Implementación de un lenguaje de definición de operaciones complejas en Gestión de Modelos para la herramienta MOMENT

Abel Gómez Llana

Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Cno. de Vera, s/n. 46022 Valencia.

Dirigido por

Isidro Ramos y
José Á. Carsí

b

Índice general

Índice de figuras

Listados de código

Capítulo 1

Introducción

En la iniciativa MDA, un artefacto software es considerado como un modelo. Un modelo en este contexto es la especificación de la funcionalidad, estructura y/o comportamiento de un sistema o aplicación [18], de forma que permite el desarrollo de éstas de forma automática mediante técnicas de programación generativas [7]. El proceso de desarrollo software en esta filosofía se basa en el refinamiento de los artefactos software desde el espacio del problema (donde se capturan los requisitos de la aplicación), al espacio de la solución (donde se especifica el diseño y desarrollo del producto software final). Durante este proceso de refinamiento se aplican diversas operaciones a los modelos, como por ejemplo, transformaciones o integraciones de distintos modelos. Tradicionalmente, este tipo de tareas se han resuelto de manera ad-hoc para un contexto o metamodelo específico.

Para proporcionar un marco de trabajo genérico se propuso una nueva disciplina llamada Gestión de Modelos en [2]. Ésta considera los modelos y las correspondencias entre ellos como ciudadanos de primer orden, proporcionando un conjunto de operadores independientes de metamodelo y basados en teoría de conjuntos para tratar con ellos (Merge, Cross, Diff, ModelGen, etc.). Estos operadores proporcionan una solución reutilizable y componible para las tareas descritas anteriormente.

En el seno del grupo de investigación, y empleando la experiencia adquirida en la aplicación de la lógica ecuacional de pertenencia para resolver problemas en ingeniería del software [3], se ha desarrollado una herramienta (llamada MOMENT) que da soporte algebraico a estos operadores mediante un eficiente sistema de reescritura de términos —Maude [27]— desde un entorno de modelado industrial.

El entorno de modelado elegido para integrar el soporte algebraico de MO-

MENT es Eclipse Modeling Framework (EMF) [11]. La integración de estas tecnologías permite aprovechar 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, este trabajo muestra cómo se ha diseñado un Lenguaje Específico de Dominio (Domain Specific Language) para MOMENT. En la definición de este lenguaje 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. La especificación final del operador en Maude se obtiene de forma automática mediante técnicas de generación automática de código.

1.1. Objetivos

Este trabajo muestra cómo se ha resuelto en MOMENT la definición de operadores complejos mediante la aplicación a un caso de estudio. Los objetivos perseguidos son:

- Adaptar el fron-end de la herramienta MOMENT para dar soporte a la ejecución de diferentes operadores implementando los mecanismos de reconfiguración del kernel, permitiendo la carga bajo demanda de los módulos de los operadores simples sólo en el momento en que son requeridos.
- Modelar los conceptos del lenguaje de especificación de operadores complejos e implementar este modelos mediante el framework EMF de forma que se apliquen las técnicas de ingeniería dirigida por modelos.
- Implementar los mecanismos de proyección a código Maude a partir de un modelo correspondiente a la definición de un operador complejo.
- Proporcionar una interfaz de usuario amigable para la definición y ejecución de operadores complejos.

La consecución de estos objetivos resultará en un conjunto de nuevo plugins que extenderán la implementación de MOMENT mediante los mecanismos de extensibilidad estándares de Eclipse.

1.2. Descripción del documento

El documento se estructura de la siguiente manera: el capítulo 1 presenta la introducción al trabajo realizado, el capítulo 2 introduce las bases de la ingeniería dirigida por modelos y los estándares más empleados. El capítulo 3describe las tecnologías y herramientas que dan soporte a la ingeniería dirigida por modelos. A continuación, el capítulo 4 introduce el caso de estudio. Los capítulos 5 y 6 explican los lenguajes específicos de dominio y el lenguaje específico de dominio diseñado para MOMENT respectivamente. Por último, los capítulos 7 y 8 presentan algunos trabajos relacionados y las conclusiones del trabajo. Por último se presentan los anexos A, B y C.

Capítulo 2

Ingeniería Dirigida por Modelos.

La evolución de la tecnología en el campo de la Ingeniería del Software ha permitido el desarrollo de sistemas cada vez más complejos. Esto en gran medida ha sido posible gracias a la introducción de técnicas que han posibilitado el incremento del nivel de abstracción en la descripción de problemas y sus soluciones. En la década de los 80 se dio un gran paso en este sentido mediante la aparición de las herramientas CASE (Computer Aided Software Engineering —Ingeniería de Software Asistida por Ordenador—), cuyo objetivo era dotar de métodos para el desarrollo de software creando herramientas que les dieran soporte. Estas herramientas permitían a los desarrolladores expresar sus diseños mediante representaciones gráficas, como diagramas de estructura o máquinas de estados, elevando el nivel de abstracción de la especificación de los sistemas software. No obstante esta tecnología no tuvo la aceptación que cabía esperar. El motivo hay que buscarlo en las limitaciones de los procesos de traducción que trasladaban las representaciones gráficas de los sistemas (mediante lenguajes gráficos de propósito general) a una plataforma o tecnología específica.

Los avances en el desarrollo de lenguajes de programación durante las pasadas dos décadas han conseguido elevar el nivel de abstracción en el desarrollo de software, aliviando los impedimentos de los primeros esfuerzos en la tecnología CASE. Los lenguajes basados en el paradigma de la orientación a objetos, como Java, C++ o C#, han dotado de una mayor expresividad en la codificación de sistemas, siendo su uso común en la mayor parte de los desarrollos, en detrimento de lenguajes estructurados clásicos, como Fortran o C. No obstante, la modificación y mantenimiento de los sistemas desarrollados se ha convertido en una tarea que implica un esfuerzo excesivo y tedioso.

La Ingeniería Dirigida por Modelos (Model Driven Engineering, MDE) tiene como objetivo organizar los niveles de abstracción y las metodologías de desarrollo, todo ello promoviendo el uso de modelos como artefactos principales a ser construidos y mantenidos. Un modelo está constituido por un conjunto de elementos que proporcionan una descripción sintética y abstracta de un sistema, concreto o hipotético. El término MDE fue propuesto por primera vez por Stuart Kent [17], en lo que se define como una marco general para la especificación de los modelos y tareas de modelado necesarias para llevar a cabo un proyecto de desarrollo software desde principio a fin. Cualquier especificación puede ser expresada con modelos, y éstos pueden tener cualquier nivel de abstracción y expresar cualquier aspecto de un sistema. El proceso de desarrollo se convierte en un proceso de refinamiento y transformación entre modelos, de manera que el nivel de abstracción cada vez es menor, hasta que en un último paso se genera código para una plataforma específica. Un proceso MDE debe definir claramente la secuencia de modelos a desarrollar en cada nivel y describir cómo derivar a partir de un modelo un modelo de menor nivel de abstracción. El sistema a desarrollar es inicialmente descrito por un modelo que captura los requerimientos, independientemente de los detalles específicos de la plataforma o de cualquier decisión de implementación. Se trata de un modelo con el mayor nivel de abstracción posible, una descripción del problema a abordar.

La aplicación de las propuestas de MDE a las herramientas CASE solamente pasaba por un escollo: la ausencia de lenguajes de modelado y metodologías de desarrollo estándar que dieran soporte a los sistemas software en todo su ciclo de vida a través de estas herramientas. Además, la existencia de estándares permitiría una mayor interoperabilidad.

La Ingeniería Dirigida por Modelos es un campo en la Ingeniería del Software que, durante años, ha representado los artefactos software como modelos con el objetivo de incrementar la productividad, calidad, y reducir los gastos en el proceso de desarrollo de software. Recientemente, existe un interés creciente en este campo. Prueba de ello es la aproximación de Model Driven Architecture [20], apoyada por la OMG.

El Desarrollo Dirigido por Modelos ha evolucionado del campo de la Ingeniería Dirigida por Modelos. En él, no sólo las tareas de diseño y generación de código están involucradas, sino que también se incluyen las capacidades de trazabilidad, tareas de meta-modelado, intercambio y persistencia de modelos, etc. Para poder abordar estas tareas, las operaciones entre modelos, transformaciones, y consultas sobre ellos son problemas relevantes que deben ser resueltos. En el contexto de MDA se abordan desde el punto de vista de los estándares abiertos. En este caso, el estándar Meta Object Facility (MOF) [22], proporciona un mecanismo para definir metamodelos.

El estándar Query/Views/Transformations (QVT) [21] indica cómo proporcionar soporte tanto para transformaciones como para consultas. A diferencia de otros lenguajes nuevos, QVT se apoya en el ya existente lenguaje *Object Constraint Language* (OCL) para realizar las consultas sobre los artefactos software.

Dentro de la ingeniería dirigida por modelos se ha propuesto una nueva disciplina denominada Gestión de Modelos. Ésta considera los modelos y las correspondencias entre ellos como entidades de primer orden, proporcionando un conjunto de operadores independientes de metamodelo y basados en teoría de conjuntos para tratar con ellos (Merge, Cross, Diff, ModelGen, etc.). Estos operadores proporcionan una solución reutilizable y componible para las tareas descritas anteriormente. En el capítulo 2.2 se describirá en mayor detalle este campo.

2.1. Estándares abiertos en Ingeniería Dirigida por Modelos.

2.1.1. MDA.

Para dar respuesta a estos problemas en el contexto de MDE, el Object Management Group (OMG) [13] ha lanzado la iniciativa Model Driven Arquitecture (MDA) [20], como una aproximación a la especificación e interoperabilidad de sistemas basada en el uso de modelos formales. En MDA, los modelos independientes de la plataforma (platform-independent models, PIMs) son inicialmente expresados en un lenguaje de modelado independiente de la plataforma, como UML. El modelo independiente de la plataforma es traducido a un modelo específico para la plataforma considerada (platform-specific model, PSM), por ejemplo, la plataforma Java, usando reglas formales. Por último, y a partir del modelo específico para la plataforma, se genera el código del sistema en el lenguaje de programación objetivo (Java, C#,...). Además, se propone la automatización de las transformaciones entre modelos y de la generación de código, pudiendo centrar el proceso de desarrollo de software en las tareas de modelado.

MDA define un gran número de estándares de OMG:

- *UML (Unified Modelling Language)*, 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 orientado a

objetos.

- *CWM (Common Warehouse Metamodel)*, un vocabulario específico para el dominio de los sistemas relacionados con la minería o explotación de datos.
- *OCL (Object Constraint Language)*, un vocabulario que permite expresar consultas y restricciones sobre modelos. En una sección posterior nos centraremos en este lenguaje, en torno el cual gira el desarrollo de este proyecto.
- *QVT (Query/View/Transformation)*, un vocabulario que utiliza OCL para expresar transformaciones y relaciones de equivalencia sobre modelos.
- *XMI (XML Metadata Interchange)*, un vocabulario que permite el intercambio de modelos vía XML.
- *MOF (Meta Object Facility)*, es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo.

2.1.2. MOF.

Como hemos comentado, el grupo OMG ha propuesto un marco de trabajo en el ámbito de la ingeniería de modelos denominado MDA (Model Driven Architecture). Éste pretende establecerse como un estándar «de facto» en este ámbito. MDA es un proceso de desarrollo de software. Por lo tanto el objetivo es producir sistemas informáticos ejecutables.

MOF (Meta Object Facility) es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo. Mediante MOF 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 con la siguiente distribución:

- 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.
- El nivel inferior, denominado M0, es en el que se sitúan los datos, es decir las instancias del sistema bajo estudio.

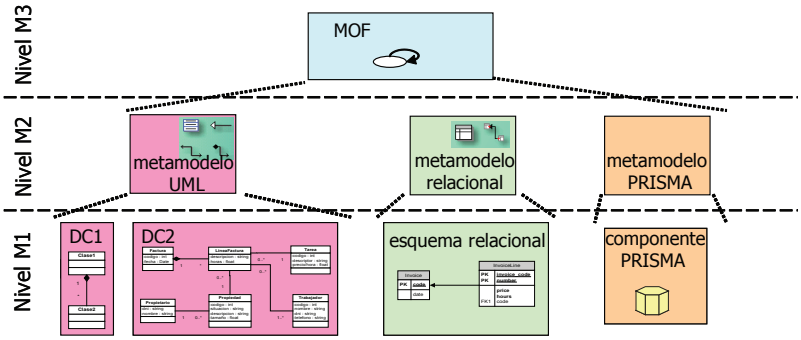

Figura 2.1: Arquitectura de niveles de MOF.

Esta estructura de cuatro capas (representada en la figura 2.1) 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 informalmente «relación instancia-de». Decimos que un modelo «x» es una instancia de un vocabulario «x+1», al que denominamos metamodelo. El modelo está en el nivel inferior, nivel de instancia; 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 instancia, vemos que también «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 instancias, como en el nivel instancia, 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 instancia 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 orientado 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.

2.2. 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.

Hasta ahora éste 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 de código y otras facilidades que automatizan gran parte del trabajo. En el contexto de la ingeniería de modelos ocurre todo lo contrario (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). Este disciplina, introducida por P. Bernstein en [2], pretende proporcionar una infraestructura específica y productiva para trabajar con los procesos de transformación y refinamiento de 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 [1], algunos ejemplos presentados en dicha fuente son:

1. 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.
2. 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.
3. El operador *Match*, que toma dos modelos y obtiene una correspondencia (mapping) entre ellos.
4. 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.
5. 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.

Nótese que en la aproximación original de Bernstein sobre los operadores de Gestión de Modelos se hace necesario proporcionar el modelo de correspondencias como argumento a la aplicación del operador. Esto no será así en los operadores que se emplearán posteriormente en la herramienta MOMENT, ya que en ella, las relaciones entre dos modelos se representan de forma implícita por medio de un morfismo de equivalencia que se define entre dos modelos desde un punto de vista más abstracto y reusable. Sin embargo, los mappings explícitos entre dos modelos son también útiles cuando no existe ninguna definición de este morfismo entre dos metamodelos.

2.2.1. Aproximaciones existentes.

Como primera aproximación a la gestión de modelos, encontramos RONDO [19]. 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 estan implementados de forma imperativa.

Capítulo 3

Soporte tecnológico a la Ingeniería dirigida por modelos

3.1. Eclipse

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 en 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.

La plataforma Eclipse es un framework para contruir IDEs. Se describe como «un entorno de desarrollo integrado para todo y nada en particular» [23]. Simplemente define la estructura básica de un IDE. Herramientas específicas extienden este framework, y se «enchufan» en él para definir un IDE particular colectivamente.

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 visivilidad 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. A continuación describiremos los principales *frameworks* del proyecto Eclipse directamente relacionados con sus capacidades de modelado *Eclipse Modeling Framework* (EMF) y *Graphical Modeling Framework* (GMF).

3.1.1. Eclipse Modeling Framework

Eclipse Modeling Framework es un framework de modelado para Eclipse. Este framework de modelado y generación de código permite definir un modelo de tres formas diferentes mediante Java anotado, XML Schema, o UML. Un modelo EMF es la representación de alto nivel común que une a las tres.

Un modelo EMF puede ser definido de cualquiera de estas tres maneras, siendo la potencia del framework y del generador la misma. A su vez, una vez definido un modelo EMF de cualquiera de estas formas, pueden obtenerse las otras de forma automática.

EMF es básica y simplemente un framework para describir un modelo y posteriormente poder generar otros elementos a partir de él. EMF es una tecnología que

se mueve en la dirección de MDA, pero de forma lenta, ya que intenta integrar las ventajas del modelado pero desde el punto de vista del programador.

Un modelo EMF es esencialmente el subconjunto de los diagramas de clases de UML y podría considerarse como una implementación del lenguaje MOF propuesto por el OMG. Esto es, un simple modelo de las clases o datos de la aplicación. Por esto, un amplio porcentaje de los beneficios del modelado pueden obtenerse en un entorno de desarrollo Java estándar. La correspondencia estre un modelo EMF y el código Java que lo implementa es sencilla y natural.

3.1.1.1. Definiendo un modelo EMF.

No obstante a lo comentado anteriormente, un modelo se describe utilizando conceptos a un mayor nivel de abstracción que las meras clases y métodos. Notaríamos por ejemplo, si observáramos la implementación de EMF, que los atributos corresponden a sendos métodos, para consultar y establecer sus valores. Igualmente, éstos, tienen la capacidad de notificar a los observadores (como una vista —View— de la interfaz, por ejemplo), o guardarse y recuperarse de un almacenamiento persistente. Las referencias son aún más potentes puesto que pueden ser bidireccionales, en cuyo caso la integridad referencial se mantiene. Las referencias pueden también persistirse entre diferentes recursos (documentos), donde entra en juego resolución delegada y la carga por demanda.

Para definir un modelo deberemos por tanto, disponer de una terminología común para describirlo. Y lo que es más importante, para implementar las herramientas de EMF y el generador, se requiere un modelo para la información.

El (Meta) modelo Ecore.

El modelo empleado para representar modelos en EMF se denomina Ecore. Ecore es también a su vez un modelo EMF, esto implica que Ecore es su propio metamodelo (o expresado en otras palabras, Ecore es un meta-metamodelo).

Gracias a Ecore, es posible definir los vocabularios locales de dominio que permiten el trabajo con modelos en distintos contextos.

Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de metamodelos. Para ello, proporciona elementos útiles para describir conceptos y las relaciones entre ellos. En la figura 3.1 se muestra un subconjuto simplificado este

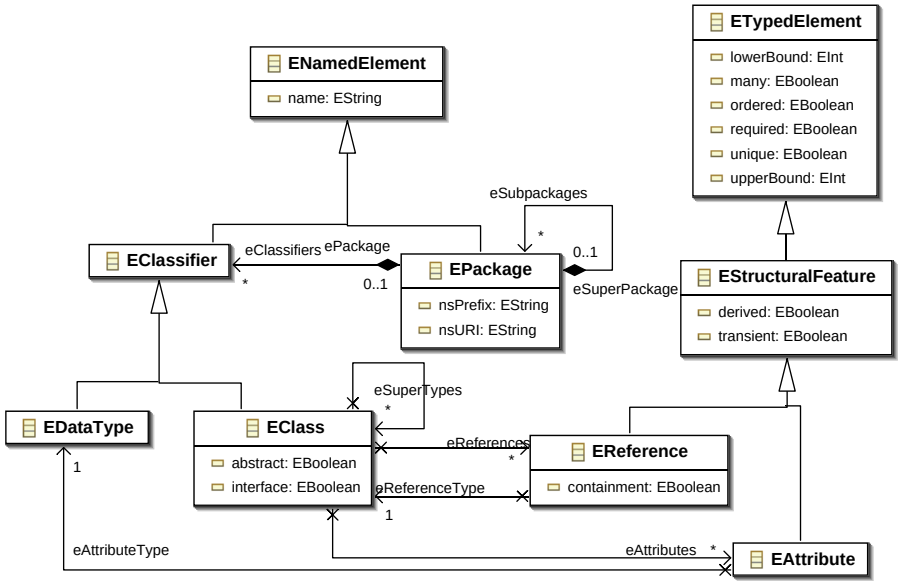

Figura 3.1: Subconjunto simplificado del modelo Ecore.

modelo.

