# UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN


# PROTOTIPO DE INTEGRACIÓN DE UNA HERRAMIENTA DE GESTIÓN DE MODELOS

**Proyecto final de carrera**
**Septiembre de 2005, Valencia**

**Realizado por:**
José Iborra

**Dirigido por:**
Artur Boronat Moll
Dr.. José A. Carsí Cubel

## Abstract

La gestión de modelos es una disciplina que persigue la manipulación genérica de modelos, entendidos como estructuras de datos que validan una relación "se ajusta a" con respecto a un vocabulario o metamodelo. Por ejemplo un modelo UML, un esquema relacional, un documento XML que valida un DTD o esquema XML determinado. La manipulación de estos artefactos, denominados modelos en este contexto, se realiza mediante una serie de operadores genéricos. Los operadores deben de ser aplicables a cualquier modelo sea cual sea su metamodelo. Además los operadores deben proporcionar la propiedad de composición, de tal forma que el resultado de una operación pueda ser utilizado como entrada de otra.

El paradigma de desarrollo de software basado en modelos (MDE, *Model Driven Engineering*) persigue realizar la descripción del sistema software mediante modelos simples. Realizando transformaciones sobre los modelos se llega a un nivel suficiente de refinamiento que permite obtener de forma automática el código de la aplicación. En MDE el ingeniero de software no se encarga de diseñar los modelos, tarea reservada para especialistas de dominio en el problema o sistema bajo estudio, sino que su responsabilidad se centra en el diseño de los procesos de transformación y refinamiento. De esta manera, la gestión de modelos constituye una aproximación genérica y reutilizable para dar soporte al problema de las operaciones con modelos que se presenta en MDE.

MOMENT es un sistema de gestión de modelos que se apoya en el formalismo de las especificaciones algebraicas, desarrollado mediante el sistema algebraico Maude. Este proyecto presenta la integración entre MOMENT y un entorno de trabajo (industrial) MDE como es EMF sobre la plataforma Eclipse.

# Tabla de contenidos

# Lista de figuras

# Lista de tablas

# Lista de ejemplos

# Capítulo 1. Introducción

El trabajo realizado en este proyecto se ubica en el contexto de un esfuerzo de investigación en torno a MOMENT(MOdel ManageMENT), un sistema de gestión de modelos en desarrollo actualmente que viene a proporcionar un conjunto de herramientas para facilitar, entre otras cosas, el avance tecnológico en la disciplina de Ingeniería guiada por modelos(MDE, *Model Driven Engineering*). Este trabajo está siendo realizado en el Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia[1].

Los esfuerzos de este proyecto han ido dirigidos a integrar MOMENT con un entorno de MDE industrial, de forma que los formalismos algebraicos empleados en MOMENT resulten transparentes al usuario final y que este usuario se mueva en todo momento en el ámbito de MDE. A lo largo de esta memoria se presentan con más detalle MOMENT, la gestión de modelos y el trabajo concreto desarrollado en este proyecto, así como todos los conceptos básicos necesarios para una adecuada comprensión.

## 1.1. Motivación

MDE es una innovación en el marco del desarrollo de aplicaciones software surgida en los últimos años para solucionar una serie de problemas en los procesos de desarrollo de software actuales, resultado de la evolución de los procesos tradicionales. MDE se desmarca de estos procesos tradicionales en varios sentidos. Principalmente, se considera a los modelos como ciudadanos de primer orden, utilizándolos para describir el problema o sistema bajo estudio. En segundo lugar,

[1] http://moment.dsic.upv.es

el papel jugado por el código fuente es mucho más reducido que en las aproximaciones tradicionales, ya que todo el código es generado automáticamente a partir de los modelos. Se habla entonces de compiladores de modelos.

Sin embargo la adopción de procesos MDE no está siendo tan rápida como cabría esperar, y es que a día de hoy MDE carece, tanto de un conjunto suficiente de estándares e implementaciones de dichos estándares, como de herramientas que faciliten el trabajo en MDE.

La gestión de modelos es una disciplina que persigue la manipulación genérica de artefactos software, entendidos como estructuras de datos que validan una relación "se ajusta a" con respecto a un vocabulario o metamodelo. Por ejemplo un modelo UML, un esquema relacional, un documento XML que valide un DTD o esquema determinado. La manipulación de este tipo de artefactos, denominados modelos[2] en este contexto, se realiza mediante una serie de operadores genéricos. Los operadores deben de ser aplicables a cualquier modelo sea cual sea su metamodelo. Además los operadores deben proporcionar la propiedad de composición, de tal forma que el resultado de una operación pueda ser utilizado como entrada de otra.

De esta manera, la gestión de modelos constituye una aproximación genérica y reutilizable para dar soporte al problema de las operaciones con modelos que se presenta en MDE. Mediante la gestión de modelos debería ser posible producir herramientas y metodologías que soporten y faciliten los procesos de trabajo en MDE.

MOMENT es una herramienta de gestión de modelos que se apoya en el formalismo de las especificaciones algebraicas, implementada sobre el sistema algebraico Maude[18]. En MOMENT los modelos de un metamodelo determinado se representan como términos de un álgebra, álgebra correspondiente al metamodelo. MOMENT permite trabajar con los modelos como si estos fueran conjuntos (en el sentido matemático de la palabra), proporcionando los operadores correspondientes como Unión, Intersección, Diferencia, etc. Se proporcionan también otras abstracciones características de la gestión de modelos, como las transformaciones de modelos y la relación de equivalencia que se puede definir entre dos modelos (denominada *mapping* en el argot de gestión de modelos). Estos

[2]Nótese que esta definición de modelo es más amplia que la proporcionada en el contexto de MDE

conceptos se estudiarán con más profundidad en la Sección 2.2, “La gestión de modelos”.

Para manipular los modelos, MOMENT requiere que sean convertidos o proyectados a su expresión algebraica. Meediante los apropiados mecanismos de conversión, es factible integrar el espacio tecnológico en el que se ubica MOMENT con el espacio tecnológico de una implementación dada de MDE. Evidentemente, cuanto más transparente sea esta integración, mayor será la difusión y aceptación del trabajo realizado. En cualquier caso, es deseable que el ingeniero de MDE pueda extraer los beneficios de la aplicación de los procesos de gestión de modelos ofrecidos por MOMENT sin tener que conocer el formalismo de las especificaciones algebraicas que subyace.

# 1.2. Objetivos

El objetivo perseguido es conseguir un prototipo que permita integrar los operadores de gestión de modelos definidos en MOMENT con un marco de trabajo MDE de forma transparente. El marco elegido para ello ha sido EMF(Eclipse Modelling Framework)[20]. La plataforma de desarrollo es Java, trabajando sobre la plataforma Eclipse[19]. A continuación se presenta un prototipo que cumple con dichos requisitos y se indican las interacciones de este proyecto con otros en el contexto de MOMENT.

## 1.2.1. Contexto del proyecto: MOMENT

Integrar MOMENT con EMF pasa por integrar primero Maude en Eclipse. Este primer paso es el objetivo del proyecto desarrollado en [15] de forma simultánea a éste.

El segundo paso consiste en proporcionar los mecanismos necesarios para interoperar EMF y MOMENT. Para llegar a esta meta se han desarrollado un conjunto de conversores o proyectores capaces de convertir los principales artefactos software que intervienen en los procesos de MDE entre ambos espacios tecnológicos, Maude y EMF.

El tercer y último paso consiste en proporcionar los mecanismos necesarios para invocar operaciones de MOMENT desde Eclipse. Esto se consigue mediante el

desarrollo de un conjunto de “plug-ins” que se montan sobre la plataforma Eclipse y proporcionan un conjunto de metáforas gráficas para la definición e invocación de operadores de gestión de modelos de MOMENT desde Eclipse.

En este proyecto han sido estudiados el segundo y tercer paso, que se presentan a continuación con más detalle incluyendo los requisitos que se deben satisfacer.

### 1.2.2. Interoperabilidad Maude - EMF

El principal objetivo de este módulo es proporcionar los mecanismos que permitan representar o proyectar modelos y metamodelos EMF en MOMENT, así como obtener el modelo EMF original a partir de su proyección en MOMENT.

Hay que tener en cuenta que el mecanismo de proyección de metamodelos es unidireccional, mientras que la proyección de modelos es bidireccional. Esto implica que el proceso de proyección de modelos debe ser capaz de recuperar toda la información que había en el modelo original, tanto si se parte desde Maude como EMF, aunque en la práctica el caso de interés es el que parte de EMF.

### 1.2.3. Invocación de operaciones de MOMENT desde Eclipse

El objetivo de este módulo es realizar la automatización de los pasos o procesos necesarios para invocar operaciones en MOMENT, y ofrecer una interfaz de usuario que permita hacerlo desde Eclipse. El subconjunto de funcionalidades de MOMENT que se debe soportar es el siguiente:

- Los creación de operadores compuestos utilizando los operadores básicos de teoría de conjuntos.

- La invocación de operaciones intramodelo (esto es, las operaciones en que todos los modelos son del mismo metamodelo).

No siendo requisito en esta primera versión del prototipo soportar el resto de características de MOMENT, como:

- El operador de transformaciones *ModelGen*, y los operadores que trabajan con *mappings*.
- Las operaciones extramodelo (esto es, operaciones con modelos de distintos tipos, o dicho de otra manera, operaciones que involucran a más de un metamodelo).
- La definición manual por parte del usuario de *mappings*.

## 1.3. Contribuciones

A lo largo de este proyecto se estima que se han completado las siguientes contribuciones:

- Se ha logrado integrar satisfactoriamente MOMENT con un entorno de trabajo con modelos muy utilizado como es EMF, cumpliendo todos los objetivos demandados. Gracias a este trabajo se da vía libre a la utilización de las capacidades de MOMENT en problemas aplicados.
- Se ha logrado ocultar los procesos de trabajo de MOMENT bajo un automatismo, consiguiendo que el usuario final no necesite conocer el formalismo algebraico subyacente.
- Se ha demostrado la validez y aplicación de procesos formales en el desarrollo de software moderno, cuando se acompañan de herramientas que faciliten la aplicación práctica, como son Eclipse y EMF.

## 1.4. Contenido

A continuación se presenta un breve resumen sobre la estructura de los contenidos que se quieren comunicar en esta memoria.

En primer lugar, se ha dado una introducción que presenta los objetivos del proyecto y proporciona la información del contexto en el que éste se ha desarrollado, junto con una breve introducción a la Gestión de Modelos y a MOMENT.

En el Capítulo 2 se estudia el problema de la integración de MOMENT con un entorno MDE industrial. En primer lugar se presenta el paradigma MDE contex-

tualizado para EMF; se estudia el vocabulario básico o metametamodelo de EMF, denominado *Ecore*, que permite definir metamodelos en EMF. A continuación se estudia en más detalle la gestión de modelos y MOMENT; se presenta también un breve resumen de las principales aproximaciones existentes en este campo. Finalmente, se hace una breve reseña sobre las facilidades utilizadas para realizar la implementación del prototipo.

En el Capítulo 3 se estudian en detalle las correspondencias entre MOMENT y EMF en los conceptos utilizados para representar modelos y metamodelos. La formalización de estas correspondencias permitirá luego realizar la implementación de los proyectores que faciliten la interoperabilidad entre ambos. En segundo lugar, se presenta el funcionamiento de MOMENT a nivel de usuario para realizar operaciones de gestión de modelos.

En el Capítulo 4 se presenta el proceso de análisis que se ha seguido para construir los proyectores y el módulo de interacción automática con MOMENT. Aquí se detalla como se ha abordado el problema de generación del código Maude que representa en MOMENT a los modelos y metamodelos. Se explica como se ha dado solución al problema inverso construyendo un compilador basado en la gramática abstracta para un subconjunto de la sintaxis de Maude. Finalmente, los procesos de usuario de MOMENT se analizan en profundidad para obtener un modelo que permite realizar de forma automática todos los pasos, aplicando los proyectores para ello.

El Capítulo 5 proporciona detalles relativos a la implementación, incluyendo la estructura de ficheros y de paquetes. Se estudia también com se ha realizado la integración de la aplicación en la plataforma Eclipse.

Finalmente, en el Capítulo 6 se resumen los objetivos alcanzados y se indican los trabajos futuros que se plantean para perfeccionar el trabajo realizado.

# Capítulo 2. Fundamentos

## 2.1. Ingeniería de modelos

### 2.1.1. Motivación

Es un hecho bien conocido que el desarrollo de software es un proceso altamente complicado, muy costoso, y con una gran proporción de fallos. Hoy en día existen lenguajes como Java o C# que permiten aumentar la productividad de los programadores, tecnologías como HTTP o CORBA que simplifican el desarrollo de sistemas distribuidos, etc. Sin embargo, la complejidad de los sistemas construidos ha aumentado de forma paralela a este avance en la tecnología de desarrollo de software, y a una velocidad notablemente mayor. La ingeniería guiada por modelos es una de las disciplinas que intentan proporcionar los mecanismos y métodos para manejar esta complejidad.

En el contexto de MDE, los modelos se consideran como el artefacto principal en el proceso de desarrollo. Se utilizan varios modelos para describir un sistema informático, con el objetivo de mejorar los procesos tradicionales de producción de software automatizando gran cantidad del trabajo, gracias al superior nivel de abstracción proporcionado por los modelos.

Según este sistema, los desarrolladores describen el sistema desde el espacio del problema utilizando modelos. Mediante diversos procesos de refinamiento, estos modelos se trasladan al espacio de la solución mediante operaciones, de una forma semiautomatizada.

MDE permite o intenta aportar en los siguientes aspectos de la producción de software: productividad del proceso de desarrollo, portabilidad del sistema a

otras tecnologías, interoperabilidad del sistema con otras tecnologías, y finalmente documentación y mantenimiento del sistema.

### 2.1.2. Modelos y Metamodelos

En el contexto de la ingeniería de modelos (MDE, Model Driven Engineering), un modelo está constituido por un conjunto de elementos que proporcionan una descripción sintética y abstracta de un sistema, concreto o hipotético. En la elaboración de un modelo los procesos mentales que deben intervenir son principalmente la abstracción y la clasificación. Mediante la abstracción se eliminan todos los detalles del sistema bajo estudio que no son relevantes en el modelo, y mediante la clasificación se busca agrupar los elementos según sus propiedades, aún cuando en el sistema bajo estudio no se hallen agrupados de la misma manera. El objetivo último de un modelo es la transmisión de conocimientos entre personas, y para ello no debe superar un nivel adecuado de complejidad. Si es necesario, es posible utilizar varios modelos para describir un sistema bajo estudio. Por ello se dice que los modelos funcionan a un nivel de abstracción superior que un programa, cuya función es la especificación absoluta del sistema.

De igual manera que en el contexto de la programación existen diferentes lenguajes de programación, y se utiliza uno u otro según el problema bajo estudio, en la ingeniería de modelos existen diferentes metamodelos. Un metamodelo proporciona un vocabulario para definir modelos, es por lo tanto un vocabulario de modelos o un lenguaje de programación de modelos. El símil entre metamodelo y lenguaje de programación no es completo y es necesario resaltar una diferencia importante: un metamodelo es también un modelo, mientras que no se puede decir que un lenguaje de programación es un programa.

### 2.1.3. MOF

La OMG ha propuesto, ya no tan recientemente, un marco de trabajo en el ámbito de la ingeniería de modelos denominado MDA (Model Driven Architecture). MDA se perfila como la apuesta de la OMG para conseguir 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 voca-

bularios, es decir nuevos metamodelos, 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 cual se sitúan los datos, es decir las instancias del sistema bajo estudio.

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

Resulta interesante destacar que esta asignación fija de niveles puede resultar confusa en ocasiones, y 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 y por lo tanto es relativa. Esta relación se denomina informalmente "relación instancia-de". Decimos que un modelo `x` es una instancia de un vocabulario `y` 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 `y`ya que si ahora lo situamos en el nivel instancia, vemos que también `y`, necesariamente, está definido por un vocabulario `z`. 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 (*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 sobre modelos.

- QVT (*Query-View-Transformation*), un vocabulario que extiende OCL para expresar relaciones entre modelos

### 2.1.4. EMF

EMF(Eclipse Modelling Framework), es una aproximación MDE parcial sobre la plataforma Eclipse, desarrollada por la fundación Eclipse. A diferencia de MDA, que es una especificación o un conjunto de especificaciones que deben ser implementadas, EMF es una implementación concreta[1]. El núcleo de EMF es Ecore, un vocabulario basado en MOF[2] que se ubica en el nivel M3 y se utiliza para definir metamodelos EMF.

