

La primera cuestión importante hace referencia a los artefactos origen y destino de una transformación de modelo. Si estos artefactos son programas, se usa el término de transformación de programas. Si los artefactos software son modelos, entonces se habla de transformación de modelos. Sin embargo, todas las transformaciones de programas pueden llevarse a cabo como transformaciones de modelos, siempre y cuando sea posible clasificar la estructura de los modelos que intervienen en la transformación. De forma más específica, algunos sistemas pueden representarse como árboles estrictos, grafos o conjuntos de elementos.

2.1.1. Número de modelos origen y destino

Una característica importante en la transformación de modelos es el número de modelos origen y destino que pueden intervenir. Así, hay cuatro tipos de transformaciones:

- 1-1. Un modelo origen y un modelo destino
- 1-n. Un modelo origen y varios modelos destino
- n-1. Varios modelos origen y un solo modelo destino
- n-n. Varios modelos origen y varios modelos destino

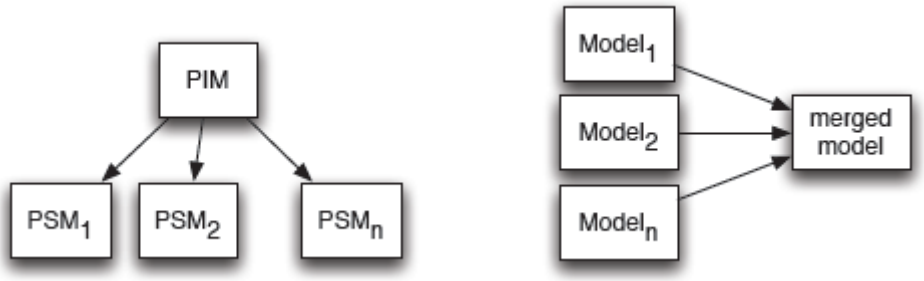


Figura 16. Ejemplos de transformación de modelos.

2.1.2. Espacio tecnológico

El concepto de espacio tecnológico ya ha sido introducido en el apartado 4.3 de la sección I, y viene determinado por el metamodelo que utiliza (nivel M3). Dada una transformación de modelos, sus modelos origen y destino pueden pertenecer a un mismo o a diferentes espacios tecnológicos. En este último caso, son necesarias herramientas y técnicas que definen transformaciones que establecen un puente entre espacios tecnológicos distintos.

2.1.3. Transformaciones endógenas versus transformaciones exógenas

Basándonos en el lenguaje en el que se expresan los modelos origen y destino de una transformación, puede establecerse una distinción entre transformaciones de tipo endógeno y exógeno. Las transformaciones endógenas son transformaciones son aquellas transformaciones que comparten el mismo metamodelo, mientras que las transformaciones exógenas se definen entre modelos que conforman a metamodelos diferentes.

Las transformaciones endógenas en las que solo interviene un único modelo, el modelo origen y el modelo destino es el mismo, y reciben el nombre de transformaciones *in-place*. El resto de transformaciones endógenas en las que interviene más de un modelo, reciben el nombre de transformaciones *out-place*. Todas las transformaciones exógenas son de tipo *out-place*.

2.1.4. Transformaciones horizontales versus transformaciones verticales

Una transformación horizontal es una transformación donde los modelos origen y destino residen en el mismo nivel de abstracción (por ejemplo refactorización y migración). Una transformación vertical es una transformación donde los modelos origen y destino residen en niveles de abstracción diferentes (por ejemplo refinamiento y generación de código).

3. CLASIFICACIÓN DE APROXIMACIONES SOBRE LA TRANSFORMACIÓN DE MODELOS

Llevar a cabo una transformación de modelos, tomando uno o varios modelos como entrada y produciendo uno o varios modelos como salida, requiere una compresión clara de la sintaxis abstracta y la semántica tanto del origen como del destino. Una técnica común para la definición de la sintaxis abstracta de modelos y de la interrelación entre los elementos de los modelos es el meta-modelado. La práctica ha demostrado que para lenguajes visuales de modelado, existen grandes ventajas cuando la implementación se realiza en términos del meta-modelo del lenguaje. Existen determinadas herramientas que permiten definir un lenguaje visual específico de dominio mediante la especificación de un meta-modelo, por ejemplo Dome, GME, MetaEdit+, ParadigmP. UML está también especificado en términos de un meta-modelo, el cual está implementado en un gran número de herramientas como Rational Rose, Objecteering o Together.

En general, estas herramientas proporcionan al usuario tres aproximaciones arquitectónicas para la definición de transformaciones: manipulación directa del modelo, la representación intermedia y soporte para lenguaje de transformación.

- Manipulación directa. Las herramientas ofrecen a los usuarios acceso a una representación interna del modelo y una API para manipularla.
- Representación intermedia. La herramienta puede exportar el modelo en un formato estándar, normalmente XML. Una herramienta externa puede coger el modelo exportado y transformarlo.
- Soporte para lenguaje de transformación. La herramienta ofrece un lenguaje que proporciona un conjunto de constructores y mecanismos para expresar, componer y aplicar transformaciones de forma explícita.

Una herramienta que posee una (o varias) de estas aproximaciones, permite la especificación de transformaciones de modelos. A continuación, se destacan algunas de las ventajas y de las limitaciones de las diferentes aproximaciones.

Una ventaja de la aproximación basada en manipulación directa es que el lenguaje usado para acceder y manipular la API, puede ser tanto un lenguaje común como VisualBasic o Java, como una variante propietaria de un lenguaje común, por lo que los desarrolladores necesitan muy poco esfuerzo para escribir transformaciones. Además, en general, éstos se encuentran más cómodos codificando algoritmos complejos con lenguajes procedurales. Una desventaja de esta aproximación es que la API normalmente restringe el tipo de transformaciones que pueden realizarse. También, dado que los lenguajes de programación son de propósito general, carecen de abstracciones de alto nivel para la especificación de transformaciones. En consecuencia, la codificación de transformaciones puede llevar mucho tiempo, ser engorrosa y los algoritmos de transformación difíciles de mantener.

Respecto a la aproximación de representación intermedia, muchas herramientas de UML permiten la exportación e importación de modelos a/desde XMI. XMI es un estándar basado en XML para el intercambio de modelos UML, por lo que también es posible utilizar herramientas de XML como XSLT para realizar transformación de modelos. El principal inconveniente de esta aproximación es que al realizar la transformación en una herramienta externa, pueden surgir problemas relacionados con la sincronización de modelos, la violación de restricciones de integridad y la dificultad en llevar a cabo un diálogo interactivo con el usuario.

El soporte para lenguaje de transformación, como su propio nombre indica, proporciona un lenguaje específico de dominio para especificar transformaciones de modelos y, en consecuencia, ofrece el mayor potencial de las tres aproximaciones. En este contexto, hay muchos lenguajes que pueden usarse para especificar y ejecutar transformaciones, algunos de ellos incluso ofrecen constructores visuales. Estos lenguajes pueden ser declarativos, imperativos o una mezcla de ambos.

4. CARACTERÍSTICAS DESEABLES DE UN LENGUAJE DE TRANSFORMACIÓN DE MODELOS

Las características deseables y recomendables por un lenguaje de transformación de modelos que soporte desarrollo dirigido por modelos según [SeKo03] son:

- Ejecutabilidad
- Estar implementado de manera eficiente
- Ser completamente expresivo y no ambiguo
- Facilitar productividad al desarrollador de forma clara, precisa y concisa
    - El lenguaje debería ser declarativo haciendo implícitos conceptos o mecanismos que puedan ser interpretados intuitivamente por el contexto
    - El lenguaje debería diferenciar claramente la descripción de reglas seleccionadas del modelo origen de las reglas usadas en la producción del modelo destino
    - El lenguaje debería ofrecer construcciones gráficas de forma que los objetos representados sean más concisos e intuitivos en comparación con los representados en formas textuales
- Proporcionar los medios para combinar transformaciones y formar transformaciones compuestas, ofreciendo al menos operadores para secuenciación, selección condicional y repetición de transformaciones
- Ofrecer la forma de definir condiciones bajo las cuales se ejecuten las transformaciones
- Preservación

5. CRITERIOS DE ÉXITO PARA UN LENGUAJE DE TRANSFORMACIÓN O UNA HERRAMIENTA

Los criterios de éxito para un lenguaje de transformación de modelos o una herramienta que soporte transformación de modelos [MeVa05] son:

- **Validación y *testing* de transformaciones**. Puesto que las transformaciones pueden considerarse como programas especiales, es necesario aplicar técnicas sistemáticas de validación y *testing* a las transformaciones para asegurar que tienen el comportamiento deseado.

- **Tratar con modelos incompletos o inconsistentes**. Es importante transformar modelos en una fase temprana del ciclo de desarrollo software, cuando los requisitos son descritos en lenguaje natural. Esto, con frecuencia, da como resultado modelos ambiguos, inconsistentes o incompletos, por lo que es necesario tener mecanismos que permitan detectar estos problemas y ayuden a resolverlos, de forma que los modelos puedan ser transformados.
- **Agrupar, componer y descomponer transformaciones**. La posibilidad de componer transformaciones existentes en nuevas transformaciones compuestas es muy útil para incrementar la legibilidad, modularidad, mantenibilidad y escalabilidad de un lenguaje de transformación. La descomposición de transformaciones complejas en otras más sencillas, requiere de mecanismos de control que especifican cómo han de combinarse estas transformaciones más pequeñas.
- **Bidireccionalidad**. Los lenguajes y herramientas que tienen la propiedad de bidireccionalidad requierem unas pocas reglas de transformación más, puesto que cada transformación puede utilizarse en dos direcciones diferentes: para transformar modelos origen en modelos destino, y la transformación inversa, transformar el modelo o los modelos destino en el modelo o los modelos origen.
- **Soporte de trazabilidad y propagación de cambios**. Para soportar trazabilidad, el lenguaje de transformación o la herramienta necesitan proporcionar mecanismos para mantener un enlace explícito entre los modelos origen y destino de una transformación de modelo. Para soportar propagación de cambios, el lenguaje de transformación o la herramienta, tienen que tener mecanismos de comprobación de consistencia y un mecanismo de actualización incremental.
- **Interoperabilidad**. La herramienta debería también ser interoperable o fácil de integrar con otras herramientas usadas en los procesos de ingeniería del software. Para lograrlo, la herramienta necesita proporcionar puentes a otros espacios tecnológicos.
- **Extensibilidad**. La flexibilidad de una herramienta depende de la facilidad con la que la herramienta puede extenderse con una nueva funcionalidad.
- **Estandarización**. La herramienta de transformación debería cumplir con todos los estándares.

6. ESTANDARIZACIÓN DE LA TRANSFORMACIÓN DE MODELOS

La utilización de MDA está ganando cada vez más interés entre la comunidad del software. MDA es un *framework* para el desarrollo de software dirigido por modelos que define tres pasos para ir de un alto nivel de diseño a la realización:

Paso 1: Un modelo de sistema software se construye de forma que es independiente de cualquier implementación tecnológica. Este tipo de modelos reciben el nombre de modelos independientes de plataforma (PIM).

Paso 2: Un PIM se transforma en uno o varios modelos específicos de plataforma (PSM), usando una estrategia particular de *mappings*. Un PSM especifica un sistema usando constructores implementados en una tecnología determinada.

Paso 3: Los PSMs son transformados en código.

Está claro que para entender la potencia de la iniciativa MDA, es necesario ser capaz de describir la transformación entre un PIM y diferentes PSMs. Por esta razón, la aplicación de MDA en el desarrollo del software y de sus actividades relacionadas, son los principales estímulos para estandarizar lenguajes de transformación de modelos.

La especificación *Common Warehouse Metamodel* (CWM) de la OMG, define una manera genérica para describir transformaciones de caja blanca y de caja negra. En CWM, las transformaciones de caja negra asocian los elementos origen y destino sin describir cómo se obtiene uno a partir del otro. Las transformaciones de caja blanca, describen los enlaces entre los elementos origen y destino en términos de un elemento explícito llamado “transformación”, que a su vez está asociado a otro término denominado “expresión procedural”. Una expresión procedural define la operación de transformación; puede ser expresada en cualquier lenguaje capaz de coger elementos origen y producir elemento(s) destino. Por tanto, CWM no proporciona mecanismos o lenguajes para llevar a cabo transformaciones, sino que es usado para describir *mappings*.

Para cubrir este vacío, la OMG inició la búsqueda de un lenguaje estándar para la especificación de transformaciones, que ha finalizado con la especificación *Query View Transformation* (QVT). Con QVT se pretende potenciar definitivamente la iniciativa MDA dentro de la ingeniería del software.

V. EL LENGUAJE QVT-RELATIONS

1. LA ESPECIFICACIÓN QVT

Para soportar transformaciones se ha seguido el estándar *Query View Transformation* (QVT) [MOFQVT] propuesto por la OMG. La especificación de QVT depende de otros dos estándares de la OMG como son MOF 2.0 y OCL 2.0. De esta manera, la utilización de la especificación de QVT para especificar transformaciones, aporta reutilización de tecnología que sigue estándares y reducción de la curva de aprendizaje de la herramienta.

La especificación QVT se define a través de dos dimensiones ortogonales: la dimensión del lenguaje y la dimensión de interoperabilidad, cada una de las cuales tiene una serie de niveles. La intersección de niveles de las dos dimensiones define un punto de cumplimiento QVT (QVT - compliance).

La dimensión del lenguaje define los diferentes lenguajes de transformación presentes en la especificación QVT. Concretamente son tres: Relations, Core y Operational, y la principal diferencia entre ellos es su naturaleza declarativa o imperativa. Para integrar QVT en MOMENT se ha elegido el lenguaje Relations de cara a aprovechar su naturaleza declarativa, su trazabilidad implícita y la sintaxis sencilla que ofrece para definir transformaciones y relaciones de equivalencia entre modelos.

En la dimensión de la interoperabilidad se encuentran aquellas características que permiten a una herramienta que cumple el estándar QVT interoperar con otras herramientas. Concretamente, son cuatro: sintaxis ejecutable, XMI ejecutable, sintaxis exportable y XMI exportable.

- La sintaxis ejecutable se traduce en una implementación que facilite la importación o lectura, y posterior ejecución de una sintaxis concreta que describa una transformación definida en el lenguaje dado por la dimensión del lenguaje.
- XMI ejecutable dictamina que una implementación debe facilitar la importación o lectura, y posterior ejecución de una serialización XMI de una transformación que conforma con el metamodelo de MOF del lenguaje dado por la dimensión del lenguaje.
- La sintaxis exportable establece que una implementación debe proporcionar facilidad para exporta una transformación entre modelos en la sintaxis concreta del lenguaje dado por la dimensión del lenguaje.
- XMI exportable es el nivel de interoperabilidad que debe facilitar la exportación de transformaciones entre modelos como serializaciones XMI que conformen con el metamodelo MOF del lenguaje dado por la dimensión del lenguaje.

Dado que MOMENT está integrado en la herramienta industrial EMF, que soporta serialización de modelos siguiendo el estándar XMI de la OMG, se da soporte implícito de importación y exportación de modelos serializados como XMI.

En definitiva, en cuanto interoperabilidad se refiere, en MOMENT se permite la ejecución de modelos que conforman al metamodelo QVT Relations, la ejecución de cualquier especificación textual de una transformación expresada con el lenguaje Relations, así como la importación y exportación de modelos serializados con formato XMI. Por tanto, tras integrar el lenguaje Relations de QVT en MOMENT, se obtienen los siguientes puntos de cumplimiento QVT: QVT-Relations-Sintaxis ejecutable, QVT-Relations-XMI ejecutable y QVT-Relations-XMI exportable.

		Interoperabilidad			
Lenguaje		Sintaxis Ejecutable	XMI Ejecutable	Sintaxis Exportable	XMI Exportable
	Core				
	Relations	X	X		X
	Operacional				

Tabla 1. Cumplimiento de QVT en MOMENT

La especificación QVT tiene una naturaleza híbrida: declarativa e imperativa, con la parte declarativa dividida en una arquitectura de dos niveles que formará el *framework* para la ejecución semántica de la parte imperativa.

1.1. Arquitectura de dos niveles

La parte declarativa de esta especificación está estructurada en una arquitectura de dos capas:

- Un metamodelo Relations conocido, y un lenguaje que soporta *pattern matching* de objetos complejos y creación de plantillas de objetos. La trazabilidad entre los elementos del modelo involucrados en la transformación se crea implícitamente.
- Un metamodelo Core, y un lenguaje definido usando extensiones mínimas de EMOF y OCL. Todas las clases de trazabilidad están definidas explícitamente como modelos de MOF, y la creación y eliminación de una instancia de clase de trazabilidad se define de la misma manera que la creación y eliminación de cualquier otro objeto.

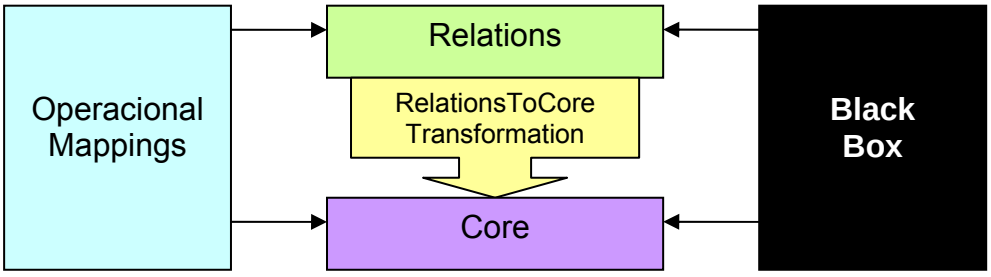


Figura 17. Relaciones entre los metamodelos QVT.

1.1.1. Relations

El lenguaje Relations proporciona una especificación declarativa de relaciones entre modelos MOF. Soporta *pattern matching* de objetos complejos y de manera implícita crea clases de trazabilidad y sus instancias correspondientes, de forma que permite guardar todo lo que ocurre durante la ejecución de una transformación.

1.1.2. Core

Este es un lenguaje/modelo que solo soporta *pattern matching* sobre un conjunto de variables, evaluando condiciones sobre variables de un conjunto de modelos. Core trata a todos los elementos del modelo origen, destino y de trazabilidad de manera simétrica y su potencia es equiparable a la del lenguaje Relations. Esto se debe, a que por su relativa simplicidad su semántica puede ser definida de manera más sencilla, aunque la definición de transformaciones sea más verbosa. Además, los modelos de trazabilidad han de ser definidos de manera explícita y no pueden deducirse de la definición de la transformación como ocurría en el lenguaje Relations.

El modelo Core debe ser implementado directamente, o simplemente usado como referencia para la semántica del Relations, la cual está mapeada a Core a través de una transformación.

1.1.3. Analogía de máquina virtual

Es posible trazar una analogía con la arquitectura Java™, donde el lenguaje Core es como *Java Byte Code* y la semántica de Core se comporta como la máquina virtual Java. El lenguaje Relations desempeña el rol del lenguaje Java, y la transformación estándar de Relations a Core es la especificación de un compilador Java que genera *Byte Code*.

1.2. Implementaciones imperativas

Además de los lenguajes Relations y Core, que tienen la misma semántica en diferentes niveles de abstracción, existen dos mecanismos para invocar implementaciones imperativas de transformaciones desde Relations o desde Core: un lenguaje estándar, Operational Mappings, y una operación MOF no estándar de caja negra (*Black-box*). Cada relación define una clase que será instanciada para dar soporte a trazabilidad entre elementos de dos modelos que están siendo transformados. Esta clase define un *mapping* uno-a-uno hacia la signatura de una operación que implementa el Operational Mappings o la *Black-Box*.

1.2.1. Lenguaje Operational Mappings

Este lenguaje está especificado como estándar para proporcionar implementaciones imperativas. Proporciona extensiones OCL con efectos laterales que da un mayor estilo procedural y una sintaxis más cercana a la programación imperativa.

Este lenguaje define operaciones de *mappings* que pueden ser usadas para implementar una o más Relations a partir de una especificación Relations, cuando sea difícil proporcionar una especificación declarativa más pura. Estas operaciones pueden invocar otras operaciones de *mappings* con el objetivo de crear modelos de trazabilidad entre elementos de modelos.

Es posible escribir transformaciones completas utilizando operaciones de *mapping*, denominándose este tipo de transformaciones: transformaciones operacionales.

1.2.2. Implementación *Black-box*

Las operaciones MOF deben poder derivarse de Relations haciendo posible integrarlas con cualquier implementación de una operación MOF con la misma signatura. Esto resulta beneficioso por las siguientes razones:

- Permite codificar algoritmos complejos en cualquier lenguaje de programación con correspondencia a MOF.
- Permite el uso de librerías específicas de dominio para calcular ciertas propiedades del modelo. Por ejemplo, en matemáticas, en ingeniería y en otros muchos campos existen grandes librerías que almacenan algoritmos específicos de dominio, que serían muy difíciles de expresar usando OCL.
- Permite ocultar la implementación de algunas partes de una transformación.

Sin embargo, esta integración puede resultar peligrosa puesto que la implementación asociada a esta integración tiene libre acceso a las referencias de objetos en los modelos. Las implementaciones *Black-box* no tienen una relación implícita hacia Relations, y cada *Black-box* debe implementar explícitamente una relación que sea responsable de conservar la trazabilidad entre los elementos del modelo relacionado con la implementación de la operación MOF.

En aras de extender la analogía con la arquitectura Java, la habilidad para invocar implementaciones "*Black-Box"* y *"Operational Mappings"*, puede considerarse equivalente con la llamada a la *Java Native Interface* (JNI).

1.3. Escenarios de ejecución

Las semánticas de los lenguajes Core y Relations da lugar a los siguientes escenarios de ejecución:

- Relaciones de equivalencia (transformaciones *Checkonly*) para verificar equivalencia entre modelos.
- Transformaciones de una dirección.
- Transformaciones bidireccionales.
- Habilidad para establecer relaciones entre modelos preexistentes, herramientas o mecanismos.
- Transformaciones incrementales.

Las aproximaciones de Operational Mappings y Black-box restringen estos escenarios, puesto que solo permiten la especificación de una transformación en una sola dirección. Sin embargo, el resto de escenarios definidos anteriormente son ejecutables con implementaciones imperativas o híbridas.

1.4. Metamodelos MOF

Esta especificación define tres paquetes principales, uno por cada lenguaje definido: QVTCore, QVTRelation y QVTOperational.

El paquete “QVTBase” define una estructura común para las transformaciones.

El paquete “QVTOperational” hereda del “QVTRelation” y usa el mismo *framework* para trazabilidad definido en este paquete. Además, usa expresiones imperativas definidas en el paquete “ImperativeOCL”.

Todo QVT depende del paquete “EssentialOCL” que proporciona OCL 2.0 y todos los paquetes de lenguajes dependen del paquete EMOF.

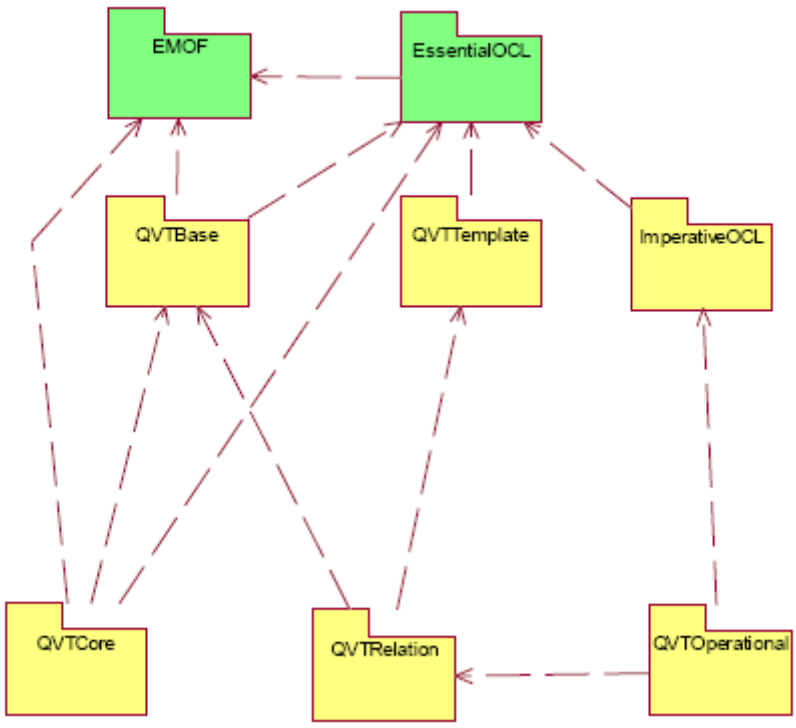


Figura 18. Dependencias de paquetes en la especificación QVT.