El elemento más importante es *EClass*, que modela el concepto de clase, con una semántica similar al elemento Clase de UML. *EClass* es el mecanismo principal para describir conceptos mediante Ecore. Una *EClass* está compuesta por un conjunto de atributos y referencias, así como por un número de superclases (el símil con UML sigue siendo aplicable). A continuación se comentan el resto de los elementos aparecidos en el diagrama:

- *EClassifier*. Tipo abstracto que agrupa a todos los elementos que describen conceptos
- *EDataType* se utiliza para representar el tipo de un atributo. Un tipo de datos puede ser un tipo básico como int o float o un objeto, como por ejemplo java.util.Date.
- *EAttribute*. Tipo que permite definir los atributos de una clase. Éstos tienen nombre y tipo. Como especialización de ETypedElement, EAttribute hereda un conjunto de propiedades como cardinalidad (lowerBound, upperBound), si es un atributo requerido o no, si es derivado, etc.
- *EReference*. Permite modelar las relaciones entre clases. En concreto EReference permite modelar las relaciones de asociación, agregación y composición

que aparecen en UML. Al igual que EAttribute, es una especialización de *ETypedElement*, y hereda las mismas propiedades.

Además define la propiedad containment mediante la cual se modelan las agregaciones disjuntas (denominadas composiciones en UML).

- *EPackage* agrupa un conjunto de clases en forma de módulo, de forma similar a un paquete en UML. Sus atributos más importantes son el nombre, el prefijo y la URI. La URI es un identificador único gracias al cual el paquete puede ser identificado unívocamente.

Las similitudes de EMF con UML son evidentes, y es que EMF es un subconjunto de MOF, el cual a su vez está basado en los elementos del diagrama de clases de UML.

La cuestión de porqué no se ha utilizado UML como lenguaje de modelado es sencilla: Ecore es un subconjunto pequeño y simplificado de UML. UML soporta un modelado mucho más ambicioso que el soporte básico que se proporciona en EMF. Por ejemplo, UML permite modelar el comportamiento de una aplicación, a parte de su estructura de clases.

En el contexto de EMF, un metamodelo está constituido por las clases contenidas en un *EPackage*. Sólo se considera este caso simple; otros casos, como por ejemplo un metamodelo compuesto por más de un *EPackage*, no han sido tenidos en cuenta, sin que esto conlleve pérdida de genericidad o aplicabilidad.

Finalmente, para evitar confusiones cabe mencionar que en la documentación de EMF se utiliza la expresión modelo core para designar metamodelos. Dicha expresión hace referencia a modelos Ecore, es decir, modelos definidos utilizando el metamodelo Ecore. En cualquier caso el significado es el mismo: un metamodelo está constituido por un EPackage y un conjunto de *EClassifiers*

La creación de un modelo.

Ahora que ya disponemos de estos objetos Ecore para representar un modelo en memoria, el framework EMF puede leer de ellos para, entre otras cosas, generar código de implementación. La principal cuestión ahora es, ¿Cómo se crea un modelo Ecore?

Si se comienza mediante interfaces Java, el generador de EMF introspeccionará el código y construirá el modelo core. Si por el contrario se comienza a partir de un

esquema XML el modelo se construirá a partir de éste. En caso de que se comience con UML, existen 3 posibilidades:

1. Edición directa en Ecore. Se puede editar un modelo en Ecore directamente, por ejemplo, por medio del editor en árbol de ejemplo de EMF, o mediante Omondo.

2. Importar desde UML. El asistente de nuevo proyecto EMF proporciona esta opción para archivos de Racional Rose (archivos .mdl) únicamente. Esto se debe a que fue la herramienta con la que se inició la implementación del mismo EMF.

3. Exportar desde UML. Básicamente es la misma opción que la anterior, salvo que la conversión se invoca desde la herramienta UML en lugar del asistente de nuevo proyecto EMF.

Serialización en XMI.

Hemos comentado que un modelo «conceptual» puede ser representado físicamente de al menos tres formas diferentes: código Java, XML Schema, o un diagrama UML. Pero de hecho, aún existe una cuarta forma de persistir un modelo que es la que se utiliza como representación canónica: XMI (XML Metadata Interchange).

La razón de utilizar XMI se debe a que es un estándar para serializar metadatos, lo cual es Ecore. Además, salvo el código Java, el resto de formas son opcionales. Si se utilizara Java para representar un modelo se debería inspeccionar el conjunto de archivos Java cada vez que se deseara representarlo.

Por esto, XMI es la elección más razonable para la forma canónica de Ecore. Es de hecho la forma más cercana a la tercera forma de representarlo (UML). El problema radica en que cada herramienta de UML tiene su propio formato de persistencia. Un archivo XMI de Ecore es una serialización estándar XML de los metadatos que EMF utiliza.

3.1.1.2. Generación de código.

La principal ventaja de EMF, como la del modelado en general es el aumento en la productividad que resulta de la generación automática de código. Dado un modelo Ecore que hemos definido, es posible obtener una implementación con unos

pocos clicks. Todo lo que hay que hacer es crear un proyecto usando el Asistente para un nuevo proyecto EMF, que automáticamente lanza el generador y seleccionar Generar código del modelo desde un menú.

3.1.2. Graphical Modeling Framework

GMF surge de la necesidad por parte de los desarrolladores de tener que usar frecuentemente los frameworks EMF y GEF de Eclipse para el desarrollo de sus herramientas basadas en modelos. GEF (Graphical Editing Framework) es el framework de Eclipse que permite a los desarrolladores crear editores gráficos *ricos* a partir de un modelo de aplicación. GEF está formado por dos plugins. El plugin *org.eclipse.draw2d* proporciona las herramientas para renderizar y ordenador los elementos al mostrar los gráficos. El framework GEF emplea una arquitectura de «modelo—vista—controlador».

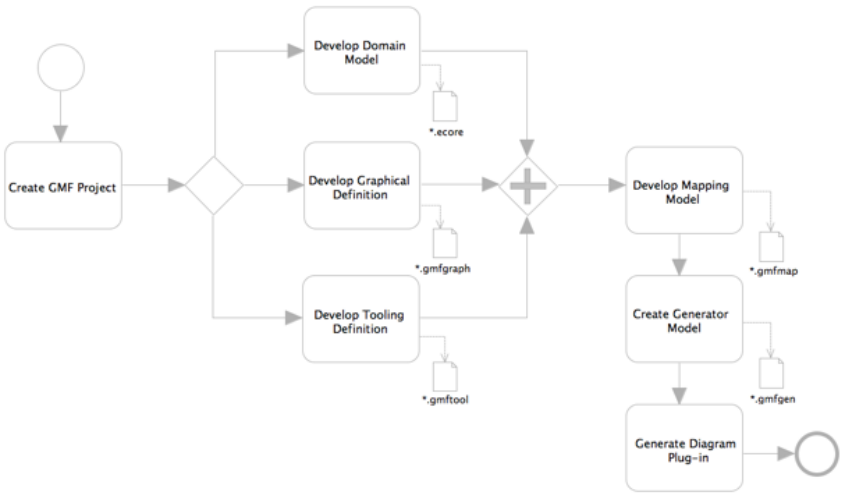

Figura 3.2: Proceso de creación de un editor gráfico mediante GMF.

La figura 3.2 ilustra los principales componentes y modelos usados durante un desarrollo basado en GMF. En la parte central de la figura se observa que un proyecto GMF parte de 3 modelos: *Domain Model*, *graphical Definition* y *Tooling Definition*. El primero de ellos, se corresponde con el modelo EMF para el que deseamos crear el nuevo editor gráfico. El segundo describe cuáles serán las primtivas gráficas que se dibujarán en el entorno de ejecución basado en GEF pero sin definicr ningún tipo de correspondencia con los elementos del modelo dominio para los que van a proporcionar capacidades de representación y edición. El tercer modelo permite

definir las herramientas que se mostrarán en la paleta de dibujo del editor así como otros elementos de la interfaz gráfica (menús, barras de herramientas, etc.).

Generalmente, una definición gráfica puede ser igualmente válida para diferentes dominios. Por ejemplo, en el diagrama de clases de UML encontramos diferentes elementos que son extremadamente parecidos en su apariencia y estructura. Un objetivo de GMF es que una definicón gráfica pueda ser reutilizada por distintos dominios. Esto se consigue mediante un modelo separado llamado *Mapping Model* que permite enlazar los elementos gráficos y las definiciones de herramienta con los elementos deseados del modelo dominio.

Una vez se han definido los enlaces apropiados, GMF proporciona un modelo generador para permitir afinar los últimos detalles de implementación para la fase de generación automática de código. La obtención del plugin de un editor basado en un modelo generador obtendrá un último modelo, llamado modelo *notacional*. El entorno de ejecución de GMF es el que enlaza este modelo notacional con el modelo dominio cuando el usuario está trabajando con un diagrama. A su vez, éste también proporciona las capacidades de persistencia y sincronización para ambos.

3.2. MOMENT. Un framework para la Gestión de Modelos.

MOMENT [4] es una herramienta que da soporte a los estándares propuestos por el OMG para dar soporte a transformaciones. La herramienta proporciona un soporte algebraico para las tareas de transformación y consulta de modelos mediante un eficiente sistema de reescritura de términos —Maude— y desde un entorno de modelado industrial —Eclipse Modeling Framework (EMF)—. Respecto a Maude, MOMENT aprovecha las capacidades de modularidad y parametrización de este sistema para proporcionar un entorno de transformación y consulta de modelos de forma genérica e independiente de metamodelo.

3.2.1. Espacios Tecnológicos y puentes.

Como se ha observado, MOMENT se desarrolla en dos ámbitos tecnológicos completamente diferenciados. A un lado, se encuentra la parte de interfaz del usuario y modelado, implementada como un nuevo framework para Eclipse; y al otro lado,

el motor de cálculo: el álgebra de MOMENT ejecutándose sobre Maude. A cada uno de estos ámbitos diferenciados se le denomina «espacio tecnológico».

El concepto de espacios tecnológicos fue introducido en [18] en la discusión sobre el enlace de tecnologías heterogéneas. 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 [16]. 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.

3.2.1.1. Puentes tecnológicos.

Cada espacio tecnológico tiene unas características que le hacen especialmente apropiado para resolver un tipo de problemas. Muchas veces sin embargo 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.

En MOMENT los operadores de gestión de modelos [2] han sido especificados algebráicamente utilizando el formalismo Maude como se ha comentado anteriormente. 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

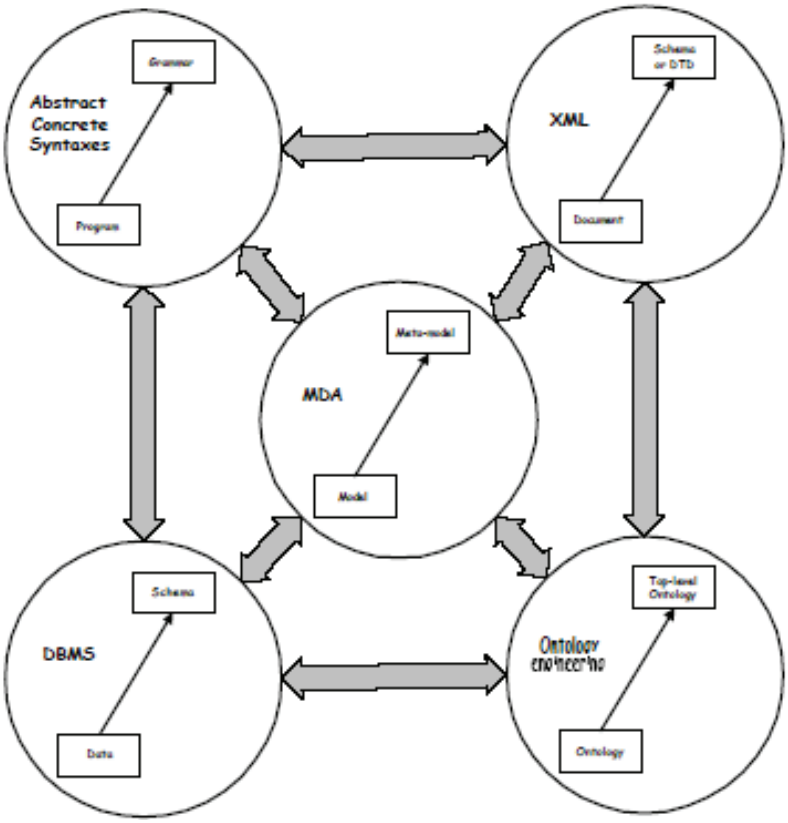

Figura 3.3: Cinco espacios tecnológicos y diversos puentes entre ellos. [Kurt02].

disponer de unos puentes tecnológicos entre ambos espacios tecnológicos. Este problema es resuelto en [15]. En dicho proyecto se resuelve la definición y construcción de estos puentes tecnológicos entre el ET de Maude y el ET de EMF.

Estos puentes creados 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.

3.2.2. Visión global del framework MOMENT.

En MOMENT los modelos se especifican como conjuntos de elementos de forma independiente del metamodelo, de manera que los operadores pueden acceder a los elementos sin conocer la representación de un modelo. La interfaz de MOMENT está integrada en EMF, de manera que el formalismo de especificaciones algebraicas queda totalmente transparente al usuario.

Para ilustrar el funcionamiento de MOMENT, se indica un pequeño ejemplo de integración de esquemas XML. Para ello se ha definido una parte del metamodelo del lenguaje de definición XML (XSD), mostrado en notación UML en la figura 3.4.

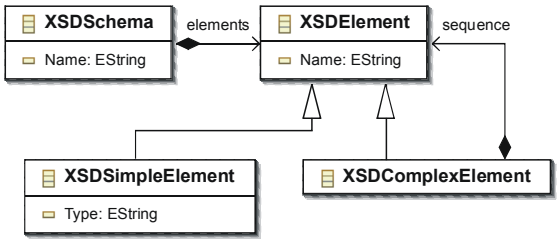

Figura 3.4: Parte del metamodelo XSD.

Utilizando el editor en forma de árbol que proporciona EMF, definimos los esquemas XML A y B en la 3.5. Se aplica el operador Merge a ambos, obteniendo el esquema XML integrado C y dos modelos de trazas (mapAC y mapBC) que enlazan los elementos de los modelos de entrada con los elementos del modelo de salida. La invocación del operador es la siguiente: $< C, mapAC, mapBC >= Merge(A, B)$.

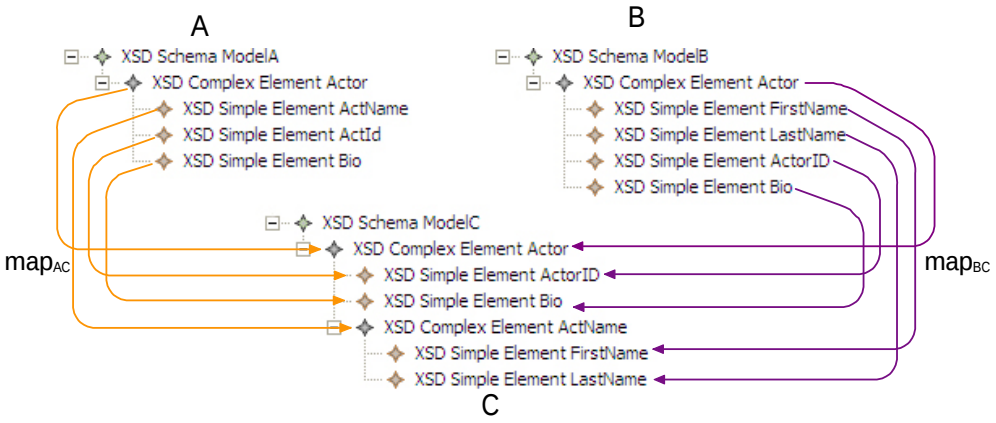

Figura 3.5: Aplicación del operador Merge.

Para poder realizar la integración de los esquemas XML, estos deben ser traducidos a términos en Maude, para que el operador Merge pueda aplicarse sobre ellos.

Entre las numerosas herramientas que dan soporte a la ingeniería de Modelos, MOMENT utilizará EMF como entorno de modelado para nuestra herramienta de Gestión de Modelos por su situación dentro del marco de la Ingeniería de Modelos. EMF permite tratar con gran variedad de artefactos software, como esquemas XML, modelos UML (definidos en entornos visuales de modelado como Rational Rose), esquemas relacionales (a través de Rational Rose), ontologías, entre otros. Además, EMF es utilizada por las principales herramientas de IBM, aportando una visión industrial a nuestro enfoque de Gestión de Modelos.

3.2.3. Marco conceptual para la representación de artefactos software en Maude.

Siguiendo un enfoque de Ingeniería de Modelos, para tratar con artefactos software utilizamos la terminología que define el estándar Meta-Object Facility de la iniciativa MDA. Este estándar, como se comentó en el apartado 2.1.2, presenta una arquitectura de cuatro capas de modelado que permite clasificar artefactos software con diferente propósito: M3 (metametamodelos), M2 (metamodelo), M1 (modelo), M0 (sistema real).

Una estrategia para trabajar con metamodelos consiste en definir una sintaxis

básica en el nivel M3, que pueda ser utilizada para definir artefactos software en niveles inferiores. En EMF, el metametamodelo se llama Ecore y proporciona una serie de primitivas de modelado: un subconjunto del diagrama de clases del metamodelo UML. Estas primitivas se utilizan para definir metamodelos en el nivel M2, constituyendo un paradigma de modelado. Como por ejemplo, el lenguaje de definición de esquemas XML (XML Schema Definition language — XSD).

Los elementos de un metamodelo son utilizados como tipos para definir los elementos que constituyen un modelo en el nivel M1. En el caso del metamodelo XSD, un modelo es un esquema XML específico. Los elementos de un modelo también se comportan como tipo para definir información en el nivel M0 de la arquitectura MOF. Por ejemplo, un esquema XML define los elementos que se pueden utilizar en un documento XML.

3.2.4. Proyecciones de artefactos software EMF sobre Maude.

Un espacio tecnológico se caracteriza por el soporte tecnológico que se proporciona a un determinado paradigma de modelado. Cada paradigma de modelado se organiza entorno a un metametamodelo común y persigue unos objetivos específicos.