EMF proporciona entre, otras cosas, un sistema de persistencia basado en XMI[3] y un editor gráfico para crear modelos Ecore (nivel M2). Además se facilitan, o existen como parte de otro proyectos, una serie de puentes que facilitan la inte-

[1] Quizás sería más correcto decir que EMF es un “puente” Ecore-Java, donde Ecore es un subconjunto de MOF, y por lo tanto se puede decir que EMF es un “puente” parcial MOF-Java.
[2] Concretamente, Ecore es un subconjunto de MOF
[3] XMI *(XML Metadata Interchange)* es un estándar OMG para el intercambio de modelos vía XML.

roperabilidad con otras tecnologías, como UML 1.3, UML 2.0[28], XML Schema[29]. Además, gracias a la buena aceptación a nivel industrial y académico de EMF, continuamente aparecen nuevos puentes con otras tecnologías, por ejemplo con el campo de las ontologías en [30], etc.

EMF constituye una base ideal para el desarrollo de proyectos MDE.

#### 2.1.4.1. Ecore

Gracias a Ecore, es posible definir los vocabularios locales de dominio, denominados metamodelos, que permiten la creación o modificación de modelos en distintos contextos.

Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de metamodelos. Para ello Ecore proporciona elementos útiles para describir conceptos y las relaciones entre ellos. El diagrama de clases UML de la Figura 2.1, "Características del metamodelo Ecore" muestra un subconjunto de Ecore. 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` | Tipo que modela el concepto de tipo básico de dato |
| `EAttribute` | Tipo que permite definir los atributos de una clase. 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. |

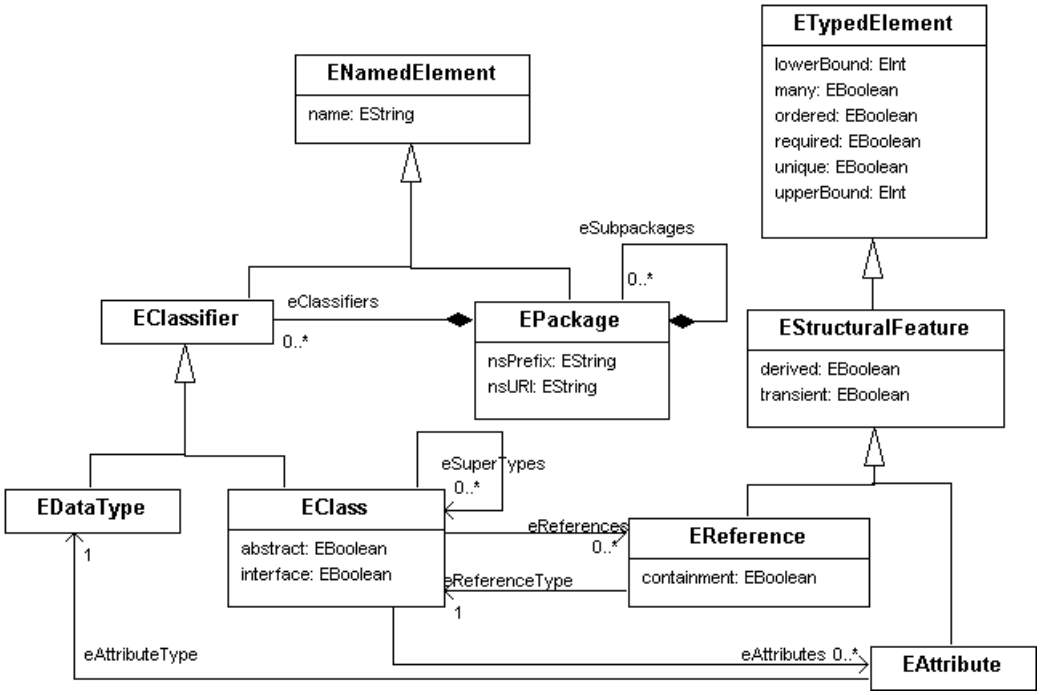



**Figura 2.1. Características del metamodelo Ecore**

`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` `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.

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.

Para evitar confusiones cabe mencionar que en la documentación de EMF [1] se utiliza la expresión *core model* 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`

Una vez se ha definido un metamodelo mediante Ecore, la aplicación más directa y evidente es la creación de modelos. Por ejemplo, se podría haber definido un metamodelo navigacional para representar flujos de información, y se quiere aplicar a un problema de diseño de un sitio web, para modelar los flujo de navegación de los visitantes. EMF permite generar de forma automática el código Java completo de un editor gráfico sencillo a partir de la definición de un metamodelo. Si bien es cierto que el editor generado puede no ser adecuado en todos los casos, sí que posibilita el ciclo completo de trabajo en MDE con metamodelos y modelos.

## 2.2. La gestión de modelos

### 2.2.1. Motivación

En un proceso de ingeniería MDE se parte de un conjunto de modelos que describen el sistema de interés de manera abstracta. A partir de estos modelos, y mediante una serie de procesos de refinamiento y transformación, se pretende obtener de manera automática el artefacto software ejecutable final. Es en estos procesos donde se centra el trabajo del ingeniero MDE. Mientras que los modelos

serán creados por analistas o especialistas de dominio, el ingeniero MDE debe encargarse de establecer los denominados *mappings* o relaciones de transformación que permitirán refinar los modelos originales, produciendo como resultado el sistema informático requerido en la tecnología de implementación deseada. Con tan sólo sustituir estos *mappings* es posible obtener el sistema en otra tecnología de implementación.

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

Uno de los principales problemas encontrados es la falta de infraestructuras y herramientas de soporte que permitan, no sólo crear y trabajar con estos *mappings*, sino manipular modelos en general. En el contexto de los lenguajes de programación estamos acostumbrados a una gran variedad de lenguajes, potentes entornos integrados de programación, herramientas para gestionar versiones del código y otras facilidades que automatizan gran parte del trabajo. En el contexto de la ingeniería de modelos ocurre todo lo contrario (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 *mappings* entre modelos, modificar modelos o *mappings*, generar un modelo a partir de otro a partir de un *mapping* o generar la representación equivalente de un modelo en otro metamodelo.

Evidentemente este modelo 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

El objetivo de la disciplina de la gestión de modelos es proporcionar una infraestructura específica y productiva para trabajar con los procesos de transformación y refinamiento de modelos, de forma genérica y reutilizable; genérica en el sentido de que las herramientas proporcionadas sean aplicables a cualquier vocabulario (metamodelo), y entre vocabularios; 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 [16], algunos ejemplos son :

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

### 2.2.2. MOMENT

MOMENT [13] es un sistema de gestión de modelos que se apoya en el formalismo de las especificaciones algebraicas. La característica fundamental de MOMENT es la representación utilizada para los modelos y metamodelos.

Un metamodelo se representa como un álgebra cuya signatura se compone de dos partes: un conjunto de sorts[4] que denota a los tipos del metamodelo (los elementos que componen el metamodelo), y un número de operadores que permiten manipular términos de estos sorts. La segunda parte de la signatura, esto es, los operadores, se define de forma genérica y reutilizable para cualquier metamodelo.

Un modelo se representa en MOMENT como un conjunto de términos algebraicos, en el sentido matemático de la palabra conjunto. Estos términos algebraicos, que representan a los elementos del modelo, son susceptibles de ser manipulados mediante los operadores provenientes del álgebra de su metamodelo. MOMENT facilita los operadores clásicos de teoría de conjuntos. Algunos ejemplos de operadores básicos en MOMENT son:

- El operador binario *Merge*, que permite realizar la unión disjunta de dos modelos.
- El operador binario *Diff*, que permite obtener la diferencia
- El operador binario *Intersection*, para obtener la intersección
- El operador unitario *SetPreferred*, que se utiliza para solucionar conflictos en caso de que aparezcan elementos repetidos (equivalentes) en una operación con varios modelos. *SetPreferred* permite marcar cuál es el modelo que tiene la prioridad.

Otro operador básico en MOMENT, que proviene en este caso de la disciplina de gestión de modelos, es el operador unitario *ModelGen*. La aplicación de *ModelGen* sobre un modelo m del metamodelo A, especificando como destino el metamodelo B, obtiene la representación equivalente del modelo m en el vocabulario definido por el metamodelo B. Por ejemplo, mediante ModelGen se

[4]Para más detalles sobre el formalismo de las especificaciones algebraicas se recomienda acudir a [10]

puede obtener el esquema entidad-relación (metamodelo relacional) correspondiente a un modelo UML.

La aplicación de cualquiera de estos operadores produce un modelo resultado y además, a excepción de *SetPreferred*, un conjunto de *mappings* o modelos de trazabilidad que contienen de manera explícita las correspondencias entre cada uno de los modelos origen y el modelo resultado. MOMENT proporciona un conjunto de operadores básicos que permiten realizar operaciones con ellos:

- El operador binario *Compose*, que permite componer dos *mappings*.
- Los operadores binarios *Range* y *Domain*, que toman un modelo y un *mapping* y permiten seleccionar los elementos del modelo que participan en el *mapping*.

En MOMENT los operadores de gestión de modelos comentados anteriormente han sido especificados algebraicamente utilizando el formalismo Maude[18]. 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.

#### 2.2.2.1. Maude

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

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

- Maude soporta de una manera sistemática y eficiente la reflexión lógica. Esto permite a Maude ser un lenguaje extremadamente potente y extensible, permitiéndole soportar un álgebra de operaciones de composición de módulos extensible, además permite muchas aplicaciones avanzadas de metaprogramación y metalenguaje.

- Maude es potente. Puede modelar casi todo, desde el conjunto de los números naturales hasta un sistema biológico donde se programe el lenguaje Maude a sí mismo. Cualquier cosa que se pueda escribir, hablar o describir mediante el lenguaje humano, se puede expresar con instrucciones Maude.

- Maude es simple. Su semántica está basada en los fundamentos de la teoría de clases, el cual es bastante intuitivo y directo, hasta el punto que un matemático puede describirla formalmente con símbolos y letras griegas. Comparado con la mayoría de los lenguajes, Maude no tiene mucha sintaxis que memorizar, solo un pequeño conjunto de palabras claves y símbolos, y algunas directivas generales y convenciones a seguir.

- Maude está bien establecido. O, depende de cómo se mire, puede llegar a ser extremadamente abstracto. Su diseño permite tanta flexibilidad que la sintaxis puede parecer más bien abstracta. Sin embargo, la mayoría de los lenguajes de programación, distancian al programador del modelo a implementar con infinitud de palabras claves y sintaxis cuyo diseño y protocolo está más o menos oculto al usuario. En el modelo Maude, está tan embellecido cada objeto gracias a que cada uno de ellos se encuentra observado en todo momento por la mirada del programador, es por ello que la mayoría de los modelos han sido probablemente actualizados por los usuarios.

- Maude es desafiante. Puede llegar a solucionar lo más difícil o complejo. Esto desafía al programador a ser astuto, en vez de resolver el problema afrontándolo con una serie de variables globales y funciones que a menudo son un caos de por sí.

Aunque Maude es un intérprete, su rendimiento es tal que puede ser usado en aplicaciones serias con un rendimiento competitivo y muchas ventajas sobre código convencional. Ilustramos esto con un ejemplo. Un componente, que se usaba para probar si una traza de eventos satisfacía cierta fórmula dada en lógica lineal temporal (LTL), fue escrito en Maude para un proyecto de la NASA. El componente tenía una prueba trivial de corrección, ocupaba apenas una página y fue desarrollado en unas horas. Este componente reemplazó un componente similar escrito en Java que tenía aproximadamente 5000 líneas y que llevó más de un mes para ser desarrollado por un programador con experiencia. El código Java traducía a formula LTL en un autómata de büchi y era aproximadamente tres veces más lento que el código Maude.

La implementación actual de Maude puede ejecutar reescrituras sintácticas con velocidades típicas de medio millón a varios millones de reescrituras por segundo, dependiendo de la aplicación particular y la máquina en la que funcione (la estimación anterior supone un Pentium a 900Mhz). La razón por la cual el intérprete de Maude consigue alto rendimiento es que las reglas de reescritura son cuidadosamente analizadas y semicompiladas mediante algoritmos muy eficientes.

Se llama Core Maude al intérprete de Maude 2.0 implementado en C++ y que provee toda la funcionalidad básica del lenguaje. Full Maude es una extensión de Maude, escrito en Maude, que dota a Maude de un potente y extensible álgebra de módulo en el cual los módulos de Maude pueden ser combinados conjuntamente para construir módulos más complejos. Los módulos pueden ser parametrizados, y pueden ser instanciados usando los "views" (vistas). Los parámetros son teorías especificando los requerimientos semánticos para una correcta instanciación. Las teorías mismas pueden ser parametrizadas. Módulos orientados a objetos (que también pueden ser parametrizados) soportan objetos, mensajes, clases y herencia. También es posible subir y bajar en la torre de reflexión usando comandos de Full Maude.

### 2.2.3. Otras aproximaciones existentes

RONDO [16] es el sistema de gestión de modelos desarrollado por P. Bernstein. En RONDO los modelos se representan en forma de grafos dirigidos, y se facilitan un conjunto de operadores de alto nivel para manipularlos, similares a los descritos en el apartado La gestión de modelos. La traducción de instancias de modelos como grafos se hace mediante unos conversores especiales, desarrollados para cada metamodelo en concreto. Los operadores de manipulación están implementados en RONDO mediante tecnología imperativa.

AMMA / ATLAS[22] es una iniciativa de gestión de modelos basada en el lenguaje de transformación ATL, un prototipo para la transformación de modelos del que ya existe una implementación sobre Java compatible con EMF. Sin embargo los esfuerzos de gestión de modelos están aún en una fase temprana y es demasiado pronto ara realizar valoraciones.

## 2.3. Ingeniería de Software

### 2.3.1. Eclipse

Eclipse es la plataforma de desarrollo elegida para construir el prototipo.

Eclipse es una plataforma moderna para la integración y desarrollo de aplicaciones cliente. Esencialmente Eclipse es un programa escrito en Java, de código abierto y licencia no restrictiva, con gran compatibilidad en todos los S.O. modernos y aceptación por parte de usuarios e industria. Una visión más arquitectónica de Eclipse lo describiría como un contenedor de componentes que ofrece una serie de funcionalidades básicas a éstos, como configuración, interfaz gráfica, interacción con otros componentes, actualizaciones automáticas de componentes, etc.

### 2.3.2. Java 5

Java 5 es el lenguaje utilizado en la implementación del prototipo. Java 5 es la última encarnación del conocido lenguaje de programación, extendido en esta versión con nuevas construcciones sintácticas y nuevas librerías base.

# Capítulo 3. Interoperabilidad EMF-Maude

## 3.1. Espacios Tecnológicos

El concepto de espacios tecnológicos fue introducido en [14] 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. 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.

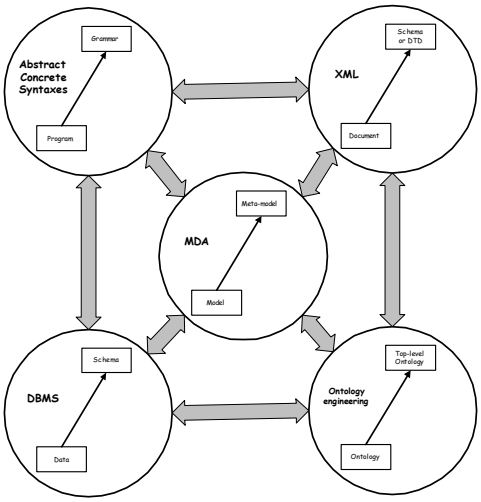



**Figura 3.1. Espacios Tecnológicos**

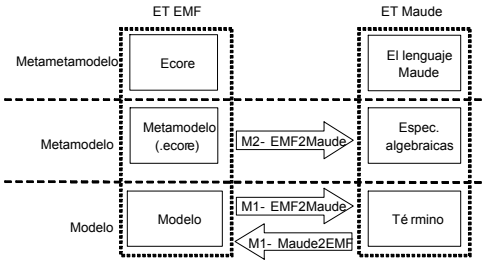



**Figura 3.2. Puentes Tecnológicos**

En MOMENT los operadores de gestión de modelos [16] han sido especificados algebraicamente utilizando el formalismo Maude. El ET de Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET también puede ser visto como un paradigma de modelado , considerando el álgebra universal de Maude como el lenguaje de definición de metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

MOMENT representa un modelo como una estructura de términos algebraicos, caracterizados por una especificación algebraica que proviene del metamodelo. Para poder utilizar MOMENT desde la ingeniería de modelos será necesario

disponer de unos puentes tecnológicos entre ambos espacios tecnológicos. El primer problema que este proyecto resuelve es la definición y construcción de estos puentes tecnológicos entre el ET de Maude y el ET de EMF.