2. EL LENGUAJE RELATIONS

2.1. Transformaciones y tipos de modelos

En el lenguaje Relations, una transformación entre modelos candidatos se especifica como un conjunto de relaciones que han de tener lugar para llevar a cabo la transformación. Un modelo candidato es cualquier modelo que conforme a un tipo de modelo o metamodelo, que es una especificación de los diferentes tipos que pueden tener los elementos del modelo que lo conforma. Los modelos candidatos tienen un nombre y los tipos de los elementos que pueden contener, los cuales quedan restringidos al paquete al que se hace referencia en la declaración del modelo candidato. Por ejemplo:

```
transformation uml2Rdbms (uml : SimpleUML, rdbms : SimpleRDBMS)
```

En esta declaración de transformación llamada “uml2Rdbms”, hay dos modelos candidatos: “uml” y “rdbms”. El modelo “uml” se declara referenciando el paquete “SimpleUML” como su metamodelo, y el modelo “rdbms” hace lo propio con el paquete “SimpleRDBMS”. Una *“transformation”* puede invocarse tanto para comprobar la consistencia de dos modelos como para transformar un modelo en otro.

2.1.1. Dirección de ejecución de una transformación

Una *“transformation”* invocada para realizar una transformación se ejecuta en una dirección determinada seleccionando uno de los modelos candidatos como modelo destino u objetivo. El modelo destino debe estar vacío o contener elementos para ser relacionados por la transformación. El procedimiento de transformación se basa en comprobar que todas las relaciones definidas en la transformación se cumplen y para aquellas relaciones que no se cumplen, se intenta cumplirlas creando, eliminando o modificando el modelo destino.

2.2. Relaciones y dominios

Las relaciones en una transformación definen restricciones que deben satisfacer los elementos de los modelos candidatos. Una relación o *relation* definida para dos o más dominios, y un par de predicados *when* y *where*, especifica una relación que se debe satisfacer entre los elementos de los modelos candidatos.

Un dominio o *domain* es una variable tipada que puede hacer *matching* en un modelo de un metamodelo dado. Un dominio tiene un patrón que puede ser visto como un grafo de nodos de objetos, cuyas propiedades y links de asociación se originan de una instancia del tipo del dominio. Alternativamente, un patrón puede ser considerado como un conjunto de variables y un conjunto de restricciones, que los elementos del modelo vinculados a esas variables deben satisfacer para logar una correspondencia válida de ese patrón. Un patrón de dominio o *domain pattern* puede considerarse como una plantilla para objetos, y sus propiedades han de ser localizadas, modificadas o creadas en un modelo candidato para satisfacer la relación.

En el siguiente ejemplo, se declaran dos dominios que corresponderán a elementos de los modelos “uml” y “rdbms”. Cada dominio especifica un patrón simple,

un paquete con un nombre y un esquema con un nombre. Ambas propiedades de "nombre" se corresponden con la misma variable "pn", lo que implica que deberían tener el mismo valor:

```
relation PackageToSchema
{
   domain uml p: Package {name=pn}
   domain rdbms s: Schema {name=pn}
}
```

2.2.1. Cláusulas *when* y *where*

Como se muestra en el siguiente ejemplo de la relación "ClassToTable", una relación puede estar limitada por dos conjuntos de predicados, una cláusula *when* y una cláusula *where*. La cláusula *when* define las condiciones bajo las cuales se necesita satisfacer la relación, por lo que la relación "ClassToTable" se cumplirá solo cuando se cumpla la relación "PackageToSchema", es decir, cuando el paquete contiene la clase y el esquema contiene la tabla. La cláusula *where* define la condición que deben satisfacer todos los elementos de todos los modelos que participan en la relación, y puede restringir cualquiera de las variables en la relación y sus dominios. Por tanto, siempre que se cumpla la relación "ClassToTable", la relación "AttributeToColumn" también se cumplirá.

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

Las cláusulas *when* y *where* pueden contener cualquier expresión OCL además de las expresiones de invocación de la relación.

La invocación de relaciones permite componer relaciones complejas a partir de relaciones más simples.

2.2.2. Relaciones *Top-level*

Una transformación contiene dos tipos de relaciones: las *top-level* y las *no-top-level*. La ejecución de una transformación requiere que todas sus relaciones *top-level* se cumplan, mientras que las relaciones *no-top-level* solo han de ser satisfechas cuando son invocadas directa o transitivamente desde la cláusula *where* de otra relación.

```
Transformation uml2Rdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
      top relation PackageToSchema { . . . }
      top relation ClassToTable { . . . }
      relation AttributeToColumn { . . . }
}
```

Una relación *top-level* tiene la palabra clave *"top"* que la distingue sintácticamente. En el ejemplo anterior, tanto "PackageToSchema" como "ClassToTable" son relaciones *top-level* mientras que "AttributeToColumn" es una relación *no-top-level*.

2.2.3. *Check* y *Enforce*

El que una relación sea de transformación o no viene determinado por el dominio destino, que estará marcado como *checkonly* o *enforce*. Cuando una transformación se satisface en la dirección de un dominio *checkonly*, simplemente se comprueba si existe una correspondencia válida en el modelo relevante que satisfaga la relación. Cuando una transformación se ejecuta en la dirección de un dominio *enforce*, si la comprobación falla, se modifica el modelo destino tanto como sea necesario para que satisfaga la relación.

En el ejemplo siguiente, el dominio para el modelo "uml" está marcado como *checkonly* y el dominio para el modelo "rdbms" está marcado como *enforce*.

```
relation PackageToSchema
{
   checkonly domain uml p: Package { name=pn }
   enforce domain rdbms s: Schema  { name=pn }
}
```

Si estuviéramos ejecutando en la dirección de "uml" y existiera un esquema en "rdbms" para el cual no tenemos un paquete correspondiente con el mismo nombre en "uml", se detectaría una inconsistencia pero no se crearía el paquete puesto que el modelo "uml" es *checkonly* y no *enforce*.

Sin embargo, si estuviéramos ejecutando la transformación "uml2Rdbms" en la dirección de "rdbms", entonces para cada paquete en el modelo de "uml", la relación primero comprobaría si existe un esquema con el mismo nombre en el modelo "rdbms", y si no es así, se crearía un nuevo esquema en ese modelo con el nombre dado. Para considerar una variación del caso propuesto, si ejecutamos la transformación en la dirección de "rdbms" y no hay un paquete con el mismo nombre que en "uml", entonces el esquema se eliminará del modelo "rdbms" para forzar la consistencia en el dominio *enforce*.

La aplicación de las reglas depende únicamente del dominio destino.

2.3. *Pattern Matching*

Los patrones o *patterns* asociados a los dominios reciben el nombre de *"object template expressions"*. Para ilustrar cómo tiene lugar el *matching* de estos patrones utilizaremos el ejemplo del "ClassToTable" utilizado con anterioridad.

"ClassToTable" define varias expresiones de tipo *"object template expression"* que se utilizarán para hacer *pattern matching* en los modelos candidatos. Por ejemplo, el siguiente *"object template expression"* está asociado al dominio de "uml":

```
c: Class {
           namespace = p: Package { },
           kind = 'Persistent',
           name = cn
         }
```

Las combinaciones resultantes de un *"object template expression"* unen elementos de un modelo candidato con variables declaradas en el dominio. Un *"object template expression"* debe efectuarse en un contexto donde algunas de las variable del domino estén enlazadas con elementos de un modelo. En este caso el *"object template expression"* solo encuentra enlaces para tres variables del dominio.

El *pattern matching* ligará todas las variables de la expresión ("c", "p", y "cn"), empezando desde la variable raíz del domino "c" de tipo "Class". En este ejemplo, la variable "p" debería también estar ligada como resultado de la evaluación de la expresión "PackageToSchema(p,s)" en la cláusula *when*. El proceso de *matching* pasa por filtrar todos los objetos de tipo "Class" en el modelo "uml", eliminando cualquier objeto que no tenga los mismos valores literales que el *"object template expression"*. En el ejemplo, cualquier "Class" con su propiedad "kind" distinta de "'Persistent'" es eliminada.

Para propiedades que se comparan con variables, como "name=cn", aparecen dos casos. Si la variable "cn" tiene un valor asignado, entonces cualquier "Class" que no tenga el mismo valor para su propiedad "name" será eliminada. Si la variable "cn" es libre, como sucede en el ejemplo, se le asignará el valor de la propiedad del nombre de todas las clases que han sido eliminadas filtrando, debido a una incompatibilidad con otras comparaciones de la propiedad.

Entonces el *matching* avanza a propiedades cuyos valores son comparados con *"object template expressions"* anidadas. Por ejemplo, la propiedad "namespace = p: Package { }" hará *matching* solo con aquellas clases cuya propiedad "namespace" tenga una referencia no nula a un "Package". Al mismo tiempo, la variable "p" apuntará al "Package". Sin embargo, puesto que en nuestro ejemplo "p" toma valor en la cláusula *when*, solo se hará *pattern matching* con aquellas clases cuyas propiedades "namespace" tengan una referencia al mismo paquete al que se refiere "p".

Se permite el anidamiento profundo de *"object template expressions"*, y el *matching* y la toma de valores de las variables se realiza recursivamente hasta que haya una serie de tuplas de valores que correspondan a variables de un dominio y su *"object template expression"*. Por ejemplo las tres variables: "c", "p" y "cn", crean una 3-tupla, y cada *match* válido dará como resultado una única tupla que represente la unión de un valor con una variable.

En una invocación a una relación dada, puede haber varios *matchings* válidos para una *"template expression"*, la manera de procesar esta multiplicidad depende de la dirección en la que ejecutemos la relación.

Por ejemplo, si se ejecuta la relación "ClassToTable" con "rdbms" como modelo destino, entonces por cada *matching* válido del dominio "uml", tendría que existir al menos un *matching* válido del dominio "rdbms" que satisfaga la cláusula *where*. Si para un *matching* válido del dominio "uml", no existe un *matching* válido del dominio "rdbms", entonces puesto que el dominio "rdbms" es *enforce*, se crean objetos y las propiedades se ponen en la *"template expression"* asociada con el dominio "rdbms". También, para cada *matching* válido del dominio "rdbms", tendría que existir al menos un *matching* válido del dominio "uml" que satisfaga la cláusula *where* (esto exige que el dominio "uml" esté marcado como *checkonly*); en otro caso los objetos se eliminarán del modelo "rdbms" por lo que no seguirá siendo un *matching* válido.

2.4. *Keys* y creación de objetos usando patrones

Como se mencionó anteriormente, un *"object template expression"* también sirve como plantilla para crear un objeto en un modelo destino. Cuando para un *matching* válido dado de un patrón de dominio origen, no existe un *matching* en el patrón del dominio destino, entonces se utiliza el *"object template expression"* del dominio destino como plantilla para crear objetos en el modelo destino. Por ejemplo, cuando se ejecuta "ClassToTable" con "rdbms" como modelo destino, el siguiente *"object template expression"* sirve como plantilla para crear objetos en el modelo "rdbms":

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

La plantilla asociada con "Table" especifica que un objeto de este tipo debe crearse con las propiedades "schema", "name", "column" y "primaryKey" inicializadas a los valores de la *"object template expression"*. Análogamente, las plantillas asociadas con "Column", "PrimaryKey", etc, especifican como deben crearse sus respectivos objetos.

Cuando se creen objetos, se debe asegurar que no se creen objetos duplicados cuando los objetos requeridos ya existen. Sin embargo, en ocasiones, simplemente se querrá actualizar objetos existentes, lo que deriva en un problema sobre cuándo actualizar objetos existentes o no permitir la creación de un objeto cuando éste ya existe con el fin de evitar objetos duplicados. MOF permite etiquetar como identificador una sola propiedad de una clase, sin embargo, para muchos metamodelos esto resulta insuficiente. El metamodelo Relations introduce el concepto de *Key*, que define un conjunto de propiedades de clase que identificarán de manera única el objeto instancia de una clase del modelo. Una clase puede tener múltiples *keys* al igual que ocurre en las bases de datos relacionales.

Por ejemplo, continuando con el ejemplo de la relación "ClassToTable", queremos especificar que en los modelos de tipo "simpleRDBMS", una tabla está identificada únicamente por dos propiedades: su nombre y el esquema al que pertenece.

```
key Table { schema, name };
```

Las *keys* se utilizan en el tiempo de la creación de objetos si un *"object template expression"* tiene las propiedades que corresponden a una *key* de la clase asociada, entonces las *keys* se utilizan para localizar un *matching* de objeto en el modelo; se crea un objeto solo cuando no existe un matching de objeto.

En el ejemplo, tomando el caso en el que se tiene una clase persistida llamada "foo" en el modelo "uml", y existe una tabla con un *matching* de nombre "foo" en un *matching* de esquema en el modelo "rdbms" pero la tabla no tiene *matching* de valores con las propiedades "column" y "primaryKey". En este caso, de acuerdo con el *matching* semántico de patrones, el modelo "rdbms" no tiene un *matching* válido para el patrón asociado a la "Table", por lo que es necesaria la creación de objetos que cumplan esa relación. Sin embargo, desde que la tabla existente hace *matching* con las propiedades especificadas por la *key*, no es necesario crear una nueva tabla, tan solo hay que actualizar las propiedades "column" y "primaryKey" de la tabla.

2.5. Restricciones sobre expresiones

Para garantizar ejecutabilidad existe un algoritmo que fuerza una relación en la dirección de un modelo destino dado; las expresiones que ocurren en la relación son requeridas para satisfacer las siguientes condiciones:

- Deber ser posible organizar las expresiones que ocurren en la cláusula *when*, el dominio origen y la cláusula *where*, en un orden secuencial que contenga solo los siguientes tipos de expresiones:
    - Una expresión de la forma

      <objeto>.<propiedad> = <variable>

      Donde la <variable> es una variable libre, y el <objeto> es tanto una variable asociada a un *"object template expression"* de un patrón de dominio opuesto, como una variable que toma su valor de una expresión precedente en el orden de la expresión
    - Una expresión de la forma

      <objeto>.<propiedad> = <expresión>

      Donde <objeto> es tanto una variable asociada a un *"object template expression"* de un patrón de dominio, como una variable que toma su valor de una expresión precedente. No hay ocurrencias de variables libres en <expresión>.
    - Ninguna otra expresión tiene ocurrencias de variables libres (todas las ocurrencias de variables deberían tomar valor de expresiones precedentes).
- Debería ser posible organizar expresiones que ocurran en el dominio destino, en un orden secuencial que contenga solamente los siguientes tipos de expresiones:
    - Una expresión de la forma

      <objeto>.<propiedad> = <expresión>

Donde <objeto> es tanto una variable asociada a un *"object template expression"* de un patrón de domino como una variable que toma su valor de una expresión precedente. No hay ocurrencias de variables libres en <expresión> (las ocurrencias de variables solo deberían aparecer en expresiones precedentes).

- Ninguna otra expresión tiene ocurrencias de variables libres

2.6. Propagación de cambios

En las relaciones, el efecto de la propagación de cambios desde un modelo origen a uno destino, es semánticamente equivalente a ejecutar una transformación en la dirección del modelo destino. La semántica de la creación y eliminación de objetos garantiza que solo se verán afectadas aquellas partes requeridas en el modelo destino. En primer lugar, la semántica de comprobar antes de transformar, asegura que los elementos del modelo destino que satisfacen la relación no son modificados. En segundo lugar, la selección de objetos basada en *keys*, asegura que los objetos existentes son actualizados debidamente. Por último, la semántica de borrado, asegura que un objeto es eliminado solo cuando no es requerido por ninguna regla.

Una implementación puede utilizar libremente algoritmos de propagación de cambios siempre que sea consistente con la semántica comentada.

2.7. Transformaciones *In-Place*

Una transformación puede considerarse "*In-Place*" cuando sus modelos candidatos origen y destino están asociados al mismo modelo en tiempo de ejecución. Este tipo de transformaciones tienen las siguientes características:

- Una relación es re-evaluada después de cada modificación inducida en la instancia de un patrón objetivo del modelo.
- La evaluación de una relación se detiene cuando todas las instancias de patrones satisfacen la relación.

2.8. Integración de operaciones *Black-Box* en relaciones

Una relación opcionalmente puede estar asociada a una implementación operacional para forzar un dominio. La operación *Black-box* se invoca cuando la relación se ejecuta en la dirección del dominio forzado y la relación se evalúa a falso mediante comprobaciones semánticas. La operación invocada es responsable de crear los cambios necesarios en el modelo para satisfacer la relación especificada. La signatura de la operación puede derivarse de la especificación del dominio de la relación; un parámetro de salida correspondiente al dominio forzado, y un parámetro de entrada correspondiente a cada uno de los otros dominios.

Las Relations implementadas usando *Operational Mappings* u operaciones *Black-Box*, están restringidas de la siguiente manera:

- Su dominio debe ser primitivo o contener una plantilla de objeto simple (sin subelementos).

- No deben definirse variables dentro de las cláusulas *when* y *where.*

2.9. Ejecución de una transformación en modo *checkonly*

Una transformación puede ser ejecutada en modo *checkonly*. En este modo, la transformación simplemente comprueba si las relaciones se cumplen en todas las direcciones. No se realiza ningún forzado en ninguna dirección, independientemente de cómo estén marcados los dominios, *checkonly* o *enforce*.

2.10. Sintaxis y semántica abstracta

2.10.1. Paquete QVTBase

Este paquete contiene un conjunto de conceptos básicos, muchos reutilizados de las especificaciones de EMOF y OCL, que estructuran transformaciones, sus reglas y sus modelos de entrada y salida. También introduce la noción de patrón, o en inglés *pattern*, como un conjunto de predicados sobre variables en expresiones OCL. Estas clases se extienden en paquetes de lenguajes específicos para proporcionar semánticas de lenguajes específicos.

2.10.1.1. Transformation

Una transformación o en inglés *transformation*, define como un conjunto de modelos pueden ser transformados en otro. Ésta contiene un conjunto de reglas, o en inglés *rules*, que especifican el comportamiento de su ejecución. Se ejecuta sobre un conjunto de modelos cuyos tipos están especificados por un conjunto tipos de modelos o metamodelos asociados con la transformación.

Sintácticamente, una transformación es una subclase de “Package” y de “Class”. Como “Package” proporciona un espacio de nombres para las reglas que contiene; como “Class” puede definir propiedades y operaciones-propiedades para especificar valores de configuración.

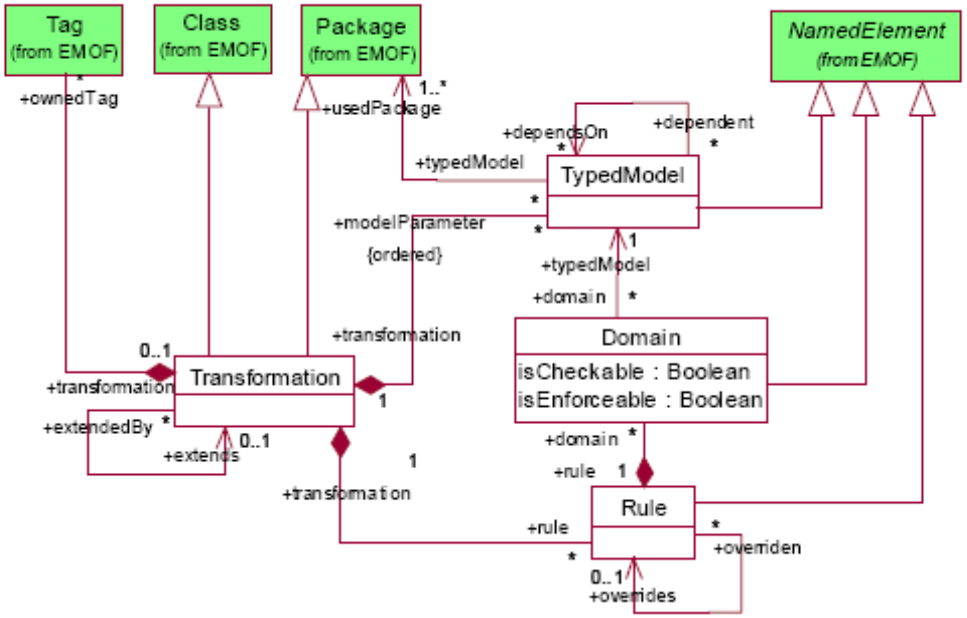


Figura 19. Paquete QVTBase - Transformaciones y reglas.

2.10.1.2. TypedModel

Un tipo de modelo o en inglés *TypedModel*, especifica un parámetro nominado y tipado de una transformación. En ejecución, un modelo que se pasa como parámetro a una transformación, solo puede contener aquellos elementos cuyos tipos están especificados en el conjunto de paquetes de modelos asociados con el tipo de modelo. Una transformación, siempre es ejecutada en una determinada dirección, seleccionando uno de los tipos de modelo como objetivo o destino de la transformación.

2.10.1.3. Domain

Un *domain* o dominio especifica un conjunto de elementos de modelo de un tipo de modelo que son de interés para una regla. *Domain* es una clase abstracta, cuyas subclases concretas son responsables de definir el mecanismo exacto de especificación, por el cual han de especificarse el conjunto de elementos de modelo de un dominio.

Un domino debe estar marcado como *checkonly* o *enforce*. Un dominio de tipo *checkonly* declara que su regla solo es requerida para comprobar, que los elementos de modelos especificados por el dominio existen en el modelo destino, y devolver errores cuando no lo son. Un dominio de tipo *enforce* declara que su regla debe asegurarse, y forzar cuando sea necesario, que los elementos de modelo especificados por el dominio existan en el modelo destino, cuando la transformación se ejecute en la dirección del tipo de modelo asociado con el dominio.

2.10.1.4. Rule

Una regla o en inglés *rule*, especifica como los elementos de modelos especificados por sus dominios son relacionados con otros, y como han de

computarse por elementos de modelos de otros dominios. *Rule* es una clase abstracta cuyas subclases concretas son responsables de especificar la semántica de cómo se relacionan los dominios y cómo son computados por otros.

Una regla puede sobrescribir otra regla de manera condicional. La regla que sobrescribe se ejecuta en lugar de la sobrescrita cuando se satisfacen las condiciones de sobrescritura.

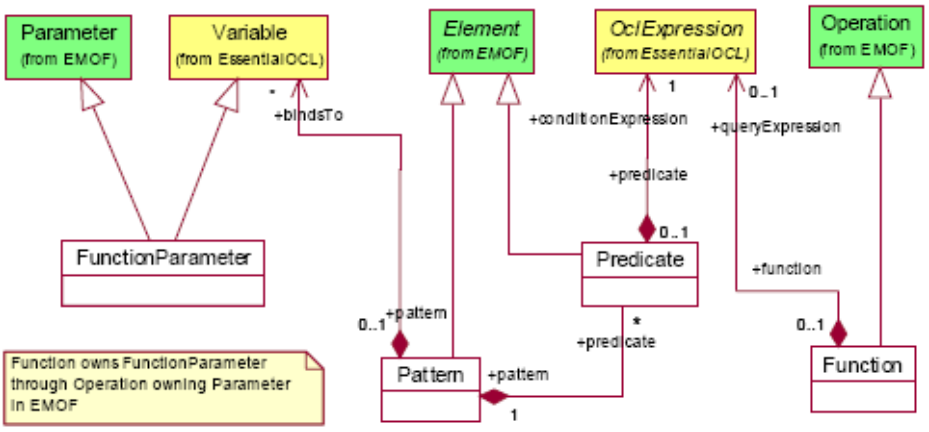


Figura 20. Paquete QVTBase - Patrones y funciones.

2.10.1.5. Function

Una *function o* función, es una operación libre de efectos laterales contenida en una transformación. Una función debe producir el mismo resultado siempre que sea invocada con los mismos argumentos. Una función puede especificarse mediante una expresión OCL, o mediante una implementación *black-box*.

2.10.1.6. FunctionParameter

Un *function parameter* o parámetro de función, especifica el parámetro de una función.

Sintácticamente, es una subclase de las clases “Parameter” y “Variable”. Por ser subclase de “Variable”, permite a las expresiones OCL que especifican una función acceder a los parámetros como variables nominadas.

2.10.1.7. Predicate

Un predicado o en inglés *predicate*, es una expresión “booleana” contenida en un patrón. Está especificado por una expresión OCL que contiene referencias a las variables del patrón que tiene el predicado.

2.10.1.8. Pattern

Un patrón o en inglés *pattern*, es un conjunto de variables y predicados, que cuando son evaluados en el contexto de un modelo, dan como resultado un conjunto de valores para las variables.