El espacio tecnológico EMF se caracteriza por las facilidades que ofrece para representar una buena variedad de artefactos software como modelos y por su interoperabilidad con otras herramientas industriales de modelado. El espacio tecnológico Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas. Este ET también puede ser visto como un paradigma de modelado, considerando el lenguaje 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.

Una especificación algebraica constituye la visión como instancia de un determinado metamodelo, proporcionando la descripción sintáctica de las primitivas (llamadas constructores en el campo de las especificaciones algebraicas), necesarias para especificar un artefacto software en el nivel M1. Cuando la especificación algebraica es interpretada como álgebra, se obtiene la visión como tipo del metamodelo, donde los constructores se pueden utilizan para definir artefactos software en el nivel M1. Éstos son representados sintácticamente como términos, representando la información en forma de árbol.

Para manipular modelos EMF con los operadores algebraicos de MOMENT se han definido una serie de proyecciones entre ambos espacios tecnológicos tal y como muestra la figura 3.6. Estas proyecciones permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF.

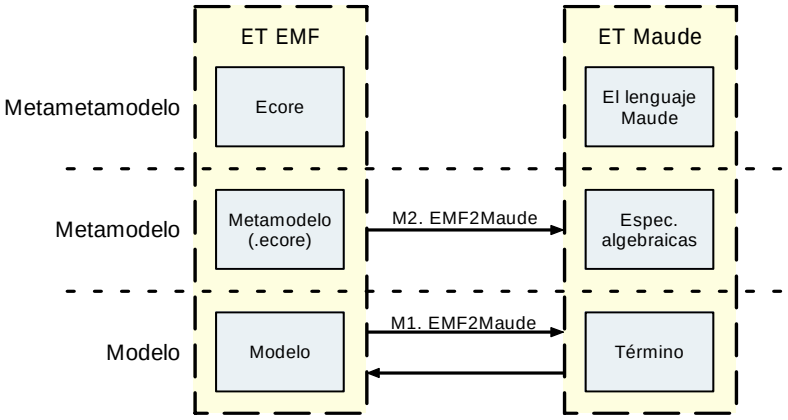

Figura 3.6: Enlaces entre el ET EMF y el ET Maude.

3.2.4.1. Interoperabilidad en el nivel M2

En el nivel de metamodelos se establece un enlace unidireccional que permite la proyección de un metamodelo EMF sobre el ET Maude, obteniendo una especificación algebraica. De este modo, un metamodelo se interpreta como un álgebra que proporciona los constructores necesarios para definir modelos y las operaciones necesarias para manipularlos, en el contexto de la Gestión de Modelos.

Este enlace es unidireccional pues los metamodelos se especifican mediante herramientas visuales de modelado a través de EMF (el nombre que hemos asignado a este enlace es M2-EMF2Maude). Este hecho permite hacer transparente el uso del formalismo Maude al usuario final del framework MOMENT.

MOMENT trabaja directamente sobre un álgebra de operadores genéricos de manipulación de modelos. Estos operadores pueden ser adaptados a un metamodelo específico haciendo uso de las capacidades de parametrización que ofrece Maude, basándose en el concepto formal de Pushout [10] de teoría de categorías.

En el diagrama del mecanismo de paso de parámetros (figura 3.7), TRIV constituye el parámetro formal del módulo parametrizado MOMENT-OP(X::TRIV). La

especificación sigXSD constituye el parámetro actual para el módulo parametrizado. sigXSD proporciona los constructores correspondientes a las primitivas de un metamodelo específico. Esta especificación algebraica está relacionada con el parámetro formal mediante la vista vXSD. Como ejemplo se ha utilizado el metamodelo XSD simplificado. En él, a partir de la metainformación que describe la clase XSDSimpleElement, que indica como definir un elemento simple en un esquema XML, se obtiene un constructor que permite definir un elemento simple en un esquema XML como un término del álgebra.

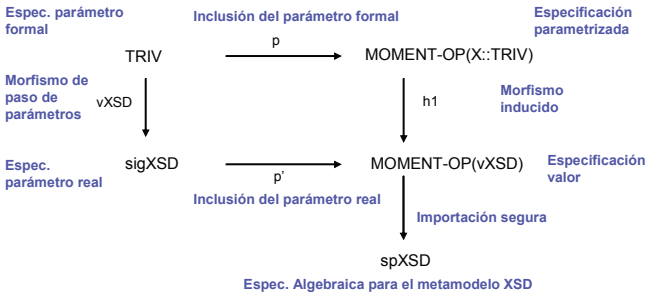

Figura 3.7: Diagrama del mecanismo de paso de parámetros en Maude.

El módulo sigXSD es importado por el módulo spXSD, en el que se extiende la presentación axiomática de los operadores genéricos adaptándolas a un metamodelo específico. Por ejemplo, el operador Merge, cuando se utiliza en el metamodelo XSD, permite integrar esquemas XML. Para definir este tipo de integración de forma más precisa, se pueden añadir relaciones de equivalencia que tengan en cuenta las primitivas del metamodelo XSD: elemento simple, elemento complejo, etc. Estas relaciones de equivalencia específicas al metamodelo XSD se añaden al módulo spXSD en forma de axiomas. La especificación algebraica resultante contituye el álgebra de un metamodelo. El módulo spXSD también es generado automáticamente por la herramienta MOMENT a partir de la especificación de los nuevos axiomas en interfaces visuales.

3.2.4.2. Interoperabilidad en el nivel M1

Existe otro tipo de enlace entre el ET EMF y el ET Maude en el nivel de modelos. Este enlace es bidireccional y consta de dos tipos de proyecciones:

- M1-EMF2Maude: Este mecanismo proyecta un modelo EMF, definido mediante un metamodelo EMF, sobre el ET Maude como un término. Para pro-

yectar un modelo sobre el ET Maude, la herramienta MOMENT consulta el correspondiente metamodelo y obtiene el constructor de la correspondiente especificación algebraica, que es necesario para especificar el término de forma automática.

- M1-Maude2EMF: Este mecanismo proporciona la proyección inversa a la anterior, obteniendo un modelo EMF a partir de un término Maude. En este paso, cuando la herramienta MOMENT lee un término que representa un modelo, determina las primitivas del metamodelo EMF que debe utilizar para construir el modelo EMF a partir de los símbolos de los constructores utilizados en el término. Con estas primitivas se construye el modelo dinámicamente y se persiste en formato XMI.

Los enlaces, que han sido descritos entre el ET EMF y el ET Maude, permiten la aplicación de operadores algebraicos sobre modelos definidos de forma gráfica mediante entornos industriales de modelado. Por ejemplo, supongamos que se desea realizar la integración de dos esquemas XML, cuyo metamodelo ha sido definido mediante Ecore. El proceso seguido es el siguiente:

1. Se obtiene la especificación algebraica spXSD a partir del metamodelo XSD.
2. Se proyectan los esquemas XML A y B a términos del álgebra, interpretada a partir de spXSD.
3. Se aplica el operador Merge a ambos términos y se obtiene el término resultante mediante el mecanismo de reducción de Maude. Como resultado de la operación de integración se obtiene el término C[1], que representa el esquema XML integrado.
4. Finalmente, el término C es proyectado al ET EMF como un modelo.

3.2.5. Presentación del álgebra de operadores de Gestión de Modelos de MOMENT.

Puesto que las correspondencias entre modelos se definen de forma implícita, es posible aplicar un operador genérico a dos modelos cualquiera. De la misma manera,

[1]El operador Merge también produce dos modelos de trazabilidad que relacionan los modelos de entrada A y B con el modelo de salida C, respectivamente. Ambos modelos se han obviado para simplificar el ejemplo.

permite obtener de forma automática el modelo de correspondencias entre ambos modelos.

La forma de funcionamiento, es la siguiente: en MOMENT, los operadores están definidos en un módulo parametrizado llamado MOMENT-OP. De esta forma, los operadores están definidos de forma genérica. Para aplicar estos operados a modelos específicos, este módulo debe ser instanciado pasando un metamodelo como parámetro actual. Esta tarea la realiza automáticamente la herramienta MOMENT.

A continuación, mostramos algunos ejemplos de operadores de Gestión de Modelos indicando sus entradas, salidas y semántica. Estos serán los operadores necesarios para resolver el ejemplo de propagación de cambios del capítulo 4:

3.2.5.1. Operadores comunes

1. *Cross* y *Merge*: Estos operadores corresponden a operaciones de conjuntos bien definidas: intersección y unión disjunta respectivamente. Ambos operadores reciben dos modelos (A y B) como entradas, y producen un tercer modelo (C). El operador *Cross* devuelve un modelo C que contiene elementos que participan en ambos modelos de entrada (A y B); mientras que el operador *Merge* devuelve un modelo C que contiene los elementos que pertenecen tanto al modelo de entrada A como a B, eliminando los elementos duplicados. Ambos operadores también devuelven a su vez sendos modelos de enlaces (map_{AC} y map_{BC}) que relacionan los elementos de cada modelo de entrada con los elementos del modelo resultante. Por ejemplo: $< C, map_{AC}, map_{BC} >= Cross(A, B)$.

2. *Diff.* Este operador realiza la diferencia entre dos modelos de entrada (A y B). La diferencia entre dos modelos (C) es el conjunto de elementos del modelo A que no corresponden a ningún elemento del modelo B. Este operador devuelve un único modelo de enlaces (map_{AC}). El modelo de mappings map_{BC} es innecesario puesto que se sabe a priori que será vacío.

3. *ModelGen. ModelGen* realiza la traslación de un modelo A, que conforma a un metamodelo origen MMA, a un metamodelo MMB destino, obteniendo el modelo B. Esta transformación implica tratar con dos metamodelos. Esto es perfectamente factible en nuestra aproximación, dada la modularidad y reusabilidad que las especificaciones algebraicas proporcionan. Este operador produce a su vez un modelo de correspondencias (map_{AB}) relacionando los elementos del modelo de entrada con los elementos del modelo generado. Por ejemplo: $< B, map_{AB} >= ModelGenMMA2MMB(A)$.

Como se observa en los ejemplos mencionados, toda aplicación de un operador sobre uno o varios modelos de entrada para producir los correspondientes modelos de salida, producen de forma automática los modelos de correspondencias que relacionan las entradas (modelos dominio) con las salidas (modelos rango). indicando que ambos lados de cada enlace representan el mismo elementos en modelos diferentes. Diversas aproximaciones, como por ejemplo RONDO, especifican operadores basados en estas correspondencias para tratar con modelos. Esto implica que las correspondencias entre ambos modelos, deben definirse de forma explícita para aplicarse sobre los modelos [19]. Sin embargo, en MOMENT, esta relación se establece de forma implícita mediante un morfismo de equivalencia definido a nivel de metamodelo (nivel M2), y no de modelo (nivel M1), de forma más abstracta y reusable.

3.2.5.2. Operadores de soporte a la navegación

Los operadores que proporcionan soporte para la navegación lo hacen a través de un modelo de trazabilidad con los siguiente elementos: dos modelos de entrada (A y B); un modelo de trazabilidad (map_{AB}) que relaciona los elementos de dos modelos de entrada y que ha sido generado automáticamente por un operador, o manualmente por un usuario; un modelo (A') que es un submodelo de A (esto es, que A' solo contiene elementos que pertenecen a A); y un modelo (B') que es un submodelo de B. Los operadores de trazabilidad considerados aquí son:

1. *Domain* y *Range*. Estos operadores proporciones la navegación hacia delante y hacia atrás a través de un modelo de trazabilidad, respectivamente. Ambos operadores obtienen un modelo como resultado que no es un modelo de trazabilidad.

   El operador *Domain* toma tres modelos como entrada: un modelo de trazabilidad (map_{AB}), un modelo dominio (A), y un modelo rango (B'). El operador navega los enlaces del modelo de trazabilidad y devuelve un submodelo de A (A'), como se muestra en la figura 3.8.a.

   El operador *Range* también recibe tres entradas: un modelo de trazabilidad (map_{AB}), un modelo dominio (A'), y un modelo rango (B). Este operador realiza la operación inversa a la anterior: navega el los enlaces de trazabilidad que tienen elementos de A' como elementos dominio y devuelve un submodelo del modelo rango B (B'), como se muestra en la figura 3.8.b.

2. *SelectMappingsByDomain* y *SelectMappingsByRange*. Estos operadores producen un modelo de trazabilidad como salida y permiten seleccionar parte de

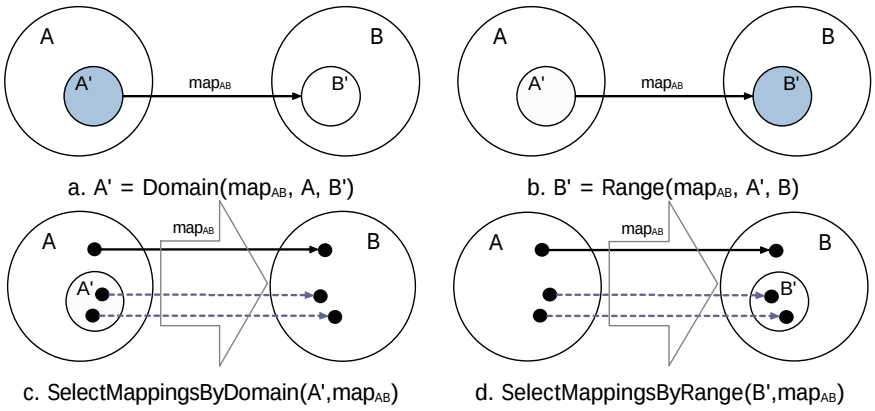

Figura 3.8: Operadores genéricos para navegación de las trazas.

un modelo de trazabilidad.

El operador *SelectMappingsByDomain* recibe dos modelos de entrada: un modelo dominio (A') y un modelo de trazabilidad (map_{AB}). El operador extrae los enlaces de trazabilidad del modelo map_{AB} que tienen elementos del modelo A' como elementos dominio y devuelve este submodelo. Los enlaces de trazabilidad que se añadir al modelo de trazabilidad de salida se muestran en la figura 3.8.c con una línea punteada.

El operador *SelectMappingsByRange* recibe dos modelos de entrada: un modelo rango (B') y un modelo de trazabilidad (map_{AB}). En este caso, el operador extrae los enlaces de trazabilidad del modelo map_{AB} que tienen elementos de B' como elementos rango, y devuelve este submodelo, como se muestra en la figura 3.8.d.

Capítulo 4

Caso de estudio

En este caso de estudio, utilizaremos un escenario de propagación de cambios que se asemeja al introducido en [19]. Lo ilustramos mediante un ejemplo específico basado en el modelo de Purchase Order utilizado en [5]. En este texto se presenta inicialmente un modelo UML simplificado (véase la Figura 6.6) para una aplicación que modele una orden de compra.

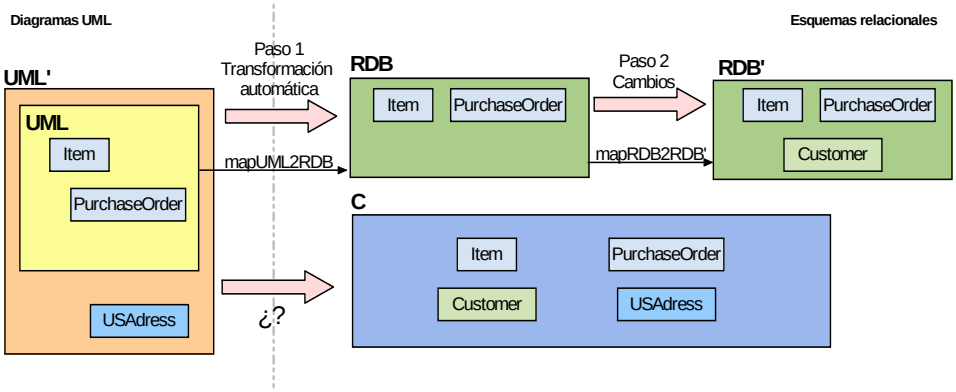

Figura 4.1: Ejemplo de propagación de cambios.

Para construir una nueva aplicación que almacene la información en una base de datos relacional, reusaremos la metainformación que describe el diagrama UML. Aplicando un mecanismo de transformación (paso 1), obtenemos la nueva base de datos 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 para proporcionar soporte a la trazabilidad ($map_{UML2RDB}$).

Tras obtener una base de datos relacional semáticamente equivalente al diagrama UML original, se continúa con el desarrollo del nuevo sistema. Esto puede implicar cambios en la aplicación y en la base de datos (paso 2), obteniendo el esquema relacional (RDB'). Estos cambios son trazados y almacenados por la herramienta que gestiona la manipulación del modelo, o directamente por el usuario ($map'_{RDB2RDB}$).

Una vez se ha desarrollado el nuevo sistema, pueden producirse cambios en los requisitos del sistema, requiriendo modificaciones. Podemos tomar estas modificaciones, como la modificación realizada en [5] sobre el modelo de la orden de compra (véase la figura 6.10). Es más sencillo modificar el esquema UML que modificar la base de datos RDB. En este punto, la aplicación del mecanismo de transformación aplicado en el paso 1 descartaría los cambios aplicados de RDB a RDB'.

Una solución a este ejemplo de propagación de cambios puede ser realizado usando operadores de gestión de modelos. En esta aproximación, puede construirse un operador complejo ($PropagateChanges$) que defina todas las operaciones a realizar para obtener el modelo final C.

Capítulo 5

Lenguajes específicos de dominio.

5.1. 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.

5.1.1. 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.

5.1.2. 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.

5.1.3. 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.

5.1.3.1. 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.

5.1.3.2. 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.

5.2. ¿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.

5.3. Ventajas e inconvenientes de usar 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.

5.3.1. 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.

5.3.2. 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.

Capítulo 6

DSL para la definición de operadores complejos en MOMENT

6.1. Introducción. Infraestructura tecnológica para la integración de un en MOMENT

Un metamodelo, tal y como se ha descrito en el capítulo 2, es un vocabulario que permite describir artefactos software. Esta definicón es por tanto aplicable o trasladable al dominio de los Lenguajes Específicos de Dominio. De esta manera, un DSL es un metamodelo, y cada uno de los programas que pueden ser definidos mediante dicho DSL son los correspondientes modelos (considerándose que un modelo es «instancia de» un metamodelo). Considerar los DSLs desde el punto de vista del metamodelado proporciona un mayor nivel de abstracción a la hora de tratar con los lenguajes, ya que nos permite independizar la definición de un determinado *programa* de la metáfora en la que es representada (textual, visual, etc). Además, si el *framework* de metamodelado proporciona capacidades de generación de código (como es el caso de EMF), esto facilita las tareas de implementación de los mecanismos de procesado y persistencia.