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

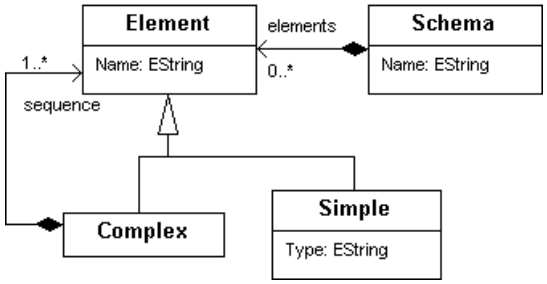



**Figura 3.3. Metamodelo miniXSD**

A lo largo del capítulo se utiliza un metamodelo simple, denominado **miniXSD** (ver Figura 3.3, “Metamodelo miniXSD”) como ejemplo ilustrativo. **miniXSD** es una versión simplificada del metamodelo para XML Schema [27]. Un esquema XML está compuesto por elementos, que pueden ser simples o complejos; un elemento complejo se define como una agregación de elementos; nótese que es un tipo recursivo. Aunque no aparece en la figura, estas clases están definidas en un EPackage de nombre “miniXSD”, prefijo “mXSD” [1] y cuya URI es “http://es.upv.dsic/issi/moment/miniXSD”

[1]Nótese que el prefijo y el nombre no tienen por qué ser distintos. De hecho en este ejemplo eran iguales, pero para distinguir mejor su uso se han cambiado

## 3.3. Puente EMF -> Maude a nivel M2

A continuación se presenta en detalle la especificación del puente unidireccional entre EMF y Maude a nivel M2. En esta sección se utilizan por primera vez patrones de correspondencia con la finalidad de indicar como se proyectan los elementos de un metamodelo EMF en Maude. *Lamentablemente no existe ninguna notación estándar para expresar la generación de artefactos texto a partir de un modelo; es por ello que los patrones aquí presentados utilizan una notación ad-hoc sencilla.*

El puente EMF->Maude nivel M2 se utiliza para abastecer de metamodelos a MOMENT. Así pues, la función de este puente es obtener una especificación algebraica de un metamodelo, que en este contexto es un modelo Ecore.

La especificación algebraica que proyecta a un metamodelo está compuesta por tres módulos Maude:

| | |
|---|---|
| *Una signatura* | Módulo que contiene las signaturas de los constructores para todas las clases no abstractas del metamodelo.<br><br>Las signaturas, como veremos más adelante, sirven para crear instancias o *términos* como se denominan en el ET Maude, del metamodelo. |
| *Una vista o parámetro formal* | Este módulo se utiliza en el proceso de parametrización. |
| *Un módulo de operaciones auxiliares y axiomas específicos.* | El tercero de los módulos contiene operaciones auxiliares, como los selectores de atributos, que son utilizadas por MOMENT para la manipulación de modelos. Contiene además los axiomas específicos de metamodelo proporcionados por el usuario. |

Una vez expuesta la composición básica de una especificación algebraica, es importante identificar los patrones de correspondencia con el ET EMF, utilizando para ello el ejemplo de miniXSD.

## 3.3.1. Módulo Signatura

El objetivo del módulo signatura es declarar la lista de tipos que componen el metamodelo. En el Ejemplo 3.1, "Signatura de miniXSD" se puede observar el aspecto del módulo signatura que representa al metamodelo miniXSD. Está compuesto por los elementos siguientes:

❶ La declaración y el nombre del módulo.

❷ Declaración de los módulos importados por este módulo, indicada por la palabra clave **pr**

❸ Enumeración de sorts que componen la signatura.

❹ Relaciones de subsorts entre los sorts.

❺ Signatura de los constructores, una para cada sort. Un constructor es una operación en Maude, indicada por la palabra clave **op**, seguida del nombre y los parámetros[2], representados por el símbolo _. A continuación viene la relación de tipos de los parámetros, y finalmente el tipo del constructor. Por ejemplo, el constructor `miniXSD-Schema` toma tres parámetros, de tipo Qid, String y Set, y en conjunto el constructor tiene el tipo miniXSD-Schema

A continuación se comentan en detalle los patrones de generación de código que permiten obtener de forma totalmente automática el módulo signatura de un metamodelo EMF

### 3.3.1.1. Declaración del módulo signatura

La declaración incluye el nombre del módulo y la lista de dependencias (módulos a importar). Todos los módulos de signatura importan el módulo DATATYPE. La declaración de un módulo en Maude contextualizada para este caso tiene el siguiente aspecto:

```
fmod sigminiXSD is
  pr DATATYPE .
```

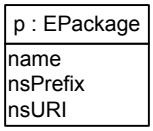
[2]Esta afirmación no es del todo exacta pero resulta una simplificación adecuada

### Ejemplo 3.1. Signatura de miniXSD

```
fmod sigminiXSD is ❶
  pr DATATYPE . ❷

  sorts mXSD-Schema mXSD-Element mXSD-Simple mXSD-Complex mXSDNode .  ❸
  subsorts mXSD-Schema mXSD-Element < miniXSDNode . ❹
  subsorts mXSD-Simple mXSD-Complex < mXSD-Element .

  *** op mXSD-Schema: Qid, elements,
  op `(mXSD-Schema___`( : Qid String OrderedSet{Qid} -> mXSD-Schema [ctor] . ❺

  *** op mXSD-Simple: Qid, Name, Type,
  op `(mXSD-Simple___`( : Qid String -> mXSD-Simple [ctor] .

  *** op mXSD-Complex: Qid, Name, sequence,
  op `(mXSD-Complex___`( : Qid String OrderedSet{Qid} -> mXSD-Complex [ctor] .
```

El nombre del módulo se obtiene añadiendo el prefijo “sig” al nombre del `EPackage`.

#### Tabla 3.1. Plantilla 1: Módulo signatura

| **NOMBRE** |
|---|
| Módulo signatura |
| **DESCRIPCION** |
| Permite obtener la declaración del módulo signatura de la especificación algebraica correspondiente a un metamodelo EMF determinado. El contenido de dicho módulo se especifica a lo largo de esta sección. |
| **ENTRADA** |
|  |
| **SALIDA** |
| `fmod sig<p.name>❶ is`<br>`  pr DATATYPE .`<br>`  Contenido del módulo signatura❷`<br>`endfm` |

**Nota**

❶ El texto en cursiva entre las llaves (<) y (>) indica un símbolo que debe ser sustituido por su valor. Se emplea la notación `clase.miembro` con el punto(.) para acceder a miembros de una clase.

❷ El texto en cursiva sin llaves es equivalente. Las llaves se utilizan para desambiguar cuando la frase lo requiere.

**Ejemplo.**

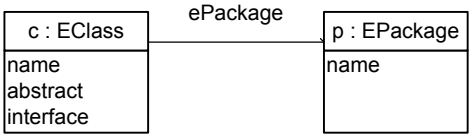
Para el metamodelo miniXSD el resultado es:

```
fmod sigminiXSD is
  pr DATATYPE .
    .....
    .....
endfm
```

### 3.3.1.2. Lista de sorts de la signatura

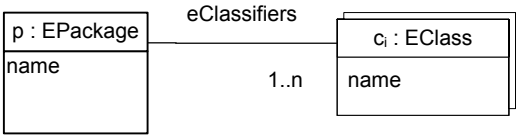
Una clase en el metamodelo da lugar a un sort en la signatura. El nombre del sort se obtiene a partir de la concatenación del prefijo del EPackage con el nombre de la clase utilizando como separador el guión (-).

**Tabla 3.2. Plantilla 2: Sort correspondiente a un tipo**

| **NOMBRE** |
|---|
| Sort correspondiente a un tipo |
| **DESCRIPCION** |
| El sort se obtiene a partir de la concatenación del prefijo del EPackage con el nombre de la instancia de EClass, utilizando como separador el carácter - . |
| **ENTRADA** |
|  |
| **SALIDA** |
| *<p.prefix>-<c.name>* |

**Sort adicional "Node" .** Además de los sorts provenientes del metamodelo, MOMENT requiere que se añada un sort artificial que es el representante abstracto de todos los sorts del metamodelo. El nombre de este sort artificial se obtiene añadiendo el sufijo Node al prefijo del EPackage. En el ejemplo de miniXSD el sort que se obtendría es `mXSDNode`

**Tabla 3.3. Plantilla 3: Lista de sorts de la signatura de un metamodelo**

| **NOMBRE** |
|---|
| Lista de sorts de la signatura de un metamodelo |
| **DESCRIPCION** |
| La lista de sorts se compone a partir de los sorts de todas las instancias de EClass que componen el metamodelo, separados por espacios; el orden no es importante. Adicionalmente se requiere un sort artificial cuyo nombre se obtiene al añadir el sufijo Node al final del prefijo del EPackage |
| **ENTRADA** |
|  |
| **ALGORITMO** |
| Sea $sort_n$ el sort correspondiente a la clase $c_n$ obtenido mediante la Plantilla 2 |
| **SALIDA** |
| `sorts` *sort₁ sort₂ ... sortₙ* `<p.prefix>Node .` |
| **DEPENDENCIAS** |
| la Plantilla 2 |

**Ejemplo.**

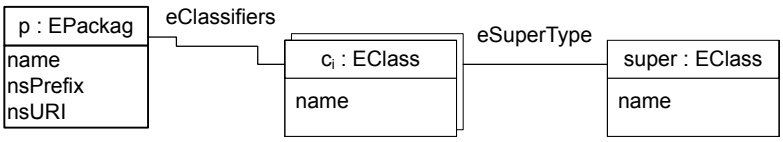
La declaración completa de sorts obtenida para miniXSD es:

```
sorts mXSD-Schema mXSD-Element mXSD-Simple mXSD-Complex mXSDNode .
```

### 3.3.1.3. Relación de subsorts

La relación de subsorts se obtiene a partir de las relaciones de herencia en el ET EMF. Cada jerarquía de herencia en el metamodelo da lugar a una jerarquía de subsorts. Adicionalmente todos los sorts del metamodelo han de ser subsorts del sort adicional “Node”.

**Tabla 3.4. Plantilla 4: Jerarquías de herencia de tipos de un metamodelo**

| **NOMBRE** |
|---|
| Jerarquías de herencia de tipos de un metamodelo |
| **DESCRIPCION** |
| Para obtener la relación de subsorts del módulo signatura, es necesario disponer previamente de las mencionadas jerarquías de herencia del metamodelo. Aquí se describe el algoritmo que las produce.<br><br>Hay que tener en cuenta que MOMENT requiere que todos los sorts del metamodelo sean subsorts del Sort adicional “Node”, para esto se introduce una jerarquía adicional que incluye todos los tipos 'libres', es decir sin supertipos, como subtipos de este sort artificial. |
| **ENTRADA** |
|  <br>Las clases que componen el metamodelo y las relaciones de herencia entre ellas |
| **ALGORITMO** |
| *Sea clases el conjunto de clases del metamodelo:*<br>clases = p.eClassifiers()<br><br>*Las jerarquías de herencia se definen como:*<br>jerher={<c1,c2> \| c1 ϵ clases, c2 ϵ clases, c1 ϵ c2.eSuperTypes()}<br>superclass : jerher -> EClass<br>subclasses : jerher -> {EClass}<br>∀jh ϵ jerher, jh = <superclass(jh),subclasses(jh)><br><br>*Se define top_sorts, la lista de clases que no tienen supertipo en el metamodelo:*<br><br>Sea sup : EClass -> {EClass} -- *La lista de supertipos locales*<br>sup(c) = {s\|s ϵ c.eSuperTypes, s.ePackage == c.ePackage} |

```
 en
  top_sorts = {c|c ϵ clases, sup(c)={}}

Mediante top_sorts se define la jerarquía de herencia adicional:
 jerherNode = <node,top_sorts>
  donde node = <p.prefix>Node

Que coloca al sort artificial en lo más alto de la jerarquía de tipos del
metamodelo.
```

**SALIDA**

El algoritmo proporciona las jerarquías de herencia así como la jerarquía adicional.

Mediante la Plantilla 4 se obtienen las relaciones de subsorts en Maude.

**Tabla 3.5. Plantilla 5: De jerarquía de herencia a relación de subsort**

| **NOMBRE** |
|---|
| De jerarquía de herencia a relación de subsort |
| **DESCRIPCION** |
| Permite obtener la relación de subsort que modela en Maude una jerarquía de herencia en un metamodelo EMF |
| **ENTRADA** |
| Una jerarquía de herencia de la forma <superclase, {subclases}> |
| **ALGORITMO** |
| Sea $sort_n$ el sort correspondiente a la subclase N, y sorts el correspondiente a la superclase, ambos obtenidos mediante la aplicación de ver la Plantilla 2 |
| **SALIDA** |
| subsorts $sort_1$ $sort_2$ ... $sort_n$ < sortS . |
| **DEPENDENCIAS** |
| la Plantilla 2<br><br>la Plantilla 4 |

**Ejemplo.**

En este caso vemos como a partir de la relación de herencia existente entre `Simple` y `Complex` con `Element`, se obtiene que `mXSD-Simple` y `mXSD-Complex` deben ser subsorts de `mXSD-Element` Además, los sorts de nivel superior, en este caso los correspondientes a `Schema` y a `Element`, son subsorts del sort artificial `miniXSDNode`. El resultado completo es:

```
subsorts mXSD-Schema mXSD-Element < miniXSDNode .
subsorts mXSD-Simple mXSD-Complex < mXSD-Element .
```

#### 3.3.1.4. Signatura de los constructores

La especificación algebraica debe proveer un constructor para cada EClass no abstracta del metamodelo. Un constructor es una operación en Maude que crea un término de un sort. La signatura de una operación en Maude tiene el siguiente aspecto:

```
op nombre _ _ _ _: tipo1 tipo2 tipo3 tipoN -> tipoOp .
```

Tras la palabra clave "op" viene la declaración sintáctica de la operación, delimitada por el carácter :, donde el carácter _ indica un punto de aparición de un operando. Finalmente se declaran los tipos de los parámetros y el tipo de retorno de la operación. Por ejemplo, la suma aritmética se definiría en Maude como:

```
op _+_ : Int Int -> Int .
```

En MOMENT las operaciones de los constructores tienen una definición sintáctica sencilla, con el nombre de la operación seguido de todos los parámetros de los que conste. Por lo tanto una operación de cuatro parámetros se declararía

```
op name _ _ _ _ : ..... -> ... .
```

O de forma equivalente, ya que los espacios son opcionales:

```
op name ____: ..... -> ... .
```

Para indicar que una operación es un constructor, se añade la etiqueta " `[ctor]` " al final, con lo que resultaría:

```
op name ____: ..... -> ... [ctor] .
```

Los parámetros del constructor provienen de los atributos y referencias que posea la clase correspondiente, más un parámetro añadido que se utiliza para almacenar un identificador, como se verá más adelante. Por lo tanto como paso previo para poder componer la signatura del constructor, es necesario obtener los tipos Maude equivalentes para cada atributo y referencia de la EClass.

## Representación de las referencias de la clase

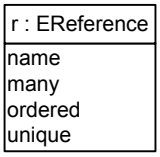
Veamos el caso más simple, correspondiente a las referencias. Una referencia, contextualizada en MDE, es el concepto que modela las asociaciones entre objetos. En el contexto de MOMENT los objetos se modelan como términos algebraicos. Para poder modelar las referencias se hace uso de identificadores únicos, representados como constantes del tipo

`Qid`

Así pues cada término que modele a un objeto debe tener un id único. Éste es el primer parámetro del constructor de un sort en una especificación algebraica en MOMENT.

Una referencia da lugar a un parámetro en el constructor cuyo tipo es el de los conjuntos de ids. MOMENT facilita distintos tipos de conjuntos para modelar los distintos tipos de referencias.