2.10.2. Paquete QVTTemplate

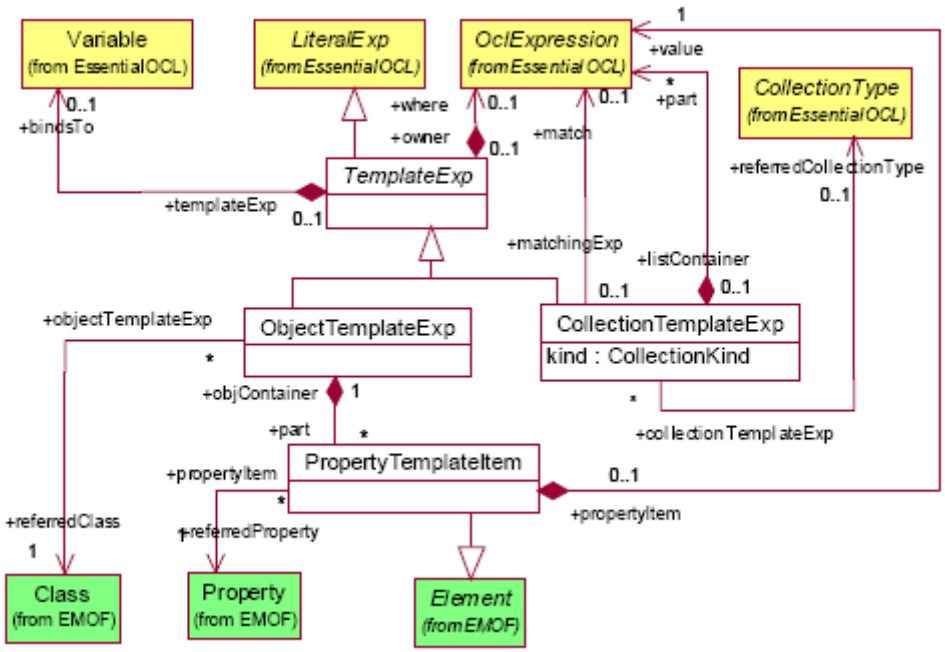


Figura 21. Paquete QVT Template.

2.10.2.1. TemplateExp

Una *template expression* o plantilla de expresión, especifica un patrón que puede combinarse con elementos de modelo en un modelo candidato de una transformación. El elemento de modelo combinado debe estar ligado a una variable y a su vez, esta variable debe ser usada en otras partes de la expresión. Solo se produce el *matching* con elementos de modelo cuando la expresión *where* asociada con la *template expression* se evalúa a *true*. Una *template expression* debe poder hacer *matching* tanto con un solo elemento de modelo, como con una colección de elementos de modelo.

2.10.2.2. ObjectTemplateExp

Una *object template expression*, especifica un patrón que debe poder hacer *matching* solo con un elemento de modelo, y tiene un tipo especificado por la clase referida. Se especifican mediante una colección de "*property template ítems*", cada uno de los cuales se corresponde con atributos diferentes de la clase referida.

2.10.2.3. CollectionTemplateExp

Una *collection template expression* especifica un patrón que hace *matching* con una colección de elementos. El tipo de la colección con la que hace *matching* una plantilla, viene dada por el tipo de la colección referida. Una *collection template expression* puede definirse de tres maneras diferentes: enumeración, comprensión y selección de miembro. La interpretación del modelo en los tres casos es la siguiente:

- Enumeración. El conjunto de expresiones "parte" especifica exactamente los elementos que comprenden la colección.
- Comprensión. La expresión obtenida del *matching*, que puede ser tanto una variable como una plantilla de objeto, que toma como valor un elemento de la colección. Si se usa una plantilla de objeto, entonces cada elemento de la colección debe hacer *matching* con el patrón especificado por una *object template expression*. Cada elemento en la colección debe satisfacer adicionalmente la expresión "parte".
- Selección de miembro. La expresión obtenida del *matching* solo puede ser una variable cuyo valor es un elemento de la colección. Si el tipo de la colección es una secuencia o un conjunto ordenado, la expresión toma como valor el primer elemento de la secuencia o del conjunto ordenado.

2.10.2.4. PropertyTemplateItem

Los *property template items* se utilizan para especificar restricciones en los valores de las ranuras del elemento modelo que empareja el contenedor de la *object template expression*. La restricción está en la ranura, que es una instancia de la propiedad referida y de la expresión que contiene la restricción, y viene dada por el valor de la expresión.

2.10.3. Paquete QVTRelation

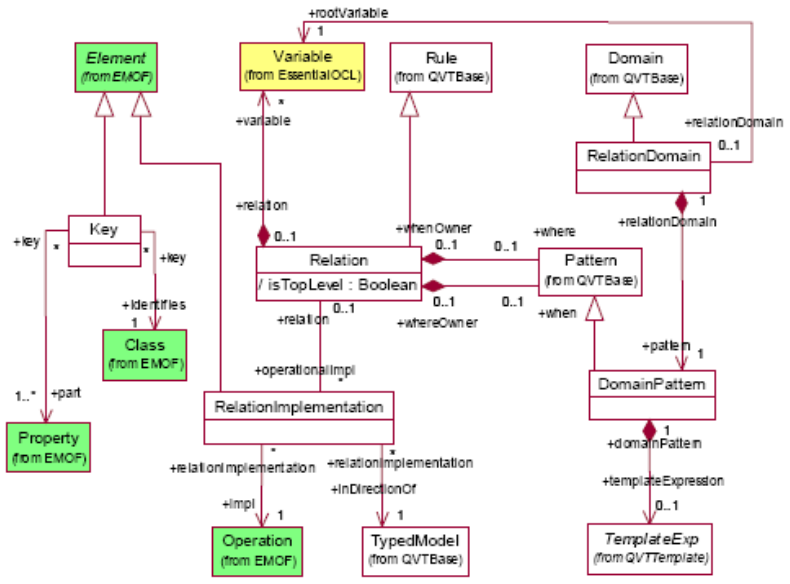


Figura 22. Paquete QVT Relation.

2.10.3.1. Relation

Una relación o en inglés *relation*, es la unidad básica de especificación de comportamiento de una transformación en el lenguaje Relations. Ésta es una subclase concreta de "Rule". Una *relation* especifica la relación que debe cumplirse entre los elementos modelo de un conjunto de modelos candidatos, que conforma a los *typed models* de la transformación que contiene la *relation*. Una relación está definida por dos o más *relation domains*, que especifican los elementos modelo que han de estar relacionados, una cláusula *when* que especifica las condiciones necesarias bajo las cuales la relación debe cumplirse, y una cláusula *where* que especifica la condición que debe satisfacerse por los elementos modelo que están siendo relacionados.

2.10.3.2. RelationDomain

La clase "RelationDomain" especifica los dominios de una relación. Es una subclase concreta de "Domain". Un dominio de relación o en inglés *relation domain*, tiene una variable tipada llamada *root variable*, que puede hacer *matching* en un modelo de un tipo de modelo dado. Una relación de dominio especifica un conjunto de elementos modelo de interés en términos de un patrón de dominio (*domain pattern*), que puede ser visto como un grafo de nodos de objetos, con sus propiedades y *links* de asociación, y con un distinguido nodo raíz que es asociado a la variable raíz de la relación de dominio.

2.10.3.3. DomainPattern

La clase "DomainPattern" es una subclase de la clase "Pattern". Un patrón de dominio o en inglés *domain pattern*, puede especificar un grafo arbitrariamente complejo en términos de una *template expression* basada en *object template expressions*, *collection template expressions* y *property template items*. Un *domain pattern* tiene una distinguida *template expression* raíz, que es requerida para ser asociada a la variable raíz de la relación de dominio que contiene el patrón de dominio. Un *object template expression* puede tener otras *template expressions* anidadas con una profundidad arbitraria.

2.10.3.4. Key

Una *key* define un conjunto de propiedades de una clase que identifica de manera única una instancia de clase en un modelo. Una clase puede tener múltiples *keys*.

2.10.3.5. RelationImplementantion

Una *RelationImplementation*, especifica de manera opcional una implementación operacional de *black-box* para forzar un dominio de una relación. La operación *black-box* se invoca cuando la relación se ejecuta en la dirección del tipo de modelo asociado con el dominio forzado, y la relación se evalúa a falso de acuerdo con la comprobación semántica. La operación invocada es responsable de hacer los cambios pertinentes en el modelo para satisfacer la relación especificada. La signatura de la operación puede derivarse de la especificación del dominio de la relación como un parámetro de salida, correspondiente al dominio forzado, y como un parámetro de entrada, correspondiente a cada uno de los otros dominios.

2.11. Gramática abstracta del lenguaje Relations

```
<topLevel> ::= ('import' <filename> ';' )* <transformation>*
<filename> ::= <identifier>
<transformation> ::= 'transformation' <identifier> '('
                        <modelDecl> (; <modelDecl>)* ')'
                        ['extends' <identifier> (',' <identifier>)* ]
                        '{'
                        <keyDecl>* ( <relation> | <query> )*
                        '}'
<modelDecl> ::= <modelId> ':' <metaModelId> (, <metaModelId>)*
<modelId> ::= <identifier>
<metaModelId> ::= <identifier>
<keyDecl> ::= 'key' <classId> '{' <propertyId> (, <propertyId>)* '}' ';'
<classId> ::= <identifier>
<propertyId> ::= <identifier>
<relation> ::= ['top'] 'relation' <identifier> ['overrides' <identifier>]
                '{'
                <varDeclaration>*
                (<domain> | <primitiveTypeDomain>)+
                <when>? <where>?
                '}'
<varDeclaration> ::= <identifier> (, <identifier>)* ':' <typeCS> ';'
<domain> ::= [<checkEnforceQualifier>]
                'domain' <modelId> [ <identifier> ] ':' <typeCS>
                '{' <propertyTemplate>* '}' [ '{' <oclExpressionCS> '}' ]
                ['implementedby' <OperationCallExpCS>]
                ';'
<propertyTemplate> ::= <identifier> '=' <oclExpressionCS>
<primitiveTypeDomain> ::= 'primitive' 'domain' <identifier> ':' <typeCS> ';'
<checkEnforceQualifier> ::= 'checkonly' | 'enforce'
<when> ::= 'when' '{' <oclExpressionCS> '}'
<where> ::= 'where' '{' <oclExpressionCS> '}'
<query> ::= 'query' <pathNameCS> '(' [<paramDeclaration> (','
              <paramDeclaration>)*] ')' ':'
              [<paramDeclaration> (','
              <paramDeclaration>)*]
              (
              ';' | '{' <oclExpressionCS> '}'
```

VI. INTEGRACIÓN Y SOPORTE DEL LENGUAJE QVT-RELATIONS EN MOMENT

1. PROCESO DE EJECUCIÓN QVT-RELATIONS EN MOMENT

En el proceso de transformación, el usuario interactúa con la plataforma MOMENT especificando un programa QVT utilizando el lenguaje Relations. MOMENT, proporciona un editor textual con coloreado de sintaxis que facilita la labor de especificar transformaciones.

Una vez especificado el programa QVT-Relations se inicia un proceso automático y totalmente oculto al usuario, que culmina con la ejecución en Maude de la transformación definida en el programa y la devolución al usuario de los resultados obtenidos. Este proceso se divide en tres partes fundamentales: análisis, proyección a Maude, y ejecución y proyección de los resultados a EMF (Figura 23).

Para llevar a cabo estas tres fases, son necesarios un conjunto de componentes funcionales que forman la arquitectura de MOMENT QVT:

- QVT *Parser*: encargado de analizar un programa QVT Relations de entrada y generar su modelo correspondiente.
- MOMENT *Registry*: repositorio de artefactos generados y/o utilizados por MOMENT.
- MOMENT Relations: encargado de generar y proyectar código Maude a partir de la especificación de una transformación.
- OCL *Parser*: encargado de analizar expresiones OCL y generar su código Maude correspondiente.

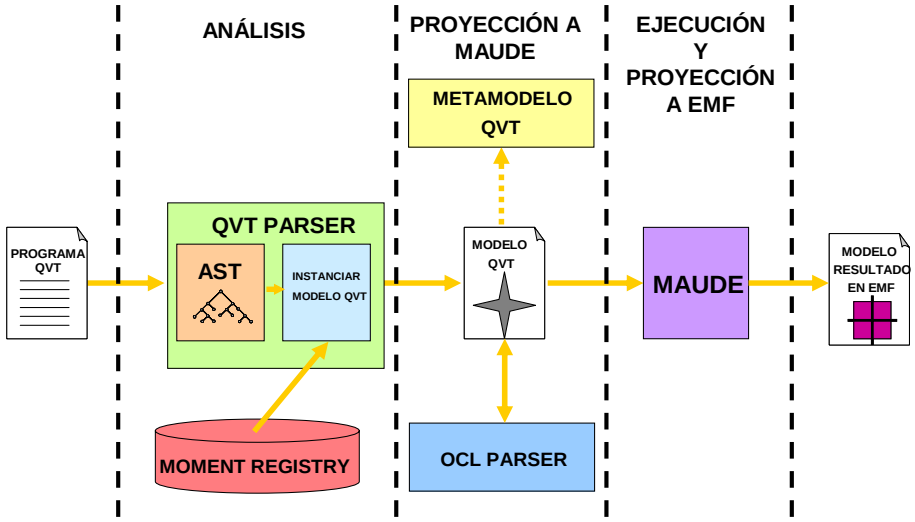


Figura 23. Proceso de ejecución de un programa QVT-Relations en MOMENT.

Para una mejor compresión del funcionamiento de las diferentes fases que componen el proceso de ejecución de programas QVT-Relations en la plataforma MOMENT, se utilizará la transformación QVT-Relations definida en el apéndice de la especificación del estándar MOF QVT [MOFQVT] y que puede encontrarse en el Anexo III, que especifica la transformación de un diagrama de clases en UML a su correspondiente modelo relacional. Concretamente, se mostrará la regla de transformación “ClassToTable” en cada una de las etapas del proceso. Esta regla establece la correspondencia entre una clase perteneciente al metamodelo origen (UML) y una tabla perteneciente al metamodelo destino (relacional), de forma que a

partir del término clase perteneciente a uno de los dominios de la regla, se obtienen los términos tabla, columna y clave primaria, que pertenecen al dominio destino de la regla.

1.1. Análisis

Esta etapa comienza con un análisis léxico y sintáctico del programa QVT-Relations. Recorriendo el árbol de sintaxis abstracta (AST) generado por esta etapa, se crea un modelo QVT-Relations como instancia del metamodelo de QVT-Relations, el cual se encuentra integrado en la plataforma MOMENT. Durante este recorrido es necesario consultar los metamodelos respectivos a los modelos participantes en la transformación, y obtener de este modo los tipos y las propiedades de sus elementos. Para realizar estas consultas, es necesario que los metamodelos estén registrados en el registro de modelos de EMF. Es por esto, por lo que esta fase interactúa con el MOMENT *Registry*, evitando al usuario la tarea de registrar los metamodelos requeridos por una transformación cada vez que se inicie la herramienta o cada vez que se ejecute la transformación. MOMENT *Registry* será explicado detalladamente en la sección VII de este documento.

Teniendo en cuenta la regla de transformación “ClassToTable” mencionada anteriormente, puede observarse en la parte derecha de la Figura 24 el código QVT-Relations asociado a la regla. Este código constituye el programa de entrada de la fase de análisis. Como resultado de esta fase, se obtiene el modelo QVT-Relations asociado al código del programa, tal y como puede verse en la parte izquierda de la Figura 24.

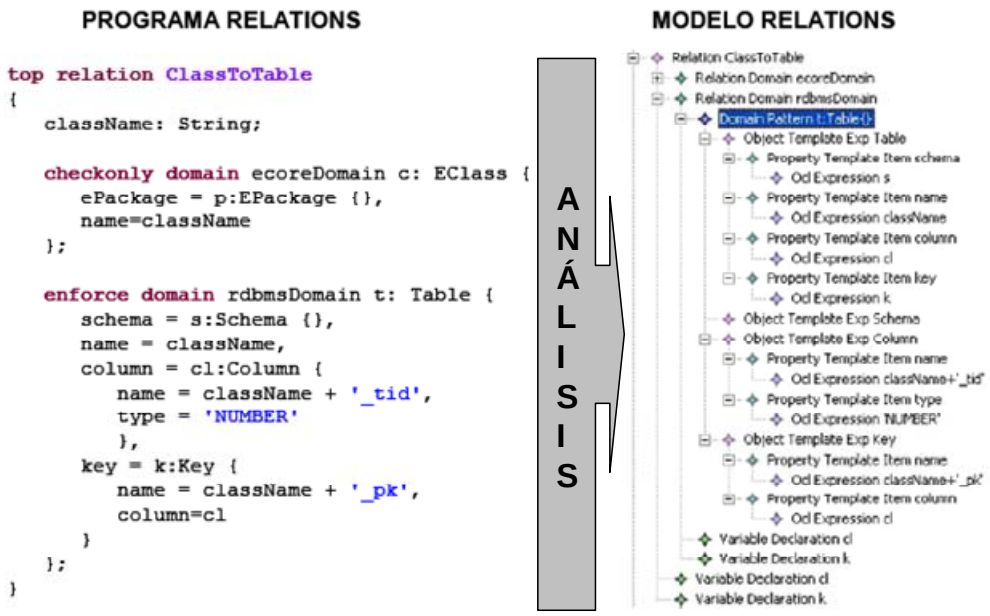


Figura 24. Fase de análisis para la regla “ClassToTable”.

1.2. Proyección al espacio tecnológico de Maude

Los metamodelos origen y destino son compilados en teorías basadas en lógica ecuacional [BoI05]. Los conceptos del metamodelo se proyectan como *sorts* y las propiedades como constructores y operadores de consulta. Así, se crea una especificación algebraica en el espacio tecnológico de Maude para cada metamodelo participante en la transformación. Además, se toman todos los modelos origen (en la capa M1 de MOF) y se obtiene para cada uno de ellos el término algebraico que conforma la especificación algebraica definida para su respectivo metamodelo [BoCR06].

La nueva especificación algebraica obtenida extiende a las especificaciones ya existentes de los metamodelos participantes en la transformación, mediante un conjunto de reglas de reescritura que describen las guías para realizar la transformación, constituyendo el contexto donde se definen las relaciones semánticas entre los metamodelos origen y destino.

Seguidamente, el modelo QVT Relations resultado de la etapa de análisis se proyecta en otra especificación algebraica. Una parte muy importante de esta proyección es la traducción de las expresiones OCL utilizadas en la especificación de la transformación, y que se realiza a través del componente OCL *Parser* [BoO06].

Siguiendo con el ejemplo de la regla “ClassToTable”, esta fase recibe como entrada el modelo QVT-Relations mostrado en la Figura 23 y que ha sido obtenido de la fase de análisis. Este modelo es proyectado a Maude dando como resultado el código Maude presente en el Anexo IV. Dada la complejidad del código y para dar una ligera noción del significado del mismo, en la Figura 25, se muestran unas líneas de código que lo resumen. En éstas se aprecia la ecuación “TransformElements” que lleva a cabo su ejecución a través de los mecanismos de *pattern-matching* y de recursión. La generación de los términos resultantes “Table”, “Column” y “Key” se realiza a través del operador “New”, el cálculo de sus respectivos identificadores se consigue con el operador “AddOID” y la asignación de valores y contenido a cada una de las propiedades de estos términos se obtiene por medio del operador asignación “<--” y las navegaciones del modelo origen partiendo de su respectivo dominio en la regla, como es el caso de “ecoreEClass0 :: name”.

```
*** regla 1: ClassToTable
eq TransformElements (ClassToTable, ? Set{ecoreEClass0) ? ecoreModel0, TargetModel, Tuple2) =
  Set {
    AddOID(
      ( (New ("Table", MM ((empty-set).Set{rdbms}))).rdbmsNode
         :: schema <-- ((p1
            ((ModelGenRule (PackageToSchema,
                ? ((Set{((ecoreEClass0 :: ePackage (ecoreModel0)))} -> flatten) ? ecoreModel0,
                TargetModel, Tuple2)
            ))))
         :: name <-- (((ecoreEClass0 :: name)))
         :: column <-- (Set { AddOID (
                           (New(“Column”, MM((empty-set).Set{rdbms}))).rdbmsNode
                           :: name <-- ((((ecoreEClass0 :: name) + "_tid" ))))
                           :: type <-- ("NUMBER"))
                       })
         :: key <-- (Set { AddOID (
                  (New("Key", MM((empty-set).Set{rdbms}))).rdbmsNode
                  :: name <-- (((((ecoreEClass0 :: name) + "_pk"))))
                  :: column <-- (Set {AddOID (
                                 (New("Column", MM ((empty-set).Set{rdbms}))).rdbmsNode
                                 :: name <-- (((((ecoreEClass0 :: name) + "_tid"))))
                                 :: type <-- ("NUMBER") )))
         )})
    )}
```

Figura 25. Proyección a Maude modelo QVT-Relations de la regla “ClassToTable”.

1.3. Ejecución de la transformación y proyección al espacio tecnológico EMF

Las reglas de reescritura especificadas en la extensión de la especificación algebraica se aplican automáticamente reescribiendo el término origen en un término del álgebra destino al tiempo que se crea un término con información de trazabilidad. Estos términos son finalmente proyectados al espacio tecnológico de EMF de manera que el primo de ellos corresponde al modelo destino (resultado de la transformación) y el segundo al modelo de trazabilidad.

2. DIFERENCIAS CON EL ESTÁNDAR

Para poder dar soporte al lenguaje QVT-Relations en MOMENT de manera satisfactoria ha sido necesario realizar una serie de modificaciones respecto a la especificación estándar de QVT y al lenguaje Relations de QVT desarrollados en la sección V de este documento.

Como se comentó a lo largo del apartado 1 de esta sección, en MOMENT se requiere del metamodelo del lenguaje Relations para poder ejecutar programas QVT-Relations. En la definición de este metamodelo, se han integrado los paquetes “QVT Base”, “QVT Relation” y “QVT Template” bajo un único paquete denominado “QVT Package”. Además, ha sido necesario modificar el nombre de algunas asociaciones en el metamodelo con el fin de evitar ambigüedades tanto en la definición del metamodelo como en la posterior implementación del mismo. También se han eliminado las dependencias con los paquetes referentes a OCL, puesto que las expresiones OCL son evaluadas en tiempo de ejecución por una herramienta proporcionada por MOMENT destinada a tal fin. El metamodelo resultante puede consultarse en el Anexo V.

Con el fin de ejecutar programas QVT-Relations en MOMENT ha sido necesario modificar el metamodelo definido para QVT-Relations así como la sintaxis definida en la gramática del apartado 2.11 de la sección V. Esta modificación se debe, a que por estrategias de ejecución, no es posible diferenciar el modelo destino de una transformación. Para paliar este problema se ha modificado el metamodelo QVT-Relations introduciendo una nueva clase llamada “Transformation model”, destinada a contener información (nombre y tipo) de cada uno de los modelos candidatos que participan en la transformación. Esta clase es una clase agregada de la clase “Transformation” por partida doble, por un lado, una clase “Transformation” puede contener una o varias clases “Transformation model” (modelos origen), y por otro lado, puede contener cero o una clase “Transformation model” (modelo destino). La cardinalidad mínima es cero puesto que si el programa QVT-Relations se ejecuta en modo *checkonly* no existe un modelo destino. Respecto a la modificación en la gramática, se ha introducido la palabra reservada *target* en la declaración de modelos candidatos de la transformación, que indica cuál es el modelo destino. De esta manera, se ha redefinido en la gramática la regla sintáctica *transformation* y se ha creado una nueva regla llamada *modelDeclTarget*. Estos cambios pueden verse a continuación:

```
<transformation> ::= 'transformation' <identifier> '('
                     <modelDecl> (; <modelDecl>)* [ ; <modelDeclTarget>] ')'
                     ['extends' <identifier> (',' <identifier>)* ]
                     '{'
```

```
                        <keyDecl>* ( <relation> | <query> )*
                        '}'
<modelDeclTarget> ::= 'target' <modelId> ':' <metaModelId>
```

El soporte para QVT-Relations de MOMENT está integrado dentro de EMF. En EMF los modelos se identifican mediante un identificador llamado URI. En la declaración de los modelos candidatos de la transformación es necesario especificar el tipo del modelo, o lo que es lo mismo, su metamodelo. Así, en MOMENT cuando se especifica el tipo de un modelo candidato es necesario indicar la URI del metamodelo.

3. SOPORTE ACTUAL

Actualmente, el soporte para el lenguaje QVT-Relations de MOMENT permite:

- Ejecutar transformaciones en una dirección
- Ejecutar transformaciones incrementales
- Transformaciones *In-Place*
- Exportar, importar y/o ejecutar transformaciones serializadas en XMI

Actualmente, respecto al estándar QVT, en MOMENT no es posible:

- Ejecutar transformaciones bidireccionales
- Ejecutar transformaciones en modo *"checkonly"*
- Utilizar operaciones *Black-Box* (característica opcional del estándar)

4. HERRAMIENTAS DESARROLLADAS

Para dar soporte a transformaciones y relaciones de equivalencia especificadas usando el lenguaje Relations de QVT, se ha desarrollado un *plug-in* en Eclipse llamado "es.upv.dsic.issi.moment.qvt.parser". Este *plug-in* realiza la fase de análisis del proceso de transformación en MOMENT. A lo largo de este documento nos referiremos a este *plug-in* como *parser* de QVT o QVT-*Parser*.

4.1. Descripción

En este apartado se presenta una descripción a alto nivel del *plug-in*. Se presentarán las necesidades a las que el sistema debe dar soporte, las funciones que debe realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo.

4.1.1. Funciones del *plug-in*

Este *plug-in* deberá proporcionar la siguiente funcionalidad:

- Analizar léxica y sintácticamente un programa textual QVT-Relations e informar debidamente al usuario de los posibles errores contenidos en el programa.
- Construir un árbol de sintaxis abstracta (AST) como resultado del análisis sintáctico del programa textual QVT-Relations de entrada.
- Crear de forma automática un modelo QVT-Relations como instancia del metamodelo QVT-Relations a partir del AST generado.
- Serializar como fichero XMI el modelo QVT-Relations creado.