En MOMENT, siguiendo la filosofía MDE, todos los artefactos se representan mediante modelos en el espacio tecnológico de EMF. Mediante los puentes tecnológicos definidos entre el espacio tecnológico de EMF y el de Maude, se puede comunicar el *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, ejecutar una operación 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.

6.1.1. Creación de editores

6.1.1.1. 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 [14].

6.1.1.2. Editor gráfico

La creación de editores gráficos o visuales puede realizarse mediante los plug-ins GEF [8] Y GMF [9] 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.

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

6.1.2.1. 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.

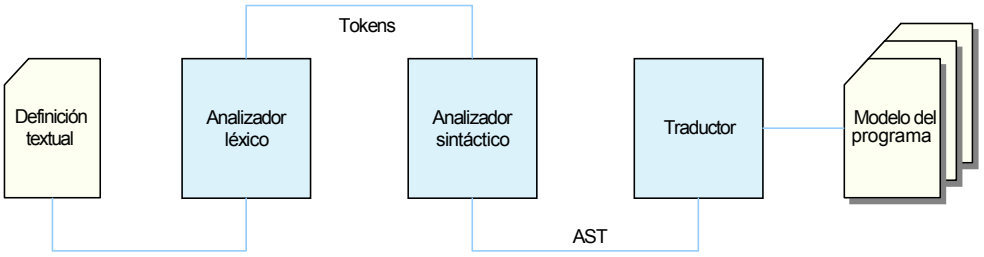

Figura 6.1: Proceso de compilación/traducción.

En la figura 6.1, 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 y generando el código final del proceso de compilación/traducción (código máquina, etc). En nuestro caso se generará un modelo EMF donde se reflejen propiedades y relaciones entre elementos que han de estar presentes en el modelo final. 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.

6.1.2.2. 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 [25] o ya en castellano, a la guía escrita en [6].

6.1.3. 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 [24].

6.2. Diseño del DSL de operaciones complejas.

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) [12] 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.

6.2.1. 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.

6.2.1.1. 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).

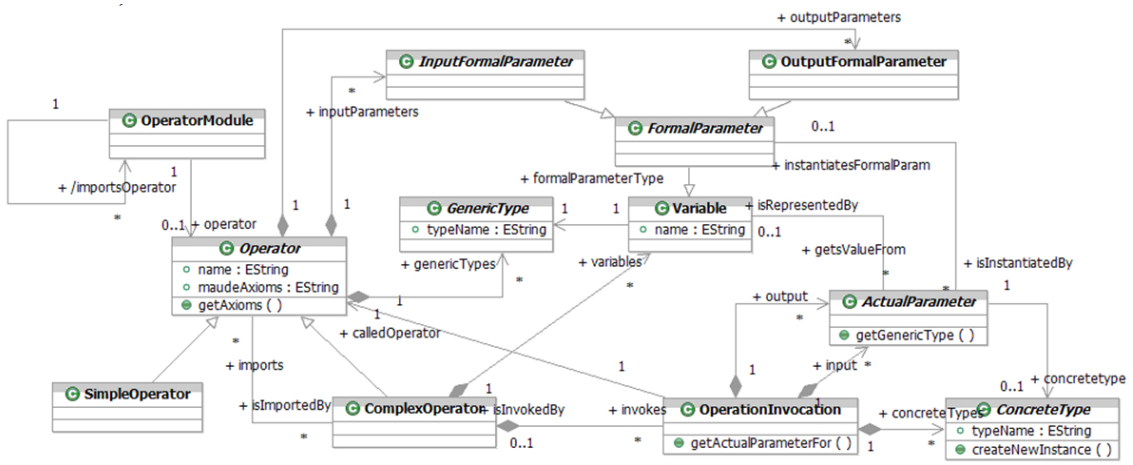

Figura 6.2: Modelo simplificado para la especificación de operadores complejos en MOMENT.

El diagrama de clases de la figura 6.2 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.

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 6.2, 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.

6.2.1.2. 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 6.3 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).

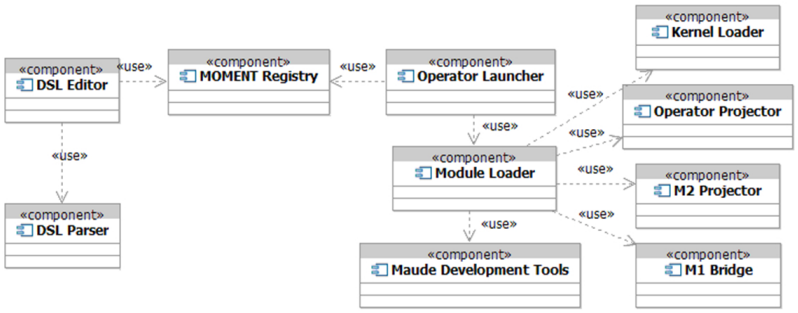

Figura 6.3: Componentes de MOMENT relacionados con la ejecución de operaciones de gestión de Modelos.

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*.

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 3.2.5. 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.

6.2.2. 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 A para dar soporte textual a la definición de operadores complejos.

Por ello, y tal como se observa en la figura 6.3, 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.

6.2.3. El DSL de MOMENT.

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

Toda definición de operador complejo sigue el siguiente esquema:

```
1    //Importación de operadores simples
2    #import ‘‘ruta_operador’’   //importación por ruta
3    #import <nombre_operador>   //importación por nombre
```

```
4
5     //Declaración de metamodelos
6     metamodel metamodelo1, metamodel2, ..., metamodelon;
7
8     //Declaración de variables
9     MMX modelo1, modelo2, ..., modelon;
10
11    //Declaración del operador
12    operator nombre_operador (parámetros_de_entrada) : <
         parametros_de_salida> {
13
14      //Sentencias
15      <argumentos_salida> = operador_simple_importado (
           argumentos_entrada);
16
17      return <modelos_resultado>;
18    }
```

Listado 6.1: Esquema de declaración textual de un operador complejo.

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 (línea 2) o por nombre del operador (línea 3). Para poder importar un operador por su nombre, es necesario que éste se encuentre registrado en el MOMENT Registry.

2. La declaración de metamodelos (línea 6) 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.

```
1     metamodel MM1, MM2;
```

```
2     operator nombre_operador (MM1 modelo1, MM2 modelo2,...)
         : <...> {...}
```

Listado 6.2: Declaración de un operador complejo.

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.

6.2.4. Aplicación de MOMENT al caso de estudio

6.2.4.1. Pasos a realizar

El problema mostrado en el caso de estudio 4 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:

«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:

```
1     // Declaración de importaciones de operadores simples
2     // Declaración de metamodelos
3
4
5
6     operator PropagateChanges(
7             MM1 Uml, MM1 UmlMd, MM2 RdbMd,
8             TraceabilityMetamodel MapUmlIni2RdbMd,
                 Transformation Uml2Rdbms)
```

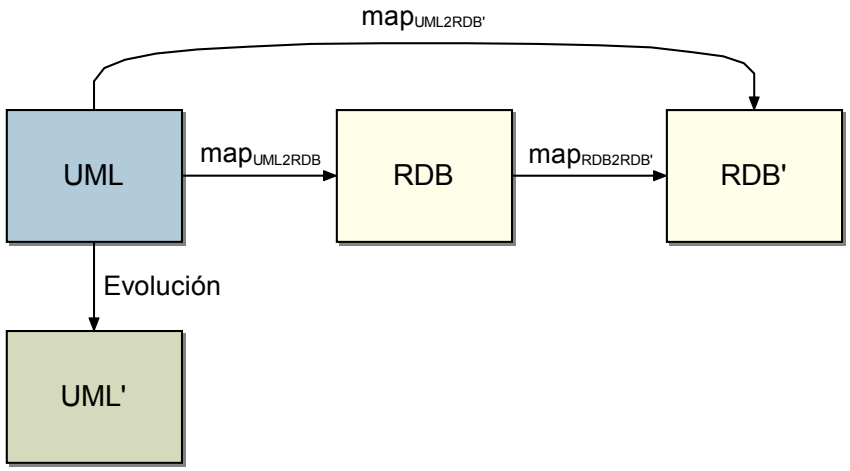

Figura 6.4: Esquematización del problema del caso de estudio.

```
9           : <MM2, TraceabilityMetamodel>
10      {
11
12              // Obtención del modelo resultado
13              <UmlUnmd, MapUml2UmlUnmd, MapUmlMd2UmlUnmd> = Cross(
                    Uml, UmlMd);
14              <RdbMdMd> = Range(mapUml2RdbMd , UmlUnmd, RdbMd);
15              <UmlNew, MapUmlMd2UmlNew, MapUmlUnmd2UmlNew> = Diff(
                    UmlMd, UmlUnmd);
16              <RdbNew, MapUmlNew2RdbNew> = ModelGen(Uml2Rdbms,
                    UmlNew);
17              <Result, MapRdbUnmd2Result, MapRdbNew2Result> =
                    Merge(RdbUnmd, RdbNew);
18
19              // Generación del modelo de trazabilidad
20              <MapUmlUnmd2RdbUnmd> = RestrictDomain(UmlUnmd,
                    MapUml2RdbMd);
21              <MapUmlUnmd2Result> = Compose(MapUmlUnmd2RdbUnmd,
                    MapRdbUnmd2Result);
22              <MapUmlNew2Result> = Compose(MapUmlNew2RdbNew,
                    MapRdbNew2Result);
23              <MapUmlMd2Result, Map1, Map2> = Merge(
                    MapUmlUnmd2Result, MapUmlNew2Result);
24
25              return <Result, MapUmlMd2Result>;
26      }
```

Listado 6.3: Operador complejo de propagación de cambios.

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 6.5. 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.

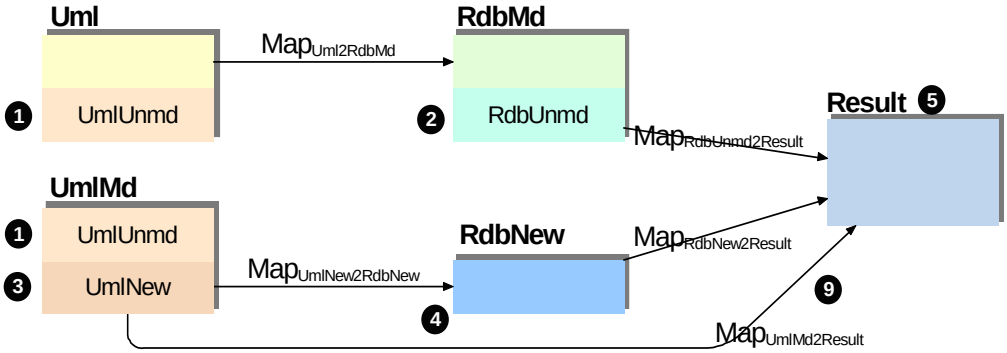

Figura 6.5: Solución al problema del caso de estudio.

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 A. 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 B.

6.2.4.2. Ejemplo de ejecución

La figura 6.6 muestra el modelo simplificado para una aplicación que gestione órdenes de compra introducido en [5].

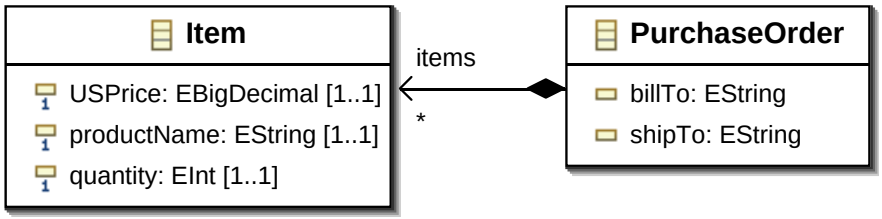

Figura 6.6: Modelo Purchase Order simplificado (UML)

Se desea obtener un modelo relacional equivalente. Para el ejemplo, tomaremos un metamodelo simplificado del metamodelo relacional. En la figura siguiente se muestra en notación UML.

Mediante la herramienta MOMENT, se ha obtenido tras la aplicación del operador *ModelGen* un esquema de base de datos relacional equivalente al modelo *Purchase Order*. De la misma manera, se ha obtenido un modelo de trazabilidad, también generado automáticamente por *ModelGen*.

A continuación la figura 6.8 muestra el modelo ecore *Purchase Order* simplificado, su correspondiente modelo equivalente generado automáticamente para el metamodelo relacional, y el modelo de trazabilidad que los relaciona .

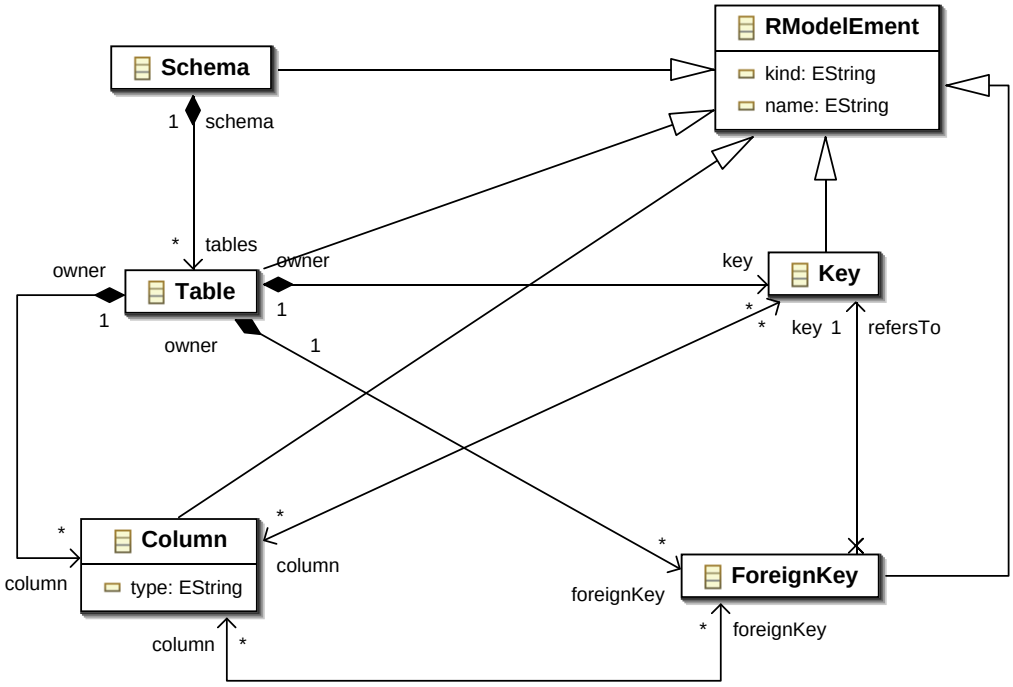

Figura 6.7: Metamodelo relacional simplificado

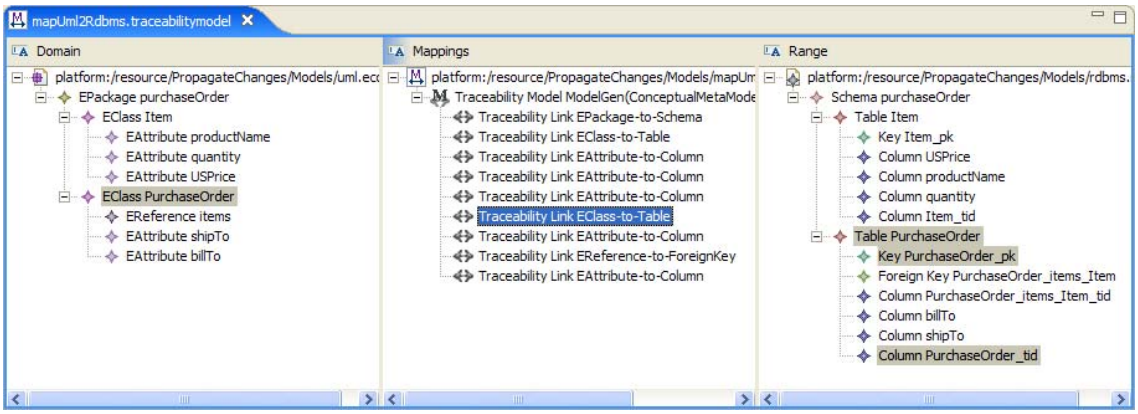

Figura 6.8: Vista de los modelos «UML», modelo de trazabilidad «mapUML2RDB», y «RDB» en el editor de trazabilidad de MOMENT.

Tras la generación del esquema relacional, se sigue desarrollando la base de datos. Para este ejemplo realizaremos dos tipos de cambio, en primer lugar, se va a modificar el tipo de la columna «productName» de «http://www.eclipse.org/emf/-2002/Ecore#//EString» a, por ejemplo, «VARCHAR(50)».

En segundo lugar, se van a realizar modificaciones en la base de datos, añadiendo una tabla llamada *Customer*. Esta tabla almacenará los datos de clientes. Cada orden de compra se relacionará con una entrada de esta tabla, por lo que se deberán crear también las correspondientes columnas y claves ajenas en la tabla *PurchaseOrder*. El nuevo modelo relacional queda tal como se muestra en la figura 6.9.