**Tabla 3.6. Plantilla 6: Representación de una EReference en MOMENT**

| **NOMBRE** |
| --- |
| Representación de una EReference en MOMENT |
| **DESCRIPCION** |
| Las referencias se representan en MOMENT como conjuntos de ids. Atendiendo a las características de ordenación y unicidad de la EReference se representa por un tipo de conjunto u otro |
| **ENTRADA** |
|  |
| **ALGORITMO** |
| Sea `tipo_conjunto()` la función que devuelve el tipo de conjunto asociado a una `EReference` (o a una `EStructuralFeature` en general). `tipo_conjunto` se define mediante la siguiente tabla: |

| r.ordered | r.unique | Maude |
|---|---|---|
| false | false | Bag |
| true | false | Sequence |
| false | true | Set |
| true | true | OrderedSet |

**SALIDA**

```
tipo_conjunto(r) {Qid}
```

## Representación de los atributos de la clase

En EMF un atributo tiene asignado un tipo de datos mediante la asociación *eType* de `EAttribute`. Ecore ofrece un conjunto de tipos de datos por defecto, por ejemplo `EString`, `EInt`, `EBoolean`, `EFloat`, etc. Además de esto, es posible crear nuevos tipos de datos mediante instancias de la metaclase `EDatatype`. En MOMENT un atributo da lugar a un parámetro en el constructor, cuyo tipo se deduce a partir del `EDatatype` asociado al atributo.

**Tabla 3.7. Plantilla 7: Tipo equivalente de un EAttribute**

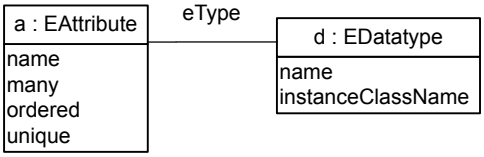
**NOMBRE**

Tipo equivalente de un EAttribute

**DESCRIPCION**

Los atributos se representan en MOMENT como datos. Si la cardinalidad es múltiple como una lista de datos, si es simple como un dato simple. El tipo se obtiene a partir de la asociación *eType*.

**ENTRADA**




**ALGORITMO**

Sea

```
tipos : {<String, String>}
```

una tabla, representada como lista de pares, que contiene las equivalencias de tipos (ver Apéndice C, *Correspondencia de tipos EMF <-> Ecore*).

Se define `tipo_eq`, la función que realiza la conversión de tipos:

```
tipo_eq : EAttribute -> String
tipo_eq att = [j|(i,j)<-tipos, i = att.eType.instanceClassName] | "String"
```

Donde si no se encuentra un tipo en la tabla, el tipo equivalente siempre será `String`.

**SALIDA**

```
if a.many = true
  tipo_eq(a)

else
  tipo_conjunto(a) { tipo_eq(a) }
```

Donde `tipo_conjunto` es la función definida en la Plantilla 6

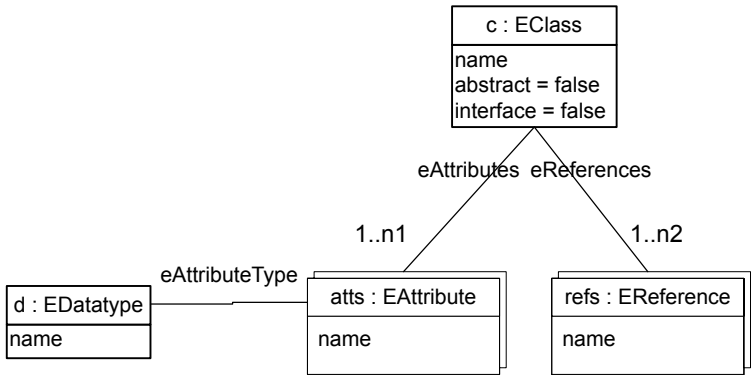
**DEPENDENCIAS**

la Plantilla 6 - Para obtener el tipo de conjunto si es un atributo múltiple.

## Signatura de los constructores

**Tabla 3.8. Plantilla 8: Signatura de los constructores**

| NOMBRE |
| --- |
| Signatura de los constructores |
| **DESCRIPCION** |
| La especificación algebraica debe proveer un constructor para cada tipo no abstracto del metamodelo. El nombre del constructor es el nombre del sort correspondiente al tipo (obtenido mediante la Plantilla 2). La signatura del constructor se construye a partir de las `EStructuralFeatures` de la `EClass` que representa el sort.<br><br>Los parámetros del constructor provienen de los atributos y referencias de la EClass. |
| **ENTRADA** |




**ALGORITMO**

Sea $sort_c$ el sort correspondiente a c

Sea `nparams = c.eAttributes.size() + c.eReferences.size()`

Sea $\{\_\}_m$ la repetición del símbolo '_' m veces.

Sea $tipo_{attn}$ el tipo obtenido para el atributo n. Idem para $tipo_{refn}$.

**SALIDA**

$sort_c$ `( $\{\_\}_{nparams}$ `( : Qid $tipo_{att1}$ $tipo_{att2}$ .. $tipo_{attn1}$ $tipo_{ref1}$ $tipo_{ref2}$ .. $tipo_{refn2}$ -> $sort_c$ [ctor].

**DEPENDENCIAS**

la Plantilla 2 - Para obtener el sort

la Plantilla 7 - Para obtener $tipo_{att}$

la Plantilla 6 - Para obtener $tipo_{ref}$

**Ejemplo.**

Para la clase `Schema`, se tiene un `constructor` que toma tres parámetros correspondientes a las tres *`Structural Features`* de la clase

```
op `( mXSD-Schema___ `( : Qid String OrderedSet{Qid} -> mXSD-Schema [ctor] .
```

## 3.3.2. Modulo de vista o parámetro formal

El módulo vista forma parte del mecanismo de parametrización . Tiene una extensión muy reducida, no más de tres líneas como en este caso, ya que se compone

**Ejemplo 3.2. Modulo Vista de miniXSD**

```
view vminiXSD❶ from TRIV to sigminiXSD❷ is
sort Elt to miniXSDNode❸ .
endv
```

únicamente de declaración. Los patrones de correspondencia resultan por tanto muy sencillos en este caso.

En el Ejemplo 3.2, “Modulo Vista de miniXSD” se muestra el módulo vista obtenido para el metamodelo miniXSD. Los elementos variables que en él aparecen son:

❶ El nombre del módulo se obtiene añadiendo el prefijo v al principio del nombre del EPackage
❷ El nombre del módulo signatura que define este metamodelo. Se obtiene lógicamente siguiendo las indicaciones dadas anteriormente
❸ El nombre del sort virtual “.....Node”, representante abstracto de todos los sorts de la signatura, que apareció en la especificación del módulo signatura.

La siguiente plantilla recoge las correspondencias con el metamodelo:

**Tabla 3.9. Plantilla 9: Módulo vista de la especificación algebraica**

| **NOMBRE** |
|---|
| Módulo vista de la especificación algebraica |
| **DESCRIPCION** |
| Permite obtener el módulo vista a partir de un EPackage |
| **ENTRADA** |
| p : EPackage<br>name<br>nsPrefix<br>nsURI |
| **SALIDA** |
| view v<*p.name*> from TRIV to sig<*p.name*> is<br>sort Elt to <*p.name*>Node .<br>endv |

### 3.3.3. Módulo de operaciones auxiliares y axiomas de usuario

Este módulo tiene una doble funcionalidad. Por un lado es el punto de introducción de axiomas específicos de usuario. Mediante el mecanismo de axiomas de usuario, MOMENT permite que se facilite información añadida acerca de un metamodelo para adaptar o retocar el comportamiento de los operadores de gestión de modelos en este metamodelo específico. Esta es una característica de MOMENT que no se explora en detalle en esta memoria. Para más información se recomienda consultar [12].

Por otro lado este módulo extiende la presentación axiomática del operador de selección '::', que permite seleccionar un atributo o referencia de una instancia, de manera equivalente al '.' en OCL o en los lenguajes orientados a objetos en general. Este operador forma parte del soporte proporcionado por MOMENT para trabajar con consultas OCL. Además del operador de selección, también se extiende la presentación algebraica del operador de actualización '->'. Gracias a que se puede extender la presentación axiomática de dichos operadores genéricos para adaptarla a un metamodelo específico, se evita trabajar con la definición genérica de los operadores, que utiliza técnicas de introspección y es por lo tanto varios niveles de magnitud más lenta. El soporte de OCL es otra característica de MOMENT relacionada con las transformaciones de modelos y el operador `ModelGen` que tampoco se explora en esta memoria. A pesar de ello, lo que sí se contempla es la generación automática de estas ecuaciones axiomáticas.

Aunque el objetivo de esta sección no es explicar la sintaxis de Maude ni los mecanismos ecuacionales utilizados por MOMENT, no está de más comentar que las ecuaciones axiomáticas mencionadas están definidas mediante el mecanismo de *pattern matching* [4].

En el Ejemplo 3.3, “Módulo de Operaciones auxiliares para miniXSD” se muestra el módulo que representaría a miniXSD. El contenido está estructurado de la siguiente manera:

❶ Declaración del módulo, nótese como en este caso es un módulo no parametrizado
❷ Declaración de variables

## Ejemplo 3.3. Módulo de Operaciones auxiliares para miniXSD

```
fmod spminiXSD is ❶
pr MOMENT-OP(vminiXSD) .
pr specore .
❷ vars OID1 OID2 IdValue: Qid .
  vars Name1 Type1 StringValue : String .
  vars elements1 sequence1 OSet : OrderedSet{QID} .
  vars Model Set : Set{ vminiXSD } .

❸ eq (mXSD-Schema OID1  Name1  elements1) :: OID = OID1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: OID <-- IdValue = (mXSD-Schema IdValue
Name1  elements1) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Schema OID1  Name1  elements1) :: Name = Name1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: Name <-- StringValue = (mXSD-Schema OID1
StringValue elements1) .
  op elements : -> Fun{vminiXSD} [ctor] .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements = elements1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements (Model) = (Model -> select (hasId
 ; ? elements1 ; empty-set)) -> sortedBy (firstThan ; ? elements1 ; empty-set) .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements <-- OSet = (mXSD-Schema OID1 Name1
 OSet) .

  eq (mXSD-Simple OID1  Name1 Type1 ) :: OID = OID1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: OID <-- IdValue = (mXSD-Simple IdValue  Name1
 Type1 ) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Name = Name1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Name <-- StringValue = (mXSD-Simple OID1
StringValue Type1) .
  op Type : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Type = Type1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Type <-- StringValue = (mXSD-Simple OID1 Name1
 StringValue) .

  eq (mXSD-Complex OID1  Name1  sequence1) :: OID = OID1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: OID <-- IdValue = (mXSD-Complex IdValue
 Name1  sequence1) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Complex OID1  Name1  sequence1) :: Name = Name1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: Name <-- StringValue = (mXSD-Complex OID1
 StringValue sequence1) .
  op sequence : -> Fun{vminiXSD} [ctor] .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence = sequence1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence (Model) = (Model -> select( hasId
 ; ? sequence1 ; empty-set)) -> sortedBy(firstThan ; ? sequence1 ; empty-set) .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence <-- OSet = (mXSD-Complex OID1
Name1 OSet) .

❹
      aquí axiomas de usuario
endfm
```

❸ Axiomatización de operaciones auxiliares para trabajar en OCL[3] sobre las entidades del metamodelo

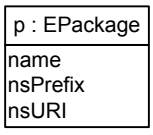
❹ Los axiomas específicos de metamodelo opcionales introducidos por el usuario.

El módulo de operaciones auxiliares es visiblemente más extenso que los demás, demostrando la necesidad imperativa de implementar estos puentes tecnológicos capaces de generarlo automáticamente. A continuación se presenta el análisis de las plantillas de correspondencia para este módulo.

### 3.3.3.1. Declaración del módulo

La siguiente plantilla muestra como se realiza la declaración del módulo de operaciones auxiliares.

**Tabla 3.10. Plantilla 10: Declaración del módulo de operaciones auxiliares**

| **NOMBRE** |
|---|
| Declaración del módulo de operaciones auxiliares |
| **DESCRIPCION** |
| Especifica como se declara el módulo de op. auxiliares en la especificación algebraica de un metamodelo |
| **ENTRADA** |
|  |
| **ALGORITMO** |
| Sea $eq\_clase_i$ las ecuaciones obtenidas para la clase i mediante la plantilla la Plantilla 12.<br><br>Sea $eq\_datatype_i$ las ecuaciones obtenidas para el EDatatype i mediante la plantilla la Plantilla 15. |
| **SALIDA** |
| `fmod sp<p.name> is`<br>`pr MOMENT-OP{v<p.name>) .`<br>`pr specore .`<br><br>`  Declaración de variables utilizadas`<br><br>`eq_clase1` |

[3]Object Constraint Language, ver [5]

```
  eq_clase2
   ...
  eq_clasei

 eq_datatype1
  eq_datatype2
   ...
  eq_datatypei

  Axiomas de usuario

endfm
```

**DEPENDENCIAS**

la Plantilla 12

la Plantilla 15

### 3.3.3.2. Ecuaciones para una clase

Se han de generar el selector y el actualizador que permiten trabajar con el identificador de un término del sort correspondiente a la clase. Además, cada atributo y cada referencia de la clase requieren sus respectivas ecuaciones para el la función de selección y la función de actualización.

**Tabla 3.11. Plantilla 11: De constructor a término con variables libres**

**NOMBRE**

De constructor a término con variables libres

**DESCRIPCION**

Este patrón permite obtener para un constructor un término que lo instancie con variables libres, para utilizarlo en la parte izquierda de una ecuación axiomática. La sintaxis que permite invocar un constructor tiene el siguiente aspecto:

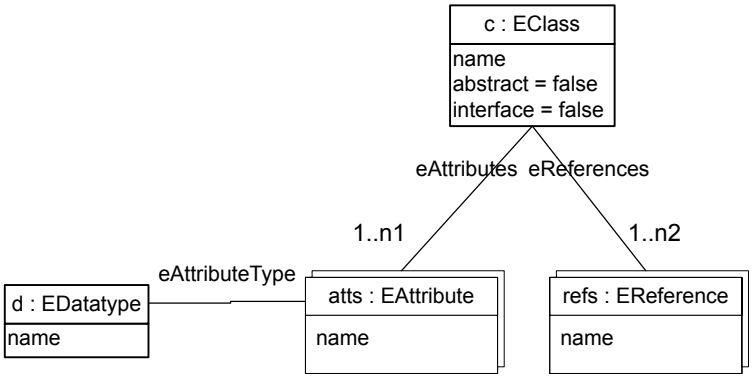
```
(cons-name param1 param2 ... paramn)
```

Para obtener un término con variables libres, basta con utilizar variables para todos los `param`$_i$ . Como nombre de cada variable se puede tomar el nombre del propio atributo (o referencia) de la clase al que representa, o simplemente un nombre aleatorio.

Cada variable libre debe ser declarada de forma previa a su utilización, y su tipo debe corresponder con el especificado en el constructor. La sintaxis para declarar una variable es:

```
var nombre : tipo .
```

**ENTRADA**




Sea el constructor obtenido para la clase c mediante la Plantilla 8, cuya signatura tiene el siguiente aspecto:

```
op cons-name _____ : Qid tipo1  tipo2  tipo3  ... tipon  -> tipo
```

**SALIDA**

```
var var0 : Qid .
var var1 : tipo1 .
var var2 : tipo2 .
  ...
var varn : tipon .

( cons-name var0 var1 ... varn )
```

**Tabla 3.12. Plantilla 12: Ecuaciones para una clase**

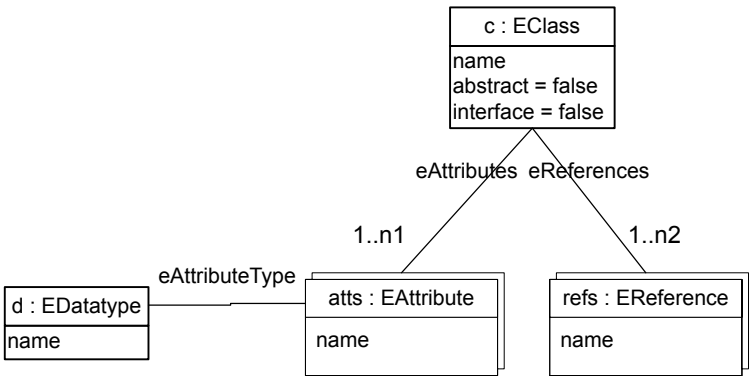
**NOMBRE**

Ecuaciones para una clase

**DESCRIPCION**

Especifica las ecuaciones relativas al soporte OCL para una clase

**ENTRADA**




**ALGORITMO**

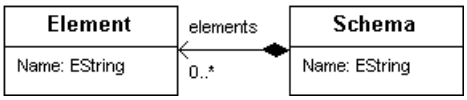
Sea `freeterm` un término del constructor para c con variables libres, obtenido mediante la Plantilla 11.