4.1.2. Dependencias

Para que el *plug-in* funcione correctamente se deberá disponer de una distribución de Eclipse con el *framework* EMF instalado.

Además, de manera interna este *plug-in* depende de la librería de “antlr” y de los siguientes *plug-ins* de MOMENT:

- es.upv.dsic.issi.moment.qvt.relations

  Este *plug-in* contiene todas las clases Java que componen el metamodelo QVT-Relations. La dependencia de este *plug-in* es necesaria para poder crear modelos QVT-Relations.

- es.upv.dsic.issi.moment.ui.console

  Este *plug-in* proporciona el soporte para la consola de MOMENT, a través de la cual se mostrarán al usuario aquellos mensajes de error, depuración o información.

- es.upv.dsic.issi.moment.registry

  Este *plug-in* permite acceder al repositorio “MOMENT *Registry*” para cargar y consultar aquellos metamodelos que son utilizados para la declaración de tipos de modelos en los modelos candidatos de una transformación.

  MOMENT *Registry* se describirá detalladamente en la sección VII de este documento.

4.2. Diseño

El *parser* de QVT ha sido diseñado como un caso especial de compilador/traductor siguiendo la analogía presentada en el apartado 4.2 de la sección III. Se trata entonces de "compilar" el programa QVT-Relations que se encontrará codificado en un fichero de texto con extensión “qvtext, y “traducirlo” en una representación más adecuada en forma de modelo QVT-Relations. El lenguaje con el que ha de trabajar el compilador es el lenguaje Relations de QVT.

4.3. Implementación

En la implementación del *parser* de QVT se ha implementado un analizador léxico, un sintáctico, y un semántico para hacer las veces de traductor. La construcción de cada uno de estos analizadores ha sido implementada en una gramática de ANTLR y todos ellos han sido generados automáticamente.

4.3.1. Analizador léxico

La gramática definida para el analizador léxico se ha obtenido a partir de la sintaxis abstracta del lenguaje Relations mostrada en el apartado 2.11 de la sección V y puede consultarse en el Anexo VI.

En esta gramática se han definido todas las palabras reservadas y símbolos del lenguaje QVT-Relations, y aquellos elementos que no deben pasarse al analizador sintáctico como son los comentarios, espacios en blanco y caracteres de retorno de carro.

Respecto a los comentarios se ha seguido la sintaxis del lenguaje C. Así, los comentarios de línea pueden definirse como:

- // comentario
- /* comentario */

Los comentarios multilínea vendrán delimitados por los caracteres /*, que delimitan el comienzo, y los caracteres */, que marcan el final del comentario multilínea.

Para evitar ambigüedades en la gramática ANTLR que especifica el analizador léxico, se ha variado el valor de “k”. Este valor, indica a ANTLR el número de símbolos de anticipación antes de decidir la elección de una regla.

4.3.2. Analizador sintáctico

Para construir el analizador sintáctico, se ha derivado una implementación de una sintaxis concreta a partir de la sintaxis abstracta del lenguaje Relations, mostrada en el apartado 2.11 de la sección V, y añadiendo las diferencias respecto al estándar indicadas en el apartado 2 de esta sección. Las diferencias son únicamente de implementación: una sintaxis concreta refina la abstracta añadiéndole construcciones para solucionar problemas, debido principalmente a ambigüedades que surgen en el proceso de implementación.

Esta gramática puede consultarse en el Anexo VII y las principales modificaciones realizadas se deben a:

- Expresiones OCL. Estas expresiones se encuentran incrustadas en la especificación de programas QVT-Relations pero no son procesadas por el *parser* de QVT. Para ignorarlas se definieron sumideros en aquellas partes de las reglas donde aparecen estas expresiones. Con esto, surgió el problema de cómo salir de los sumideros y de no ignorar *tokens* válidos.

Aún así, para expresiones OCL de la forma *"if-then-else"* fue necesario definir reglas de pre procesamiento para evitar ambigüedades y errores en el proceso de análisis.

- Anidamiento de *Object Template Expressions* en *Domain patterns*. Los *Domain patterns* permiten anidamientos de profundidad n de *Object Template Expressions*. Para soportar esta característica se han añadido nuevas reglas sintácticas que detectan los anidamientos y los procesan débidamente
- Partes derechas de cláusulas *When* y *Where*. Como partes derechas de estas cláusulas pueden haber tanto expresiones OCL como llamadas a *relations*, por tanto se ha añadido una nueva regla sintáctica que detecta el tipo de parte derecha de la cláusula y la procesa debidamente.

Al igual que en el analizador léxico, también se ha modificado el valor de k para evitar ambigüedades. Además, se han utilizado construcciones ANTLR que ayudan a solucionar ambigüedades. Estas construcciones tienen la siguiente estructura:

$$(token_1\ token_2\ \ldots\ token_n) \Rightarrow regla \mid reglas$$

Su significado es el siguiente: si se detectan los *tokens* indicados por ($token_1$ $token_2$ … $token_n$), automáticamente ANTLR evalúa la regla o reglas especificadas en la parte derecha del símbolo "=>".

En la especificación de esta gramática también se han etiquetado los diferentes nodos que formarán el árbol de sintaxis abstracta, de manera que puedan ser reconocidos y procesados adecuadamente por el traductor.

4.3.3. Traductor

La construcción del traductor del programa QVT-Relations a su modelo QVT-Relations correspondiente, se ha realizado en lo que correspondería al analizador semántico. En esta fase se recorre todo el AST de manera ordenada mediante un mecanismo de *pattern matching* que proporciona ANTLR para los nodos del árbol. Analizando los elementos del árbol es posible instanciar los pertinentes objetos elementos del modelo QVT-Relations así como establecer las relaciones y dependencias entre estos objetos. Este traductor es la parte más compleja del *parser* de QVT y es donde se concentra la mayor parte de la lógica del *parser*.

4.4. Archivos resultantes

Como resultado de la implementación se han creado dos ficheros QVTparser.g y QVTtreewalker.g:

- QVTparser.g

  Este fichero contiene dos gramáticas ANTLR que implementan el analizador léxico y el sintáctico respectivamente.

- QVTtreewalker.g

  Este fichero contiene una gramática ANTLR que recorre el AST de manera apropiada para poder crear el modelo QVT-Relations correspondiente.

Ambos ficheros se encuentran ubicados en la carpeta “grammar” del *plug-in* “es.upv.dsic.issi.moment.qvt.parser”.

La compilación ANTLR de ambos ficheros produce los ficheros Java que implementan los analizadores especificados por las gramáticas. Estos ficheros se encuentran en la ruta del *plug-in* “/src/es.upv.dsic.issi.moment.qvt.parser.generated”.

5. TRABAJOS RELACIONADOS

5.1. XSLT

XSLT es un lenguaje estándar de la *World Wide Web Consortium* (W3C) y cuenta con un extenso soporte tecnológico [GeL02]. A pesar de satisfacer la gran mayoría de los requisitos necesarios por un lenguaje de transformación usado en *Model Driven Engineering* (MDE), resulta bastante complicado escribir transformaciones XSLT porque el usuario ha de tener en cuenta los esquemas XSD de entrada [BoCR04].

Como principales características, podemos decir que XSLT tiene una naturaleza híbrida pues permite tanto construcciones imperativas como declarativas. En su parte declarativa, la aplicación del *pattern-matching* se hace de forma ordenada y recursiva, por lo que XSLT es determinista. Además, desde la perspectiva de metamodelado, efectúa la transformación de grafos, que son árboles, como representación de modelos, y utiliza XPath como lenguaje de consulta.

Algunas de las restricciones existentes son: no tener disponibles transformaciones unidireccionales, ausencia de soporte de trazabilidad implícita, carencia de mecanismos automáticos de composición y no tener definida su sintaxis como instancia de MOF. Asimismo, XSLT no tiene una equivalencia directa con QVT y es difícil establecer una correspondencia entre ambos.

5.2. ATL

ATL forma parte del *framework* de gestión de modelos AMMA [ATL] que se encuentra integrado en Eclipse y EMF. Utiliza un lenguaje propietario llamado ATLAS para definir transformaciones que se ejecuta sobre JAVA y que proporciona un entorno de depuración. La naturaleza de ATLAS es declarativa, aunque lleva mucho tiempo utilizando también construcciones imperativas, y las transformaciones son expresadas como reglas de transformación ya que ATL cumple el estándar MOF.

ATL posee un algoritmo de ejecución preciso y determinista, sin embargo no se apoya sobre ningún método formal. No proporciona mecanismos para la validación de las transformaciones. Para llevar a cabo transformaciones compuestas se necesita ejecutar cada una de las transformaciones participantes una a una.

Aunque la sintaxis de ATL es muy similar a la de QVT, ésta no es interoperable con QVT. No obstante, es posible definir transformaciones entre ambos espacios tecnológicos tal como se explica en [JoK06].

5.3. VIATRA

Viatra actualmente forma parte del framework VIATRA2 [VIATRA], que ha sido escrito en Java y que se encuentra integrada en Eclipse. Viatra provee un lenguaje textual para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente.

La naturaleza del lenguaje es declarativa y está basada en técnicas de descripción de patrones, sin embargo es posible utilizar secciones de código imperativo. Se apoya en métodos formales como la transformación de grafos (GT) y la máquina de estados abstractos (ASM) para ser capaz de manipular modelos y realizar tareas de verificación, validación, seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño.

Como puntos débiles podemos resaltar que Viatra no se basa en los estándares MOF y QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el *framework*.

5.4. EPSILON

Epsilon es una plataforma desarrollada como un conjunto de *plug-ins* (editores, asistentes, pantallas de configuración, etc) sobre Eclipse. Presenta el lenguaje metamodelo independiente *Epsilon Object Language* que se basa en OCL [KoP06]. Puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender con nuevos lenguajes específicos de dominio. Tres son los lenguajes definidos en la actualidad: *Epsilon Comparison Language* (ECL), *Epsilon Merging Language* (EML), *Epsilon Transformation Language* (ETL), para comparación, composición y transformación de modelos respectivamente. Se da soporte completo al estándar MOF mediante modelos EMF y documentos XML a través de JDOM.

La finalidad perseguida con la extensión del lenguaje OCL es: dar soporte al acceso de múltiple modelos, ofrecer constructores de programación convencional adicionales (agrupación y secuencia de sentencias), permitir modificación de modelos, proveer depuración e informe de errores, así como conseguir una mayor uniformidad en la invocación.

Soporta mecanismos de herencia, trazabilidad e introduce mecanismos de comprobación automática del resultado en composición y transformación de modelos [KoPP06].

VII. MOMENT *REGISTRY* Y ARQUITECTURA DE MOMENT

1. MOTIVACIÓN

Tras la integración en MOMENT del DSL para definición de operadores complejos y del soporte para el lenguaje de transformación de modelos QVT-Relations, aparecen cuatro tipos de artefactos software: metamodelos, transformaciones, relaciones de equivalencia y operadores. En MOMENT, estos artefactos son modelos y según su naturaleza pueden ser generados o utilizados por la herramienta. Por tanto, conforme aumente la interacción de un usuario con la herramienta, el número de estos modelos crecerá progresivamente complicando cada vez más al usuario trabajar en MOMENT.

Una posible solución a este problema es que el propio usuario mantenga en el *workspace* de Eclipse uno o varios proyectos para organizar todos los artefactos derivados de la utilización de MOMENT. Sin embargo, esta solución no es del todo aceptable, pues determinadas funcionalidades de MOMENT, como por ejemplo las transformaciones, requieren de artefactos para su funcionamiento. De esta manera, y siguiendo con el ejemplo, cada vez que un usuario quisiera analizar un programa QVT-Relations, debería indicarle al sistema cuáles son y dónde se encuentran los metamodelos de los modelos candidatos de la transformación. Además, aparece un problema tecnológico derivado de la integración de MOMENT en EMF, y es que el usuario debería encargarse manualmente de registrar en EMF todos aquellos modelos con los que desee trabajar. Obviamente, esto supone un obstáculo en la automatización de la funcionalidad y una pérdida de transparencia en el funcionamiento de la herramienta.

Con el fin de evitar estos problemas se integra en MOMENT un repositorio, denominado MOMENT *Registry*, con la finalidad de permitir al usuario tener almacenados y clasificados los diferentes tipos de artefactos con los que va a trabajar. Así, este repositorio permite la automatización de aquellas funcionalidades que utilizan o generan metamodelos, transformaciones, operadores o relaciones de equivalencia, haciéndolas más transparentes y cómodas al usuario.

También, es importante destacar que la integración del MOMENT *Registry* en MOMENT acarrea una “reorientación” de su arquitectura, pues varios componentes o *plug-ins*, por su funcionamiento, necesitan interactuar directamente con el MOMENT *Registry*.

2. MOMENT *REGISTRY*

2.1. Análisis y requisitos

De manera global, el MOMENT *Registry* puede considerarse como un repositorio que permite el almacenamiento y la organización de los diferentes artefactos generados y/o utilizados por MOMENT. En MOMENT, estos artefactos son modelos y pueden clasificarse en cuatro tipos: metamodelos, transformaciones, operadores y relaciones de equivalencia.

La generación de estos artefactos por parte de MOMENT se produce como consecuencia del propio funcionamiento de la herramienta, debido a que determinadas utilidades de MOMENT, como el *QVT Parser* y el *DSL Parser*, producen modelos

como resultado de su ejecución. A su vez, tanto el QVT *Parser* y el DSL *Parser* hacen uso de artefactos, metamodelos y operadores respectivamente, para poder funcionar.

Otra de las funciones que ha de cumplir el MOMENT *Registry* es la de poder suministrar en todo momento cualquiera de los artefactos que tiene registrados, con independencia de cuál sea la ubicación de éste. Es decir, si en el MOMENT *Registry* se ha registrado un artefacto X que se ubicaba en la ruta de directorios A y posteriormente este artefacto ha sido movido a una ruta A', el MOMENT *Registry* podrá seguir suministrando este artefacto cuando le sea pedido. Hay que destacar, que la forma en la que un componente de MOMENT pide un artefacto al repositorio varía según el tipo del mismo.

2.1.1. Metamodelos

Por metamodelos, entendemos aquellos "tipos" de modelos a partir de los cuales podemos especificar modelos instancia. En el MOMENT *Registry* se registrarán todos aquellos metamodelos que puedan ser utilizados en la especificación de los modelos candidatos de una transformación o de una relación de equivalencia. De esta manera, cuando se analice un programa QVT-Relations, será el propio *parser*, de manera automática, el que le pida al MOMENT *Registry* que cargue y le proporcione aquellos metamodelos utilizados en la especificación de modelos candidatos.

Sin la presencia del MOMENT *Registry*, el usuario tendría que encargarse de registrar manualmente en el registro de EMF los metamodelos utilizados en la especificación de un programa QVT-Relations. Por tanto, la justificación de registrar metamodelos en el repositorio se traduce en un proceso de análisis automático y transparente de las especificaciones textuales de programas QVT-Relations. Además, en un entorno de desarrollo, los metamodelos pueden sufrir cambios en su especificación, por lo que el MOMENT *Registry* debería incluir un mecanismo de "recarga", que permitiera actualizar el repositorio con la nueva versión del metamodelo, sin necesidad de eliminarlo y volverlo a añadir.

Todo componente de MOMENT puede pedir al MOMENT *Registry* un metamodelo determinado indicando su URI. Las peticiones se realizan por URI, porque el principal componente de MOMENT que solicita metamodelos al repositorio es el QVT *Parser*, que como se indicó en el apartado 2 de la sección VI, en toda especificación textual de un programa QVT-Relations, los metamodelos de los modelos candidatos se referencian por URI.

2.1.2. Transformaciones y relaciones de equivalencia

En MOMENT, las transformaciones y relaciones de equivalencia son modelos QVT-Relations obtenidos tras analizar especificaciones textuales de programas QVT-Relations. La diferencia entre una transformación y una relación de equivalencia radica en que las relaciones de equivalencia no tienen *relations* de tipo *enforce* y por tanto solo realizan comprobaciones.

La idea de permitir el registro de transformaciones y relaciones de equivalencia en el MOMENT *Registry* se fundamenta en la posibilidad de desarrollar interfaces gráficas que permitan la ejecución de transformaciones y relaciones de equivalencia de manera sencilla e intuitiva.

La forma de pedir transformaciones o relaciones de equivalencia al MOMENT *Registry* se realiza indicando el nombre de la transformación o de la relación de equivalencia. Así, en futuras interfaces gráficas de ejecución de transformaciones y relaciones de equivalencia, se mostrará una lista con todas aquellas que han sido registradas en el repositorio con anterioridad, pudiéndose pedir al MOMENT *Registry* aquella que se desee ejecutar.

2.1.3. Operadores

En MOMENT los operadores están representados como modelos y pueden ser de dos tipos: simples o complejos. La diferencia entre un operador simple y uno complejo, es que en la definición del complejo se incluyen invocaciones a otros operadores existentes. Por tanto, en la definición de operadores complejos es necesario "importar" aquellos operadores que se utilizan en las invocaciones a operadores. En el lenguaje de definición de operadores complejos hay dos formas de importar operadores:

- **Por ruta del operador**. Cuando la importación del operador se realiza por ruta, en el análisis de la definición textual de un operador complejo el *parser* obtiene el operador a través de su ruta. Esto significa, que una vez obtenemos el modelo del operador complejo, si cambiáramos la ruta de algún operador importado, la ejecución del operador complejo fallaría. Para evitar este problema, en la definición textual del operador complejo, habría que modificar la ruta del operador importado afectado por el cambio de ubicación y volver a analizarlo.
- **Por nombre del operador**. Cuando la importación del operador se realiza por nombre, en el proceso de análisis de la definición textual del operador complejo, el *parser* le pide al MOMENT *Registry* el operador indicado. En este caso, el repositorio es capaz en todo momento de suministrar aquellos operadores que le sean pedidos, por lo que no habrá problemas siempre y cuando los operadores importados por un operador complejo estén registrados en el MOMENT *Registry*. Esta manera de importar operadores resulta muy útil para el usuario, ya que únicamente ha de preocuparse de registrar una única vez cada operador en el repositorio.

Las peticiones de operadores al MOMENT *Registry* se realizan por nombre del operador puesto una de las opciones de importar operadores en el DSL *Parser* es mediante el nombre del operador.

2.1.4. Interacción con el MOMENT *Registry*

MOMENT *Registry* proporciona una serie de métodos que se describirán en el apartado 2.3.1.2, que permiten añadir, eliminar y obtener artefactos del repositorio, así como visualizar el contenido del mismo. Estos métodos son los que utilizan los diferentes *plug-ins* de MOMENT que necesitan interactuar con el MOMENT *Registry*.

Con el fin de que un usuario pueda utilizar el repositorio es necesario incluir una interfaz gráfica. Esta interfaz se denomina MOMENT *REGISTRY* UI y muestra al usuario el contenido del repositorio clasificado según se trate de metamodelos, transformaciones, operadores o relaciones de equivalencia. Además, permite el registro de nuevos artefactos en el repositorio y la eliminación de artefactos existentes mediante la acción de una serie de botones destinados a tal fin.

2.1.5. Resumen de requisitos

Los requisitos finales que ha de cumplir el MOMENT *Registry* son los que se enumeran a continuación:

- Almacenamiento de metamodelos, transformaciones, operadores y relaciones de equivalencia
- Registro, recarga y eliminación de metamodelos
- Suministro de un metamodelo a partir de su URI
- Registro y eliminación de transformaciones
- Suministro de una transformación a partir de su nombre
- Registro y eliminación de operadores
- Suministro de un operador a partir de su nombre
- Registro y eliminación de relaciones de equivalencia
- Suministro de una relación de equivalencia a partir de su nombre

Por parte de la MOMENT *Registry* UI, los requisitos que han de cumplirse son:

- Visualización de todos los metamodelos registrados
- Permitir el registro, la recarga y la eliminación de metamodelos
- Visualización de todos las transformaciones registradas
- Permitir el registro y la eliminación de transformaciones
- Visualización de todos los operadores registrados
- Permitir el registro y la eliminación de operadores
- Visualización de todas las relaciones de equivalencia registradas
- Permitir el registro y la eliminación de relaciones de equivalencia

2.2. Diseño de la solución

2.2.1. Fundamentos del diseño

El MOMENT *Registry* ha sido diseñado como un caso especial de repositorio cuya función principal es proporcionar una capa intermedia entre la persistencia física de los diferentes artefactos software empleados por MOMENT y la capa lógica (de utilización), de manera que si cambia la persistencia física todo lo demás debería seguir funcionando correctamente.

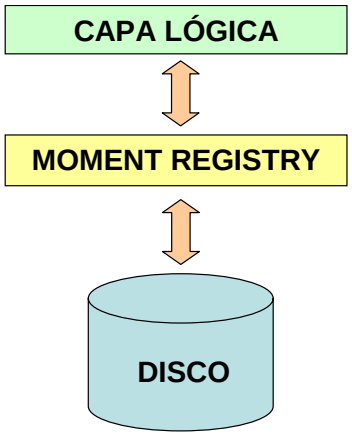


Figura 26. Arquitectura del MOMENT *Registry*.

Para lograr la independencia de la persistencia física de los artefactos, en el momento de registrar cualquier artefacto, el MOMENT *Registry* crea una copia privada del mismo y la almacena en disco. En todo momento, el repositorio mantiene para cada artefacto dos rutas de directorio: la ruta original donde se encuentra persistido y la ruta de la copia. Así, siempre que se le pide un artefacto, primero intenta obtenerlo de la ruta original pero si no es posible obtenerlo porque el fichero ha sido movido o eliminado, entonces se recurre a la ruta de la copia. De esta manera, el MOMENT *Registry* siempre va a poder suministrar cualquier artefacto que tenga registrado.

Las copias privadas de los artefactos se encuentran en una carpeta privada del *plug-in* MOMENT *Registry*. En Eclipse, los proyectos se almacenan en una carpeta denominada *workspace*. Dentro de esta carpeta, se encuentra otra carpeta llamada ".metadata" que contiene aquellos ficheros de configuración y de estado del *workspace*. A su vez, dentro de la carpeta ".metadata" existe otra carpeta llamada ".plugins", dentro de la cual, se da la posibilidad a los *plug-ins* que lo necesiten de guardar aquellos ficheros necesarios para su funcionamiento. Por tanto, en el caso del MOMENT *Registry*, será aquí donde se almacenen las copias de los artefactos que se registren. Con el fin de mantener los artefactos organizados según el tipo que sean, se crean cuatro carpetas para contener cada uno de los tipos: metamodelos, transformaciones, operadores o relaciones de equivalencia.

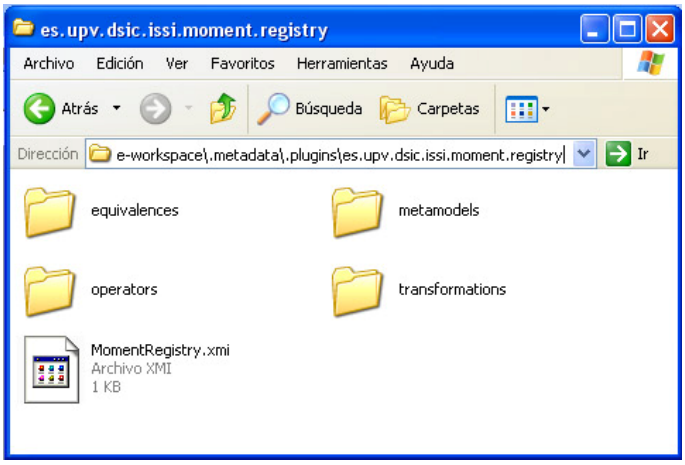


Figura 27. Almacenamiento de copias de artefactos en el MOMENT *Registry*.

2.2.2. Filosofía del diseño

Como es sabido, la gestión de modelos es una disciplina dentro de la ingeniería dirigida por modelos. En MOMENT, la filosofía del desarrollo dirigido por modelos ha estado presente en todas las fases de desarrollo y es por esta razón por la que los modelos son los principales artefactos de la plataforma.

En el diseño del MOMENT *Registry* también se ha tenido en cuenta esta filosofía, hasta el punto de que el propio MOMENT *Registry* es también un modelo, y como tal, puede ser serializado en XMI y cargado en EMF. Así, es posible persistir el repositorio en disco y tenerlo disponible cada vez que se inicie la herramienta MOMENT. El acceso al repositorio es posible ya que, en el momento de iniciar MOMENT, se carga en EMF el modelo MOMENT *Registry* y a través de los métodos que proporciona se puede interactuar con él.

La persistencia física del modelo MOMENT *Registry* se hace en el directorio privado del *plug-in* que implementa el MOMENT *Registry*. El nombre de este *plug-in* es "es.upv.dsic.issi.moment.registry", por tanto el modelo persistido se ubicará en la ruta "nombre_workspace/.metadata/.plugins/es.upv.dsic.issi.moment.registry".