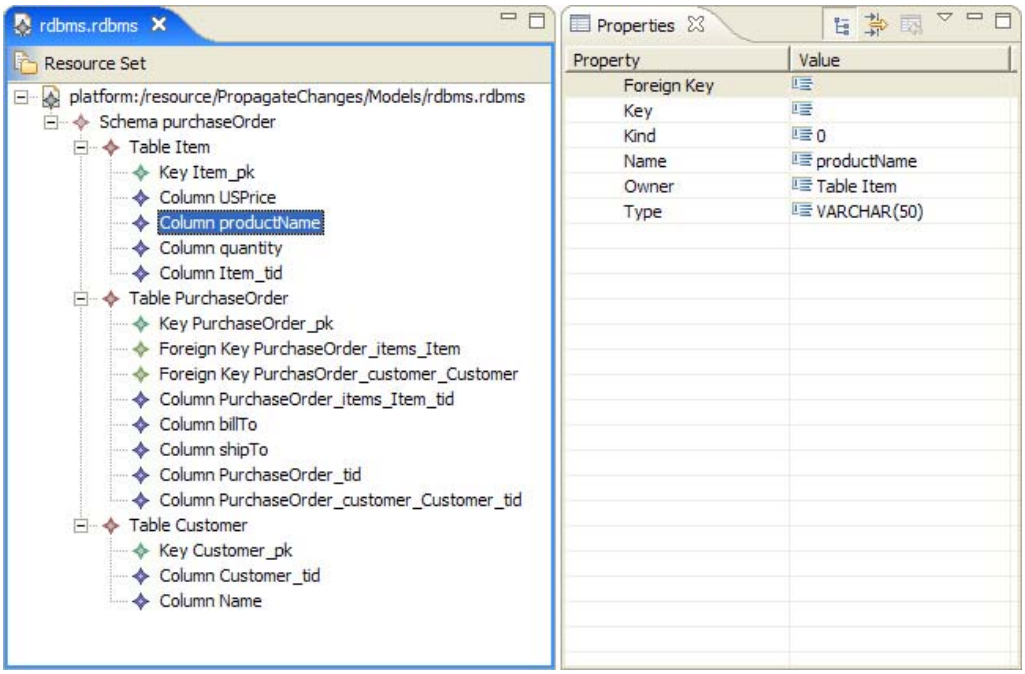

Figura 6.9: Modelo relacional Purchase Order modificado (RDB')

Tras generar esta primera base de datos, se modifican los requisitos del sistema, por lo que el diagrama UML para la orden de compra original se transforma en el modelo Purchase Order completo (figura 6.10), mostrado en [5]:

Para propagar estos cambios deberemos aplicar el operador de propagación de cambios enunciado en el apartado 6.2.4.1. Mediante el operador *Cross* se obtiene la parte común entre los modelos UML y UML' (la parte no modificada en UML'); y mediante el operador *Range*, se obtiene del modelo RDB' los elementos

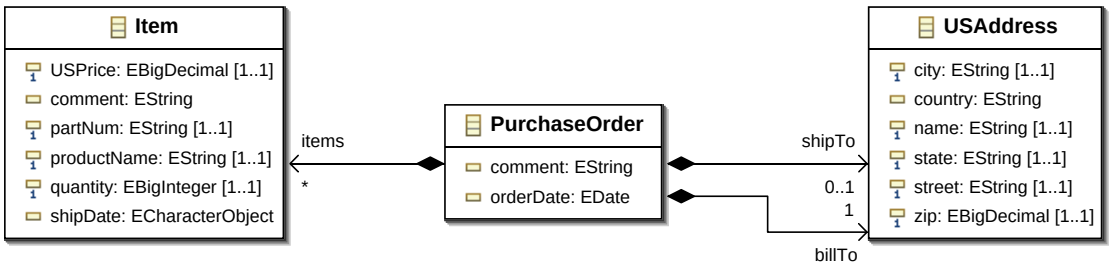

Figura 6.10: Modelo Purchase Order completo (UML')

que corresponden a esta parte no modificada (RDB").

Este paso nos servirá para, por ejemplo en el caso de estudio, eliminar las columnas shipTo y billTo, que en el nuevo modelo UML ya no aparecen como atributos, sino como referencias.

Las siguientes dos operaciones obtienen la parte nueva añadida por UML' sobre UML (newUML), y posteriormente, generan el correspondiente modelo relacional (newRDB). Por último solo nos resta componer RDB" y newRDB.

La ejecución produce el modelo de la figura 6.11.

Se puede observar que el tipo de productName se ha mantenido en «VARCHAR(50)», en lugar de haber sido machacado de nuevo por el valor «http://www.eclipse.org/emf/2002/Ecore#//EString». Se puede observar también, cómo se incluyen las tablas, claves y columnas generadas manualmente en la base de datos, así como el cambio de tipo de la columna productName. También se observa como se han aplicado correctamente los cambio de los atributos shipTo y billTo del diagrama UML a referencias del diagrama UML', con las respectivas columnas y claves ajenas.

6.3. 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 anteriormente. Éstos se desarrollarán detalladamente en los siguientes apartados.

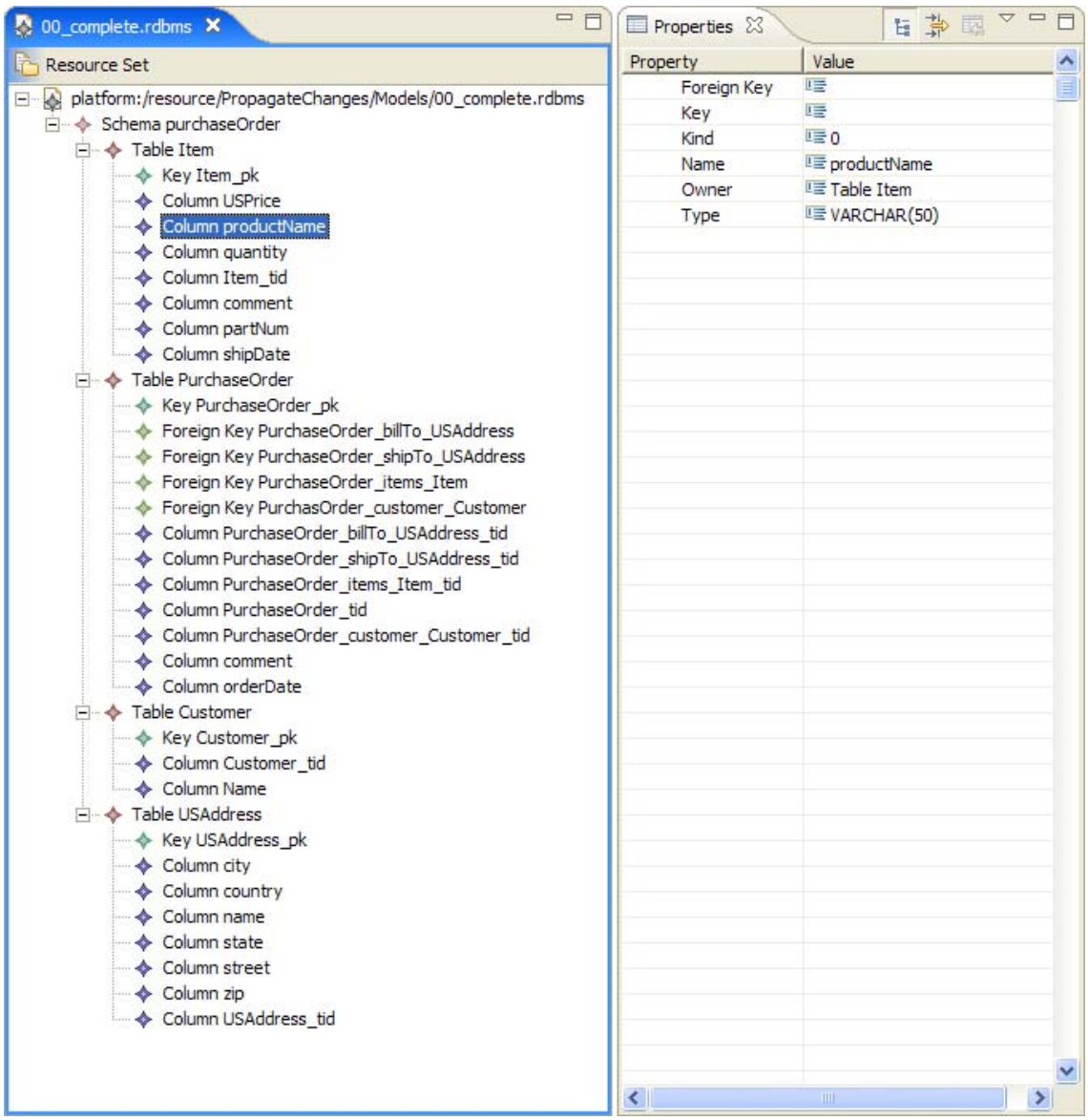

Figura 6.11: Modelo obtenido tras la aplicación del operador de propagación de cambios.

6.3.1. Editor textual para la definicón de operadores complejos.

6.3.1.1. 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.

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 A.

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».

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.

6.3.1.2. 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

3.1 de este documento.

6.3.1.3. 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.

MomentOpEditor

Esta es la clase principal del editor. Hereda de org.eclipse.ui.editors.text.TextEditor 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.

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:

```
1     public static String[] keyWords = {
2             "import", "#import", "metamodel", "operator" , "
                 string", "float",
3             "rat", "int", "qid", "bool", "traceabilitymetamodel"
                 , "transformation"
4             };
5
6     private static Color KEY_WORDS_COLOR =
7                     new Color(Display.getCurrent(), new RGB(128,
                         0, 64));
8
9     private static Color STRING_COLOR =
```

```
10                    new Color(Display.getCurrent(), new RGB(0, 0,
                          255));
11
12      private static Color COMMENT_COLOR =
13           new Color (Display.getCurrent(), new RGB (192,192,192))
                 ;
```

Listado 6.4: Coloreador de sintaxis. Definición de palabras clave.

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 comentarios y al resto de caracteres que conforman el código de definición de operadores complejos:

```
1      IToken keyWordsToken = new Token(new TextAttribute(
          KEY_WORDS_COLOR,null,SWT.BOLD));
2
3      IToken stringToken = new Token(new TextAttribute(
          STRING_COLOR));
4
5      IToken commentToken = new Token (new TextAttribute (
          COMMENT_COLOR));
```

Listado 6.5: Coloreador de sintaxis. Definición de los tokes. language

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:

```
1      setRules(new IRule[] {
2        // Add rule for processing instructions
3        keyWordsRule,
4        new SingleLineRule("\"", "\"", stringToken),
5        new SingleLineRule("'", "'", stringToken),
6        new EndOfLineRule ("//",commentToken),
7        new MultiLineRule("/*", "*/", commentToken,'\n',true),
8        new WhitespaceRule(new WhitespaceDetector())
9       });
```

Listado 6.6: Coloreador de sintaxis. Definición de Reglas.

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.

6.3.1.4. 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

6.3.2. Soporte gráfico para la invocación del proceso de compilación.

6.3.2.1. 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.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.

6.3.2.2. 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.

6.3.2.3. 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.

```
1     void run(IAction action {
2
3       DSLParserPlugin.getDefault().createModelFromTextSpec(file)
           ;
4
5     }
```

Listado 6.7: Método run(...) para la invocación del parser del DSL.

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

```
1     public void selectionChanged(IAction action, ISelection
         selection) {
2
3       file = null;
4
5       if(selection instanceof IStructuredSelection) {
6          IStructuredSelection sel = (IStructuredSelection)
              selection; Object selElem =
7          sel.getFirstElement(); if(selElem instanceof IFile)
8             file = (IFile)selElem;
9       }
10    }
```

Listado 6.8: Método selectionChanged(...).

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 construir el modelo, entonces en la llamada DSLParserPlugin.getDefault().createModelFromTextSpec(file) del método run(...) se pasa como argumento el fichero que se ha de analizar.

6.3.2.4. 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

6.3.3. Compilador/traductor de operadores complejos definidos textualmente.

6.3.3.1. 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 *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.

6.3.3.2. Diseño.

El DSL Parser ha sido diseñado como un caso especial de compilador/traductor siguiendo la analogía presentada en el apartado 6.1.2 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 A.

6.3.3.3. 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 A.

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:

```
1    // comentario
2
3    /* comentario */
```

Listado 6.9: Comentarios en el DSL de definición de operaciones complejas

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 A. En esta implementación 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 pro-

porciona ANTLR para los nodos del árbol. Recorriendo los elementos del árbol se evalúan ciertas reglas semánticas y se instancian los pertinentes objetos 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.

6.3.3.4. 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 mediante 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.

6.3.4. Modelo EMF del lenguaje específico de dominio

6.3.4.1. Descripción

En este apartado se describe de forma general el 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.engine.core y nos podremos referirnos a él simplemente como *MOMENT Engine Core*.

Funciones del plug-in

El plugin *MOMENT Engine Core* contiene la funcionalidad básica proporcionada por MOMENT. Las funciones más importantes son:

- Proyección de artefactos software de EMF a Maude.
- Recuperación de artefactos software de Maude a EMF.
- Carga del proceso Maude que se encargará de las ejecuciones en el espacio tecnológico Maude.
- Configuración y carga del kernel de MOMENT en el proceso Maude.

La estructura básica del plugin es generada de forma automática mediante los generadores de código de EMF. La figura 6.12 muestra el listado de clases del modelo EMF del plugin *MOMENT Engine Core*.

Las modificaciones que se han hecho sobre este plugin son las siguientes:

- Extender el modelo EMF del plugin para dar soporte a las clases del DSL.

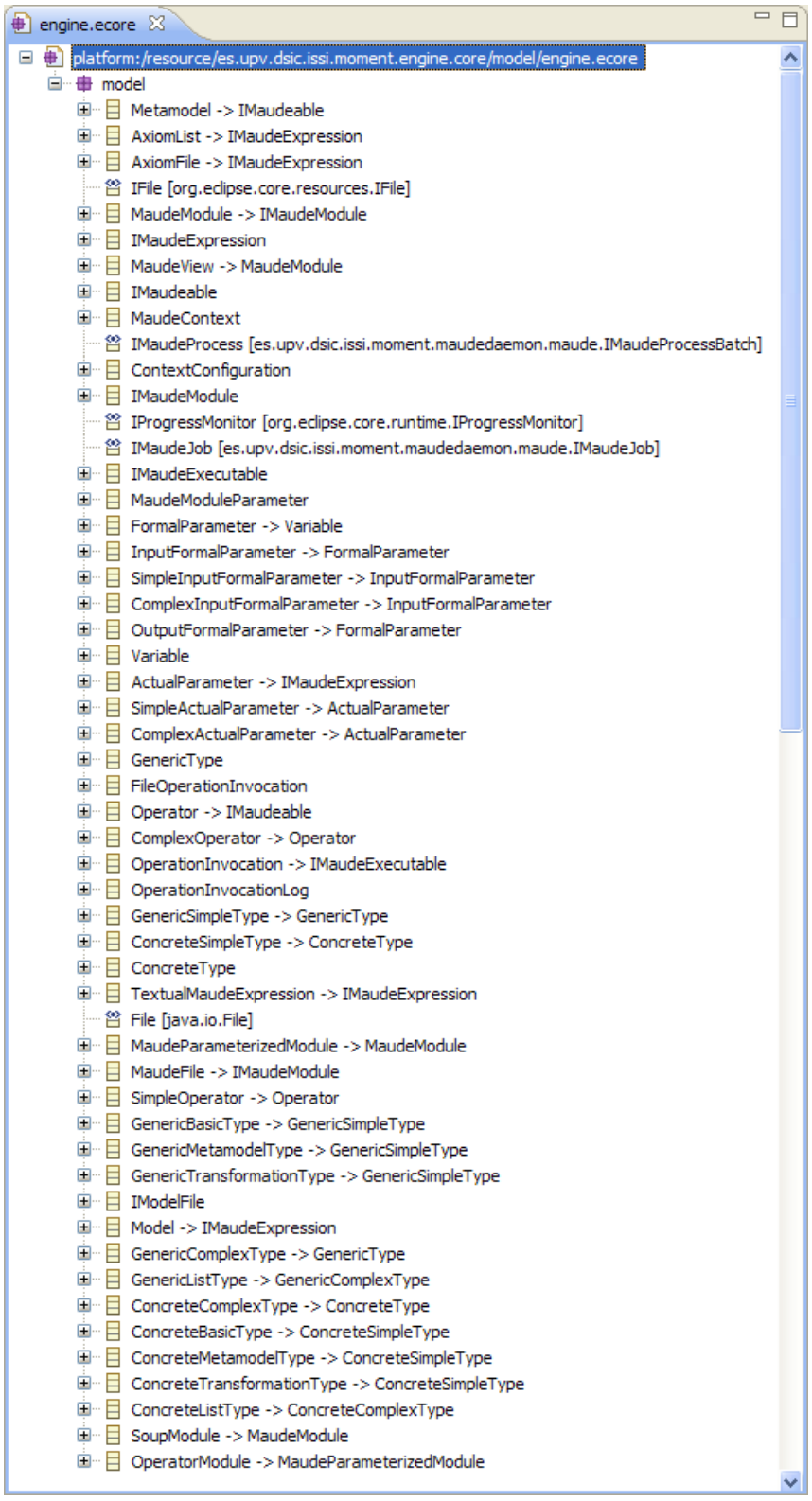

Figura 6.12: Vista general del modelo EMF del plugin *MOMENT Engine Core*.

- Regenerar el código del plugin.
- Reorganizar los módulos Maude de los operadores simples.
- Dar soporte para la reconfiguración del kernel de MOMENT en tiempo de ejecución.
- Añadir el soporte para la proyección a código Maude de los operadores complejos.

Dependencias

Las dependencias del plugin son las siguientes:

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 *ANTLR* y de los siguientes plug-ins de MOMENT:

- es.upv.dsic.issi.moment.mdt.maudedaemon. Proporciona soporte para la creación de un proceso Maude y la interacción con él. Éste plugin puede encontrarse en [26].
- es.upv.dsic.issi.moment.qvt.relations. El plugin es.upv.dsic.issi.moment.qvt.relations proporciona soporte para la definición de transformaciones QVT como una instancia del modelo QVT implementado en EMF. A su vez, proporciona las capacidades de proyección de transformaciones de programas QVT a Maude. Esta dependencia es necesaria para la ejecución del operador *ModelGen*.
- 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.traceability.metamodels.basic. El plugin es.upv.dsic.issi.moment.traceability.metamodels.basic corresponde con la implementación del modelo EMF que en MOMENT da soporte para trazabilidad. Se requiere para poder representar los modelos de enlaces que toda operación en MOMENT devuelve.

6.3.4.2. Diseño

La funcionalidad del plugin se ha añadido de dos maneras: mediante los mecanismos de generación de código de EMF, y mediante el enriquecimiento del código generado con fragmentos de código escritos de forma manual.

6.3.4.3. Implementación

Modelo del DSL

El modelo simplificado del DSL se mostraba en la figura 6.13. A continuación, la figura 6.14 muestra resaltadas las principales clases añadidas al modelo del *MOMENT Engine Core* para dar soporte a nuestro lenguaje de definición de operadores complejos.

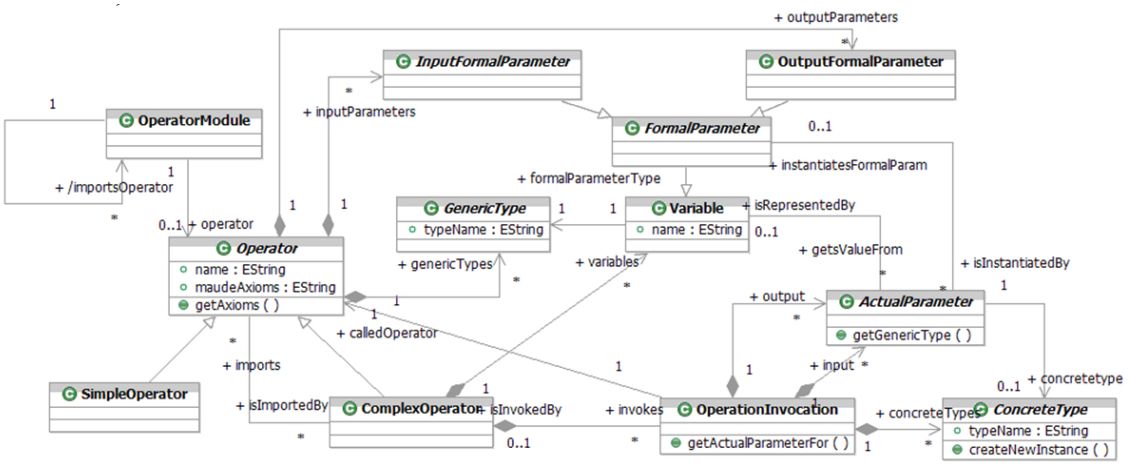

Figura 6.13: Modelo simplificado para la especificación de operadores complejos en MOMENT.