Sea `var0` el nombre dado a la primera variable, la que tiene tipo Qid.

Sea $\texttt{freeterm}_{x}$ el término obtenido a partir de `freeterm` reemplazando `var0` por `x`.

**SALIDA**

```
eq freeterm :: OID = var1 .
eq freeterm :: OID <-- IdValue = freeterm_IdValue
          .
```




**Ejemplo.** Se aplica la plantilla a la clase Schema del metamodelo miniXSD, cuyo constructor, obtenido mediante la Plantilla 8, es:

```
op `(mXSD-Schema___`) : Qid String OrderedSet {QID} -> mXSD-Schema [ctor] .
```

El término con variables libres correspondiente, obtenido mediante la Plantilla 11:

```
(mXSD-Schema OID1 Name1 elements1)
```

donde se han utilizado los nombres de los atributos/referencias para asignar nombres a las variables libres[4] Las ecuaciones buscadas que proporciona la la Plantilla 12 son:

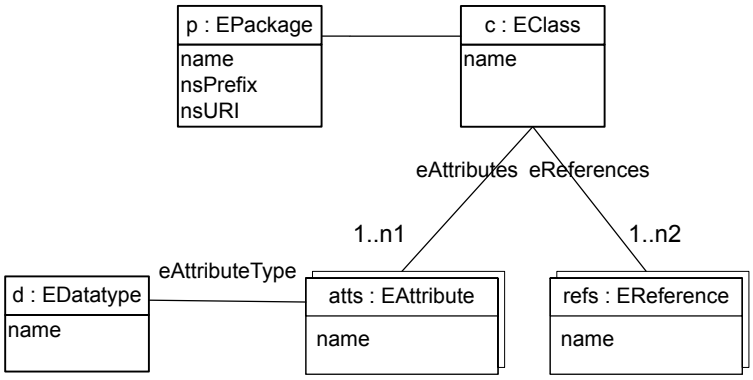
```
eq (mXSD-Schema OID1  Name1  elements1) :: OID = OID1 .
eq (mXSD-Schema OID1  Name1  elements1) :: OID <-- IdValue = (mXSD-Schema IdValue
  Name1  elements1) .
```

Las declaraciones de variables necesarias son las siguientes:

```
var Name1 : String .
var elements1 : OrderedSet{QID} .
var OID : Qid .
```

### 3.3.3.3. Ecuaciones para un atributo

**Tabla 3.13. Plantilla 13: Ecuaciones para un atributo**

| **NOMBRE** |
| --- |
| Ecuaciones para un atributo |
| **DESCRIPCION** |
| Especifica las ecuaciones relativas al soporte OCL para un atributo |
| **ENTRADA** |
|  <br>Esta plantilla especifica las ecuaciones a las que da lugar el atributo **i**. |
| **ALGORITMO** |
| Set $att_i$ el atributo de la clase c.<br><br>Sea `freeterm` un término del constructor para c con variables libres, obtenido mediante la Plantilla 11. |

[4]el 1 al final en los nombres es para evitar confusiones en ejemplos más adelante

Sea `var_i` el nombre dado a la variable que corresponde al atributo i.

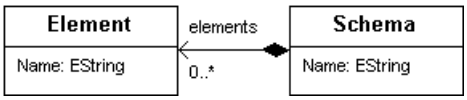
Sea $tipo_i$ el tipo de la variable `var_i`.

Sea $freeterm_x$ el término obtenido a partir de `freeterm` reemplazando `var_i` por `x`.

En la ecuación para el actualizador se utiliza la variable `newValue`. `newValue` es también una variable libre, así que el nombre elegido es arbitrario y se puede utilizar cualquier otro.

**SALIDA**

```
var newValue : tipo_i  .

op att_i.name : <tipo_i>Fun{v<p.name>} [ctor] .
eq freeterm :: att_i.name = var_i .
eq freeterm :: att_i.name <-- newValue = freeterm_newValue  .
```




**Ejemplo.** De forma análoga al ejemplo de la sección anterior, se aplica la plantilla al atributo *Name* de la clase `Schema`. El constructor sabemos que es:

```
op `(mXSD-Schema___`) : Qid String OrderedSet {QID} -> mXSD-Schema [ctor] .
```

En el constructor el segundo parámetro, de tipo “String”, es el que representa al atributo Name. Por lo tanto éste es el tipo equivalente del atributo. El término con variables libres correspondiente, obtenido mediante la Plantilla 11, es:

```
(mXSD-Schema OID1 Name1 elements1)
```

Y las ecuaciones buscadas son:

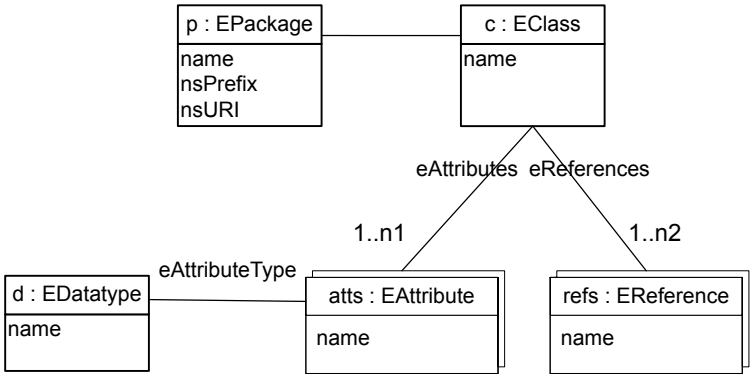
```
op Name : -> StringFun{vminiXSD} [ctor] .
eq (mXSD-Schema OID1  Name1  elements1) :: Name = Name1 .
eq (mXSD-Schema OID1  Name1  elements1) :: Name <-- newValue = (mXSD-Schema OID1
newValue elements1) .
```

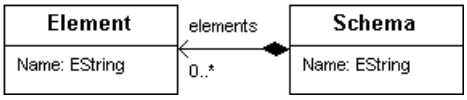
Además, hay que declarar la variable `newValue`:

```
var newValue : String .
```

### 3.3.3.4. Ecuaciones para una EReference

**Tabla 3.14. Plantilla 14: Ecuaciones para una EReference**

| **NOMBRE** |
|---|
| Ecuaciones para una EReference |
| **DESCRIPCION** |
| Especifica las ecuaciones relativas al soporte OCL para una referencia |
| **ENTRADA** |
|  <br>Esta plantilla especifica las ecuaciones a las que da lugar la referencia **i**. |
| **ALGORITMO** |
| Set $ref_i$ la referencia de la clase c.<br><br>Sea `freeterm` un término del constructor para c con variables libres, obtenido mediante la Plantilla 11.<br><br>Sea `var_j` el nombre dado a la variable que corresponde a la referencia i. El índice j se define como: j = n1 + i<br><br>Sea $freeterm_x$ el término obtenido a partir de `freeterm` reemplazando `var_j` por `x`. |
| **SALIDA** |
| `op` *$ref_i$.name* `:Fun{v<`*p.name*`>} [ctor] .`<br>`eq` *freeterm* `::` *$ref_i$.name* `=` *var_j* `.`<br>`eq` *freeterm* `::` *$ref_i$.name* `(Model) = (Model -> select (asId; ? var_j ; empty-set )) -> sortedBy (firstThan; ? var_j ; empty-set) .`<br>`eq` *freeterm* `::` *$ref_i$.name* `<-- newValue =` *$freeterm_{newValue}$* `.` |




**Ejemplo.** Repitiendo el ejemplo de secciones anteriores, se aplica la plantilla a la asociación *elements* de la clase Schema. Las ecuaciones obtenidas en esta ocasión son:

```
op Name : -> StringFun{vminiXSD} [ctor] .
eq (mXSD-Schema OID1  Name1  elements1) :: Name = Name1 .
eq (mXSD-Schema OID1  Name1  elements1) :: elements (Model) = (Model -> select
(hasId; ? elements1 ; empty-set)) -> sortedBy (firstThan; ? elements1 ; empty-set)
 .
eq (mXSD-Schema OID1  Name1  elements1) :: Name <-- newValue = (mXSD-Schema OID1
newValue elements1) .
```

El tipo de la variable `newValue` en este caso es:

```
var newValue : OrderedSet{Qid} .
```

### 3.3.3.5. Ecuaciones para un EDatatype

**Tabla 3.15. Plantilla 15: Ecuaciones para un EDatatype**

| **NOMBRE** |
|---|
| Ecuaciones para un EDatatype |
| **DESCRIPCION** |
| Especifica las ecuaciones que proyectan un EDatatype |
| **ENTRADA** |
| d : EDatatype<br>name<br>instanceClassName |
| **ALGORITMO** |
| Sea $tipo_{dat}$ el tipo de datos Maude equivalente para este EDatatype, obtenido mediante la Plantilla 7. |
| **SALIDA** |
| ceq Is<*$tipo_{dat}$*>(DT) = true<br>if (DT :: name) == *d.name* . |

### 3.3.3.6. Apuntes sobre la aplicación de las plantillas

Las plantillas describen las correspondencias entre EMF y Maude y pueden servir como guía para la implementación, pero no la sustituyen. A la hora de realizar

la implementación de las correspondencias para este módulo, hay que tener en cuenta los siguientes puntos:

- No se puede declarar una variable dos veces.
- Las variables libres se pueden reutilizar en distintas ecuaciones, siempre que se respeten los tipos. Por ejemplo, en el caso de las ecuaciones para la operación de actualización habrá que utilizar distintos nombres para la variable `newValue`, según el tipo del atributo o referencia.

## 3.4. Puente bidireccional a nivel M1

La función de este puente es proyectar modelos en Maude, para que puedan ser manipulados por MOMENT, y trasladarlos de vuelta a EMF una vez concluida la operación. Por lo tanto a diferencia del puente a nivel M2, el puente a nivel M1 ha de ser bidireccional.

Como modelo se entiende cualquier artefacto EMF, no sólo instancias del metamodelo Ecore como ocurría con los metamodelos. Véase en el Ejemplo 3.4, "Un modelo de miniXSD proyectado en términos Maude" un ejemplo de un modelo miniXSD, es decir, una instancia de miniXSD tomando ahora miniXSD como metamodelo[5]. De igual forma que no es habitual referirse a los modelos UML como "modelos del metamodelo UML", en adelante se hablará aquí de "modelos miniXSD".

El Ejemplo 3.4, "Un modelo de miniXSD proyectado en términos Maude", muestra a la izquierda un esquema XSD que define un elemento complejo llamado Actor. El elemento Actor está compuesto por un elemento ActorID y un elemento Bio, ambos simples, y un nombre definido por un elemento complejo. La representación gráfica mostrada proviene del editor en árbol automáticamente generado por EMF para miniXSD.

A la derecha de la figura se muestra la representación en MOMENT del modelo. Como ya se ha comentado, dicha representación está formada por términos

[5]También se puede decir que miniXSD es un modelo del metamodelo Ecore, o simplemente, un modelo Ecore

Maude. Es posible entender por intuición la estructura de esta representación, sabiendo que:

- Aunque no es lo mismo, en este contexto los términos son los representantes de las instancias del modelo. Así pues, diríamos que "Actor" es un objeto de la clase `Complex` o que el término con id ID1 es un término del sort `mXSD-ComplexElement`.

- Un término se construye a partir de la signatura de su constructor. El primer componente del término es el nombre del constructor, que en los constructores definidos para MOMENT resulta ser igual al nombre del sort. Los componentes de un constructor van separados por espacios.

- Las instancias del tipo de datos simple ObjectId se construyen en Maude utilizando una cadena prefijada con un apóstrofe.

- Para expresar conjuntos de IDs se utiliza la sintaxis

  ```
  Set{ id1, id2, ... id_n }
  Bag{ id1, id2, ... id_n }
  OrderedSet{ id1 :: id2 :: ... id_n }
  Sequence{ id1 :: id2 :: ... id_n }
  ```

  dependiendo del tipo de conjunto. Nótese que si el conjunto es ordenado (OrderedSet, Sequence) el separador a utilizar es "::", mientras que si no lo es se utiliza una coma (,).

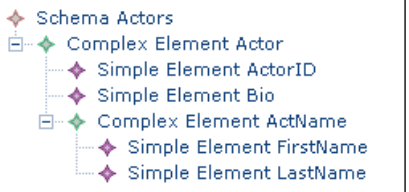
Con esta información y recordando la signatura de los constructores de la especificación[6]:

```
op mXSD-Schema: Qid, elements
op mXSD-Complex: Qid, Name, sequence
op mXSD-Simple: Qid, Name, Type
```

se puede comprender de forma intuitiva y mirando al Ejemplo 3.4, como se realiza la representación de un modelo en términos algebraicos

[6]En realidad lo que se muestra no es la signatura sino el comentario que la acompaña, resulta más útil en este contexto e igual de válido.

**Ejemplo 3.4. Un modelo de miniXSD proyectado en términos Maude**




```
Set {
(mXSD-Schema  'ID0 "Actors" OrderedSet{'ID1} )
(mXSD-Complex 'ID1 "Actor" OrderedSet{'ID2 :: 'ID3 :: 'ID4})
(mXSD-Simple  'ID2 "ActorID" "String" )
(mXSD-Simple  'ID3 "Bio" "String" )
(mXSD-Complex 'ID4 "ActName" OrderedSet{'ID5 :: 'ID6})
(mXSD-Simple  'ID5 "FirstName" "String" )
(mXSD-Simple  'ID6 "LastName" "String" )
}
```

El proceso de proyección a nivel M1 ha de tener en cuenta además los siguientes puntos:

- Tipos de datos. Las correspondencias se muestran en el Apéndice C, *Correspondencia de tipos EMF <-> Ecore*
- Generación y resolución de ObjectIds únicos en el ET Maude.

A continuación se presenta la gramática que define la sintaxis abstracta del sublenguaje de términos algebraicos que utiliza MOMENT para representar los modelos. Las plantillas se basan en esta gramática para generar, de forma adecuada, los términos algebraicos que representan a un modelo determinado. La notación utilizada es EBNF simplificada.

## 3.4.1. Gramática para el sublenguaje de términos

En primer lugar es necesario introducir los elementos terminales de la gramática.

- PARENT_AB, PARENT_CE, LLAVE_AB, LLAVE_CE, DOSPUNT y GUION representan los símbolos '(', ')', '{', '}', ':' y '-', respectivamente.
- CADENA representa una cadena sin espacios en la cuál además de los caracteres alfanuméricos son legales los siguientes símbolos: '%','_','$','@','#','&','.','/',':','\',,'\\'.
- El terminal CADENA también acepta cadenas entre comillas. En este caso además se aceptan espacios y no hay símbolos ilegales, es decir, todos los símbolos son legales (excepto lógicamente las comillas, que se aceptan sólo si van precedidas de una '\'.

- NUMERO acepta números tanto enteros como reales, utilizando como separador decimal el punto ('.').
- MAUDE_ID es un terminal que representa un identificador de término con las reglas léxicas de Maude al respecto.

**Gramática para el sublenguaje de términos**

```
modelo       ::= set_ab term (set_separator term)* set_ce
term          ::= PARENT_AB constructor oid atributo* referencia* PARENT_CE
constructor   ::= prefijo GUION sort
prefijo       ::= CADENA
sort          ::= CADENA
oid           ::= MAUDE_ID
atributo      ::= att_simple | att_compuesto
att_simple    ::= CADENA | ENTERO
att_compuesto ::= set_ab att_simple (set_separator att_simple)* set_ce
                  | empty_set
referencia    ::= (set_ab oid (set_separator oid)* set_ce)
                  | empty_set
empty_set     ::= "empty" GUION ("set" | "sequence" | "bag" | "orderedset")
set_ab        ::= ("Set" | "Bag" | "Sequence" | "OrderedSet") LLAVE_AB
set_separator ::= COMA | ( DOSPUNT DOSPUNT )
```

### 3.4.1.1. Elementos del modelo

El no-terminal `modelo` especifica que un modelo está constituido como un conjunto de términos:

```
modelo        ::= set_ab term (set_separator term)* set_ce
```

### 3.4.1.2. Término algebraico como instancia de un modelo

El no-terminal `term` especifica que una término algebraico, que representa a una instancia de un modelo, está constituido por un constructor, un identificador (`oid`), una lista de atributos y una lista de referencias, sin olvidar los paréntesis que abren y cierran el término:

```
term          ::= PARENT_AB constructor oid atributo* referencia* PARENT_CE
constructor   ::= prefijo GUION sort
```

Donde el constructor está formado por el prefijo del `EPackage` seguido de un guión y del nombre del sort; y el oid es un terminal MAUDE_ID.