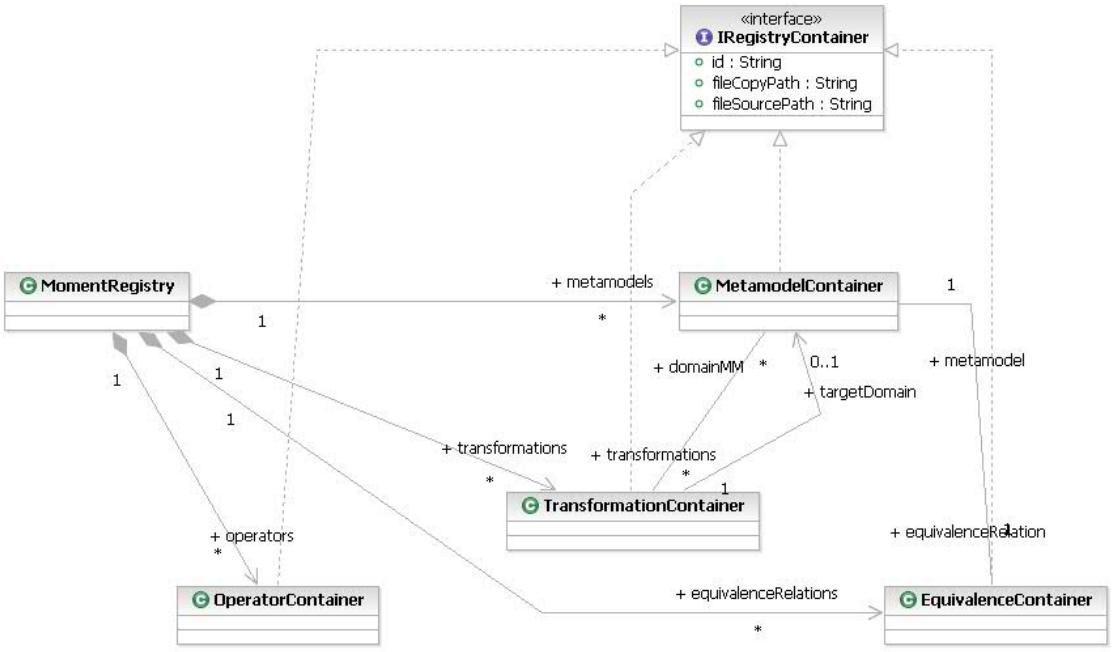


Figura 28. Modelo del MOMENT *Registry*.

Como se aprecia en la Figura 28, la clase "MomentRegistry" es la clase principal del modelo y contiene a las otras cuatro clases. Cada una de estas cuatro clases tiene como objetivo "contener" un tipo de artefacto. Así, la clase "OperatorContainer" contiene operadores, la clase "TransformationContainer" contiene modelos QVT-Relations que sean de transformación, la clase "MetamodelsContainer" contiene metamodelos y finalmente la clase "EquivalenceContainer" contiene modelos QVT-Relations de relaciones de equivalencia. Estas cuatro clases, implementan la interfaz "IRegistryContainer" de forma que así todo artefacto del MOMENT *Registry* tiene que tener un identificador (URI), una ruta de fichero origen y una ruta de fichero copia. Gracias a esta interfaz es posible mantener la independencia entre la lógica y la persistencia física de los artefactos software, ya comentada en el apartado anterior.

La asociación muchos a muchos entre la clase “MetamodelsContainer” y “TransformationContainer”, viene de que en una transformación pueden participar varios metamodelos como metamodelos de los modelos candidatos de la transformación. La otra asociación existente entre ambas clases, representa que en una transformación solo hay como mucho un metamodelo destino que corresponde al metamodelo del modelo destino de la transformación.

La asociación uno a muchos entre la clase “MetamodelsContainer” y “EquivalenceContainer”, viene de que en una relación de equivalencia solo hay un metamodelo como metamodelo de los modelos candidatos. Como el objetivo de las relaciones de equivalencias es comprobar la igualdad entre modelos, no tiene sentido que los metamodelos de los modelos que se quieren comparar sean distintos, puesto que si fuera así, obviamente nunca serían iguales.

2.2.3. Registro y eliminación de artefactos

Cuando se registra un artefacto en el MOMENT *Registry*, éste crea una copia privada del artefacto y a continuación se crea y se añade al repositorio un objeto contenedor del artefacto que contiene la identificación, y la ruta de directorio original y de copia del artefacto. Por ejemplo, si procede a registrar un metamodelo X persistido en la ruta de directorios A, el MOMENT *Registry* copiará ese metamodelo a su carpeta privada “metamodels” (ruta de directorio A’). A continuación crea un objeto “MetamodelContainer”, pone como “Id” la URI del metamodelo, A como ruta original y A’ como ruta de la copia, y lo añade al repositorio.

La eliminación de un artefacto se traduce en la eliminación de su copia privada y la supresión en el repositorio del objeto contenedor del artefacto. Para el ejemplo anterior, se eliminaría el metamodelo de la ruta de directorio A’ y se suprimiría del repositorio el “MetamodelContainer” que contiene al metamodelo X.

2.2.4. Cachés de búsqueda

Tal como se ha visto en el apartado anterior, el MOMENT *Registry* es un modelo que contiene información sobre la persistencia física de los artefactos de MOMENT. Por tanto, cuando se le pide al repositorio un artefacto determinado, el patrón de búsqueda trivial sería el de cargar el modelo, coger todos los artefactos del tipo del artefacto que se desea buscar y recorrerlos hasta encontrar el artefacto solicitado. Esta solución tiene el inconveniente de que el tiempo de encontrar un artefacto es proporcional al número de artefactos de ese tipo registrados en el repositorio (coste O(n)).

Las búsquedas en el MOMENT *Registry* son tareas muy frecuentes y por tanto conviene que sean lo más rápidas y eficientes posible. Con el fin de optimizar las búsquedas se crean cuatro cachés, una para cada tipo de artefacto. Cada caché consiste en un conjunto de pares identificador-objeto y son inicializadas automáticamente la primera vez que se utiliza el MOMENT *Registry*. En el proceso de inicialización, se cogen todos los artefactos de un tipo contenidos en el modelo MOMENT *Registry* y para cada uno de ellos se inserta en la caché una entrada identificador-objeto. Por ejemplo, en la inicialización de la caché de metamodelos, se cogen todos los objetos “MetamodelsContainer” contenidos en el modelo, y para cada uno de ellos se inserta en la caché un par URI-Objeto_MetamodelContainer.

La actualización de las cachés se realiza dinámicamente, es decir, si un artefacto es borrado del repositorio, automáticamente es borrado de su caché correspondiente. Análogamente, si se registra un artefacto en el repositorio, automáticamente se inserta en la caché.

Como resultado de la utilización de cachés, las búsquedas se efectúan sobre las cachés y no directamente sobre el modelo del MOMENT *Registry*, lo que se traduce en una mejora de la eficiencia de la búsqueda.

2.2.5. Seguridad: El registro de EMF

De manera similar a lo que ocurre en MOMENT con el MOMENT *Registry*, para que EMF pueda trabajar con modelos, es necesario que estos modelos se registren previamente en EMF. Esta analogía provoca que haya que priorizar la utilización de un registro respecto al otro. Así, por jerarquía de importancia y funcionamiento global, se da más prioridad al registro de EMF que al MOMENT *Registry*, puesto que el MOMENT *Registry* tiene un ámbito de funcionamiento local a MOMENT y los artefactos que tiene registrados han de registrarse también en EMF para que MOMENT pueda utilizarlos. Este registro de artefactos del MOMENT *Registry* en EMF, se realiza según demanda, es decir, si por ejemplo una definición de operador complejo importa un operador, este operador es solicitado al MOMENT *Registry* y a continuación se registra en EMF.

En MOMENT hay determinados metamodelos que se registran en EMF en el momento de iniciar la herramienta. Estos metamodelos son necesarios y fundamentales para el buen funcionamiento de MOMENT, por lo que dar al usuario la posibilidad de modificar alguno de estos metamodelos, registrarlos en el MOMENT *Registry* y que posteriormente puedan ser registrados en el registro de EMF sobrescribiendo el metamodelo original, puede acarrear consecuencias inesperadas y fatales en el funcionamiento de MOMENT. Por esta razón, siempre que se registre un metamodelo en el MOMENT *Registry* se comprueba que no esté registrado en el registro de EMF, por lo que nunca se permite el registro de metamodelos críticos del funcionamiento de MOMENT.

2.2.6. La función *reload*

La función *reload* permite la actualización de un artefacto metamodelo previamente registrado. Así, es posible registrar un metamodelo en el repositorio, modificarlo y proceder a su actualización en el repositorio sin necesidad de eliminarlo y volverlo a añadir.

El comportamiento de esta función consiste en reemplazar la copia privada que se tiene del metamodelo y volver a registrar el metamodelo en EMF si ya había sido registrado previamente.

El uso de la función *reload* también tiene que ver con lo explicado en los apartados anteriores. Si por ejemplo se registra un metamodelo determinado en el MOMENT *Registry* y posteriormente se hace uso de él, éste se registrará en EMF. Entonces, si se modifica ese metamodelo y se intenta actualizar en el MOMENT *Registry* eliminándolo y registrándolo de nuevo, el MOMENT *Registry* advertirá de que ese metamodelo ya está registrado en EMF y por tanto no permitirá su registro. Así, la función *reload* sobre un metamodelo, es una manera de indicarle al MOMENT *Registry*

que ese metamodelo, si ha sido registrado en EMF, ha sido registrado debido al funcionamiento de MOMENT a nivel de usuario.

2.2.7. MOMENT *Registry* UI

El usuario ha de poder interactuar con el MOMENT *Registry* de manera que pueda registrar y eliminar artefactos, y ver en todo momento el contenido del repositorio. Para ello, se diseña una capa de interfaz gráfica sobre la capa lógica del MOMENT *Registry*. Esta nueva capa llamada MOMENT *Registry* UI, permite la visualización organizada de todos los artefactos registrados en el repositorio. Como es sabido, existen cuatro tipos de artefactos, por tanto la interfaz gráfica se basa en una ventana con cuatro pestañas correspondientes a cada tipo de artefacto. Pulsando sobre cada pestaña, es posible visualizar los artefactos del tipo seleccionado registrados en el repositorio, y además se incluyen en esta vista, dos botones que permiten el registro y la eliminación de artefactos del tipo correspondiente. La visualización se realiza en forma de tabla, de forma que cada fila representa un artefacto y cada columna una propiedad. Según el tipo de artefacto variará el número de columnas. Para los metamodelos, hay tres columnas que indican el nombre, la URI y la ruta del metamodelo (ruta original). En el caso de los operadores, las transformaciones y las relaciones de equivalencia, solamente hay dos columnas que indican el nombre y la ruta original del artefacto.

El usuario tendrá acceso a la MOMENT *Registry* UI a través de pestaña de preferencias de MOMENT situada en la ventana de preferencias de Eclipse.

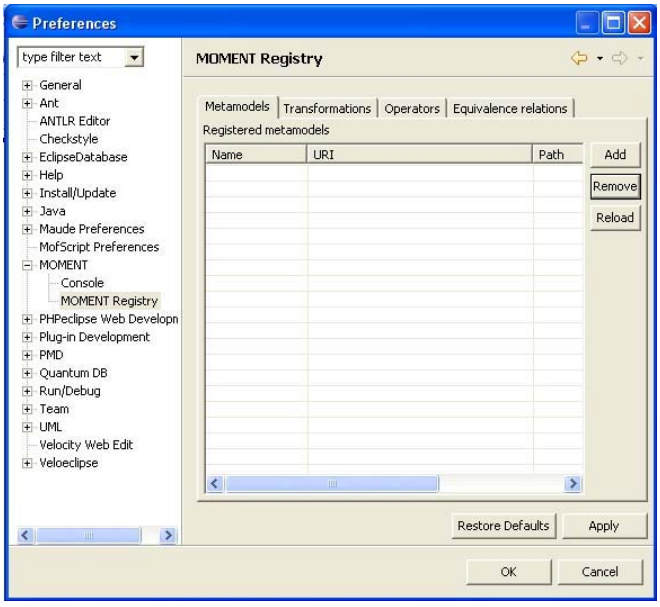


Figura 29. Interfaz gráfica del MOMENT *Registry*.

En la Figura 29 se observa como la MOMENT *Registry* UI proporciona una ventana con las cuatro pestañas correspondientes a los metamodelos, las transformaciones, los operadores y las relaciones de equivalencia. También es posible observar los botones (Add, Remove y Reload) que permiten al usuario interactuar con el repositorio y la tabla donde se muestra el contenido del MOMENT *Registry* para un tipo de artefacto determinado.

En el Anexo VIII se encuentra el manual de usuario del MOMENT *Registry*, que explica de forma detallada cómo un usuario puede interactuar con él a través de la MOMENT *Registry* UI.

2.3. Implementación

2.3.1. Moment *Registry*

La implementación del MOMENT *Registry* se ha realizado en dos etapas. En una primera etapa se han obtenido las clases Java a partir del modelo del MOMENT *Registry* (Figura 28) y en una segunda etapa se han añadido los métodos necesarios a la clase "MomentRegistry" para que sea posible soportar toda la funcionalidad requerida.

También ha sido necesario insertar código en el método "Start" de la clase "MomentRegistryPlugin" para que el modelo del repositorio se registre en el registro de EMF en el momento de iniciar el MOMENT *Registry* y se inicialicen las cachés de búsqueda. Esta clase se encuentra en el fichero "MomentRegistryPlugin.java" en el paquete "es.upv.dsic.issi.moment.registry".

2.3.1.1. Obtención clases Java

La obtención de las clases Java que forman el modelo del MOMENT Registy se realiza de forma automática utilizando EMF. Para ello es necesario crear previamente un proyecto EMF para el modelo *ecore* del MOMENT *Registry* y a continuación generar automáticamente las clases Java que forman el modelo.

La creación del proyecto EMF se realiza mediante un asistente que proporciona el propio EMF. Este asistente solicita la siguiente información:

- Nombre del proyecto. El nombre del proyecto utilizado para el MOMENT *Registry* es "es.upv.dsic.issi.moment.registry".
- Modelo *ecore*. La introducción del modelo *ecore* puede realizarse de tres formas: importando un modelo *ecore* existente, importando un modelo de clases Rational Rose o importando un modelo UML2. El modelo del MOMENT *Registry* ha sido realizado con la herramienta *Rational Software Architect* de IBM, la cual permite la exportación de modelos como modelos *ecore*. Por tanto previamente a la creación del proyecto EMF se ha exportado el modelo MOMENT *Registry* como modelo *ecore*.
- Nombre del fichero "genmodel". Este fichero es creado automáticamente por el asistente y contiene información adicional sobre cómo EMF tiene que generar el código del modelo. Para este proyecto se ha utilizado como nombre del "genmodel" el mismo nombre del modelo *ecore* del MOMENT *Registry*, aunque podría haberse utilizado cualquier otro nombre.

Una vez finalizado el asistente de creación del proyecto EMF solo falta generar el código del modelo utilizando el fichero "genmodel". Haciendo doble clic sobre este fichero, se muestra su contenido mediante un editor en forma de árbol y haciendo clic con el botón izquierdo sobre el nodo raíz, aparece un menú emergente que permite seleccionar la opción de generar el código del modelo.

2.3.1.2. Métodos implementados

A continuación se describen los métodos que se han implementado en la clase "MomentRegistry" para soportar la funcionalidad requerida del MOMENT *Registry*. Esta implementación se ha realizado sobre el fichero "MomentRegistryImpl.java" ubicado en el paquete "es.upv.dsic.issi.moment.registry.model.impl" del proyecto EMF creado para el MOMENT *Registry*.

La descripción de los métodos se realizará de forma agrupada según la finalidad que persiguen.

- *Metamodelos*

Los métodos implementados para tratar los metamodelos en el MOMENT *Registry* han sido los siguientes:

- **boolean addMetamodel(String sourcePath)**

  Este método permite registrar un metamodelo en el repositorio. Recibe como parámetro de entrada la ruta de persistencia física del metamodelo y devuelve un valor booleano para indicar si la operación ha sido realizada satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Comprobar que el metamodelo no esté registrado ni en EMF ni en el MOMENT *Registry*.
  2. Crear la copia privada del metamodelo en cuestión.
  3. Crear un "MetamodelContainer" para contener el metamodelo.
  4. Añadir el "MetamodelContainer" creado al MOMENT *Registry*.
  5. Guardar el estado del MOMENT *Registry*.
  6. Actualizar la caché de metamodelos con el nuevo metamodelo registrado.
  7. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- **boolean removeMetamodel(String uri)**

  Este método permite eliminar un metamodelo del repositorio. Recibe como parámetro de entrada la URI del metamodelo que se desea eliminar y devuelve un valor booleano para indicar si la operación se ha realizado satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Localizar el "MetamodelContainer" cuyo ID coincide con la URI pasada como parámetro al método.
  2. Eliminar el "MetamodelContainer".
  3. Guardar el estado del modelo MOMENT *Registry*.
  4. Actualizar la caché de metamodelos eliminando el metamodelo borrado.
  5. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- **boolean reloadMetamodel(MetamodelContainer mc)**

  Este método permite recargar un metamodelo del repositorio. Recibe como parámetro de entrada el "MetamodelContainer" del metamodelo que se desea recargar y devuelve un valor booleano para indicar si la operación se ha realizado satisfactoriamente o no.

  El significado de este método es poder actualizar en el MOMENT *Registry* un metamodelo que ha sido modificado después de haber sido registrado sin necesidad de eliminarlo y volver añadir. Además, ofrece las propiedades de seguridad vistas en el apartado 2.2.5. Su funcionamiento se resume en:

  1. Capturar la ruta de persistencia origen del metamodelo contenido en el "MetamodelContainer" pasado como parámetro al método.
  2. Actualizar la copia privada del metamodelo.
  3. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- *Transformaciones*

Los métodos implementados para tratar las transformaciones en el MOMENT *Registry* han sido los siguientes:

- **boolean addTransformation(String sourcePath)**

  Este método permite registrar una transformación en el repositorio. Recibe como parámetro de entrada la ruta de persistencia física del modelo QVT-Relations de la transformación y devuelve un valor booleano para indicar si la operación ha sido realizada satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Comprobar que la transformación no esté registrada en el MOMENT *Registry*.
  2. Crear la copia privada de la transformación en cuestión.
  3. Crear un "TransformationContainer" para contener la transformación.
  4. Recorrer todas las relaciones de equivalencia especificadas en el modelo y poner todos sus nombres en el ID del "EquivalenceContainer".
  5. Añadir el "TransformationContainer" creado al MOMENT *Registry*.
  6. Guardar el estado del MOMENT *Registry*.
  7. Actualizar la caché de transformaciones con la nueva transformación registrada.
  8. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- **boolean removeTransformation(String name)**

  Este método permite eliminar una transformación del repositorio. Recibe como parámetro de entrada el nombre de la transformación que se desea eliminar y devuelve un valor booleano para indicar si la operación se ha realizado satisfactoriamente o no.

  El funcionamiento de este método se resume en:

1. Localizar el “TransformationContainer” en cuyo ID aparece el nombre de la transformación pasado como parámetro al método.
2. Eliminar el “TransformationContainer”.
3. Guardar el estado del modelo MOMENT *Registry*.
4. Actualizar la caché de transformaciones eliminando la transformación borrada.
5. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- *Operadores*

Los métodos implementados para tratar los operadores en el MOMENT *Registry* han sido los siguientes:

- **boolean addOperator(String sourcePath)**

  Este método permite registrar un operador en el repositorio. Recibe como parámetro de entrada la ruta de persistencia física del modelo del operador y devuelve un valor booleano para indicar si la operación ha sido realizada satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Comprobar que el operador no esté registrado en el MOMENT *Registry*
  2. Crear la copia privada del operador en cuestión
  3. Crear un “OperatorContainer” para contener la transformación
  4. Añadir el “OperatorContainer” creado al MOMENT *Registry*
  5. Guardar el estado del MOMENT *Registry*
  6. Actualizar la caché de operadores con el nuevo operador creado
  7. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario

- **boolean removeOperator(String name)**

  Este método permite eliminar un operador del repositorio. Recibe como parámetro de entrada el nombre del operador que se desea eliminar y devuelve un valor booleano para indicar si la operación se ha realizado satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Localizar el “OperatorContainer” cuyo ID coincide con el nombre del operador pasado como parámetro al método.
  2. Eliminar el “OperatorContainer”.
  3. Guardar el estado del modelo MOMENT *Registry*.
  4. Actualizar la caché de operadores eliminando el operador borrado.
  5. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- *Relaciones de equivalencia*

Los métodos implementados para tratar las relaciones de equivalencia en el MOMENT *Registry* han sido los siguientes:

- **boolean addEquivalenceRelation(String sourcePath)**

  Este método permite registrar una relación de equivalencia en el repositorio. Recibe como parámetro de entrada la ruta de persistencia física del modelo QVT-Relations de la relación de equivalencia y devuelve un valor booleano para indicar si la operación ha sido realizada satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Comprobar que la relación de equivalencia no esté registrada en el MOMENT *Registry*.
  2. Crear la copia privada de la relación de equivalencia en cuestión.
  3. Crear un "EquivalenceContainer" para contener la relación de equivalencia.
  4. Recorrer todas las relaciones de equivalencia especificadas en el modelo y poner todos sus nombres en el ID del "EquivalenceContainer".
  5. Añadir el "EquivalenceContainer" creado al MOMENT *Registry*.
  6. Guardar el estado del MOMENT *Registry*.
  7. Actualizar la caché de relaciones de equivalencia con la nueva relación de equivalencia registrada.
  8. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario

- **boolean removeEquivalenceRelation(String name)**

  Este método permite eliminar una transformación del repositorio. Recibe como parámetro de entrada el nombre de la transformación que se desea eliminar y devuelve un valor booleano para indicar si la operación se ha realizado satisfactoriamente o no.

  El funcionamiento de este método se resume en:

  1. Localizar el "EquivalenceContainer" en cuyo ID aparece el nombre de la relación de equivalencia pasada como parámetro al método.
  2. Eliminar el "EquivalenceContainer".
  3. Guardar el estado del modelo MOMENT *Registry*.
  4. Actualizar la caché de relaciones de equivalencia eliminando la relación de equivalencia borrada.
  5. Devolver *true* si todo se ha realizado correctamente o *false* en caso contrario.

- *Cachés*

Las cachés se han implementado como tablas *Hash*. De esta manera, las tablas *Hash* o tablas de dispersión solucionan satisfactoriamente nuestro problema: nos permiten acceder asociativamente a la información y, además, lo hacen en un

tiempo medio constante, es decir, que el tiempo necesario para acceder a un elemento, no va a depender del número de elementos que almacene la estructura.

Para cada tipo de artefacto se ha implementado un tabla *hash* utilizando *HashMaps* parametrizados de Java 5.0. Cada una de estas tablas tiene un conjunto de pares clave-valor, donde la clave es un identificador del artefacto y el valor es la *resource* de ese artefacto. Las claves utilizadas han variado según el tipo de artefacto debido a que en un modelo QVT-Relations pueden definirse varias transformaciones o varias relaciones de equivalencia. Cada una de estas transformaciones o relaciones de equivalencia tiene un nombre, por tanto, para un modelo QVT-Relations habrá asociada una lista de nombres de transformaciones o relaciones de equivalencia que contiene. Así, para las tablas *hash* de transformaciones y relaciones de equivalencia se ha utilizado como clave el "recurso" del modelo y como valor la lista de nombres.

Las declaraciones de las tablas *Hash* utilizadas son las siguientes:

```
Map<String, Resource> MetamodelsCache =
                                        new HashMap <String, Resource>();
Map<Resource, List<String>> TransformationsCache =
                                  new HashMap <Resource, List<String>>();
Map<String, Resource> OperatorsCache =
                                        new HashMap <String, Resource>();
Map<Resource, List<String>> EquivalencesCache =
                                 new HashMap <Resource, List<String>>();
```

Los métodos implementados que hacen posible la inicialización de cada una de las cachés son los siguientes:

- **void initMetamodelsCache()**

  Este método inicializa la caché de metamodelos.

  Su funcionamiento se resume en: para todo "MetamodelContainer" del modelo MOMENT *Registry*, intentar crear el recurso (*resource*) para la ruta de persistencia física origen. En caso de fallo, porque por ejemplo ésta ha cambiado, crea la *resource* con la ruta de la copia. A continuación, insertar en la caché "MetamodelsCache" el par "id" del "MetamodelContainer" y la *resource* creada.

- **void initTransformationsCache()**

  Este método inicializa la cache de transformaciones.

  Su funcionamiento se resume en: para todo "TransformationContainer" del modelo MOMENT *Registry*, intentar crear el recurso (*resource*) para la ruta de persistencia física origen. En caso de fallo, porque por ejemplo ésta ha cambiado, crea la *resource* con la ruta de la copia. A continuación, insertar en la caché "TransformationsCache" el par *resource* que ha sido creada y el "id" del "TransformationContainer".