Como se indicaba en la sección 6.2.1.1, 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

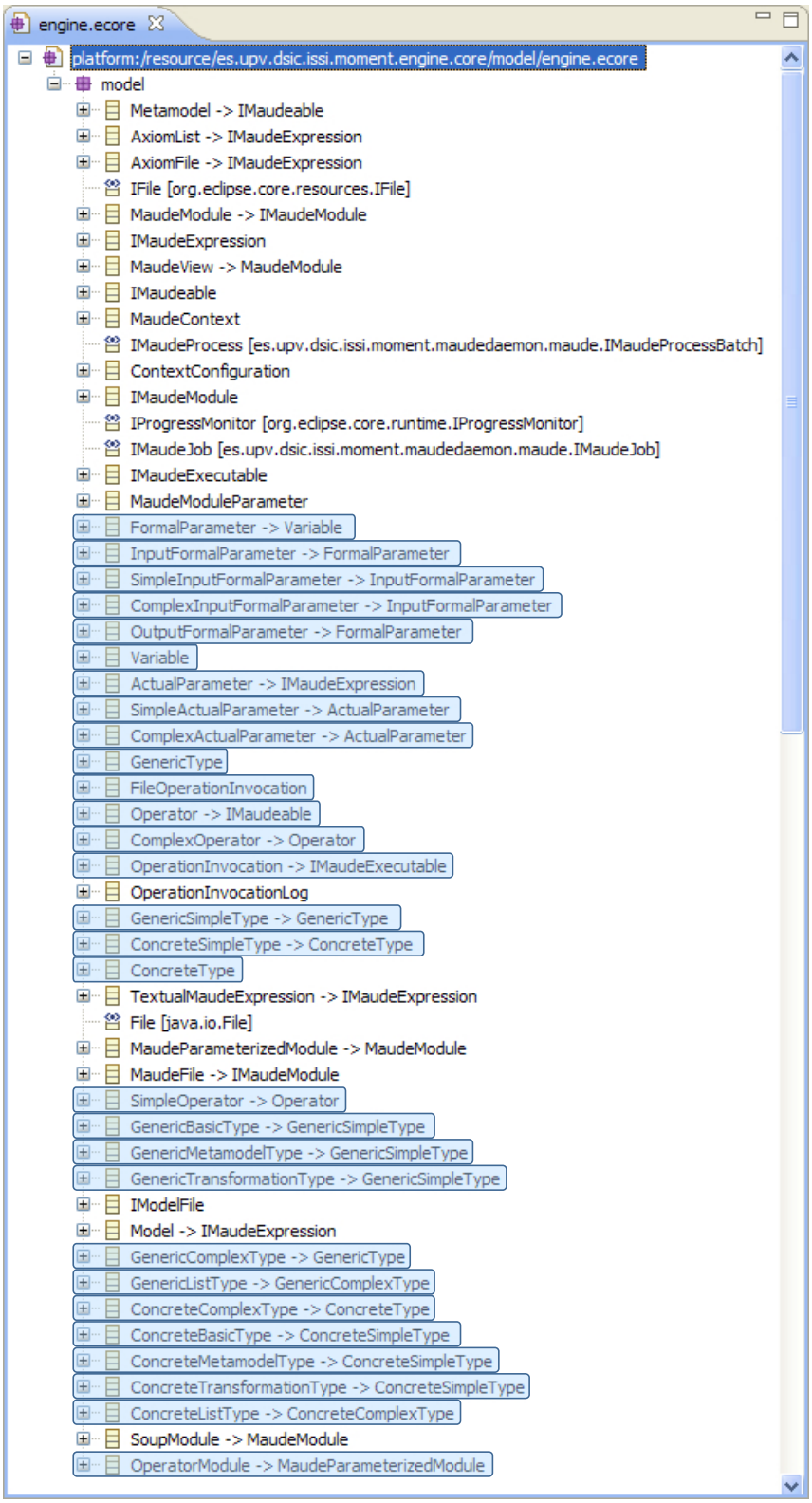

Figura 6.14: Clases del DSL del modelo del *MOMENT Engine Core*

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.

La 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 6.2, y no sobre un árbol de sintaxis abstracta construido, por ejemplo, en tiempo de compilación a partir de una gramática.

Organización de los módulos del kernel

Para reducir el consumo de memoria, y el coste temporal en la carga del kernel se han reestructurado los módulos que lo constituyen. De esta manera, con la nueva organización, los módulos se han separado en ficheros de forma que los módulos para un mismo propósito se encuentran en un mismo fichero. A su vez, se ha incluído un código numérico para identificar los grupos de ficheros, además de que se llega a la convención de nombrar en máyúsculas a los ficheros del kernel.

Los ficheros que constituyen actualmente el kernel de MOMENT son los siguientes:

- 101_PARAMETER.maude
- 110_OCL-SUPPORT.maude
- 121_DATATYPE.maude
- 130_MODEL-SUPPORT.maude
- 140_TUPLE-MODULES.maude

- 150_SIGECORE.maude
- 151_SPECORE.maude
- 160_ECORE-REFLECTION-SUPPORT.maude
- 161_ECORE-REFLECTION.maude
- 170_SIGTRACEABILITYMETAMODEL.maude
- 171_SPTRACEABILITYMETAMODEL.maude
- 181_SPTHESAURUS.maude
- 201_MOMENT-KERNEL.maude
- 301_EQUALS.maude
- 302_MODELGEN.maude
- 310_MERGE.maude
- 311_SINGLEMERGE.maude
- 312_MERGE_WITHOUT_MERGEID.maude
- 313_MERGE_NOID_CONTAINMENT.maude
- 320_CROSS.maude
- 330_DIFF.maude
- 340_MATCH.maude
- 350_MODELTOTEXT.maude
- 501_SELECTTRACESSBYDOMAIN.maude
- 502_SELECTTRACESBYRANGE.maude
- 511_COMPOSE.maude
- 512_INVERT.maude
- 601_ENDOGENOUSDOMAIN.maude
- 602_ENDOUGENOUSRANGE.maude
- 651_EXOGENOUSDOMAIN.maude

- 652_EXOGENOUSRANGE.maude
- 710_REFRESHTRACEABILITYMODELDOMAIN.maude
- 711_REFRESHTRACEABILITYMODELRANGE.maude
- 720_COMPLETEMODELREFERENCES.maude

Los códigos numéricos empleados son los siguientes:

- **100-149**: Módulos básicos del kernel: soporte paramétrico para OCL, tipos de datos, etc.
- **150-199**: Módulos básicos del kernel: soporte para los metamodelos básicos de MOMENT (ecore, trazabilidad y tesauros de sinónimos).
- **200-299**: Otros módulos básicos.
- **300-499**: Operadores básicos de MOMENT. Los operadores similares compartiran un mismo valor de las decenas en su código numérico.
- **500-599**: Operadores básicos de trazabilidad que devuelven modelos de trazabilidad.
- **600-699**: Operadores básicos de trazabilidad que devuelven cualquier modelo.
- **700-799**: Operadores auxiliares de MOMENT. Para uso interno.

Reconfiguración del kernel

Tras la reorganización de ficheros del kernel de MOMENT se puede dar soporte para la reconfiguración bajo demanda del kernel. Dé esta manera, un módulo sólo será cargado cuando sea requerido para la ejecución de un operador. La configuración del kernel de MOMENT se realiza mediante un fichero XML que establece la ruta de cada fichero del kernel, así como sus dependencias. Éste fichero se encuentra en es.upv.dsic.issi.moment.engine.core/kernel/kernel.mkconf. Este fichero puede consultarse en el anexo C.

El fichero XML *kernel.mkconf* no es mas que una instancia en XMI del modelo de configuración del kernel de MOMENT (ver figura 6.15).

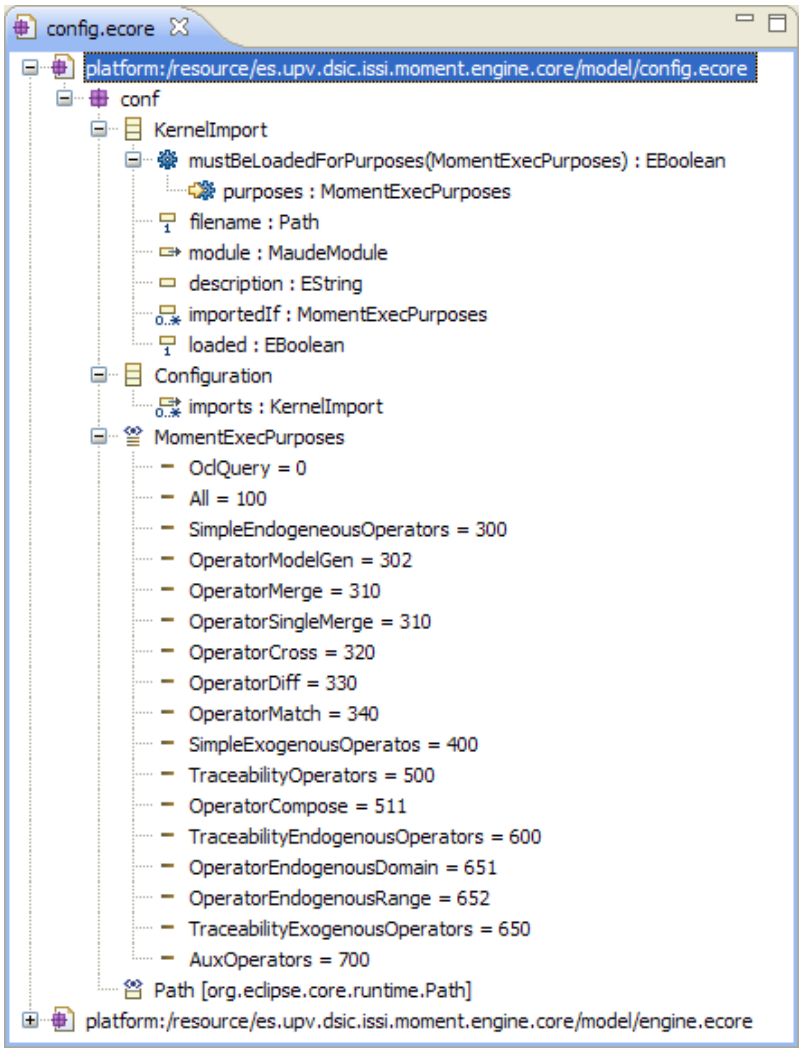

Figura 6.15: Modelo EMF del soporte para reconfiguración del kernel.

En el modelo se observa que existe un tipo enumerado, *MomentExecPurposes*, que identificada cada uno de los propósitos para los que se puede cargar el kernel. Esto es, para ejecutar un determinado operador, ejecutar una consulta OCL, etc. El modelo tiene un elemento raíz, *Configuration*, que contiene cada una de las entradas del kernel (*KernelImport*). Cada entrada contiene:

- La ruta del fichero correspondiente (atributo *path*).
- El módulo Maude que le corresponde (atributo *module*, que se establece opcionalmente en tiempo de ejecución).
- La descripción de los módulos que contiene (atributo *description*). Empleado para mostrar información al usuario en el proceso de carga.
- Información de si ya se ha cargado o no (atributo *loaded*). Se establece en tiempo de ejecución, siendo por defecto *false*.
- Información acerca de cuando se ha de cargar el módulo (atributo *importedIf*). Este atributo multivaluado almacena la lista de propósitos para los que sirve este módulo. Si se solicita la carga del kernel para un propósito concreto, y el fichero no ha sido cargado previamente (el atributo *loaded* es false), el fichero se cargará.

Al iniciarse el plugin *MOMENT Engine Core*, se cargará el fichero *kernel.mkconf*, y se mantendrá en memoria. Cuando sea necesaria la carga de cualquier módulo, el modelo «mkconf» se consultará, realizando las acciones necesarias, y actualizando el modelo «mkconf» reflejando en todo momento el estado actual del kernel.

Proyección a código Maude

La proyección a código Maude de un operador se realiza de forma mixta mediante plantilla en Velocity y código Java.

En primer lugar, un operador complejo se proyecta a Maude a partir de un OperatorModule. La clase OperatorModule genera el código Maude a partir de la plantilla opmodule.vm, que es invocada mediante el método toMaudeCode() (listado 6.10):

```
1     public String toMaudeCode() {
2       VelocityContext context = new VelocityContext();
```

```
3       context.put("module", this);
4
5       return MomentEngineCore.getConfiguration().
           getVelocityGenerator().generate("opmodule.vm",context);
6     }
```

Listado 6.10: Invocación de la plantilla de creación de un módulo Maude para un operador

El contenido de la plantilla opmodule.vm se muestra en el listado 6.11

```
1     ## context contents:
2     ##                                          "module" -
          MaudeModule
3     ##                                          "ctx" - The current
          context itself (necessary for eval)
4     ##
5
6     fmod $module.name.toUpperCase() ##
7     #if(!$module.getOperator().getGenericTypes().isEmpty())##
8     #*      *#{##
9     #*      *##foreach( $type in $module.getOperator().
          getGenericTypes())##
10    #*              *##if($velocityCount != 1)##
11    #*                      *# , ##
12    #*              *##end##
13    #*              *# $type.typeName :: TRIV##
14    #*      *##end##
15    #*      *# }##
16    #end##
17     is
18
19    #foreach( $statement in $module.statements)
20    #*      *##eval($statement.toMaudeExpression())
21    #end
22
23    endfm
```

Listado 6.11: Proyección de un operador complejo a módulo Maude

Se observa que para un determinado operador, se crea un nuevo módulo paremétrico cuyo nombre es el del operador en mayúsculas, y los parámetros cada uno de los metamodelos que intervienen en el operador:

```
1
2     fmod OPERADOR { METAMDELO1 :: TRIV , METAMDELO2 :: TRIV ,
          ...} is
```

```
3
4             *** Código del operador
5
6     endfm
```

Listado 6.12: Ejemplo de proyección de un operador complejo a módulo Maude

Un objeto de la clase OperatorModule se crea automáticamente mediante la invocación del método toMaudeModules() (véase el listado 6.13) de la clase Operator.

```
1     public EList toMaudeModules() {
2       Context context = new VelocityContext();
3       EList result = new BasicEList();
4
5       context.put("operator",this);
6       String contents = MomentEngineCore.getConfiguration().
           getVelocityGenerator().generate("operation.vm", context
           );
7
8       OperatorModule opModule = MomentEngineFactory.eINSTANCE.
           createOperatorModule();
9       opModule.setOperator(this);
10      opModule.setName(name);
11
12      opModule.getStatements().add(
13      MomentEngineFactory.eINSTANCE.createTextualMaudeExpression
           (contents));
14
15      result.add(opModule);
16      return result;
17    }
```

Listado 6.13: Invocación de la plantilla de creación de un módulo Maude para un operador

Se observa que en el momento de la creación del OperatorModule se llama a la plantilla operation.vm, que obtiene el código Maude el operador (línea 6) y posteriormente se añade a los statements del módulo (línea 12). Ese será el contenido que se colocará dentro del módulo del operador mostrado en el listado 6.12.

La plantilla de código operation.vm se muestra en el listado 6.14

```
1     #parse("symbols.vm")##
2
3     #* *#    *** Used operators imports
4     #foreach($invocation in $operator.invokes)##
```

```
5   ##* *#    pr $invocation.calledOperator.name.toUpperCase()$so
        ##
6   ##* *##foreach($type in $invocation.getFinalGenericTypes())##
7   ##*         *##if($velocityCount != 1)##
8   ##*                 *# , ##
9   ##*         *##end##
10  ##*               *#$type.typeName##
11  ##* *##end
12  ##* *# $sc .
13  #end##
14  ##*       *##if($operator.outputParameters.size() > 1)##
15  ##*   *#    *** Tuple of return types
16  ##*       *#    pr TUPLE<$operator.outputParameters.size()>
        ##
17  ##*       *#$so ##
18  ##*   *##foreach($param in $operator.outputParameters)##
19  ##*           *##if($velocityCount != 1)##
20  ##*                   *# , ##
21  ##*         *##end##
22  ##*               *#$param.formalParameterType.typeName##
23  ##*   *##end
24  ##*       *# $sc .
25  ##* *##end##
26
27  ## 1.1 - Declaración de las variables de los parámetros de
        entrada
28
29  *** Input parameters
30  #foreach($param in $operator.inputParameters)##
31  ##*      *#var $param.name : ##
32  ##*               *#$param.formalParameterType.toMaudeType()
        .
33  #end
34
35  ##
36  ##
37  ## 2 - Construcción de la signatura de la operación
38  ##
39
40  *** Operator declaration
41  op $operator.name : ##
42  #foreach($param in $operator.inputParameters)##
43  ##*      *#$param.formalParameterType.toMaudeType() ##
44  #end##
45  ##
46  ##
47  ## 2.1 - Construcción del tipo de la operación
48  ##
49  ##*       *##if($operator.outputParameters.size() > 1)##
```

```
50      #*   *#-> Tuple { ##
51      #* *##else##
52      #*   *#-> Set { ##
53      #* *##end##
54      #foreach($output in $operator.outputParameters)##
55      #*      *##if($velocityCount != 1)##
56      #*              *# , ##
57      #*      *##end##
58      #*              *#$output.formalParameterType.typeName##
59      #end##
60       } .##
61      ##
62      ##
63
64      ## 3 - Declaración de la operación
65      eq $operator.name (##
66      #foreach($param in $operator.inputParameters)##
67      #*      *##if($velocityCount != 1)##
68      #*              *#, ##
69      #*      *##end##
70      #*      *#$param.name ##
71      #end##
72      ) =
73      $operator.getBody().toMaudeExpression()
74      .
```

Listado 6.14: Obtención del código Maudeo de un operador

Las líneas 4 a 13 proyectan las importaciones de los operadores empleados en el operador complejo. Estas importaciones se proyectan de la forma

```
1       pr OPERATOR {METAMODEL1, METAMODEL2, ...} .
```

Listado 6.15: Código para la importación de un operador en Maude.