#### 3.4.1.3. Atributos de un término

Un atributo simple (cardinalidad uno) puede ser una cadena o un número. Un atributo complejo (cardinalidad múltiple) no es más que un conjunto no vacío de atributos simples, o el conjunto vacío:

```
atributo      ::= att_simple | att_compuesto
att_simple    ::= CADENA | ENTERO
att_compuesto ::= set_ab att_simple (set_separator att_simple)* set_ce
             | empty_set
empty_set     ::= "empty" GUION ("set" | "sequence" | "bag" | "orderedset")
```

#### 3.4.1.4. Referencias de un término

Una referencia está constituida por un conjunto de oids:

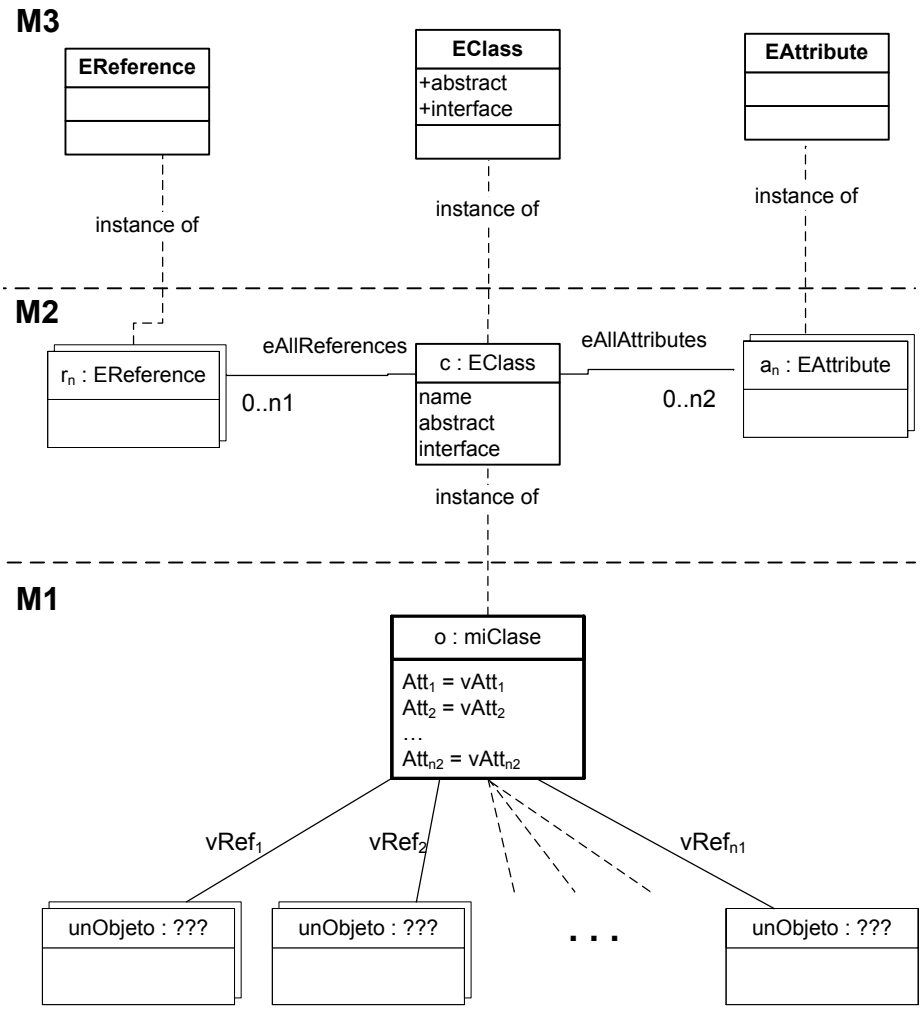
```
referencia  ::= (set_ab oid (set_separator oid)* set_ce) | empty_set
```

### 3.4.2. Formalización de las correspondencias del nivel M1

**Tabla 3.16. Plantilla 16: Representación de un modelo como un conjunto de términos algebraicos**

| NOMBRE |
|---|
| Representación de un modelo como un conjunto de términos algebraicos |
| **DESCRIPCION** |
| Un modelo se representa en MOMENT como un conjunto de términos algebraicos. |
| **ENTRADA** |
| El origen es un modelo EMF, que puede ser instancia de cualquier metamodelo. Un modelo está constituido únicamente por las instancias que lo componen |
| **ALGORITMO** |
| Sea $term_n$ el término n obtenido para la instancia n mediante la Plantilla 17 |
| **SALIDA** |
| Set { $term_1$ , $term_2$ , $term_3$ , ... $term_n$ } |
| **DEPENDENCIAS** |
| la Plantilla 17 - Para obtener la representación de las instancias en términos |

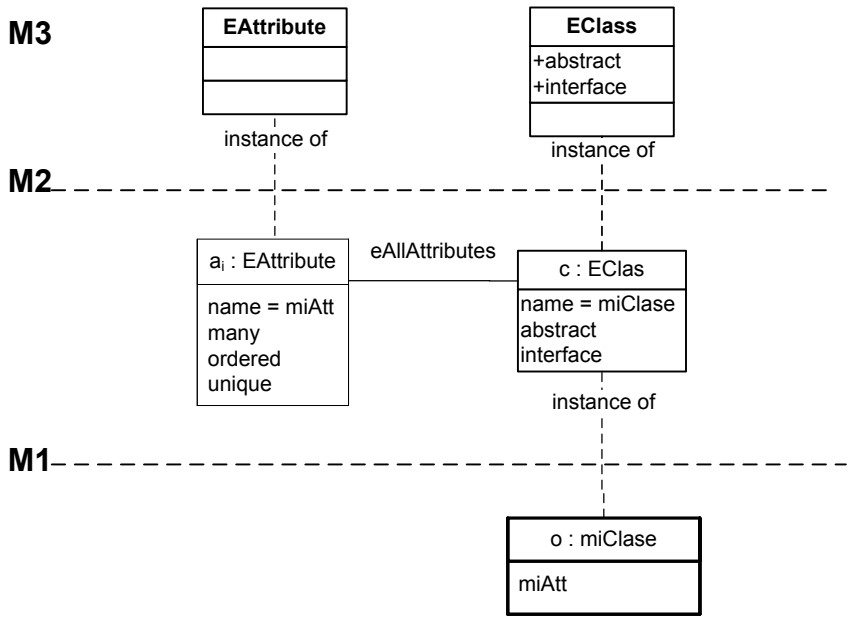
**Tabla 3.17. Plantilla 17: Representación de una instancia como un término algebraico**

| **NOMBRE** |
| --- |
| Representación de una instancia como un término algebraico |
| **DESCRIPCION** |
| Una instancia u objeto de una metaclase se representa en MOMENT como un término algebraico, cuyo sort es el correspondiente a la metaclase. Los datos o contenidos del objeto se proyectan a Maude mediante el constructor proporcionado por el sort |
| **ENTRADA** |
|  |
| **ALGORITMO** |

| |
|---|
| Sea *sort* el sort para la clase c obtenido aplicando la Plantilla 2.<br>Sea *gen_oid()* una función capaz de generar un identificador único<br>para cada objeto:<br>`gen_oid : EObject -> Qid`<br>Sea $v_{attn}$ el valor equivalente en Maude del atributo n obtenido aplicando la Plantilla 18.<br>Idem para $v_{refn}$ aplicando esta vez la Plantilla 19. |
| **SALIDA** |
| (*sort gen_oid(o)* $v_{att1}$ $v_{att2}$ ... $v_{attn2}$ $v_{ref1}$ $v_{ref2}$ ... $v_{refn1}$ ) |
| **DEPENDENCIAS** |
| la Plantilla 18 - Para obtener la representación del valor de un atributo<br><br>la Plantilla 19 - Para obtener la representación del valor de una referencia |

**Tabla 3.18. Plantilla 18: Representación del valor de un EAttribute**

| |
|---|
| **NOMBRE** |
| Representación del valor de un EAttribute |
| **DESCRIPCION** |
| Los atributos en EMF contienen datos de tipos simples, tal y como se comenta en la Plantilla 7. La representación equivalente en Maude se obtiene realizando la conversión de tipos EMF <->Maude y produciendo la representación textual de acuerdo a la sintaxis de Maude para el tipo concreto. En este aspecto la sintaxis de Maude es convencional: los Strings van entre comillas, los enteros y caracteres no, y los números reales utilizan el punto (.)<br><br>En caso de que la cardinalidad sea múltiple, el atributo da lugar a un conjunto de datos simples. |
| **ENTRADA** |




**ALGORITMO**

Sea `convert()` una función que realiza la conversión de datos, de acuerdo con el tipo equivalente Maude obtenido para att mediante `tipo_eq()` (ver la Plantilla 7).

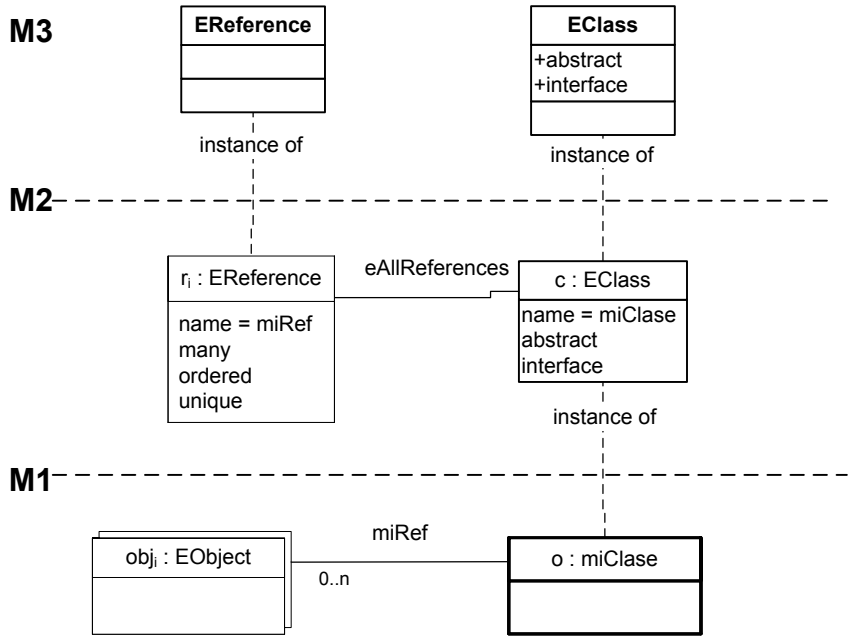
**SALIDA**

```
if a_i.many = false
  convert(o.miAtt)
else
  tipo_conjunto(a_i) { convert(o.miAtt_1), convert(o.miAtt_2), ... convert(o.miAtt_n)
 }
```

Donde:

- $att_i \equiv$ `c.eAllAttributes[i]`
- hay que tener en cuenta que si el conjunto es de tipo `Sequence` o `OrderedSet`, hay que sustituir la coma utilizada para separar los elementos por el símbolo "::" (ver punto Sección 3.4, "Puente bidireccional a nivel M1").

**Tabla 3.19. Plantilla 19: Representación del valor de una EReference**

| **NOMBRE** |
| --- |
| Representación del valor de una EReference |
| **DESCRIPCION** |
| El valor de una EReference se representa en Maude como una lista de Qids, obtenidos a partir de las instancias referenciadas en el valor de la EReference. |
| **ENTRADA** |
|  |
| **SALIDA** |
| *tipo_conjunto($r_i$)* { *gen_oid($obj_1$)*, *gen_oid($obj_2$)*, ... *gen_oid($obj_n$)* }<br><br>Donde hay que tener en cuenta que si el conjunto es de tipo `Sequence` o `OrderedSet`, hay que sustituir la coma utilizada para separar los elementos por el símbolo "::" (ver punto Sección 3.4, "Puente bidireccional a nivel M1"). |
| **DEPENDENCIAS** |
| la Plantilla 16 - Para obtener los oids |

## 3.5. Operaciones de gestión de modelos con MOMENT

Una vez los modelos y metamodelos han sido proyectados al ET Maude, MOMENT está en condiciones de manipularlos y realizar operaciones con ellos, tal y como se expone en [11].

### 3.5.1. Operaciones intramodelo/extramodelo

En gestión de modelos se clasifican las operaciones con modelos según los metamodelos origen y destino. Si son el mismo, se denominan *intramodelo*, en caso contrario se denominan *extramodelo*. El segundo tipo resulta más complejo ya que requiere trabajar con metamodelos distintos de forma simultánea.

En este prototipo sólo se ha contemplado el primer caso, el de los operadores intramodelo, dejando el caso más general como una extensión a desarrollar en trabajos futuros.

### 3.5.2. Operaciones simples/compuestas

Adicionalmente las operaciones se clasifican en compuestas y simples. Las operaciones simples consisten en la invocación de un operador básico de los ofrecidos por MOMENT, como *Merge*, *Diff*, *Intersection* o *ModelGen*.

Una operación compuesta es aquella que hace uso de uno o más operadores complejos. Un operador complejo se define como la composición de uno o más operadores, simples o complejos. Los operadores complejos son esenciales para solucionar problemas complejos mediante gestión de modelos. El ejemplo más común es el de la regresión de cambios, introducido en [16], y contextualizado para MOMENT en [15].

#### 3.5.2.1. Definición de operadores complejos en MOMENT

Un operador complejo se define como una signatura acompañada de la ecuación axiomática que la especifica. La manera más común de introducir esta definición en MOMENT para poder utilizarla, es definirla en su propio módulo, junto con

cualquier definición auxiliar que necesite para realizar su trabajo. El módulo deberá importar la definición de los operadores simples de MOMENT, así como (si lo hay) cualquier otro operador complejo que intervenga en la operación. Para introducirlo en MOMENT, el módulo que contiene el operador complejo es cargado en Maude. A partir de aquí, el operador complejo definido puede ser invocado de la misma manera que los operadores simples como se verá en el próximo apartado.

El Ejemplo 3.5, "Operador compuesto: Merge de 5 modelos" muestra el aspecto de un operador complejo que realiza la unión (*merge*) de 5 modelos. Nótense las siguientes propiedades:

- El nombre del módulo en el que se declara la operación es arbitrario.
- Dicho módulo importa el módulo del kernel de MOMENT llamado MOMENT-OP.
- En caso de que el operador complejo dependa de otro operador complejo (recuérdese que los operadores complejos se definen como la composición de otros operadores), el módulo a importar será aquel o aquellos en los que estén definidos los operadores complejos requeridos en lugar de MOMENT-OP.
- No se instancia el parámetro X de MOMENT-OP por lo tanto el módulo de la operación tiene un parámetro X.