- **void initOperatorsCache()**

  Este método inicializa la caché de operadores.

  Su funcionamiento se resume en: para todo "OperatorContainer" del modelo MOMENT *Registry*, intentar crear el recurso (*resource*) para la ruta de persistencia física origen. En caso de fallo, porque por ejemplo ésta ha cambiado, crea la *resource* con la ruta de la copia. A continuación, insertar en la caché "OperatorsCache" el par "id" del "OperatorContainer" y la *resource* creada.

- **void initEquivalencesCache()**

  Este método inicializa la caché de relaciones de equivalencia.

  Su funcionamiento se resume en: para todo "EquivalenceContainer" del modelo MOMENT *Registry*, intentar crear el recurso (*resource*) para la ruta de persistencia física origen. En caso de fallo, porque por ejemplo ésta ha cambiado, crea la *resource* con la ruta de la copia. A continuación, insertar en la caché "EquivalencesCache" el par *resource* que ha sido creada y el "id" del "EquivalenceContainer".

  - *Petición de artefactos al repositorio*

A continuación se describen los métodos que hacen posible la petición de artefactos al MOMENT *Registry*. Todos estos métodos devuelven un elemento de tipo *Resource* que se corresponde con el recurso asociado al artefacto solicitado.

- **Resource getMetamodelByUri(String uri)**

  Este método devuelve la *resource* del metamodelo del repositorio cuya URI coincide con la que se pasa como parámetro al método.

  El funcionamiento de este método se basa en buscar la URI del metamodelo en la caché de metamodelos y si existe se devuelve el metamodelo solicitado y sino se devuelve *null*.

- **Resource getTransformation(String name)**

  Este método devuelve la *resource* de la transformación del repositorio cuyo nombre coincide con el que se pasa como parámetro al método.

  El funcionamiento de este método se basa en buscar el nombre de la transformación en la caché de transformaciones y si existe se devuelve la transformación solicitada y sino se devuelve *null*.

- **Resource getOperator(String name)**

  Este método devuelve la *resource* del operador del repositorio cuyo nombre coincide con el que se pasa como parámetro al método.

  El funcionamiento de este método se basa en buscar el nombre del operador en la caché de operadores y si existe se devuelve el operador solicitado y sino se devuelve *null*.

- **Resource getEquivalenceRelation(String name)**

  Este método devuelve la *resource* de la relación de equivalencia del repositorio cuyo nombre coincide con el que se pasa como parámetro al método.

  El funcionamiento de este método se basa en buscar el nombre de la relación de equivalencia en la caché de relaciones de equivalencia y si existe se devuelve la relación de equivalencia solicitada y sino se devuelve *null*.

2.3.1.3. Modificación método "Start"

La inserción de código en el método "Start" de la clase "MomentRegistryPlugin" se debe principalmente a la necesidad de registrar el modelo del repositorio en el registro de EMF para poder trabajar directamente con él e inicializar las cachés de búsqueda para metamodelos, transformaciones, operadores y relaciones de equivalencia. Además, en la clase "MomentRegistryPlugin" se han insertado dos

atributos estáticos que facilitan la invocación de los métodos de la clase "MomentRegistry" así como la persistencia del modelo en cualquier momento. Estos atributos son los siguientes:

```
public static MomentRegistry registryInstance =
                    RegistryFactory.eINSTANCE.createMomentRegistry();
public static Resource PersistentRegistry = null;
```

El atributo "registryInstance" es del tipo "MomentRegistry" y es utilizado para poder realizar las invocaciones a los métodos implementados en la clase "MomentRegistry".

El atributo "PersistentRegistry" es del tipo "Resource" y se utiliza para guardar la *resource* del modelo MOMENT *Registry*. De esta manera, se facilita el cargado y el guardado del modelo desde otras clases.

El funcionamiento del código insertado en el método "Start" es el siguiente:

1. Creación de un fichero URI con la ruta donde se persiste el modelo del MOMENT *Registry*. Esta ruta pertenece a la ruta de directorios privada del *plug-in* y se obtiene mediante el siguiente código:

```
MomentRegistryPlugin.getDefault().getStateLocation().append(
                                                "MomentRegisty.xmi");
```

2. Creación e inicialización del paquete del modelo MOMENT *Registry*. Este paquete tiene el nombre de "RegistryPackage" y se inicializará con el nombre y con el prefijo del modelo.
3. Si el atributo "PersitentRegistry" tiene valor *null,* es decir, es la primera vez que se ejecuta el MOMENT *Registry* durante la ejecución actual de MOMENT, se inicializan los atributos "PersistentRegistry" y "registryInstance", y a continuación se inicializan las cachés de búsqueda. Si durante la inicialización de los atributos citados se capturara alguna excepción significaría que es la primera vez que se ejecuta el MOMENT *Registry* en el *workspace* de trabajo actual. Esto significa que no existe el fichero "MomentRegistry.xmi" (modelo MOMENT *Registry* persistido) en las carpetas privadas del *plug-in*. El manejador de estas excepciones crea este fichero e inicializa debidamente los atributos "registryInstance" y "PersistentInstance".

2.3.2. MOMENT *Registry* UI

La implementación de la MOMENT *Registry* UI se ha realizado en tres etapas. En una primera etapa se ha creado el proyecto del *plug-in* como ventana de preferencias. Posteriormente se ha implementado la interfaz gráfica de esta ventana (pestañas, botones, tabla, etc) y por último se han implementado los métodos necesarios para cumplir los requisitos especificados para la MOMENT *Registry* UI en el apartado 2.1.5.

2.3.2.1. Creación del *plug-in*

El proceso de creación del *plug-in* se basa en la creación de un proyecto vacío y en la utilización del mecanismo de extensiones del *worbench* de Eclipse para

configurarlo como ventana de preferencias en la pestaña de MOMENT de la ventana de preferencias de Eclipse.

Como resultado de la creación del *plug-in* se obtienen tres clases Java:

- **MomentRegistryPreferencePage**

  Esta clase extiende la clase "PreferencePage" e implementa la interfaz "IWorkbenchPreferencePage". Además, proporciona dos métodos por defecto, "Control createContents(Composite parent)" y "void init(IWorkbench workbench)". El método "createContents" es el método principal de la clase y en él se codifica la interfaz gráfica principal de la ventana de preferencias.

- **PreferenceConstants**

  Esta clase contiene las definiciones constantes de las preferencias del *plug-in*.

- **PreferenceInitializer**

  Esta clase inicializa los valores de preferencias por defecto del *plug-in*.

2.3.2.2. Implementación interfaz gráfica

La implementación de la interfaz gráfica se ha realizado utilizando SWT, que es una utilidad de código abierto diseñada para Java que proporciona portabilidad y eficiencia a las facilidades de interfaz de usuario de los sistemas operativos sobre los que se lleva a cabo la implementación.

La implementación de la interfaz gráfica se ha dividido en varios métodos:

- **Control createContents(Composite top)**

  En este método se implementa la carpeta de pestañas para los diferentes tipos de artefactos. El contenido gráfico de cada pestaña se implementa en un método independiente

- **Control createMetamodelsControl(Composite top)**

  Este método implementa los componentes gráficos de la pestaña de metamodelos (*Metamodels)*. Entre estos componentes se encuentran la tabla donde se presenta la visualización de los metamodelos registrados en el MOMENT *Registry* y los botones de registrar (*Add*), eliminar (*Remove*) y recargar (*Reload*) un metamodelo.

- **Control createTransformationsControl(Composite top)**

  Este método implementa los componentes gráficos de la pestaña de transformaciones (*Transformations)*. Entre estos componentes se encuentran la tabla donde se presenta la visualización de las transformaciones registradas en el MOMENT *Registry* y los botones de registrar (*Add*) y eliminar (*Remove*) una transformación.

- **Control createOperatorsControl(Composite top)**

  Este método implementa los componentes gráficos de la pestaña de operadores (*Operators)*. Entre estos componentes se encuentran la tabla donde se presenta la visualización de los operadores registrados en el MOMENT *Registry* y los botones de registrar (*Add*) y eliminar (*Remove*) un operador.

- **Control createEquivRelsControl(Composite top)**

  Este método implementa los componentes gráficos de la pestaña de relaciones de equivalencia (*Equivalences)*. Entre estos componentes se encuentran la tabla donde se presenta la visualización de las relaciones de equivalencia registradas en el MOMENT *Registry* y los botones de registrar (*Add*) y eliminar (*Remove*) una relación de equivalencia.

2.3.2.3. Implementación de métodos

- **void handleAddMetamodel( )**

  Este método permite añadir un metamodelo al MOMENT *Registry* implementando la acción del botón "Add" de la pestaña "Metamodels".

  El funcionamiento del método es el siguiente:

  1. Solicitud al usuario del metamodelo a registrar a través de una ventana emergente que muestra el árbol de directorios del *workspace* actual de trabajo.
  2. Captura de la ruta donde se encuentra persistido el metamodelo.
  3. Captura del paquete del metamodelo.
  4. Invocación al método "addMetamodel" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".
  5. Mostar mensaje de error si el metamodelo ya existe (addMetamodel devuelve *false*) o en caso contrario actualizar la tabla de visualización de metamodelos con el nuevo metamodelo.

- **void handleRemoveMetamodel( )**

  Este método permite eliminar del MOMENT *Registry* un metamodelo seleccionado de la tabla de visualización de los metamodelos actualmente registrados, implementando la acción del botón *"Remove"* de la pestaña *"Metamodels"*.

  El funcionamiento del método es el siguiente:

  1. Caputar el metamodelo seleccionado.
  2. Invocar al método "removeMetamodel" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".
  3. Eliminar el metamodelo de la tabla de visualización de los metamodelos registrados.

- **handleReloadMetamodel( )**

  Este método permite recargar (actualizar) un metamodelo registrado en el MOMENT *Registry*.

  El funcionamiento del método es el siguiente:

  1. Encontrar el "MetamodelContainer" cuyo "id" coincide con la URI del metamodelo seleccionado.
  2. Invocar al método "reloadMetamodel" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".
  3. Si el método devuelve *true* se muestra un mensaje informando de que la operación se ha realizado correctamente. Si devuelve *falso* se

muestra un mensaje informando de que la operación no se ha realizado debido a algún error.

- **loadMetamodelsView( )**

  Este método carga en la tabla de visualización de metamodelos todos los metamodelos registrados en el MOMENT *Registry*.

  Su funcionamiento es el siguiente:

  1. Para cada "MetamodelContainer" existente en el MOMENT *Registry*, se coge la ruta origen y la de la copia.
  2. Se intenta capturar el recurso contenido en el "MetamodelContainer" mediante la ruta origen. Si falla, se captura mediante la ruta de la copia.
  3. Se captura el paquete del metamodelo.
  4. Con la información del paquete se inserta una fila en la tabla de visualización de metamodelos.

- **int[ ] getSelectedMetamodel( )**

  Este método captura los índices de los metamodelos seleccionados en la tabla de visualización de metamodelos.

- **void handleAddTransformation( )**

  Este método permite añadir una transformación al MOMENT *Registry* implementando la acción del botón *"Add"* de la pestaña *"Transformations"*.

  El funcionamiento del método es el siguiente:

  1. Solicitud al usuario del modelo QVT-Relations que especifica la transformación o transformaciones a registrar a través de una ventana emergente que muestra el árbol de directorios del *workspace* actual de trabajo.
  2. Captura de la ruta donde se encuentra persistido el modelo QVT-Relations.
  3. Captura del paquete del modelo del QVT-Relations.
  4. Invocación al método "addTransformation" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".
  5. Mostar mensaje de error si la transformación ya existe (addTransformation devuelve *false*).
  6. En caso contrario, para cada transformación especificada en el modelo QVT-Relations, actualizar la tabla de visualización de transformaciones con los valores de la transformación.

- **void handleRemoveTransformation( )**

  Este método permite eliminar del MOMENT *Registry* una transformación seleccionada de la tabla de visualización de las transformaciones actualmente registradas, implementando la acción del botón *"Remove"* de la pestaña *"Transformations"*.

  El funcionamiento del método es el siguiente:

  1. Caputar la transformación seleccionada.
  2. Invocar al método "removeTransformation" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".

3. Eliminar la transformación de la tabla de visualización de las transformaciones registradas.

Nota: Si la transformación a eliminar se encuentra especificada en un modelo QVT-Relations en el cual hay especificadas más transformaciones, se eliminan del MOMENT *Registry* todas las transformaciones especificadas en el modelo. Caben otras soluciones posibles, pero se ha optado por esta opción pensando en futuras modificaciones sobre MOMENT-QVT en las que se pretende que en un modelo QVT-Relations solo pueda especificarse una transformación o una relación de equivalencia.

- **int[ ] getSelectedTransformation( )**

  Este método captura los índices de las transformaciones seleccionadas en la tabla de visualización de transformaciones.

- **void loadTransformationsView( )**

  Este método carga en la tabla de visualización de transformaciones todas las transformaciones registradas en el MOMENT *Registry*.

  Su funcionamiento es el siguiente:

  1. Para cada "TransformationContainer" existente en el MOMENT *Registry*, se coge la ruta origen y la de la copia.
  2. Se intenta capturar el recurso contenido (modelo QVT-Relations) en el "TransformationContainer" mediante la ruta origen. Si falla, se captura mediante la ruta de la copia.
  3. Se captura el paquete del modelo del QVT-Relations.
  4. Para cada transformación especificada en el modelo, se inserta una fila en la tabla de visualización de transformaciones.

- **void handleAddOperator( )**

  Este método permite añadir un operador al MOMENT *Registry* implementando la acción del botón *"Add"* de la pestaña *"Operators"*.

  El funcionamiento del método es el siguiente:

  1. Solicitud al usuario del modelo del operador a registrar a través de una ventana emergente que muestra el árbol de directorios del *workspace* actual de trabajo.
  2. Captura de la ruta donde se encuentra persistido el modelo del operador.
  3. Captura del paquete del modelo del operador.
  4. Invocación al método "addOperator" mediante el atributo de clase "registryInstance" implementado en la clase "MomentRegistryPlugin".
  5. Mostar mensaje de error si el operador ya existe (addOperator devuelve *false*) o en caso contrario actualizar la tabla de visualización de operadores actualmente registrados con el nuevo operador.

- **void handleRemoveOperator( )**

  Este método permite eliminar del MOMENT *Registry* un operador seleccionado de la tabla de visualización de los operadores actualmente registrados, implementando la acción del botón *"Remove"* de la pestaña *"Operators".*

El funcionamiento del método es el siguiente:

1. Caputar el operador seleccionado.
2. Invocar al método “removeOperator” mediante el atributo de clase “registryInstance” implementado en la clase “MomentRegistryPlugin”.
3. Eliminar el operador de la tabla de visualización de los operadores registrados.

- **int[ ] getSelectedOperator( )**

  Este método captura los índices de los operadores seleccionados en la tabla de visualización de operadores.

- **void loadOperatorsView( )**

  Este método carga en la tabla de visualización de operadores todos los operadores registrados en el MOMENT *Registry*.

  Su funcionamiento es el siguiente:

  1. Para cada “OperatorContainer” existente en el MOMENT *Registry*, se coge la ruta origen y la de la copia.
  2. Se intenta capturar el recurso contenido en el “OperatorContainer” mediante la ruta origen. Si falla, se captura mediante la ruta de la copia.
  3. Se captura el paquete del modelo del operador.
  4. Con la información del paquete se inserta una fila en la tabla de visualización de operadores.

- **void handleAddEquivRel( )**

  Este método permite añadir una relación de equivalencia al MOMENT *Registry* implementando la acción del botón *“Add”* de la pestaña *“Equivalences”*.

  El funcionamiento del método es el siguiente:

  1. Solicitud al usuario del modelo QVT-Relations que especifica la relación o relaciones de equivalencia a registrar a través de una ventana emergente que muestra el árbol de directorios del *workspace* actual de trabajo.
  2. Captura de la ruta donde se encuentra persistido el modelo QVT-Relations.
  3. Captura del paquete del modelo del QVT-Relations.
  4. Invocación al método “addEquivalence” mediante el atributo de clase “registryInstance” implementado en la clase “MomentRegistryPlugin”.
  5. Mostar mensaje de error si la relación de equivalencia ya existe (addEquivalence devuelve *false*).
  6. En caso contrario, para cada relación de equivalencia especificada en el modelo QVT-Relations, actualizar la tabla de visualización de transformaciones con los valores de la transformación.

- **void handleRemoveEquivRel( )**

  Este método permite eliminar del MOMENT *Registry* una relación de equivalencia seleccionada de la tabla de visualización de las relaciones de equivalencia, implementando la acción del botón *“Remove”* de la pestaña *“Equivalences”*.

El funcionamiento del método es el siguiente:

1. Caputar la relación de equivalencia seleccionada.
2. Invocar al método “removeEquivalence” mediante el atributo de clase “registryInstance” implementado en la clase “MomentRegistryPlugin”.
3. Eliminar la relación de equivalencia de la tabla de visualización de las relaciones de equivalencia registradas.

Nota: Si la relación de equivalencia a eliminar se encuentra especificada en un modelo QVT-Relations en el cual hay especificadas más relaciones de equivalencia, se eliminan del MOMENT *Registry* todas las relaciones de equivalencia especificadas en el modelo. Caben otras soluciones posibles, pero se ha optado por esta opción pensando en futuras modificaciones sobre MOMENT-QVT en las que se pretende que en un modelo QVT-Relations solo pueda especificarse una transformación o una relación de equivalencia.

- **int[ ] getSelectedEquivalence( )**

  Este método captura los índices de las relaciones de equivalencia seleccionadas en la tabla de visualización de relaciones de equivalencia.

- **loadEquivRelsView( )**

  Este método carga en la tabla de visualización de relaciones de equivalencia todas las relaciones de equivalencia registradas en el MOMENT *Registry*.

  Su funcionamiento es el siguiente:

  1. Para cada “EquivalenceContainer” existente en el MOMENT *Registry*, se coge la ruta origen y la de la copia.
  2. Se intenta capturar el recurso contenido (modelo QVT-Relations) en el “EquivalenceContainer” mediante la ruta origen. Si falla, se captura mediante la ruta de la copia.
  3. Se captura el paquete del modelo del QVT-Relations.
  4. Para cada relación de equivalencia especificada en el modelo, se inserta una fila en la tabla de visualización de relaciones de equivalencia.

2.4. *Plug-ins* resultantes

Como resultado de la implementación del MOMENT *Registry* se han obtenido dos *plug-ins* que implementan el modelo del MOMENT *Registry* y su interfaz gráfica de usuario respectivamente.

El *plug-in* “es.upv.dsic.issi.moment.registry” implementa las clases Java que codifican el modelo del MOMENT *Registry*. Los ficheros más importantes de este *plug-in* son “MomentRegistry.java” y “MomentRegistryImpl.java”.

El fichero “MomentRegistry.java” codifica la interfaz que implementa la clase “MomentRegistryImpl”, por tanto en este fichero se especifican las cabeceras de los métodos que se implementan en la clase “MomentRegistryImpl”. Este fichero se encuentra en el paquete “es.upv.dsic.issi.moment.registry.model”.

El fichero “MomentRegistryImpl.java” codifica la clase “MomentRegistryImpl” que se corresponde con la clase “MomentRegistry” del modelo y además implementa los métodos de la interfaz codificada en el fichero “MomentRegistry.java”. La mayor

parte de la implementación de los métodos descritos en el apartado 2.3 se codifican en este fichero.

3. ARQUITECTURA DE MOMENT

Tras la integración del MOMENT *Registry* en MOMENT, ha variado la dependencia entre los diferentes componentes que conforman la herramienta, convirtiéndose el componente del MOMENT *Registry* en uno de los componentes principales.

De esta manera, la arquitectura de MOMENT queda como sigue en la Figura 30, donde también se han incluido los componentes implementados como parte del soporte para transformación y manipulación de modelos (QVT *Parser* y DSL *Parser*).

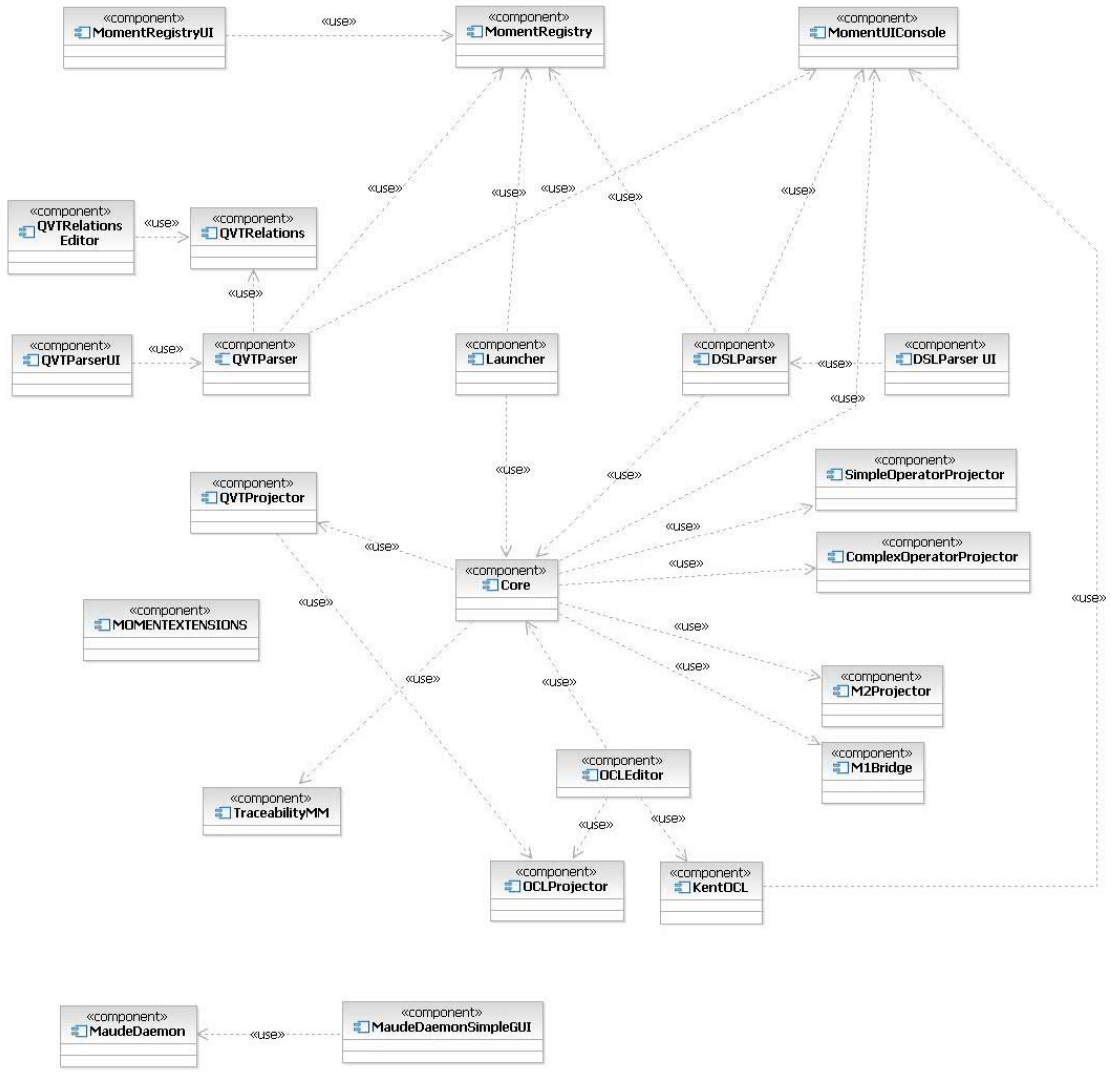


Figura 30. Arquitectura de MOMENT

A continuación se describen de forma global y abstracta los diferentes componentes presentes en la arquitectura de MOMENT:

- **MomentUIConsole**

  Este componente implementa la consola de MOMENT, a través de la cual se informará al usuario sobre posibles errores, *warnings* o información de estado.

- **MomentRegistryUI**

  Este componente implementa la interfaz gráfica de usuario que permite la interacción de un usuario con el MOMENT *Registry*. MomentRegistryUI depende del componente MomentRegistry puesto que éste proporciona los métodos que hacen posible operar directamente sobre el repositorio como por ejemplo registrar un operador.

- **MomentRegistry**

  Este componente implementa las clases Java que constituyen el modelo del MOMENT *Registry* y además proporciona los métodos necesarios para interaccionar con el repositorio. De este componente dependen todos aquellos componentes que requieren utilizar el MOMENT *Registry*.

- **QVTRelations**

  Este componente implementa las clases Java que constituyen el modelo QVT-Relations planteado en el Anexo V.

- **QVTRelationsEditor**

  Este componente implementa un editor en forma de árbol de modelos QVT-Relations. Por esta razón, depende del componente QVT-Relations.

- **QVTParserUI**

  Este componente proporciona un menú emergente que permite analizar programas QVT-Relations. Tiene una dependencia al componente "QVTParser", puesto que éste proporciona el método necesario para analizar un programa QVT-Relations.