Las líneas 14 a 26 proyectan la importación del módulo TUPLE, que da soporte al uso de tuplas. Las tuplas son necesarias para los operadores que devuelven más de un valor. El código proyectado sigue el patrón:

```
1       pr TUPLE<NUM_ARGUMENTOS_SALIDA> {TIPO_ARG1, TIPO_ARG2, ...}
            .
```

Listado 6.16: Código para la importación del módulo TUPLE.

Las líneas 24 a 36 declaran las variables de los argumentos de entrada:

```
1     var ARGUMENTO1 : TIPO_ARGUMENTO1 .
```

Listado 6.17: Código para la declaración de variables en Maude.

Las líneas 35 a 62 proyectan la signatura de la operación de la forma:

```
1     op OPERATOR : TIPO_ARGUMENTO1 , TIPO_ARGUMENTO2 , ... ->
         Tuple {TIPO_RESULTADO1 , TUPO_RESULTADO2 , ...} .
```

Listado 6.18: Declaración de una operación.

Por último, las líneas 64 a 74 proyectan la ecuación de la operación:

```
1     eq OPERATOR ( VARIABLE_ARGUMENTO1 , VARIABLE_ARGUMENTO2 ,
         ... ) =
2
3           *** Ecuaciones anidadas que obtienen el resultado
4
5     .
```

Listado 6.19: Declaración de la ecuación de una operacion.

El cuerpo de una operación se compone mediante sucesivas invocaciones al método getBody().toMaudeExpression() (listado 6.20). Este método se encarga de navegar hacia abajo y reconstruir la secuencia de invocaciones que permite obtener un argumento.

```
1
2     public IMaudeExpression getBody() {
3       String expr = "";
4       for (int i = 0; i < getOutputParameters().size(); i++) {
5         expr = expr.concat(getBodyExpressionAt(i).
             toMaudeExpression());
6         if (i < getOutputParameters().size()-1) {
7           expr = expr.concat(" , ");
8         }
9       }
10      return MomentEngineCore.getFactory().
           createTextualMaudeExpression(" ( \n" + expr + "\n ) ");
11    }
12
13    private IMaudeExpression getBodyExpressionAt(int numParam) {
14      String expr = "";
15      if (numParam > getOutputParameters().size() || numParam <
           0) {
```

```
16            return null;
17        } else {
18          OperationInvocation opInv = findInvocationForOutput((
               OutputFormalParameter)getOutputParameters().get(
               numParam));
19          if (opInv != null) {
20            expr = opInv.toMaudeCommand();
21            if (opInv.getCalledOperator().getOutputParameters().
                 size() > 1) {
22              for (Object objAP : opInv.getOutput()){
23                if (objAP instanceof SimpleActualParameter) {
24                  SimpleActualParameter sap = (
                       SimpleActualParameter) objAP;
25                  if (sap.getIsRepresentedBy() == (
                       OutputFormalParameter)getOutputParameters().
                       get(numParam))
26                    expr = "p" + (opInv.getOutput().indexOf(sap)
                         +1) + "(" + expr + ")";
27                }
28              }
29            }
30          }
31        }
32        return MomentEngineCore.getFactory().
             createTextualMaudeExpression(expr);
33      }
```

Listado 6.20: Obtención del cuerpo de la ecuación de un operador.

6.3.4.4. Archivos resultantes

Como resultado de la implementación, a continuación se listan los ficheros y paquetes más relevantes del plugin *MOMENT Engine Core*:

Directorio templates

- invocation.vm
- invocationmodelgen.vm
- M1.vm
- M2sig.vm

- M2sp.vm
- macros.vm
- module.vm
- operation.vm
- opmodule.vm
- parammodule.vm
- refreshaxdiff.vm
- refreshaxmacros.vm
- refreshaxmerge.vm
- symbols.vm
- tracInvocation.vm
- view.vm

Directorio kernel

- kernel.mkconf
- 101_PARAMETER.maude
- 110_OCL-SUPPORT.maude
- 121_DATATYPE.maude
- 130_MODEL-SUPPORT.maude
- 140_TUPLE-MODULES.maude
- 150_SIGECORE.maude
- 151_SPECORE.maude
- 160_ECORE-REFLECTION-SUPPORT.maude
- 161_ECORE-REFLECTION.maude
- 170_SIGTRACEABILITYMETAMODEL.maude

- 171_SPTRACEABILITYMETAMODEL.maude
- 181_SPTHESAURUS.maude
- 201_MOMENT-KERNEL.maude
- 301_EQUALS.maude
- 302_MODELGEN.maude
- 310_MERGE.maude
- 311_SINGLEMERGE.maude
- 312_MERGE_WITHOUT_MERGEID.maude
- 313_MERGE_NOID_CONTAINMENT.maude
- 320_CROSS.maude
- 330_DIFF.maude
- 340_MATCH.maude
- 350_MODELTOTEXT.maude
- 501_SELECTTRACESSBYDOMAIN.maude
- 502_SELECTTRACESBYRANGE.maude
- 511_COMPOSE.maude
- 512_INVERT.maude
- 601_ENDOGENOUSDOMAIN.maude
- 602_ENDOUGENOUSRANGE.maude
- 651_EXOGENOUSDOMAIN.maude
- 652_EXOGENOUSRANGE.maude
- 710_REFRESHTRACEABILITYMODELDOMAIN.maude
- 711_REFRESHTRACEABILITYMODELRANGE.maude
- 720_COMPLETEMODELREFERENCES.maude

Directorio grammars

- AttribsAndRefsPass.g
- Parser.g
- ReificationPass.g

Directorio model

- config.ecore
- config.genmodel
- engine.ecore
- engine.genmodel

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

- EngineConfiguration.java
- MomentEngineCore.java

Paquete es.upv.dsic.issi.moment.engine.core.conf

- Configuration.java
- KernelConfigFactory.java
- KernelConfigPackage.java
- KernelImport.java
- MomentExecPurposes.java

Paquete es.upv.dsic.issi.moment.engine.core.conf.impl

- ConfigurationImpl.java
- KernelConfigFactoryImpl.java

- KernelConfigPackageImpl.java
- KernelImportImpl.java

Paquete es.upv.dsic.issi.moment.engine.core.conf.util

- KernelConfigAdapterFactory.java
- KernelConfigResourceFactoryImpl.java
- KernelConfigResourceImpl.java
- KernelConfigSwitch.java

Paquete es.upv.dsic.issi.moment.engine.core.internal

- MomentIdGenerator.java
- AbstractMomentIdGenerator.java
- M1Helper.java
- MetamodelContext.java
- ModelContext.java
- MomentContext.java
- MomentEcoreEquiv.java
- SimpleIdGenerator.java
- URIIdGenerator.java
- VarMaelstrom.java

Paquete es.upv.dsic.issi.moment.engine.core.internal.postprocess

- TextProcessStrategy.java
- AbstractTextProcessStrategy.java
- ContainerRemover.java

- LastParenthAdder.java
- MetamodelCommentAdder.java
- MomentSetUnwrapper.java
- NewLinesRemover.java
- ResultSelector.java

Paquete es.upv.dsic.issi.moment.engine.core.internal.velocity

- VelocityInitializer.java
- EclipseVelocityInitializer.java
- VelocityGenerator.java

Paquete es.upv.dsic.issi.moment.engine.core.model

- ActualParameter.java
- AxiomFile.java
- AxiomList.java
- ComplexActualParameter.java
- ComplexInputFormalParameter.java
- ComplexOperator.java
- ConcreteBasicType.java
- ConcreteComplexType.java
- ConcreteListType.java
- ConcreteMetamodelType.java
- ConcreteSimpleType.java
- ConcreteTransformationType.java
- ConcreteType.java

- ContextConfiguration.java
- FileOperationInvocation.java
- FormalParameter.java
- GenericBasicType.java
- GenericComplexType.java
- GenericListType.java
- GenericMetamodelType.java
- GenericSimpleType.java
- GenericTransformationType.java
- GenericType.java
- IMaudeable.java
- IMaudeExecutable.java
- IMaudeExpression.java
- IMaudeModule.java
- IModelFile.java
- InputFormalParameter.java
- KernelConfiguration.java
- MaudeContext.java
- MaudeFile.java
- MaudeModule.java
- MaudeModuleParameter.java
- MaudeParameterizedModule.java
- MaudeView.java
- Metamodel.java
- Model.java

- MomentEngineFactory.java
- MomentEnginePackage.java
- OperationInvocation.java
- OperationInvocationLog.java
- Operator.java
- OperatorModule.java
- OutputFormalParameter.java
- SimpleActualParameter.java
- SimpleInputFormalParameter.java
- SimpleOperator.java
- SoupModule.java
- TextualMaudeExpression.java
- Variable.java

Paquete es.upv.dsic.issi.moment.engine.core.model.impl

- ActualParameterImpl.java
- AxiomFileImpl.java
- AxiomListImpl.java
- ComplexActualParameterImpl.java
- ComplexInputFormalParameterImpl.java
- ComplexOperatorImpl.java
- ConcreteBasicTypeImpl.java
- ConcreteComplexTypeImpl.java
- ConcreteListTypeImpl.java
- ConcreteMetamodelTypeImpl.java

- ConcreteSimpleTypeImpl.java
- ConcreteTransformationTypeImpl.java
- ConcreteTypeImpl.java
- ContextConfigurationImpl.java
- FileOperationInvocationImpl.java
- FormalParameterImpl.java
- GenericBasicTypeImpl.java
- GenericComplexTypeImpl.java
- GenericListTypeImpl.java
- GenericMetamodelTypeImpl.java
- GenericSimpleTypeImpl.java
- GenericTransformationTypeImpl.java
- GenericTypeImpl.java
- InputFormalParameterImpl.java
- MaudeContextImpl.java
- MaudeFileImpl.java
- MaudeModuleImpl.java
- MaudeModuleParameterImpl.java
- MaudeParameterizedModuleImpl.java
- MaudeViewImpl.java
- MetamodelImpl.java
- ModelImpl.java
- MomentEngineFactoryImpl.java
- MomentEnginePackageImpl.java
- OperationInvocationImpl.java

- OperationInvocationLogImpl.java
- OperatorImpl.java
- OperatorModuleImpl.java
- OutputFormalParameterImpl.java
- SimpleActualParameterImpl.java
- SimpleInputFormalParameterImpl.java
- SimpleOperatorImpl.java
- SoupModuleImpl.java
- TextualMaudeExpressionImpl.java
- VariableImpl .java

Paquete es.upv.dsic.issi.moment.engine.core.model.util

- MomentEngineAdapterFactory.java
- MomentEngineSwitch.java

Paquete es.upv.dsic.issi.moment.engine.core.parser

- LazyReferenceResolver.java

Paquete es.upv.dsic.issi.moment.engine.core.util

- ExtractInfoEPackage.java
- XMIzer.java

Paquete es.upv.dsic.issi.moment.engine.ui.preferences

- GeneralPreferencePage.java
- MomentGeneralPreferenceConstants.java
- PreferenceInitializer.java

Paquete es.upv.dsic.issi.moment.exception

- InvalidXMIException.java
- MaudeException.java
- MetamodelException.java
- MomentException.java
- MomentKernelException.java

Paquete es.upv.dsic.issi.moment.registry

- MomentRegistryPlugin.java

Paquete es.upv.dsic.issi.moment.registry.model

- EPackage.java
- EquivalenceRelationFile.java
- IMomentFile.java
- KernelConfigurationFile.java
- MetamodelContainer.java
- ModelFile.java
- MomentRegistry.java
- OperatorFile.java
- RegistryFactory.java
- RegistryPackage.java
- TraceModelFile.java
- TransformationFile.java

Paquete es.upv.dsic.issi.moment.registry.model.impl

- EPackageImpl.java
- EquivalenceRelationFileImpl.java
- KernelConfigurationFileImpl.java
- MetamodelContainerImpl.java
- ModelFileImpl.java
- MomentRegistryImpl.java
- OperatorFileImpl.java
- RegistryFactoryImpl.java
- RegistryPackageImpl.java
- TraceModelFileImpl.java
- TransformationFileImpl.java

Paquete es.upv.dsic.issi.moment.registry.model.util

- RegistryAdapterFactory.java
- RegistrySwitch.java

Paquete org.apache.commons.lang

- Entities.java

Capítulo 7

Trabajos relacionados.

La Gestión de Modelos es aún un campo emergen en investigación y son muy pocas las herramientas que proporcionan una visión similar a la de MOMENT para el tratamiento de los modelos. Es por ello que a continuación se presentan tan poco aproximaciones similares a la de MOMENT. De hecho, únicamente RONDO presenta una aproximación comparable a MOMENT.

7.1. RONDO.

RONDO, como se introdujo en el apartado 2.2.1, es la plataforma de Gestión de Modelos basada en los trabajos de P. Bernstein [1]. En esta herramienta, por ejemplo, el operador *Merge* (que permite la integración de dos modelos), recibe como entradas dos modelos (A y B) y un modelo de *mappings* entre ellos (map_{AB}); y produce el modelo combinado C, y dos nuevos modelos de correspondencias (map_{AC} y map_{BC}): $< C, map_{AC}, map_{BC} >= Merge(A, B, map_{AB})$.

Para la creación de estos modelos de correspondencias se proporciona una sencilla interfaz. La figura 7.1 muestra el editor simple de correspondencias implementado en Java para la plataforma RONDO. Éstos serán los mappings que se emplearán en la operación Merge en su ejemplo de propagación de cambios. Para mayor detalle se puede consultar [19].

En MOMENT, los modelos de mappings se introducen como modelos de trazabilidad. Esto se debe a que los operadores no se apoyan en ellos para aplicarse

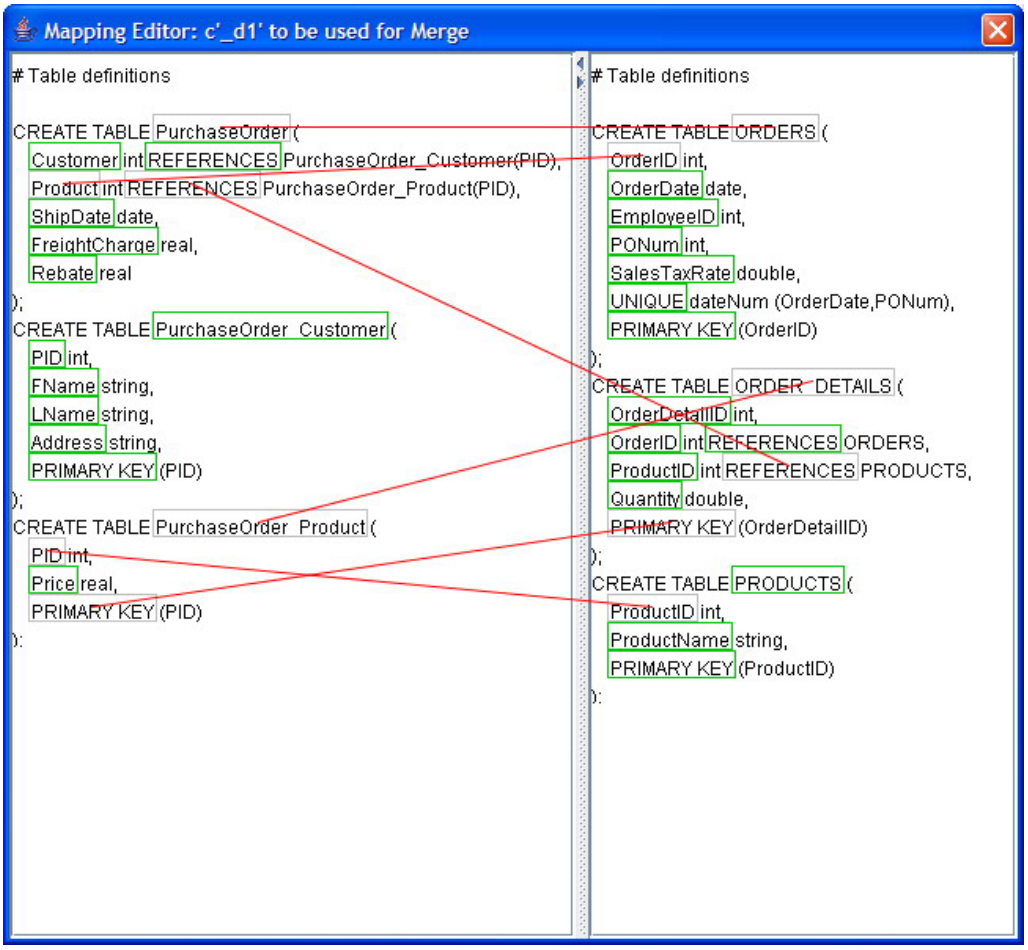

Figura 7.1: Vista del editor de correspondencias de RONDO.

a un conjunto de modelos. En MOMENT, las relaciones de trazabilidad entre los elementos de dos modelos, que se necesitan para aplicar un operador, se definen entre los elementos de sus correspondientes metamodelos de forma axiomática con los correspondientes operadores. La colección de relaciones de equivalencia entre dos metamodelos constituye un morfismo que puede ser reusado por todos los operadores del álgebra de MOMENT. Esto permite una especificación más clara de los operadores complejos. En MOMENT, el operador *Merge* es de la siguiente manera: $< C, map_{AC}, map_{BC} >= Merge(A, B)$. Los modelos de correspondencias son producidos por la aplicación de un operador simple a un conjunto de modelos, y almacena la información acerca de la tarea de manipulación realizada sobre un modelo. Por ello, se tratan estos modelos de correspondencias desde el punto de vista de la trazabilidad.

En cuanto a la interfaz de RONDO, se observa que frente a las herramientas proporcionadas por MOMENT, éstas últimas son mucho más completas, claras, y genéricas. Los editores de MOMENT permiten representar cualquier modelo, sin necesidad de modificar absolutamente ninguna línea de código. Únicamente basta registrar su correspondiente metamodelo en EMF. Por otra parte, aunque la representación de las correspondencias mediante líneas parece resultar inicialmente más expresiva, puede resultar en una pérdida de información en modelos complejos ya que el número de correspondencias mostradas simultáneamente puede ser excesivo.

A parte, la representación de las correspondencias tal y como se hace en MOMENT, mostrando directamente el modelo de trazabilidad (aunque proporcionando igualmente facilidades de navegación), permite de manera automática mostrar cualquier información que un metamodelo de trazabilidad personalizado permita recoger además de la que define el metamodelo básico. Al margen de todo esto, ésta es la forma habitual de mostrar este tipo de información en otras herramientas basadas en Eclipse.