Debido a la estructura de instanciación de álgebras en Maude hay un requisito adicional que debe ser satisfecho para poder invocar el operador complejo. En la Sección 3.3.3, "Módulo de operaciones auxiliares y axiomas de usuario" se mencionaba que el módulo de operaciones auxiliares en la especificación algebraica de un metamodelo importa el módulo del kernel MOMENT-OP. Para poder utilizar un operador complejo con un metamodelo X, su módulo de operaciones auxiliares debe importar el módulo del operador complejo en lugar de MOMENT-OP. Poniendo como ejemplo una vez más el metamodelo miniXSD, si quisiéramos utilizar el operador 5merge habría que modificar el módulo de operaciones auxiliares para que importase el módulo `5MergeOp`:

```
fmod spminiXSD is
pr 5MergeOp(vminiXSD) .
pr specore .
  vars OID1 OID2 IdValue: Qid .
```

**Ejemplo 3.5. Operador compuesto: Merge de 5 modelos**

```
fmod 5MergeOp{ X :: TRIV } is
 pr MOMENT-OP{X}

  vars model1 model2 model3 model4 model5 : Set{QID} .
  op 5Merge _____: Set{QID} Set{QID} Set{QID} Set{QID} Set{QID} -> Set{QID} .
  eq 5Merge model1 model2 model3 model4 model5 =
    GetModel(Merge
     SetPreferred(GetModel(Merge
      SetPreferred(GetModel(Merge
       SetPreferred(GetModel(Merge
        SetPreferred(model1)
        model2
       ))
       model3
      ))
      model4
     ))
     model5
    ) .

endfm

       vars Name1 Type1 StringValue : String .
       vars elements1 sequence1 OSet : OrderedSet{QID} .
       vars Model Set : Set{ vminiXSD } .

       eq (mXSD-Schema OID1  Name1  elements1) :: OID = OID1 .
       eq (mXSD-Schema OID1  Name1  elements1) :: OID <-- IdValue = (mXSD-Schema IdValue
       Name1  elements1) .
       op Name : -> StringFun{vminiXSD} [ctor] .
       .
       .
       .
     endfm
```

### 3.5.2.2. Invocación de operadores en MOMENT

La invocación de una operación en MOMENT requiere los siguientes elementos:

- Un operador de gestión de modelos.
- Uno o varios modelos en forma de términos algebraicos, junto con las especificaciones algebraicas de los metamodelos.
- La plataforma MOMENT con las especificaciones algebraicas de los metamodelos cargadas, así como los operadores compuestos en caso de estar presentes.

Para producir la invocación se utiliza una instrucción Maude que tiene el siguiente aspecto:

```
red (nombre_op param1 param2 ... paramn ) .
```

En caso de que los parámetros sean modelos (el caso más común), tendría el siguiente aspecto:

```
red ( nombre_op Set { ... } Set { ... } ... Set { ... } ) .
```

El Ejemplo 3.6, “Invocación de un merge entre dos modelos” muestra como sería la invocación de un *Merge* entre dos modelos.

**Ejemplo 3.6. Invocación de un merge entre dos modelos**

```
red (Merge Set{términos de A} Set{términos de B} ) .
```

## 3.5.3. Proceso completo de gestión de modelos en MOMENT

En esta sección se muestran los pasos que se siguen en la utilización manual de MOMENT, tal y como se ha hecho hasta la existencia de este prototipo. De esta manera se proporciona una visión de conjunto de las tareas que debe asumir la herramienta.

Los pasos a seguir para realizar una operación con modelos en MOMENT son los siguientes:

1. Arrancar el sistema Maude.
2. Cargar el kernel de MOMENT en Maude enviando los ficheros que lo componen, en el orden adecuado, al intérprete Maude.
3. Si se está ejecutando una operación compuesta, cargar el módulo que la define.
4. Cargar la signatura del o de los metamodelos que intervienen.
5. Cargar la vista del o los metamodelos

6. Cargar el módulo de operaciones auxiliares del o de los metamodelos. Si es una operación compuesta, hay que preparar previamente el módulo auxiliar para que importe la operación compleja.

7. Ejecutar y componer el comando que realiza la invocación, a partir de la definición de la operación y de la proyección en el ET Maude de los modelos que intervienen.

Tras la invocación Maude produce el modelo resultado en forma de términos que pueden ser proyectados a EMF.

## 3.6. Otros Requisitos observados

A partir de las necesidades de uso y desarrollo expresadas por el equipo de MOMENT, así como de la lectura y exploración del estado del arte en el área de MDE, se ha observado que sería muy deseable que la herramienta cumpliese con las siguientes propiedades:

1. Independencia de plataforma, en este caso de Eclipse, para poder trasladar la herramienta a otras tecnologías y/o plataformas con el mínimo esfuerzo

2. Extensibilidad y modularidad, para facilitar la adaptación a requisitos futuros

A lo largo de esta memoria se contemplan estos requisitos y cómo se han tenido en cuenta.

# Capítulo 4. Análisis y Diseño

En este capítulo se presentan los pasos que se han seguido hasta llegar al diseño actual del prototipo. Uno de los objetivos de este documento es el de servir de memoria de desarrollo para que la persona que tome el relevo en el equipo de MOMENT sea capaz de continuar con el trabajo aquí realizado. Esta sección debería ser de gran interés para esa persona o personas.

## 4.1. Los Puentes EMF -> Maude

### 4.1.1. Análisis del problema

El análisis de los niveles M1 y M2 se aborda de manera simultanea ya que estos dos puentes parten de la misma premisa, se basan en la generación de artefactos de texto a partir de un un modelo EMF. La forma más común de acceder a dichos modelos es a través de su forma codificada, ficheros XML en formato XMI 2.0

El problema consiste básicamente en animar las plantillas de correspondencia facilitadas a lo largo del Capítulo 3, *Interoperabilidad EMF-Maude*, que nos permitirán obtener los artefactos Maude necesarios para poder manipular modelos en MOMENT.

#### 4.1.1.1. Generación de IDs únicos

Por otra parte, en el nivel M1 se comentaba en la Sección 3.3 la necesidad de proporcionar un algoritmo para la generación de ObjectIDs únicos. La solución más común a este tipo de problema pasa por la utilización de números o cadenas aleatoriamente generados lo suficientemente grandes para estadísticamente garantizar la unicidad.

Para considerar la solución adoptada aquí hay que tener en cuenta el siguiente apunte. Tanto EMF como XMI a través de EMF permiten la creación de referencias a instancias externas, es decir a instancias que residen en otros modelos. XMI hace uso de un mecanismo de identificación basado en URI (*Universal Resource Identifier*) que permite codificar la información que identifica el fichero y la localización de la instancia externa. Para permitir que el puente M1 desarrollado no pierda la información acerca de estas referencias externas hay que almacenar la URI correspondiente. Sin embargo esta URI no es un concepto perteneciente al modelo, sino un artefacto que depende del sistema de codificación empleado.

La solución adoptada en este proyecte consiste en utilizar la URI que XMI genera como identificador del término algebraico. Al fin y al cabo una URI es un mecanismo para identificar.

Así en el Ejemplo 3.4 del esquema del actor, si éste reside en un fichero `c:/actors.minixsd`, los términos quedarían como sigue:

```
 Set {
(mXSD-Schema  'file:/C:/actors.minixsd#/ "Actors"
    OrderedSet { 'file:/C:/actors.minixsd#//@elements.0} )
(mXSD-Complex 'file:/C:/actors.minixsd#//@elements.0 "Actor"
    OrderedSet { 'file:/C:/actors.minixsd#//@elements.0/@sequence.0 ::
                 'file:/C:/actors.minixsd#//@elements.0/@sequence.1 ::
                 'file:/C:/actors.minixsd#//@elements.0/@sequence.2
    } )
(mXSD-Simple  'file:/C:/actors.minixsd#//@elements.0/@sequence.0 "ActorID" "String" )
(mXSD-Simple  'file:/C:/actors.minixsd#//@elements.0/@sequence.1 "Bio" "String" )
(mXSD-Complex 'file:/C:/actors.minixsd#//@elements.0/@sequence.2 "ActName"
    OrderedSet { 'file:/C:/actors.minixsd#//@elements.0/@sequence.2/@sequence.0 ::
                 'file:/C:/actors.minixsd#//@elements.0/@sequence.2/@sequence.1
    } )
(mXSD-Simple  'file:/C:/actors.minixsd#//@elements.0/@sequence.2/@sequence.0
   "FirstName" "String" )
(mXSD-Simple  'file:/C:/actors.minixsd#//@elements.0/@sequence.2/@sequence.1
   "LastName"  "String" )
}
```

## 4.1.2. Diseño de la solución

### 4.1.2.1. Aproximación XSLT

Dado que el formato de codificación está basado en XML, la primera opción considerada para abordar el problema de las proyecciones fue la creación de un conjunto de plantillas XSLT que realizasen la generación de los artefactos

Maude a partir de la codificación XMI de los modelos. XSLT (*Extensible StyleSheet Language Transformations*) es un lenguaje especializado para la transformación o generación de texto a partir de documentos XML.

Esta vía fue explorada pero se llegó a la conclusión de que las plantillas que había que diseñar resultaban excesivamente complejas y difíciles de mantener. El diseño consistía en una pareja de plantillas obtenidas a partir de la codificación XMI del metamodelo. La primera implementaba el puente M2 al aplicarla a cualquier metamodelo; la segunda, al aplicarla a un metamodelo Y, generaba una tercera plantilla capaz de realizar el puente M1 al aplicarla a modelos Y.

Se estimó que las plantillas resultaban en exceso complejas por los siguientes motivos:

- XMI es un formato de codificación y como tal está optimizado para su función, adoptando una serie de compromisos que permiten que el resultado final sea lo más compacto posible. Por ejemplo, aquellos atributos que tienen el valor por defecto no aparecen en el fichero, y los atributos derivados, cuyo valor puede calcularse a partir de los atributos simples, tampoco lo hacen. MOMENT requiere la presencia de todos los atributos de manera explícita, por tanto las plantillas tendrían que implementar la lógica para recrear esos atributos. Esto ciertamente aumenta la complejidad de unas plantillas ya complicadas, y además con los atributos derivados esta estrategia no sirve. En la metaclase EClass se hallan una serie de atributos derivados como AllSuperclasses, AllAttributes, AllReferences, etc. que son imprescindibles para MOMENT, por citar un ejemplo en el que son necesarios estos atributos derivados. Las soluciones que se consideraron a este problema no eran satisfactorias y finalmente se abandonó la vía XSLT.

- La resolución de referencias entre elementos de un mismo modelo, otro problema complejo dada la amplitud del estándar XMI, forzó a utilizar extensiones de al lenguaje XSLT incompatibles con el estándar oficial. El caso de referencias a elementos en modelos externos, caso también contemplado por XMI, presentaba dificultades aún mayores.

- XSLT es un lenguaje muy joven en su primera versión aún, y es un hecho aceptado que dicha versión 1.0 tiene una serie de limitaciones que reducen su utilidad en escenarios de cierta complejidad, como éste. La versión 2.0

está en vías de estandarización, existiendo ya algunas implementaciones en fase de pruebas. En la parte final de la elaboración de las plantillas se comprobó que las nuevas capacidades de la versión 2.0 resultaban de gran ayuda; sin embargo la escasa documentación a nivel didáctico para esta nueva versión y el todavía escaso número de herramientas que la soportasen enlentecían los progresos.

Se consideró también la opción de modificar las opciones de persistencia de modelos EMF en busca de un formato de persistencia más explícito, con el fin de reducir algunos de los inconvenientes enumerados, pero no se obtuvieron los resultados deseados. Así pues finalmente se descartó la ruta de trabajo sobre XML.

#### 4.1.2.2. Aproximación Java + Velocity

Pronto se vio que todos los problemas que aparecían eran debidos a trabajar con una codificación de los modelos, en lugar de con los modelos en sí. Esta limitación resulta aún más injustificable si se tiene en cuenta que EMF proporciona una excelente correspondencia con el espacio tecnológico del lenguaje Java en forma de generación de código.

Es por esto que se decidió trabajar con los modelos directamente mediante la completa API de EMF, con la cual es posible hacer todo tipo de consultas sobre un modelo.

Como motor de plantillas para generar los artefactos de texto, se optó por Apache Velocity[21], un motor de plantillas plenamente integrado en Java.

El resultado son unas plantillas mucho más sencillas y mantenibles, gracias a que es EMF quien se encarga ahora de cargar y resolver los modelos y las plantillas se basan en consultas simples a la API de EMF. Además, gracias a que Velocity facilita una comunicación fluida entre el motor de plantillas y Java, se puede evitar programar excesiva lógica en el lenguaje de plantillas. De esta manera es posible localizar la parte algorítmica de las plantillas en una clase Java que actúe en cooperación con la plantilla. Esto nos permite obtener unas plantillas más sencillas y fáciles de mantener, que se sirven tanto de la API de EMF como de los algoritmos implementados en Java.

**Portabilidad**

Una ventaja añadida de la solución adoptada es la portabilidad en cuanto a un hipotético cambio de tecnología en el campo MDE, es decir a otro ET en el campo MDE. Por ejemplo, la sustitución de EMF por una implementación alternativa de MOF, o incluso por una tecnología no relacionada con MOF, como las Software Factories™ de Microsoft™. Puede parecer un contrasentido, ¿acaso no se depende de la API de EMF para las consultas, así como de la topología de los modelos EMF? Sí, pero si se puede asumir que la ontología va a ser similar y que el lenguaje de programación es el mismo o existe interoperabilidad, es relativamente sencillo construir un adaptador de una API a otra, minimizando las modificaciones necesarias en el código desarrollado.

En contraposición, una solución con XML + XSLT, que a priori parece mucho más portable, podría requerir numerosos cambios en las plantillas o su total reingeniería si el formato de codificación utilizado por la nueva tecnología no es XMI. En este caso se podrían proponer soluciones alternativas como el uso de un elemento intermedio que realice la conversión, sin embargo estas soluciones suelen requerir una fuerte inversión de efectivos para dar buenos resultados y además aumentan la complejidad del sistema

Nótese que en cualquier caso, la solución adoptada en este prototipo no es dependiente del formato de codificación.

Se ha utilizado el motor de plantillas Velocity para animar las plantillas de correspondencias expuestas a lo largo del Capítulo 3, *Interoperabilidad EMF-Maude*.En algunos casos para facilitar la implementación de estas plantillas, el modelo es cargado en EMF para poder analizarlo y producir una estructura de datos más cercana a la representación en la plantilla destino. Posteriormente, el motor de plantillas recibe el resultado de este procesado y la plantilla a utilizar, y produce el correspondiente artefacto de texto.

## Nivel M2: Metamodelos

En este nivel se han especificado tres plantillas Velocity además de una clase Java (`MetamodelContext`), que preprocesa el modelo y 'rellena' las plantillas:

- `m2sig.vm` - Produce el módulo signatura de la especificación
- `m2view.vm` - Produce la vista
- `m2sp.vm` - Produce el módulo de operaciones de selección

Para un metamodelo representado por un EPackage y un conjunto de clases, las responsabilidades del preprocesado incluyen:

- El filtrado de las clases del modelo que sean abstractas, ya que no intervienen en la producción de constructores
- La definición de la jerarquía de sorts
- La obtención de la signatura de los constructores

**Nivel M1: Modelos**

Para un modelo, representado por un conjunto de instancias de conceptos del metamodelo, se ha especificado una plantilla que implementa las correspondencias de la Sección 3.4, "Puente bidireccional a nivel M1" Además como en el caso de nivel M2, se utiliza una clase Java (`ModelContext`) que realiza el preprocesado:

- La conversión de tipos de datos entre EMF y Maude
- La Identificación y declaración de los metamodelos que intervienen en el modelo
- La obtención o generación de los IDs para los términos Maude, y la representación de referencias entre instancias como referencias a IDs

# 4.2. El puente Maude -> EMF

## 4.2.1. Análisis del problema

El puente de Maude a EMF presenta un problema que poco o nada tiene que ver con los puentes en la dirección inversa. En este caso se parte de una representación textual de los modelos basada en *términos*, y el objetivo no es producir un arte-

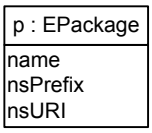
facto textual como anteriormente, sino producir un modelo. Se identifican al menos dos opciones para abordar el problema:

- Producir el modelo directamente en su forma codificada, esto es, la generación de artefactos XML en formato XMI
- Utilizar la API de EMF para construir el modelo en memoria y a partir de esta representación, generar la representación XMI automáticamente usando EMF

En este proyecto se ha optado por la segunda. En cualquiera de los dos casos estos son los elementos necesarios para poder producir un modelo:

- El conjunto de términos algebraicos que lo describen
- La identificación de los metamodelos que participan en el modelo y la relación entre las metaclases y los constructores de los términos

Evidentemente los términos son conocidos, sin embargo el segundo requisito, la identificación de los metamodelos, no está presente en los términos. La identificación de metamodelos en EMF se hace por URI, mediante el atributo nsURI de EPackage. Como en los constructores de la especificación algebraica ya se ha seguido la convención de incluir el prefijo del metamodelo al que pertenecen, es suficiente con relacionar este prefijo con la URI del EPackage al que pertenece. En esta implementación del puente M1 se hace mediante un comentario al principio de la lista de términos. Para que la implementación del puente sea capaz de identificar esta información, se espera la aparición de un comentario con el siguiente formato justo antes del conjunto de términos:




```
***$ p.nsPrefix - "p.nsURI"
```

Nótese el símbolo de dólar ($) que sucede a los tres asteriscos (sintaxis de comentario en Maude). El puente M1 reconoce mediante este símbolo que lo que viene a continuación es la información acerca de un metamodelo. Nótese también que la URI debe ir entre comillas.