- **QVTParser**

  Este componente implementa el *parser* de QVT y además construye el modelo QVT-Relations asociado al programa que ha sido analizado. La dependencia al componente QVTRelations es necesaria para poder construir modelos QVT-Relations a partir de programas QVT-Relations analizados. Su dependencia al "MomentRegistry" es necesaria para poder obtener los metamodelos de los modelos candidatos de una transformación o relación de equivalencia especificada en un programa QVT-Relations. La dependencia del "OCLProjector" aparece debido a la utilización de OCL en los programas QVT-Relations. Por último, este componente utiliza el "MomentUIConsole" tanto para informar al usuario sobre errores en el proceso de parseado, como para indicarle cuando empieza y cuando acaba el parseado.

- **QVTProjector**

  Este componente permite generar el código Maude correspondiente a un modelo QVT-Relations

- **DSLParser**

  Este componente implementa el *parser* del DSL y además construye el modelo del operador complejo asociado a la definición textual del operador complejo parseado. Depende del componente Core para poder construir los modelos de operador complejo a partir de sus definiciones textuales analizadas. Su dependencia al "MomentRegistry" es necesaria para poder obtener los operadores importados en una definición textual de operador complejo.

- **DSLParserUI**

  Este componente proporciona un menú emergente que permite analizar definiciones textuales de operadores complejos. Tiene una dependencia al componente "DSLParser", puesto que éste proporciona el método necesario para analizar una definición textual de operador.

- **Launcher**

  Este componente permite lanzar la ejecución de transformaciones y operadores. Depende del componente "Core" para poder representar los modelos como términos algebraicos y ejecutarlos en Maude.

- **Core**

  Este es el componente principal y más importante porque implementa el *kernel* de MOMENT. El *kernel* está formado por un conjunto de expresiones algebraicas en Maude que permiten el funcionamiento de MOMENT. Depende de "SimpleOperatorProjector" y de "ComplexOperatorProjector" para poder generar el código Maude de modelos de operadores simples y complejos respectivamente. La dependencia a "M2Projector" y "M1Bridge" es necesaria para poder representar en Maude los metamodelos como especificaciones algebraicas y los modelos como términos, y obtener modelos en EMF a partir de términos como resultado de ejecuciones en Maude. Por último, utiliza el "MomentUIConsole" para informar al usuario de posibles errores que puedan aparecer.

- **TraceabilityMM**

  Este componente implementa el soporte a la trazabilidad en MOMENT. Para conocer más acerca de este componente se sugiere acudir a [Gom05]

- **SimpleOperatorProjector**

  Este componente permite generar código Maude a partir de un modelo de operador simple.

- **ComplexOperatorProjector**

  Este componente permite generar código Maude a partir de un modelo de operador complejo.

- **M2Projector**

  Este componente permite representar los metamodelos como representaciones algebraicas en Maude. Para conocer más acerca de este componente se sugiere acudir a [Ibo05].

- **M1Bridge**

  Este componente implementa el puente bidireccional de interoperabilidad a nivel M1 entre EMF y Maude. Por tanto los modelos pueden representarse como términos en Maude y un término en Maude puede representarse como modelo en EMF. Para conocer más acerca de este componente se sugiere acudir a [Ibo05].

- **OCLProjector**

  Este componente permite generar el código Maude correspondiente a una expresión OCL. Para conocer más acerca de este componente se sugiere acudir a [BoO06].

- **OCLEditor**

  Este componente implementa un editor gráfico para la definición y ejecución (en Maude) de expresiones OCL 2.0 sobre modelos. Necesita del "OCLProjector" para proyectar y poder ejecutar en Maude las expresiones OCL. Igualmente depende del componente "Core" para poder proyectar a Maude los modelos y metamodelos sobre los que se definen las expresiones OCL.

- **KentOCL**

  Este componente proporciona un *parser* de expresiones OCL 2.0. Depende del "MomentUIConsole" para informar al usuario de errores y *warnings* que puedan aparecer durante el proceso de parseado.

- **MOMENTEXTENSIONS**

  Este componente consiste en un conjunto de servicios añadidos que han demostrado ser útiles para el equipo de desarrollo de MOMENT, pero que sin embargo no presentarían una utilidad evidente para un usuario final.

- **MaudeDaemon**

  Este componente proporciona los mecanismos necesarios para arrancar Maude en MOMENT-Eclipse e interactuar con él. Para conocer más acerca de este componente se sugiere acudir a [Gom05].

- **MaudeDaemonSimpleGUI**

  Este componente proporciona un sencillo IDE para desarrollar programas en Maude. Para conocer más acerca de este componente se sugiere acudir a [Gom05].

VIII. CONCLUSIONES

En este trabajo se han implementado los mecanismos necesarios para integrar en la herramienta de gestión de modelos MOMENT una serie de soportes que amplían notablemente su funcionalidad y productividad de uso. Concretamente estos soportes han sido tres:

- Soporte para transformaciones y relaciones de equivalencia entre modelos.
- Soporte para manipulación de modelos a través de operadores definidos por el usuario.
- Soporte para registrar aquellos modelos que son utilizados en MOMENT para el funcionamiento de los dos soportes anteriores.

Antes del inicio de este proyecto, MOMENT era un prototipo inicial de herramienta de gestión de modelos cuyas características principales eran genericidad, corrección, eficiencia, potencia y trazabilidad. El principal objetivo de MOMENT era y es demostrar la falsedad de la poca productividad de los sistemas formales en la ingeniería del software. Así, en pocos meses y partiendo del prototipo inicial, se ha conseguido desarrollar un segundo prototipo de MOMENT mucho más potente en cuanto a funcionalidad y eficiencia que el inicial. Además, se han seguido respetando las características iniciales de MOMENT:

- Genericidad. Ya que como se ha mostrado a lo largo de este documento, en MOMENT pueden especificarse y ejecutarse transformaciones y operadores con modelos pertenecientes a diversos metamodelos.
- Eficiencia. Ya que se hace uso de Maude, un sistema de alto rendimiento, habiéndose ejecutado las transformaciones y las operaciones de los ejemplos descritos en pocos segundos.
- Potencia. Dado el álgebra genérica que proporciona MOMENT es posible construir operadores complejos que satisfagan nuestras necesidades en tiempo record. Por ejemplo, dado el caso de estudio de propagación de cambios, basta con añadir una simple invocación más para obtener los resultados deseados. Esto desde otra herramienta de transformación presumiblemente nos hubiera supuesto la modificación de numerosos líneas de un largo código.
- Trazabilidad. Dado el soporte para trazabilidad implementado en el primer prototipo de MOMENT, la trazabilidad en las transformaciones y operaciones se obtiene de manera implícita.

Esto demuestra la validez y la potencia que los métodos formales pueden aportar al campo de la ingeniería de modelos.

Respecto a la integración del lenguaje QVT-Relations en MOMENT, podemos decir que nuestra aproximación constituye un primer prototipo ejecutable de transformaciones especificadas a través de este lenguaje, proporcionando soporte para el *pattern matching* de reglas de transformación, trazabilidad y navegación mediante expresiones OCL. En definitiva, consideramos este prototipo como una buena alternativa en el campo de las transformaciones, ya que al estar basada en estándares aprovecha la reutilización de tecnología y se consigue reducir la curva de aprendizaje de la herramienta.

En cuanto al soporte actual, la implementación de QVT-Relations en MOMENT únicamente soporta transformaciones unidireccionales. Sin embargo, es posible soportar bidireccionalidad en transformaciones de carácter endógeno o en aquellas de

carácter exógeno, siempre que no haya pérdida de información, siendo además necesario que las manipulaciones que se realicen mediante expresiones OCL sean reversibles; por ejemplo la concatenación de *strings* no es reversible.

También se está trabajando en el soporte de ejecución de transformaciones QVT Relations en modo *checkonly*, que se encargan de comprobar que las relaciones establecidas por los modelos participantes en la transformación se sostienen en todas direcciones. Así pues, en un breve espacio de tiempo, MOMENT-QVT será uno de los primeros prototipos ejecutables que cumplan completamente el lenguaje Relations de la especificación QVT.

Por último, a través de los ejemplos mostrados en este documento, queda patente la productividad alcanzable en el desarrollo software aplicando ingeniería dirigida por modelos y particularmente la gestión de modelos. Igualmente, esta productividad depende de manera directa de las herramientas de gestión de modelos que se utilicen, por lo que hoy por hoy, MOMENT es una herramienta muy potente y versátil cuyo campo de aplicación no tiene límites, no sólo dentro de la ingeniería del software sino en otros campos como la bioinformática. Tan solo es cuestión de tiempo que este tipo de herramientas se popularicen y que la ingeniería dirigida por modelos se convierta en la metodología de desarrollo por excelencia.

1. TRABAJOS FUTUROS

MOMENT es un prototipo de herramienta de Gestión de Modelos muy joven todavía. Es por esto que son numerosas las tareas que han quedado pendientes de ser realizadas en el conjunto de la herramienta y particularmente en los tres soportes desarrollados.

Respecto al soporte para transformaciones y relaciones de equivalencia queda pendiente la implementación del análisis semántico de programas QVT-Relations. También sería interesante dar soporte a la especificación gráfica o visual de programas QVT-Relations. Relacionado con este soporte queda pendiente también el soporte para la ejecución de relaciones de equivalencia (transformaciones en modo *checkonly*). Además, pretendemos implementar también mecanismos que permitan la herencia de transformaciones, así como comprobar y preservar la semántica de los modelos resultantes de las transformaciones.

En cuanto al soporte de manipulación de modelos mediante la definición de operadores, el principal trabajo que deberá realizarse es la definición e implementación de un editor gráfico o visual que permita la definición de operadores por parte del usuario. Para ello, es conveniente esperar a versiones más estables de GMF (*Graphical Modelling Framework)* [GMF] que es una herramienta integrada en Eclipse y en EMF que permite la definición de editores visuales a partir de lenguajes específicos de dominio.

Por último, relacionado con el MOMENT *Registry*, el trabajo que deberá realizarse es ampliar la funcionalidad de las herramientas QVT *Parser* y DSL *Parser* para que registren automáticamente en el repositorio los artefactos que generan. Por otro lado, queda pendiente la creación de una interfaz gráfica de usuario que permita la ejecución de transformaciones o relaciones de equivalencia seleccionándolas de una lista de transformaciones o relaciones de equivalencia registradas en el MOMENT *Registry*.

IX. BIBLIOGRAFÍA

[AMMA] **Bézivin, J., Valduriez, P., Jouault, F.** *El sitio web de ATL-AMMA*. http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/

[ANTLR] **El generador de *parsers* ANTLR.** *El sitio web de ANTRL.*: http://www.antlr.org.

[ATL] **Bézivin, J., Valduriez, P., Jouault, F.** *El sitio web de ATL*. http://www.sciences.univ-nantes.fr/lina/atl

[Ber00] **Bernstein, P.A., Levy, A.Y., Pottinger, R.A.** *A Vision for Management of Complex Models*. Microsoft Research Technical Report MSR-TR-2000-53. Junio 2000. SIGMOD'00. Diciembre 2000.

[Ber03] **Bernstein, P.A.** *Applying Model Management to Classical Meta Data Problems*. CIDR, 2003.

[BoCR04] **Boronat, A., Carsí, J.Á., Ramos, I.** *An Algebraic Baseline for Automatic Transformations in MDA.* Software Evolution Through Transformations: Model-based vs. Implementation-level Solutions Workshop (SETra'04), Second International Conference on Graph Transformation (ICGT2004), Electronic Notes in Theoretical Computer Scienc.

[BoC05] **Boronat, A., Carsí, J.Á., Ramos, I.** *Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine*. IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, Reino Unido. 2005.

[BoCR06] **Boronat, A., Carsí, J.Á., Ramos, I.** *Algebraic Specification of a Model Transformation Engine.* LNCS. Fundamental Approaches to Software Engineering (FASE'06). ETAPS'06. Vienna (Austria). March 27-29. 2006.

[BoI05] **Boronat, A., Iborrra, J., Carsí, J.Á., Ramos, I., Gómez, A.** *Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos.* Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). September 2005. Granada, Spain.

[BoO06] **Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J.Á.** *An Algebraic Specification for Generic OCL Queries within the Eclipse Modeling Framework.* Proceedings of Second European Conference on Model Driven Architecture. Bilbao (Spain). Springer LNCS. July 10th-13th 2006

[BoP04] **Boronat, A., Pérez, J., Carsí, J.Á., Ramos, I.** *Two experiencies in software dynamics*. Journal of Universal Science Computer. Vol. 10 (issue 4), Abril 2004.

[EclOv03] **Object Technology International, Inc.** *Eclipse Platform Technical Overview*, Febrero 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[EMF] **Eclipse Tools.** *El sitio web de EMF.* http://www.eclipse.org/emf/

[Fond04] **Fondement, F., Silaghi, R.** *Defining Model Driven Engineering Processes*. WiSME@UML 2004. Octubre 2004.

[GarT03] **García, E.J., Troyano, J.A.** *Guía práctica de ANTLR*. ETS de Ingeniería Informática de la Universidad de Sevilla. 2003.

[GEF] **The Graphical Editing Framework.** *El sitio web de GEF.* www.eclipse.org/gef/

[GeL02] **Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.** *Transformation: The Missing Link of MDA.* First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002

[GMF] **The Graphical Modeling Framework.** *El sitio web de GMF.* www.eclipse.org/gmf/

[GoB06] **Gómez, A., Boronat, A., Hoyos, L., Carsí, J.Á., Ramos, I.** *Definición de operaciones complejas con un lenguaje específico de dominio en gestión de modelos*. JISBD 2006, Sitges, Spain.

[Gom05] **Llana, A.** *Soporte gráfico para trazabilidad en una herramienta de gestión de modelos*. Proyecto final de carrera. Universidad Politécnica de Valencia. Valencia, Spain, Septiembre 2005.

[Ho03] **Ho, E.** *Creating a text-based editor for Eclipse*. Junio 2003. http://devresource.hp.com/drc/technical_white_papers/eclipeditor/index.jsp

[Ibo05] **Iborra, J.** *Prototipo de integración de una herramienta de Gestión de Modelos*. Proyecto final de carrera. Universidad Politécnica de Valencia. Septiembre 2005.

[JoK06] **Jouault F, Kurtev I.** *On the ArchitecturalAlignment of ATL and QVT*. Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, chapter Model transformation (MT 2006)

[KoP06] **Kolovos, D.S., Paige, R.F., Polack, F.** *The Epsilon Object Language (EOL).* Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 2006.

[KoPP06] **Kolovos, D.S., Paige, R.F., Polack, F.** *Model Comparison: A Foundation for Model Composition and Model Transformation Testing.* First International Workshop on Global Integrated Model Management (G@MMA) 2006, co-located with ICSE'06, Shanghai, China, May 2006.

[Jéz03] **Jézéquel, J.M.** *"Model-driven engineering:Basic principles and challenges"*. FMCO'03. Netherlands, Noviembre 2003.

[Mar02] **Martí-Oliet, N., Meseguer, J.** *Rewriting logic: Roadmap and bibliography*. Theoretical computer Science, 2002.

[MaudeMan] **Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcote, C.** *Maude Manual (Version 2.1.1)*. Abril 2005. http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf

[MelR03] **Melnik, S., Rahm, E., Bernstein, P.A.** *Rondo: A Programming Platform for Generic Model Management*. SIGMOD 2003. Extended ver. in Web Semantics, vol. 1, Num. 1.

[MeVa05] **Mens, T., Gorp, P.V.** *A Taxonomy of Model Transformation*.International Workshop on Graph and Model Transformation (GraMoT), 2005.

[MOF] **Meta Object Facility.** *El sitio web de MOF.* http://www.omg.org/mof/

[MOFQVT] **Especificación estándar MOF QVT.** http://www.omg.org/docs/ptc/05-11-01.pdf

[RONDO] **Melnik, S., Rahm, E., Bernstein, P.A.** *RONDO: A Programming Platform for Generic Model Management.* Proc. ACM SIGMOD 2003.

[SeKo03] **Sendall, S., Kozaczynski, W.** *Model Transformation – the Heart and Soul of Model-Driven Software Development*. IEEE Software, Special Issue on Model Driven Software Development, pages 42--45, Sept/Oct 2003.

[VELOCITY] **Sitio web de Velocity.** http://jakarta.apache.org/velocity/

[VIATRA] **Sitio web VIATRA:** http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html

X. ANEXOS

ANEXO I.

```
moment-op:
      [ import-decl-list ]
      [ metamodels-decl ]
      operator-decl

import-decl-list:
      import-decl { import-decl }

import-decl:
      #include < identifier > | #include " filename "

metamodels-decl:
      metamodel identifier { , identifier } ;

operator-decl:
      operator identifier ( formal-parameters-list ) :
      < return-types-list > { [ statements-sequence ] }

formal-parameters-list:
      formal-parameter-decl { , formal-parameter-decl }

formal-parameter-decl:
      parameter-type identifier

return-types-list:
      parameter-type { , parameter-type }

statement-sequence:
      statement { statement }

statement:
      < output-actual-parameters-list  > =
            identifier ( input-actual-parameters-list ) ;

input-actual-parameters-list:
      input-actual-parameter { , input-actual-parameter }

input-actual-parameter:
      constant | identifier

output-actual-parameters-list:
      output-actual-parameter { , output-actual-parameter }

output-actual-parameter:
      identifier

parameter-type:
      String | Float | Rat | Int | Qid | Bool |
            TraceabilityMetamodel | Transformation | identifier
```

ANEXO II.

```
fmod PROPAGATECHANGES { MM2 :: TRIV ,  MM1 :: TRIV ,  TraceabilityMetamodel :: TRIV ,
Transformation :: TRIV } is

    *** Used operators imports
    pr CROSS{ MM1 , TraceabilityMetamodel } .
    pr RANGE{ TraceabilityMetamodel , MM1 , MM2 } .
    pr DIFF{ MM1 , TraceabilityMetamodel } .
    pr MODELGEN{ MM2 , TraceabilityMetamodel } .
    pr MERGE{ MM2 , TraceabilityMetamodel } .
    pr RESTRICTDOMAIN{ MM1 , TraceabilityMetamodel } .
    pr COMPOSE{ TraceabilityMetamodel } .
    pr MERGE{ TraceabilityMetamodel , TraceabilityMetamodel } .
    *** Tuple of return types
    pr TUPLE<2> { MM2 , TraceabilityMetamodel } .


    *** Input parameters
    var Uml : Set { MM1 } .
    var UmlMd : Set { MM1 } .
    var RdbMd : Set { MM2 } .
    var MapUml2RdbMd : Set { TraceabilityMetamodel } .
    var Uml2Rdbms : Transformation .


    *** Operator declaration
    op PropagateChanges : Set{MM1} Set{MM1} Set{MM2} Set{TraceabilityMetamodel}
Transformation
                        -> Tuple { MM2 , TraceabilityMetamodel } .
    eq PropagateChanges ( Uml , UmlMd , RdbMd , MapUml2RdbMd , Uml2Rdbms ) =

        ( *** 1st output parameter: Result
            p1(Merge(
                Range(
                    MapUml2RdbMd,
                    p1(Cross(Uml,UmlMd)),
                    RdbMd
                  ),
                p1(ModelGen1(
                    Uml2Rdbms;
                    ? p1(Diff(
                            UmlMd,
                            p1(Cross(Uml,UmlMd))
                        ))
                    ? MM((empty-set).Set{MM2}
                  ))
             ))

        , *** 2nd output parameter: MapUmlMd2Result
            p1(Merge(
                Compose(
                    RestrictDomain(
                        p1(Cross(Uml,UmlMd)),
                        MapUml2RdbMd),
                    p2(Merge(
                        Range(
                            MapUml2RdbMd,
                            p1(Cross(Uml,UmlMd)),
                            RdbMd),
                        p1(ModelGen1(    Uml2Rdbms;
                                         ? p1(Diff(   UmlMd,
                                                      p1(Cross(Uml,UmlMd))
                                           ))
                                         ? MM((empty-set).Set{MM2}
                        ))
                    ))
                  ),
                Compose (
                    p2(ModelGen1(
                        Uml2Rdbms;
                        ? p1(Diff(   UmlMd,
                                     p1(Cross(Uml,UmlMd))
                          ))
                        ? MM((empty-set).Set{MM2}
                      )),
                    p3(Merge(
                        Range(   MapUml2RdbMd,
                                 p1(Cross(Uml,UmlMd)),
                                 RdbMd
                               ),
                        p1(ModelGen1(    Uml2Rdbms;
                                         ? p1(Diff(   UmlMd,
                                                 p1(Cross (Uml,UmlMd))
```

```
                                        ))
                                      ? MM((empty-set).Set{MM2}
                              ))
                      ))
                  )
              ))
          ) .

endfm
```

ANEXO III.

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
  key Table (name, schema);
  key Column (name, owner); // owner:Table opposite column:Column
  key Key (name, owner); // key of class ‘Key’;
  // owner:Table opposite key:Key

  top relation PackageToSchema // map each package to a schema
  {
    pn: String;
    checkonly domain uml p:Package {name=pn};
    enforce domain rdbms s:Schema {name=pn};
  }

  top relation ClassToTable // map each persistent class to a table
  {
    cn, prefix: String;
    checkonly domain uml c:Class {
        namespace=p:Package {},
        kind='Persistent', name=cn
    };
    enforce domain rdbms t:Table {
        schema=s:Schema {},
        name=cn,
        column=cl:Column {name=cn+'_tid', type='NUMBER'},
        key=k:Key {name=cn+'_pk', column=cl}
    };
    when {
      PackageToSchema(p, s);
    }
    where {
      prefix = '';
      AttributeToColumn(c, t, prefix);
    }
  }
  relation AttributeToColumn
  {
    checkonly domain uml c:Class {};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
      PrimitiveAttributeToColumn(c, t, prefix);
      ComplexAttributeToColumn(c, t, prefix);
      SuperAttributeToColumn(c, t, prefix);
    }
  }
  relation PrimitiveAttributeToColumn
  {
    an, pn, cn, sqltype: String;
    checkonly domain uml c:Class {
        attribute=a:Attribute {
            name=an,type=p:PrimitiveDataType {name=pn}
        }
    };
    enforce domain rdbms t:Table {
        column=cl:Column {name=cn,type=sqltype}
    };
    primitive domain prefix:String;
```

```
    where {
      cn = if (prefix = '') then an else prefix+'_'+an endif;
      sqltype = PrimitiveTypeToSqlType(pn);
    }
  }
  relation ComplexAttributeToColumn
  {
    an, newPrefix: String;
    checkonly domain uml c:Class {
        attribute=a:Attribute {name=an,type=tc:Class {}}
    };
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
      newPrefix = prefix+'_'+an;
      AttributeToColumn(tc, t, newPrefix);
    }
  }
  relation SuperAttributeToColumn
  {
    checkonly domain uml c:Class {general=sc:Class {}};
    enforce domain rdbms t:Table {};
    primitive domain prefix:String;
    where {
      AttributeToColumn(sc, t, prefix);
    }
  }
  // map each association between persistent classes to a foreign key
  top relation AssocToFKey
  {
    srcTbl, destTbl: Table;
    pKey: Key;
    an, scn, dcn, fkn, fcn: String;
    checkonly domain uml a:Association {
        namespace=p:Package {},
        name=an,
        source=sc:Class {kind='Persistent',name=scn},
        destination=dc:Class {kind='Persistent',name=dcn}
    };
    enforce domain rdbms fk:ForeignKey {
        schema=s:Schema {},
        name=fkn,
        owner=srcTbl,
        column=fc:Column {name=fcn,type='NUMBER',owner=srcTbl},
        refersTo=pKey
    };
    when { /* when refers to pre-condition */
      PackageToSchema(p, s);
      ClassToTable(sc, srcTbl);
      ClassToTable(dc, destTbl);
      pKey = destTbl.key;
    }
    where {
      fkn=scn+'_'+an+'_'+dcn;
      fcn=fkn+'_tid';
    }
  }
  function PrimitiveTypeToSqlType(primitiveTpe:String):String
  {
    if (primitiveType='INTEGER') then 'NUMBER'
    else if (primitiveType='BOOLEAN')then 'BOOLEAN'
```

```
      else 'VARCHAR'
      endif
    endif;
  }
}
```

ANEXO IV.