En cuanto a la declaración de operadores RONDO también proporciona un lenguaje propio y no documentado, similar al que MOMENT proporciona. El listado 7.1 muestra un ejemplo de éste.

```
1     // Same as above but showing more popup windows
2     operator PropagateChangesVerbose(s1, d1, s1_d1,
3                                      s2, c,  s2_c) {
4
5       // TASK 1: propagate deletions
6
7       (s1_s2, multimap) = Match(s1, s2, NGramMatch(s1, s2));
8       s1_s2 = EditMap(s1, s2, s1_s2, multimap, "s1_s2: original
```

```
          and modified schemas");
9     //  s1_s2 = EditMap(s1, s2, ReverseNonIdentical(s1, s2,
          s1_s2), multimap, "s1_s2: original and modified schemas")
          ;
10
11      d1_del = Traverse(All(s1) - Domain(s1_s2), s1_d1); //
            unmatched elements in d1
12            d1_del = EditSelector(d1, d1_del, "Elements to be
                 soft-deleted from d1");
13
14      (d1', d1'_d1) = Delete(d1, d1_del);  // soft since PK
            should stay
15            popup(d1', Obj2Text, "d1 without deleted elements");
16
17
18      // TASK 2: propagate additions/renamings
19
20      s2_add = All(s2) - Range(Forward(s1_s2)); // need to add
            backward as well; consider them unmatched here
21            s2_add = EditSelector(s2, s2_add, "New/renamed
                 elements in s2");
22            // why not select s2_add first and convert: operator
                  less generic (conversion built-in)
23
24      s2_c = EditMap(s2, c, s2_c, null, "s2_c: converted s2 to c
            ");
25
26      c_add = Traverse(s2_add, s2_c);
27            c_add = EditSelector(c, c_add, "New/renamed elements
                 to add from c");
28
29      (c', c'_c) = Extract(c, c_add);
30    //  c' = Extract(c, c_add, Dependencies(c)); // needed for
          xsd-rdb (PK/FK)
31            popup(c', Obj2Text, "c': portion of c to be added");
32
33      c'_d1' = c'_c * Invert(s2_c) * Invert(s1_s2) * s1_d1 *
            Invert(d1'_d1);
34            c'_d1' = EditMap(c', d1', c'_d1', null, "First cut c
                 '_d1' (by composition, w/o Match)");
35
36      // next two lines are optional: match surrounding elements
37      c'_d1' = Match(c', d1', c'_d1'); // OneSimilarity(c_d1));
38      // make sure that new elements remain unmatched; second
            term is similar to c_add...
39      c'_d1' = RestrictDomain(c'_d1', All(c') -
40                                                  Traverse(All(s2)
                                                       - Range(
                                                     s1_s2), s2_c
```

```
                                                        * Invert(c'_c
                                                        )));
41              // EditMap(c', d1', c'_d1', multimap, "c'_d1' after
                   Match");
42
43        do {
44              c'_d1' = EditMap(c', d1', c'_d1', null, "c'_d1' to
                   be used for Merge");
45
46             (d2, c'_d2, d1'_d2) = Merge(c', d1', c'_d1'); // new
                   version
47
48        // TASK 3: compute new mapping s1_d1
49
50          s2_d2 = s2_c * Invert(c'_c) * c'_d2 + Invert(s1_s2) *
                   s1_d1 * Invert(d1'_d1) * d1'_d2;
51          s2_d2 = EditMap(s2, d2, s2_d2, null, "new s1_d1 to be
                   stored");
52
53        } while(YesNo("Repeat merge?"));
54
55
56        return (d2, s2_d2);
57      }
58      }
```

Listado 7.1: Ejemplo de operador complejo en RONDO.

Se observa que la sintaxis parece bastante similar al lenguaje proporcionado por MOMENT, no obstante, la cuestión respecto a las correspondencias implícitas que se comentaba en párrafos anteriores, obliga en RONDO a realizar numerosas operaciones adicionales que en MOMENT resultan innecesarias. Además, se observa que en RONDO no es necesario declarar los tipos de las variables. En MOMENT esto sí es necesario puesto que se implementa el operador *ModelGen* que al ser un operador exógeno (intervienen distintos metamodelos en él) hace que los mecanismos de inferencia de tipos sean excesivamente complejos si se desea eliminar la declaración de variables explícita. Además, el lenguaje proporcionado por RONDO proporciona construcciones de tipo imperativas e instrucciones de entrada/salida, que introducen coherencias en el lenguaje confundiendo al usuario.

Capítulo 8

Conclusiones.

La Gestión de Modelos es un campo de investigación emergente en la ingeniería dirigida por modelos que destaca por su potencia en la composición de operaciones de una forma natural, intuitiva, genérica y reutilizable.

En este trabajo se ha presentado cómo se ha conseguido diseñar e implementar un DSL para la definición de operadores complejos para la herramienta MOMENT.

Maude proporciona una implementación de la lógica ecuacional de pertenencia, que ha sido utilizada para definir las operaciones de gestión de modelos mediante módulos funcionales. En estos módulos funcionales, las operaciones son descritas como funciones y dependiendo de las propiedades algebraicas que son añadidas a cada operación (asociatividad, conmutatividad, etc) se pueden componer fácilmente. Estas facilidades de composición de funciones han sido reflejadas en el lenguaje de definición de operadores complejos.

Por otra parte, el uso de una herramienta como Eclipse y EMF proporciona a esta aproximación una gran interoperabilidad y extensibilidad, a la vez que permite aplicar la propia filosofía de ingeniería dirigida por modelos al diseño del lenguaje de definición de operadores complejos. La representación de un operador a un mayor nivel de abstracción permite el aprovechamiento de técnicas de programación generativas para la obtención del código ejecutable final y facilita el diseño de diversas interfaces de usuario para la definición y modificación de operadores complejos con independencia del lenguaje empleado por el usuario. Ejemplo de esta independencia es que un operador complejo puede ser definido mediante un editor gráfico proporcionado por defecto por Eclipse, o mediante el editor textual implementado en MOMENT que proporciona una sintaxis más intuitiva.

Pero también cabe destacar que esta representación como instancia de un modelo, permite aprovechar el esfuerzo de terceros en el campo de la ingeniería dirigida por modelos en, por ejemplo, la creación semi-automática de lenguajes visuales específicos de dominio.

De esta manera, como trabajo futuro encontramos el desarrollo de un editor visual para la definición de operadores complejos aprovechando los esfuerzos del proyecto Graphical Modeling Framework (GMF) [9], que permite la generación de editores gráficos para la edición de modelos EMF mediante una metáfora gráfica específica de dominio.

Apéndice A

Gramática para la definición textual de operadores complejos.

```
1     moment-op:
2             [ import-decl-list ]
3             [ metamodels-decl ]
4             operator-decl
5     import-decl-list:
6             import-decl { import-decl }
7     import-decl:
8             #include < identifier > | #include " filename "
9     metamodels-decl:
10            metamodel identifier { , identifier } ;
11    operator-decl:
12            operator identifier ( formal-parameters-list ) : <
                 return-types-list > {
13                    [ statements-sequence ] }
14    formal-parameters-list:
15            formal-parameter-decl { , formal-parameter-decl }
16    formal-parameter-decl:
17            parameter-type identifier
18    return-types-list:
19            parameter-type { , parameter-type }
20    statement-sequence:
21            statement { statement }
22    statement:
23            < output-actual-parameters-list  > =
24                   identifier ( input-actual-parameters-list ) ;
25    input-actual-parameters-list:
26            input-actual-parameter { , input-actual-parameter }
27    input-actual-parameter:
```

```
28              constant | identifier
29      output-actual-parameters-list:
30              output-actual-parameter { , output-actual-parameter }
31      output-actual-parameter:
32              identifier
33      parameter-type:
34              String | Float | Rat | Int | Qid | Bool |
35                      TraceabilityMetamodel | Transformation |
                           identifier
```

Listado A.1: Gramática de definición de operadores

Apéndice B

Código generado para el operador de propagación de cambios del caso de estudio.

```
1
2     fmod PROPAGATECHANGES { MM2 :: TRIV ,  MM1 :: TRIV ,
         TraceabilityMetamodel :: TRIV ,  Transformation :: TRIV }
          is
3       *** Used operators imports
4       pr CROSS{ MM1 , TraceabilityMetamodel } .
5       pr RANGE{ TraceabilityMetamodel , MM1 , MM2 } .
6       pr DIFF{ MM1 , TraceabilityMetamodel } .
7       pr MODELGEN{ MM2 , TraceabilityMetamodel } .
8       pr MERGE{ MM2 , TraceabilityMetamodel } .
9       pr RESTRICTDOMAIN{ MM1 , TraceabilityMetamodel } .
10      pr COMPOSE{ TraceabilityMetamodel } .
11      pr MERGE{ TraceabilityMetamodel , TraceabilityMetamodel }
            .
12    *** Tuple of return types
13      pr TUPLE<2> { MM2 , TraceabilityMetamodel } .
14      *** Input parameters
15      var Uml : Set { MM1 } .
16      var UmlMd : Set { MM1 } .
17      var RdbMd : Set { MM2 } .
18      var MapUml2RdbMd : Set { TraceabilityMetamodel } .
19      var Uml2Rdbms : Transformation .
20      *** Operator declaration
21      op PropagateChanges : Set{MM1} Set{MM1} Set{MM2} Set{
           TraceabilityMetamodel} Transformation
```

```
22                  -> Tuple { MM2 , TraceabilityMetamodel } .
23        eq PropagateChanges ( Uml , UmlMd , RdbMd , MapUml2RdbMd ,
              Uml2Rdbms ) =
24          ( *** 1st output parameter: Result
25             p1(Merge(
26               Range(
27                 MapUml2RdbMd,
28                 p1(Cross(Uml,UmlMd)),
29                 RdbMd
30               ),
31               p1(ModelGen1(
32                 Uml2Rdbms;
33                 ? p1(Diff(
34                     UmlMd,
35                     p1(Cross(Uml,UmlMd))
36                   ))
37                 ? MM((empty-set).Set{MM2}
38               ))
39            ))
40          , *** 2nd output parameter: MapUmlMd2Result
41             p1(Merge(
42               Compose(
43                 RestrictDomain(
44                   p1(Cross(Uml,UmlMd)),
45                   MapUml2RdbMd),
46                 p2(Merge(
47                   Range(
48                    MapUml2RdbMd,
49                    p1(Cross(Uml,UmlMd)),
50                    RdbMd),
51                   p1(ModelGen1(  Uml2Rdbms;
52                          ? p1(Diff( UmlMd,
53                               p1(Cross(Uml,UmlMd))
54                          ))
55                          ? MM((empty-set).Set{MM2}
56                   ))
57                 ))
58               ),
59               Compose (
60                 p2(ModelGen1(
61                   Uml2Rdbms;
62                   ? p1(Diff( UmlMd,
63                        p1(Cross(Uml,UmlMd))
64                   ))
65                   ? MM((empty-set).Set{MM2}
66                 )),
67                 p3(Merge(
68                   Range(  MapUml2RdbMd,
69                      p1(Cross(Uml,UmlMd)),
```

```
70                      RdbMd
71                    ),
72                  p1(ModelGen1(  Uml2Rdbms;
73                         ? p1(Diff( UmlMd,
74                            p1(Cross (Uml,UmlMd))
75                         ))
76                         ? MM((empty-set).Set{MM2}
77                  ))
78                ))
79              )
80            ))
81          ) .
82    endfm
```

Listado B.1: Código generado para el caso de estudio

Apéndice C

Fichero de configuración del kernel de MOMENT.

```
<?xml version="1.0" encoding="ASCII"?>
<kernelconf:Configuration xmlns:kernelconf="http:///es.upv.
   dsic.issi/moment/kernel">
  <imports filename="101_PARAMETER.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="110_OCL-SUPPORT.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="121_DATATYPE.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="130_MODEL-SUPPORT.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="140_TUPLE-MODULES.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="150_SIGECORE.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="151_SPECORE.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="160_ECORE-REFLECTION-SUPPORT.maude">
    <importedIf>All</importedIf>
  </imports>
  <imports filename="161_ECORE-REFLECTION.maude">
```

```
28          <importedIf>All</importedIf>
29        </imports>
30        <imports filename="170_SIGTRACEABILITYMETAMODEL.maude">
31          <importedIf>All</importedIf>
32        </imports>
33        <imports filename="171_SPTRACEABILITYMETAMODEL.maude">
34          <importedIf>All</importedIf>
35        </imports>
36        <imports filename="181_SPTHESAURUS.maude">
37          <importedIf>All</importedIf>
38        </imports>
39        <imports filename="201_MOMENT-KERNEL.maude">
40          <importedIf>All</importedIf>
41        </imports>
42        <imports filename="301_EQUALS.maude">
43          <importedIf>OperatorMerge</importedIf>
44          <importedIf>OperatorSingleMerge</importedIf>
45          <importedIf>OperatorCross</importedIf>
46          <importedIf>OperatorDiff</importedIf>
47          <importedIf>OperatorMatch</importedIf>
48        </imports>
49        <imports filename="302_MODELGEN.maude">
50          <importedIf>OperatorMerge</importedIf>
51          <importedIf>OperatorSingleMerge</importedIf>
52          <importedIf>OperatorCross</importedIf>
53          <importedIf>OperatorDiff</importedIf>
54          <importedIf>OperatorModelGen</importedIf>
55        </imports>
56        <imports filename="310_MERGE.maude">
57          <importedIf>OperatorMerge</importedIf>
58        </imports>
59        <imports filename="311_SINGLEMERGE.maude">
60          <importedIf>OperatorSingleMerge</importedIf>
61        </imports>
62        <imports filename="320_CROSS.maude">
63          <importedIf>OperatorCross</importedIf>
64        </imports>
65        <imports filename="330_DIFF.maude">
66          <importedIf>OperatorDiff</importedIf>
67          <importedIf>OperatorEndogenousDomain</importedIf>
68          <importedIf>OperatorEndogenousRange</importedIf>
69        </imports>
70        <imports filename="340_MATCH.maude">
71          <importedIf>OperatorMatch</importedIf>
72        </imports>
73        <imports filename="350_MODELTOTEXT.maude"/>
74        <imports filename="501_SELECTTRACESSBYDOMAIN.maude"/>
75        <imports filename="502_SELECTTRACESBYRANGE.maude"/>
76        <imports filename="511_COMPOSE.maude">
```

```
77          <importedIf>OperatorCompose</importedIf>
78        </imports>
79        <imports filename="512_INVERT.maude"/>
80        <imports filename="601_ENDOGENOUSDOMAIN.maude">
81          <importedIf>OperatorEndogenousDomain</importedIf>
82        </imports>
83        <imports filename="602_ENDOUGENOUSRANGE.maude">
84          <importedIf>OperatorEndogenousRange</importedIf>
85        </imports>
86        <imports filename="651_EXOGENOUSDOMAIN.maude"/>
87        <imports filename="652_EXOGENOUSRANGE.maude"/>
88        <imports filename="710_REFRESHTRACEABILITYMODELDOMAIN.
             maude"/>
89        <imports filename="711_REFRESHTRACEABILITYMODELRANGE.maude
             ">
90          <importedIf>OperatorMerge</importedIf>
91          <importedIf>OperatorSingleMerge</importedIf>
92          <importedIf>OperatorCross</importedIf>
93          <importedIf>OperatorDiff</importedIf>
94          <importedIf>OperatorMatch</importedIf>
95          <importedIf>OperatorModelGen</importedIf>
96        </imports>
97        <imports filename="720_COMPLETEMODELREFERENCES.maude"/>
98      </kernelconf:Configuration>
```

Listado C.1: Fichero kernel.mkconf

Bibliografía

[1] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[2] Phillip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):55–63, 2000.

[3] A. Boronat, J. Pérez, J. Á. Carsí, and I. Ramos. Two experiences in software dynamics. *Journal of Universal Computer Science*, 10(4):428–453, 2004. `http://www.jucs.org/jucs_10_4/two_experiences_in_software`.

[4] Artur Boronat, José Iborra, José Ángel Carsí, Isidro Ramos, and Abel Gómez. Del método formal a la aplicación industrial en gestión de modelos: Maude aplicado a eclipse modeling framework. *Revista IEEE América Latina*, September 2005.

[5] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[6] Enrique José García Cota. *Guía práctica de ANTLR 2.7.2*. E.T.S. de Ingeniería Informática de la Universidad de Sevilla. Departamento de Lenguajes y Sistemas Informáticos., 2003. `http://www.lsi.us.es/~troyano/documentos/guia.pdf`.

[7] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[8] Eclipse Organization. Eclipse graphical editing framework, 2006. `http://www.eclipse.org/gef/`.

[9] Eclipse Organization. The graphical modeling framework, 2006. `http://www.eclipse.org/gmf/`.

[10] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. 1985.

[11] EMF. `http://download.eclipse.org/tools/emf/scripts/home.php`.

[12] Abel Gómez, Artur Boronat, Luis Hoyos, José Á. Carsí, and Isidro Ramos. Definición de operaciones complejas con un lenguaje específico de dominio en gestión de modelos. October 2006.

[13] Object Management Group. `http://www.omg.org`.

[14] Elwin Ho. Creating a text-based editor for eclipse 2.1. Technical report, Hewlett-Packard Development Company, L.P. `http://devresource.hp.com/drc/technical_white_papers/eclipeditor/EclipseEditor.pdf`.

[15] José Iborra. Prototipo de integración de una herramienta de gestión de modelos. Master's thesis, Universidad Politéncica de Valencia, 2005.

[16] Dragan Djurić J.M. Favreau Dragan Gašević Jean Bézivin, Vladan Devedžić and Frédéric Jouault. An m3-neutral infrastructure for bridging model engineering and ontology engineering. Geneva, Switzerland, feb 2005. Springer-Verlag.

[17] Stuart Kent. Model driven engineering. In *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002.

[18] Ivan Kurtev, Jean Bezivin, , and Mehmet Aksit. Technical spaces: An initial appraisal. *In Tenth International Conference on Cooperative Information Systems (CoopIS), Federated Conferences Industrial Track, California.*, 2002.

[19] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: a programming platform for generic model management. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 193–204, New York, NY, USA, 2003. ACM Press.

[20] Object Management Group. MDA Guide Version 1.0.1. 2003. `http://www.omg.org/docs/omg/03-06-01.pdf`.

[21] Object Management Group. MOF 2.0 QVT final adopted specification (ptc/05-11-01). 2005. `http://www.omg.org/cgi-bin/doc?ptc/2005-11-01`.

[22] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification (ptc/06-01-01), 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[23] Object Technology International, Inc. Eclipse platform technical overview, 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`.

[24] The Apache Jakarta Project. Velocity, 2006. `http://jakarta.apache.org/velocity/`.

[25] Terence Parr. The antlr parser generator, 2006. `http://www.antlr.org`.

[26] The ISSI Research Group. The MOMENT Project.

[27] The Maude System homepage. The Maude Project. `http://maude.cs.uiuc.edu`.