```
*** regla 1: ClassToTable

var  ecoreEClass0 : ecore-EClass .

eq TransformElements(
  ClassToTable ,
  ? Set{ecoreEClass0} ? ecoreModel0,
  TargetModel,
  Tuple2 ) =
    Set{
      AddOID(
        (
          (New("Table", MM((empty-set).Set{rdbms}))).
          rdbmsNode :: schema <-- (
            (p1 ((ModelGenRule (
              PackageToSchema ,
              ? ( Set{
                      ((ecoreEClass0 :: ePackage ( ecoreModel0 )))
                     } -> flatten )
              ? ecoreModel0,
              TargetModel,
              Tuple2
              )
            )))
         -> select(oclIsTypeOf;? "Schema";(empty-set).Set{rdbms})
        )
        :: name <-- (((ecoreEClass0 :: name)))
        :: column <-- (Set {
          AddOID(
           (New("Column", MM((empty-set).Set{rdbms}))).
           rdbmsNode :: name <--
                              (((((ecoreEClass0 :: name) + "_tid"))))
                               :: type <-- ("NUMBER"))
        })
        :: key <-- (Set {
          AddOID(
            (New("Key", MM((empty-set).Set{rdbms}))).
            rdbmsNode :: name <-- ((((
                                    (ecoreEClass0 :: name) + "_pk"))))
            :: column <-- (Set {
              AddOID(
               (New("Column", MM((empty-set).Set{rdbms}))).
               rdbmsNode :: name <-- ((((
                                   (ecoreEClass0 :: name) + "_tid"))))
            :: type <-- ("NUMBER"))
        })
      )}))))
      ,
      AddOID(((New("Column", MM((empty-set).Set{rdbms}))).
        rdbmsNode :: name <-- ((((
                                   (ecoreEClass0 :: name) + "_tid"))))

        :: type <-- ("NUMBER")
      ))
      ,
      AddOID(((New("Key", MM((empty-set).Set{rdbms}))).
        rdbmsNode :: name <-- (((((ecoreEClass0 :: name) + "_pk"))))
```

```
        :: column <-- (Set {
          AddOID((New("Column", MM((empty-set).Set{rdbms})))).
            rdbmsNode :: name <-- ((((
                                    (ecoreEClass0 :: name) + "_tid"))))
            :: type <-- ("NUMBER"))
          })
         ))
       } .


ceq ModelGenRule (
  ClassToTable ,
  ? Set{ecoreEClass0 , ecoreM0 } ? ecoreModel0 ,
  TargetModel,
  Tuple2 ) =
           (ModelGenRule(ClassToTable ,
             ? Set{ecoreM0 } ? ecoreModel0,
             TargetModel,
             *** Tuple2
             ModelGenRule( AttributeToColumn ,
               ? (Set{ecoreEClass0} -> flatten) ? ecoreModel0 ?
                 ((( TransformElements (
                       ClassToTable,
                       ? Set{ecoreEClass0 } ? ecoreModel0,
                       TargetModel, Tuple2 )
                  -> select (
                             oclIsTypeOf ;
                             ? "Table" ;
                             (empty-set).Set{rdbms})
                 )-> asOrderedSet
               ) -> first
             )
             ?  "" ,
             TargetModel ,

             *** Tuple2
             (Set{
               getMagma(p1(Tuple2)),
               RefreshGeneratedElements(
                 TransformElements(
                   ClassToTable ,
                   ? Set{ecoreEClass0 } ? ecoreModel0,
                   TargetModel, Tuple2
                 ) ,
                 TargetModel,
                 p1(Tuple2),
                 empty-magma)
               },
               Set{
                 getMagma(p2(Tuple2)),
                 AddOID(
                  ( New (
                       "TraceabilityLink",
                       TraceabilityMetamodel)
                  ).
                  TraceabilityMetamodelNode :: domain <--
                    OrderedSet{ (ecoreEClass0 :: OID) }
                    :: range <-- (( TransformElements(
                      ClassToTable ,
                      ? Set{ecoreEClass0 } ? ecoreModel0 ,
                      TargetModel,
```

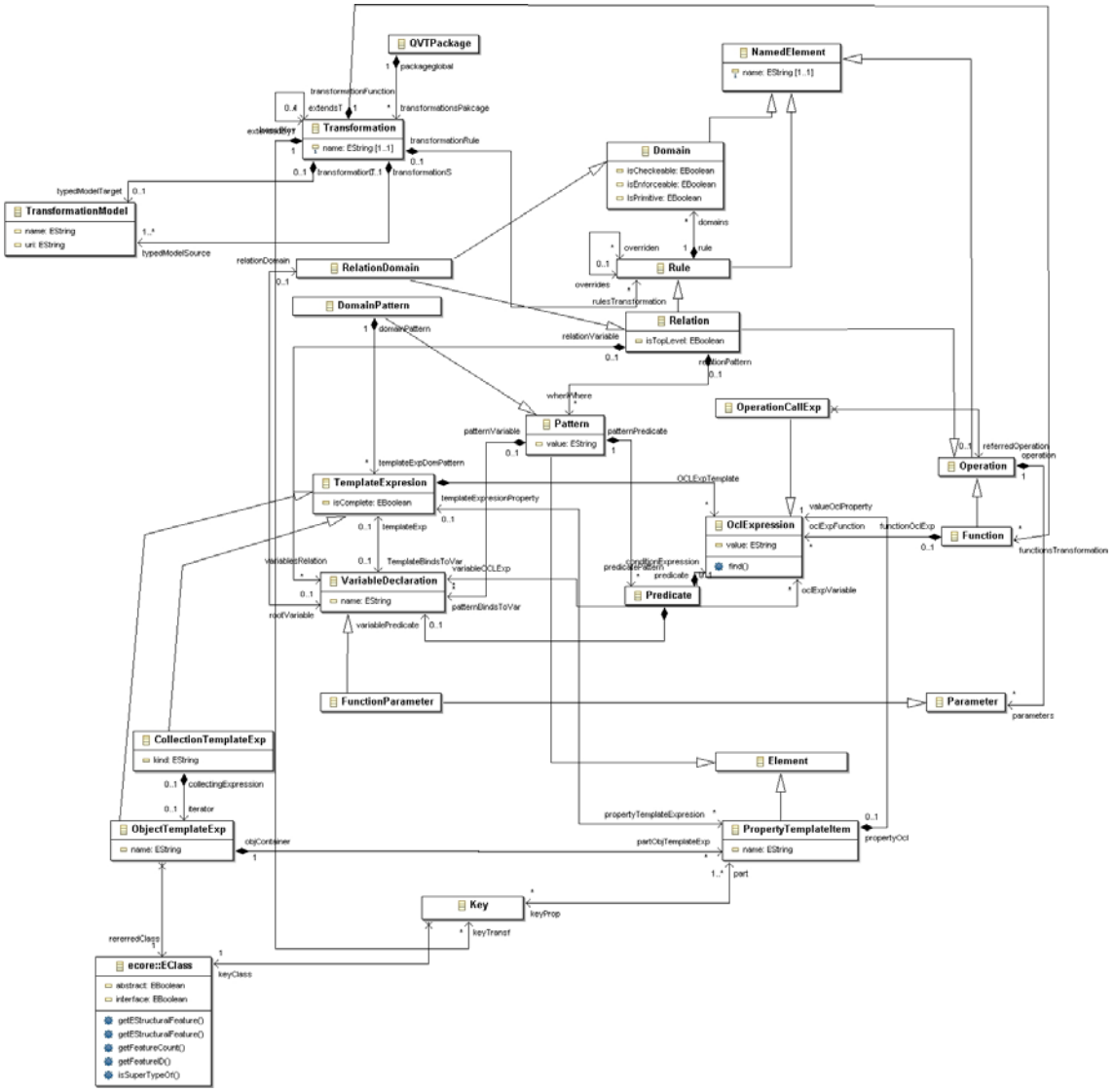
```
                      Tuple2  )
                    :: OID) -> asOrderedSet )
                    :: manipulationRule <-- "ClassToTable"
                  )
               }
            )
         ) *** where call

      ) *** close modelgenrule brackets
   ) *** close right hand side brackets

if( ( ecoreEClass0 :: oclIsTypeOf ( ? "EClass" ;
                                          (empty-set).Set{ecore} ) )  )
  [metadata "ClassToTable"] .


var attributetocolumnTABLE1 : rdbms-Table .
var attributetocolumnPREFIX1 : String .
```

ANEXO V.


QVTPackage
packageglobal
transformationFunction
extendsT
transformationsPakcage
Transformation
name: EString [1..1]
transformationRule
transformationT
transformationS
typedModelTarget
TransformationModel
name: EString
uri: EString
typedModelSource
relationDomain
RelationDomain
DomainPattern
domainPattern
NamedElement
name: EString [1..1]
Domain
isCheckeable: EBoolean
isEnforceable: EBoolean
isPrimitive: EBoolean
domains
overriden
rule
Rule
overrides
rulesTransformation
Relation
isTopLevel: EBoolean
relationVariable
relationPattern
where
Pattern
value: EString
patternVariable
patternPredicate
templateExpDomPattern
TemplateExpression
isComplete: EBoolean
templateExpression
templateExpresionProperty
OCLExpTemplate
OperationCallExp
referredOperation
operation
Operation
valueOclProperty
OclExpression
value: EString
find()
oclExpFunction
functionOclExp
Function
functionsTransformation
variablesRelation
TemplateBindsToVar
variableOCLExp
VariableDeclaration
name: EString
patternBindsToVar
rootVariable
variablePredicate
predicateExpression
predicate
Predicate
oclExpVariable
FunctionParameter
Parameter
parameters
CollectionTemplateExp
kind: EString
collectingExpression
iterator
Element
ObjectTemplateExp
name: EString
objContainer
propertyTemplateExpression
partObjTemplateExp
PropertyTemplateItem
name: EString
propertyOcl
part
Key
keyProp
keyTransf
rererredClass
keyClass
ecore::EClass
abstract: EBoolean
interface: EBoolean
getEStructuralFeature()
getEStructuralFeature()
getFeatureCount()
getFeatureID()
isSuperTypeOf()

ANEXO VI.

```
/** Analizador léxico de QVT **/
class QVTlexer extends Lexer ;

options {
      charVocabulary = '\3'..'\377';
      testLiterals = false;
      k=3;
}
tokens
{
      RELATION = "relation";
      TOP = "top";
      OVERRIDES = "overrides";
      DOMAIN = "domain";
      IMPLEMENTEDBY = "implementedby";
      PRIMITIVE = "primitive";
      CHECKONLY = "checkonly";
      ENFORCE = "enforce";
      WHEN = "when";
      WHERE = "where";
      QUERY = "query";
      IMPORT = "import";
      TRANSFORMATION = "transformation";
      KEY = "key";
      EXTENDS = "extends";
      FUNCTION = "function";
TARGET = "target";
}

/** Ignorar los espacios en blanco **/
BLANCO : (' ' | '\t' ) { $setType(Token.SKIP); };
/** Ignorar nueva línea */
NL :
( ("\r\n") => "\r\n"
  | '\r'
  | '\n'
  ) {newline();  $setType(Token.SKIP);} ;
/** Ignorar comentarios de una línea **/
COMENTARIO1: "//" (~('\r' | '\n'))* {$setType(Token.SKIP);};
/** Ignorar comentarios multi linea **/
COMENTARIO2 :
"/*"
( ('*' NL) => '*' NL // Nueva alternativa
| ('*' ~('/'|'\n'|'\r')) => '*' ~('/'|'\n'|'\r') // Modificada
| NL
| ~( '\n' | '\r' | '*' )
)*
"*/"
{ $setType(Token.SKIP); }
;
/** Paréntesis abierto **/
PARENT_AB : '(';
/** Paréntesis cerrado **/
PARENT_CE : ')';
/** Llave abierto **/
LLAVE_AB : '{';
/** Llave cerrada **/
LLAVE_CE : '}';
```

```
/** Coma **/
COMA: ',';
/** Punto y coma **/
PUNTO_COMA: ';';
/** Punto **/
PUNTO: '.';
/** Dos puntos **/
DOS_PUNTOS: ':';
/** Barra vertical **/
BARRA_VERT: '|';
/** Subguión **/
SUB_GUION: '_';
/** Igual **/
IGUAL: '=';
/** Comilla **/
COMILLA: "'";
/** Doble comilla **/
COMILLA2: '"';
/** Guión **/
GUION: "-";
/** Operador + **/
MAS: '+';
/** Barra **/
BARRA: "/";
/** Contra barra **/
CBARRA: "\\";
/** Almohadilla **/
ALMOHADILLA: "#";
/** Asterisco **/
ASTERISCO: "*";
/** Mayor **/
MAYOR: ">";
/** Menor **/
MENOR: "<";

protected LETRA: ('A'..'Z')|('a'..'z');
/** Definición de dígito **/
DIGITO: ('0'..'9');

IDENT options {testLiterals=true;}: (LETRA)(DIGITO)  | (LETRA
                                    |(SUB_GUION) | (GUION))*;
```

ANEXO VII.

```
/** Analizador sintáctico de QVTparser **/
class QVTparser extends Parser ;
options
{
buildAST = true; // Construir el AST
k=2;
exportVocab = QVTvocab;
}

tokens
{
      METAMODEL_DECL;
      METAMODEL_DECL_TARGET;
      VAR_DECL;
      PARAM_DECL;
      PRIMITIVE_DOMAIN;
      STATEMENT_WH;
      OCL;
      VALUE;
      STATEMENT;
      COLLECTION;
      QVTPACKAGE;
      DOMAIN_N1;
}

/** regla raíz **/
topLevel: (IMPORT filename PUNTO_COMA)* (transformation)*
               {##=#([QVTPACKAGE, "Package"],##);};

filename: identifier;

transformation: TRANSFORMATION^ identifier
                  PARENT_AB!
                  modelDecl (COMA! modelDecl)* (COMA! TARGET!
                                                  modelDeclTarget)?
                  PARENT_CE!
                   extends_rule)?
                   LLAVE_AB!
                   (keyDecl)* (relation | function)*
                   LLAVE_CE! ;

modelDecl:  modelId  DOS_PUNTOS! metaModelId
            { ## = #( #[METAMODEL_DECL, "MetamodelDecl"], ##);};

modelDeclTarget: modelId DOS_PUNTOS! MetaModelId
                 { ## = #( #[METAMODEL_DECL_TARGET,
                       "MetamodelDeclTarget"], ##);};

modelId: identifier;

metaModelId: (~(COMA | PARENT_CE))+ ;

extends_rule: EXTENDS^ identifier (COMA! identifier)*;

keyDecl: KEY^ classId
         LLAVE_AB! propertyId (COMA! propertyId)*
         LLAVE_CE! PUNTO_COMA!;
```

```
classId: identifier;

propertyId: identifier;

relation: (TOP)? RELATION^ identifier (OVERRIDES^ identifier)?
            LLAVE_AB! (varDeclaration)*
                      (domain | primitiveTypeDomain)+
                      (when)? (where)?
            LLAVE_CE!  ;

varDeclaration: identifier (COMA! identifier)*
                DOS_PUNTOS identifier PUNTO_COMA!
                { ## = #( #[VAR_DECL, "Var Declaration"],##);};


domain: ((checkEnforceQualifier)? DOMAIN modelID (identifier)?
         DOS_PUNTOS)=>domain1
         |
         ((checkEnforceQualifier)? DOMAIN modelID (identifier)?
            PUNTO_COMA)=>domain2;

domain1: (checkEnforceQualifier)?
               DOMAIN^ modelID (identifier)? DOS_PUNTOS identifier
               LLAVE_AB! value LLAVE_CE!
               (LLAVE_AB! (stat)? (COMA! stat)* LLAVE_CE!)?
               (IMPLEMENTEDBY^ stat)? PUNTO_COMA! ;

domain2: (checkEnforceQualifier)?
               DOMAIN! modelID (identifier)? PUNTO_COMA
               {## = #(#[DOMAIN_N1,"Domain"],##);};

primitiveTypeDomain: PRIMITIVE! DOMAIN! identifier
                     DOS_PUNTOS sumidero PUNTO_COMA!
                     { ## = #( #[PRIMITIVE_DOMAIN, "PrimitiveDomain"],
                                                                  ##);};

checkEnforceQualifier: CHECKONLY | ENFORCE;

when: WHEN^ LLAVE_AB! (statWH )* LLAVE_CE!;

where: WHERE^ LLAVE_AB! (statWH )*  LLAVE_CE!;

value: (stat)? (COMA stat)* { ## = #( #[VALUE, "Value"], ##);};

stat: ident_ocl IGUAL  parte_derecha { ## = #( #[STATEMENT,
                                          "Statement"], ##);};

stat2:  (stat) (COMA stat)*  { ## = #( #[COLLECTION, "Collection"],
                                                              ##);};

parte_derecha: (IDENT DOS_PUNTOS IDENT LLAVE_AB)
              =>  IDENT DOS_PUNTOS  IDENT LLAVE_AB (stat2 | ) LLAVE_CE
              | (IDENT PARENT_AB IDENT)
              => (IDENT PARENT_AB IDENT (COMA IDENT)* PARENT_CE)
              | (~(COMA|LLAVE_AB|DOS_PUNTOS|LLAVE_CE|PUNTO_COMA))+ ;

statWH: identifier parte_derechaWH { ## = #( #[STATEMENT_WH,
                                        "StatementWH"], ##);};

parte_derechaWH: ( PARENT_AB (identifier|blanco)?
                 (COMA (identifier|blanco))*  PARENT_CE PUNTO_COMA!
```

```
                        | (IGUAL oclIF) => IGUAL! oclIF PUNTO_COMA!
                          { ## = #( #[OCL, "Ocl"], ##);}
                        | (IGUAL sumideroWH PUNTO_COMA!)
                                    => IGUAL! sumideroWH PUNTO_COMA!
                                       { ## = #( #[OCL, "Ocl"], ##);}
                        | IGUAL! (~(IGUAL|PUNTO_COMA|LLAVE_CE))+
                          PUNTO_COMA! { ## = #( #[OCL, "Ocl"], ##);}
                         );

function: FUNCTION^ identifier
          PARENT_AB! (paramDeclaration)? (COMA! paramDeclaration)*
          PARENT_CE!  DOS_PUNTOS identifier body_function ;

body_function: LLAVE_AB!
               ( oclIF |(~("if" | PUNTO_COMA | LLAVE_CE))+) LLAVE_CE!
                { ## = #( #[OCL, "Ocl"], ##);};

oclIF: "if" oclST "then" oclST (oclELSE)? "endif";
oclELSE: "else" (oclIF | oclST);
oclST: (~(PUNTO_COMA | "if" | "then" | "else" | "endif"))+;

paramDeclaration: identifier (COMA! identifier)* DOS_PUNTOS identifier
                              {##=#(#[PARAM_DECL, "ParamDecl"],##);};

identifier: IDENT;
blanco : ((COMILLA COMILLA) | (COMILLA2 COMILLA2));

modelID: IDENT;

ident_ocl: IDENT | KEY;

sumidero: (options {greedy=false;}: .)+ ;

sumideroWH: (~(IGUAL |PUNTO_COMA |LLAVE_CE | PARENT_AB))+;
```

ANEXO VIII.

VIII.1. Instalación del MOMENT *Registry*

Para instalar Moment Engine siga los pasos siguientes:

1. Inicie Eclipse. Abra el menú “Help” y seleccione la opción “Software Update” -> “Find and Install”.
2. Elija la opción “Search for new features to install” y presione el botón “Next”.
3. En el cuadro de diálogo “Update sites to visit” debe añadir la Update Site de MOMENT si no lo ha hecho ya. Para ello, haga clic en el botón "Add Update Site" o "New Remote Site".
4. Introduzca “MOMENT Update Site” en el campo “Name” y “ftp://moment.dsic.upv.es/moment/eclipse” en el campo “URL”. Presione “OK”.
5. De vuelta al cuadro de diálogo, marque la opción “MOMENT Update Site” y presione en el botón “Next” o “Finish”.
6. En el siguiente cuadro de diálogo, despliegue la rama “Moment” y seleccione la opción “Moment Registry”. Seguidamente presione “Next”, acepte la licencia y finalice la instalación siguiendo las instrucciones que le presente Eclipse. Deberá aceptar también el reinicio de Eclipse tras finalizar la instalación.

VIII.2. Tareas comunes

VIII.2.1. Acceder al repositorio

Para acceder al repositorio MOMENT *Registry* abra el menú “Window” y seleccione la opción “Preferences…”. A continuación, en el menú de la izquierda despliegue la opción “MOMENT” y seleccione “MOMENT Registry”. Acto seguido, aparecerá la interfaz gráfica del MOMENT Registry a través de la cual es posible visualizar el contenido del respositorio e interactuar con él añadiendo o elimiando artefactos.

Como se puede apreciar en la Figura 31, en la interfaz gráfica del MOMENT *Registry*, aparecen cuatro pestañas en la parte superior “Metamodels”, “Transformations”, “Operators” y “Equivalences”, que se corresponden con los diferentes artefactos que pueden ser registrados en el respositorio (metamodelos, transformaciones, operadores y relaciones de equivalencia). Por omisión, aparece activa la pestaña “Metamodels”.

La tabla que aparece debajo de las cuatro pestañas muestra los artefactos registrados en el repositorio para la pestaña activa. En la Figura 31, se encuentra activa la pestaña de metamodelos y en la tabla pueden apreciarse tres metamodelos que se corresponden con los metamodelos registrados en el MOMENT *Registry.*

A la derecha de la tabla aparecen una serie de botones que ofrecen determinadads funcionalidades de acuerdo con la pestaña de artefactos activa. En las vistas de todos los artefactos aparecen dos botones “Add” y “Remove” que permiten añadir y eliminar artefactos al/del repositorio. En la Figura 31, para la vista de

metamodelos", aparece excepcionalmente otro botón llamado "Reload" que permite actualizar la versión de un metamodelo ya registrado.

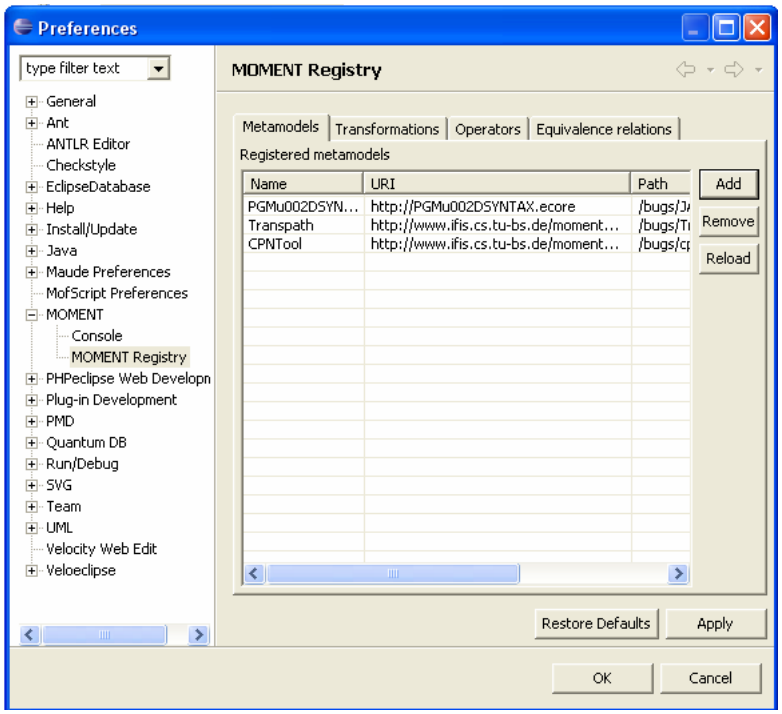


Figura 31. Vista de metamodelos del MOMENT *Registry*.

VIII.2.2. Añadir un artefacto

Los pasos a seguir para añadir un artefacto al repositorio son los mismos independientemente del artefacto que se desee añadir:

1. Acceder a la interfaz gráfica del MOMENT *Registry.*
2. Seleccionar la pestaña correspondiente al artefacto que se desee añadir.
3. Hacer clic sobre el botón "Add". Esta acción abrirá una ventana emergente que permitirá seleccionar uno o varios ficheros de los presentes en el *workspace* correspodiente a la instalación de Eclipse.
4. Seleccionar el artefacto deseado y hacer clic en el botón "Ok". Si el artefacto ya se encontraba registrado en repositorio o se encuentra registrado en el registro de EMF, se mostrará un mensaje de error.

A la hora de seleccionar un artefacto para añadirlo, es necesario tener en cuenta la extensión del fichero que se corresponde con dicho artefacto. Si el tipo de fichero seleccionado no se corresponde con el del artefacto, no se añadirá nada al respositorio. La correspondencia de tipos de ficheros con artefactos, puede verse en la siguiente tabla:

	Artefacto			
	Metamodelo	**Transformación**	**Operador**	**Relación de equivalencia**
Tipo de fichero	.ecore	.qvt	.mop	.qvt

Tabla 2. Asociación de tipos de fichero y tipos de artefactos.

VIII.2.3. Eliminar un artefacto

Los pasos a seguir para eliminar un artefacto del repositorio son los mismos independientemente del artefacto que se desee eliminar:

1. Acceder a la interfaz gráfica del MOMENT *Registry.*
2. Seleccionar la pestaña correspondiente al artefacto que se desee eliminar.
3. Seleccionar en la tabla el artefacto o artefactos que se deseen eliminar.
4. Hacer clic sobre el botón “Remove”.

VII.3. Funciones especiales: *reload*

La función *reload* tiene como objetivo actualizar la versión de un metamodelo registrado en el MOMENT *Registry*. Para ello, cuando se modifique un metamodelo que ya está registrado en el repositorio, basta con acceder a la interfaz gráfica del MOMENT Registry y seleccionar el metamodelo susceptible de la actualización. Una vez seleccionado, hacer clic en el botón “Reload” y el repositorio actualizará la versión del metamodelo registrada.