Utilizando una vez más el Ejemplo 3.4 [1] para ilustrar lo expuesto:

```
***$ mXSD - "http://es.upv.dsic/issi/moment/miniXSD"
Set{
 (mXSD-Schema  'ID0 "Actors" OrderedSet{'ID1} )
 (mXSD-Complex 'ID1 "Actor" OrderedSet{'ID2 'ID3 'ID4} )
 (mXSD-Simple  'ID2 "ActorID"   "String" )
 (mXSD-Simple  'ID3 "Bio"       "String" )
 (mXSD-Complex 'ID4 "ActName" OrderedSet{'ID5 'ID6} )
 (mXSD-Simple  'ID5 "FirstName" "String" )
 (mXSD-Simple  'ID6 "LastName"  "String" )
}
```

## 4.2.2. Diseño de la solución

Se ha abordado el problema como un caso especial de compilador. Se trata entonces de "compilar" los términos algebraicos para transformarlos en un modelo EMF, facilitado en su codificación XMI. El lenguaje con el que ha de trabajar el compilador es el subconjunto de Maude utilizado por MOMENT para expresar modelos, cuya gramática abstracta se mostraba en el capítulo anterior, en la Sección 3.4.1, “Gramática para el sublenguaje de términos”.

En la Figura 4.1, “Proceso de Análisis de lenguaje” se presenta un esquema del funcionamiento clásico de un compilador. Se dispone de una fuente de información en formato textual denominada código fuente, y se quiere llegar a una representación más adecuada en forma de modelo EMF. 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:

Analizador 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 iden-

[1] Se obvian los IDs por claridad

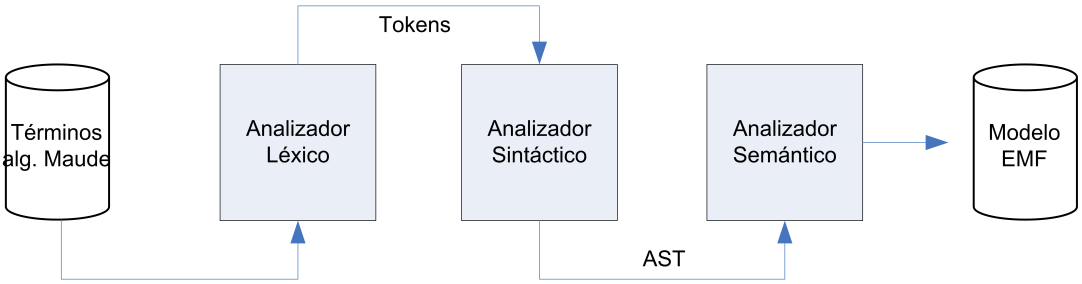



**Figura 4.1. Proceso de Análisis de lenguaje**

tificados 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*).

Analizador 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*.

Analizador semántico

El análisis semántico del árbol aplica las reglas semánticas que detectan incoherencias según el lenguaje que se esté reconociendo. Si el

AST supera esta fase es corriente enriquecerlo con una o más fases de análisis semántico. Durante estas fases, denominadas comúnmente pasadas, el AST es modificado, reestructurado, optimizado, etc. hasta conseguir el resultado final.

El programa que recorre el AST realizando una de estas fases se denomina analizador semántico o de forma más general *Tree Walker* (recorredor de árboles).

En nuestro caso después de obtener el AST enriquecido extraemos el modelo EMF que se obtiene a partir del conjunto de términos algebraicos de origen. En la Figura 4.2, "Visualización del AST de un modelo miniXSD" se muestra una visualización del AST obtenido para el modelo de Ejemplo 3.4, "Un modelo de miniXSD proyectado en términos Maude".

Para generar los analizadores léxico, sintáctico y semántico requeridos se ha empleado el generador de parsers ANTLR [24], cuyo funcionamiento es similar a los conocidos Bison y Flex. ANTLR es un generador de intérpretes de última generación. Es capaz de generar el analizador léxico, el sintáctico y además también semántico, dando cobertura de esta forma a todo el proceso de reconocimiento. ANTLR toma como entrada una gramática definida mediante un lenguaje propio cuya sintaxis está basada en EBNF. Se pueden definir los tres tipos de analizadores, es decir, léxico, sintáctico y semánticos. Lo que hace ANTLR con esta gramática es generar el código Java que implementa el analizador correspondiente. Para obtener más información de ANTLR se recomienda acudir a la página web [24] o ya en castellano, a la tesis escrita en [17].

A partir de la sintaxis abstracta mostrada en la Sección 3.4.1, "Gramática para el sublenguaje de términos" se ha derivado una sintaxis concreta que la implementa. Las diferencias son únicamente de implementación: una sintaxis concreta refina la abstracta añadiéndole construcciones para solucionar problemas, debidos principalmente a ambigüedades, que surgen en el proceso de implementación.

En este caso el principal problema encontrado ha sido la dificultad de distinguir entre atributos y referencias ya que se da un caso en el que tienen la misma re-


**Figura 4.2. Visualización del AST de un modelo miniXSD**

presentación. Esto ocurre con las instancias de un atributo o referencia con cardinalidad múltiple cuyo valor es no definido, esto es, el conjunto vacío. Para solucionarlo la implementación realizada declina la tarea de clasificación de

atributos y referencias a una fase de análisis semántico, lo cuál resulta bastante más lógico, ya que sintácticamente el sublenguaje de Maude no distingue entre ambos.

El resultado de este diseño por lo tanto es un compilador/traductor con cuatro fases:

- Análisis léxico
- Análisis sintáctico
- Fase de enriquecimiento del AST: clasificación de características estructurales (Structural Features) en atributos y referencias
- Fase de traducción: instanciación del modelo EMF a partir del AST enriquecido

Cada uno de estos analizadores ha sido implementado en una gramática de ANTLR y todos ellos han sido generados automáticamente.

La mayor parte de la lógica se concentra en la última fase que ya trabaja directamente con el AST enriquecido. Los pasos que sigue para producir el modelo deseado son los siguientes:

1. A partir del símbolo del constructor de un término se identifican el metamodelo y la clase del elemento representado por el término. El analizador semántico crea una instancia dinámicamente, que ya es la proyección en EMF del término algebraico. Conforme se crean las instancias se guarda una dupla instancia-ID en una tabla de búsqueda, para luego poder resolver las referencias por ID a referencias entre instancias.

2. Seguidamente se rellenan los atributos de la instancia creada con los valores de los subtérminos encontrados en el término, previa conversión de tipos Maude -> EMF.

3. Una vez se han procesado todos los términos se resuelven las referencias entre ellos, utilizando la tabla de búsqueda creada en el primer paso.

## 4.3. Integración de MOMENT en una herramienta de usuario

La segunda parte del proyecto se propone integrar, mediante los puentes desarrollados en la fase previa, la plataforma de gestión de modelos provista por MOMENT con un representante del campo MDE como es EMF. Para conseguirlo se ha desarrollado un prototipo que no sólo consigue automatizar todo el proceso y hacerlo transparente al usuario, sino que además lo hace de forma amigable e integrada en un entorno de trabajo estándar como es Eclipse.

A continuación se introduce de una manera abstracta el diseño del prototipo. Primero se presenta un análisis del problema que se quiere resolver y después se introduce el diseño adoptado. La intención es mostrar las correspondencias entre el espacio del problema y el espacio de la solución, en pos siempre de una mejor y más completa comprensión del artefacto software diseñado.

### 4.3.1. Análisis del problema

En primer lugar se presenta una visión global del proceso que se quiere automatizar, a continuación se desglosa el problema en varios subprocesos y se analiza cada uno por separado, extrayendo en cada uno los conceptos o elementos que participan, utilizando para ello diagramas de clases parciales (UML). Al integrar los elementos y conceptos que aparecen en todos los subprocesos se obtiene un modelo analítico del problema, que es mostrado en la última parte del capítulo. Finalmente se detallan los pasos realizados para obtener la parte central del prototipo a partir de este modelo analítico.

#### 4.3.1.1. Visión

El proceso de trabajo con modelos en el prototipo diseñado se realiza de la siguiente manera. En primer lugar el analista elige un metamodelo con el que trabajar. Mediante una herramienta de modelado capaz de producir modelos EMF, el analista compone u obtiene automáticamente una serie de modelos con los que trabaja en un proyecto. Cuando desee realizar una operación de gestión con dichos modelos, el prototipo aquí desarrollado ha de solicitar al analista:

- El metamodelo o metamodelos que intervienen.

- Los modelos con los que operar.
- La operación a realizar, posiblemente definida por una persona distinta al analista.
- Los axiomas que definen las relaciones de equivalencia entre los modelos que intervienen, posiblemente definidos utilizando otra herramienta externa a este prototipo.

A partir de aquí es responsabilidad de la herramienta realizar la ejecución de la operación y presentar los resultados. Los pasos que el prototipo ha de realiza en este proceso se detallan a continuación, resaltando además las implicaciones que tiene cada paso a nivel de implementación:

1. Inicializar el sistema Maude y cargar en él los módulos que componen el núcleo de MOMENT.
2. Obtener los metamodelos y realizar la proyección al ET Maude, incluyendo los axiomas de equivalencia proporcionados por el usuario en caso de que los haya. Esto es, generar las especificaciones algebraicas de los metamodelos, incluir en ellas los axiomas proporcionados, y cargarlas en el sistema Maude.
3. Realizar la proyección de los modelos a nivel M1 en el ET Maude. Esto es, generar las listas de términos que describen a los modelos implicados en la operación.
4. Cargar la operación en caso de que sea necesario e invocarla.
5. Recoger el resultado de la operación en el ET Maude, que serán un conjunto de términos, y proyectarlo en el ET Ecore. Esto es, de alguna manera procesar y reconocer los términos para reconstruir un modelo EMF que pueda ser posteriormente codificado en XMI para ser almacenado en el fichero indicado por el usuario como resultado.
6. Notificar al usuario si se ha producido algún error en uno de los pasos anteriores.

#### 4.3.1.2. Proceso de carga del Kernel

El primer paso a realizar en un proceso de gestión de modelos con MOMENT es la carga del núcleo o *kernel* de MOMENT en el sistema Maude. El kernel de MOMENT está compuesto por un conjunto de ficheros estático, es decir que no cambia de invocación a invocación. Estos ficheros deben ser ejecutados en el intérprete Maude en un orden determinado para lograr la correcta inicialización de MOMENT.

Hay una interacción de este proceso con el Sección 4.3.1.3, "Proceso de ejecución de operaciones" y es que tanto los módulos de los metamodelos involucrados en la invocación como el módulo de la operación invocada dependen o pueden depender, según el escenario de invocación, de los módulos que forman el kernel.

Podría ser interesante disponer de un sistema de configuración para este proceso que permita establecer la localización y el orden de carga de los ficheros que forman el kernel, así como indicar qué módulos son relevantes para el proceso de invocación y en qué sentido.

En el diagrama se muestran los elementos que es necesario tomar en cuenta para este sistema de configuración. Las clases que aparecen son las siguientes:

| | |
|---|---|
| `KernelImport` | Representa un fichero del kernel de MOMENT. Sus atributos permiten indicar el nombre del fichero así como en qué partes del proceso de invocación intervienen los módulos definidos en el fichero. |
| `MaudeModule` | Representa a un módulo de Maude. Está definido por su nombre y un número de parámetros |
| `MaudeModuleParameter` | Representa a un parámetro en un módulo Maude. Sus atributos incluyen la etiqueta (*label*) del parámetro así como el tipo (*ADTname*) |

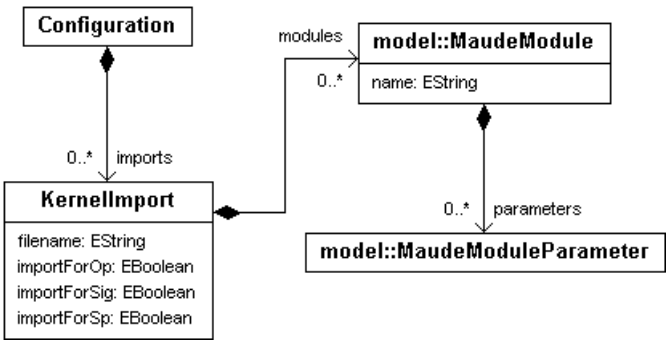



**Figura 4.3. Elementos de la configuración del kernel**

El problema de la interacción con Maude no se considera en esta fase previa de análisis y se pospone hasta la fase de diseño. Ver “MDT” más adelante en este mismo capítulo.

### 4.3.1.3. Proceso de ejecución de operaciones

La ejecución de una operación de gestión de modelos es efectivamente el resultado final de este prototipo. En esta fase se integra la mayor parte del trabajo realizado y por ello es el más complejo. Este proceso engloba los pasos 2, 3, 4 y 5 del esquema introducido en la Sección 4.3.1.1, “Visión”.Estos cuatro pasos se pueden descomponer en los siguientes problemas:

- Obtención de los metamodelos.
- Definición del operando compuesto (si es necesario).
- Proyección de elementos en el ET Maude.
- Proceso de carga en Maude de todos los módulos implicados.
- Ejecución/Invocación de la operación.
- Procesado y proyección en EMF del resultado.

#### Obtención de los metamodelos

En lugar de solicitarlos al usuario, la obtención de los metamodelos se puede realizar de forma automática, basándose en el sistema de registro de EPackages

que EMF proporciona. EMF exige registrar, mediante su API, todos los EPackages que sean metamodelos de los modelos que se vayan a manipular en una sesión de trabajo. En el caso de MOMENT únicamente es necesario extender este mecanismo para poder asociar los axiomas de usuario a cada metamodelo, tal y como muestra la Figura 4.4, "Elementos que describen el registro de Metamodelos".

| | |
|---|---|
| `MetamodelRegistry` | Proporciona los servicios que permiten obtener el metamodelo asociado a una URI o, equivalentemente, a un EPackage. Se basa en los servicios de ecore::EPackage::Registry para proveer los suyos. |
| `Metamodel` | "Wrapper" alrededor de un EPackage que modela el concepto de Metamodelo a nivel analítico. Name y nsURI son dos atributos derivados ya que su valor proviene del EPackage que contiene propiamente al metamodelo. Una de las funciones de `Metamodel` es relacionar el EPackage asociado con un conjunto de axiomas específicos de metamodelo que pueden ser proporcionados por el usuario |
| `AxiomList` | Clase que representa un conjunto de axiomas |
| `Axiom` | Clase que representa un axioma |

## Definición del operador

Si la operación a realizar es compuesta, será necesario que se haya definido previamente el operador compuesto que la realiza. Recuérdese que un operador compuesto se define como una función Maude ubicada en un módulo con unas características determinadas, que se exponen en la Sección 3.5.2.1, "Definición de operadores complejos en MOMENT". De manera más abstracta, una operador compuesto tiene un nombre, un número de parámetros, una serie de dependencias y finalmente una definición axiomática.

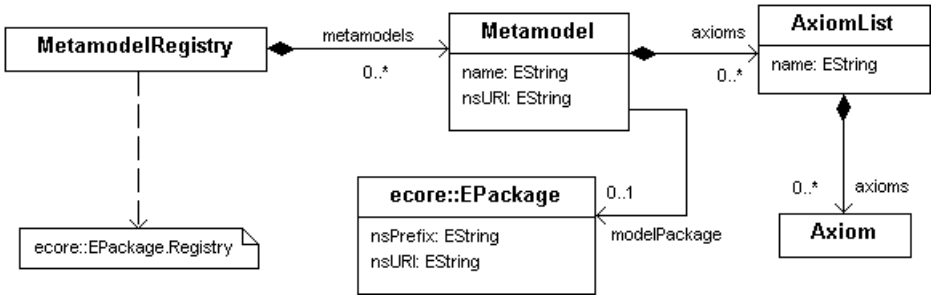



**Figura 4.4. Elementos que describen el registro de Metamodelos**

Además del código fuente del módulo de Maude que lo define, para la interacción con otros procesos es necesario conocer los siguientes atributos de la operación:

- El nombre - para poder crear el comando de invocación.
- El número de parámetros - para poder mostrar al usuario un editor de invocación.
- El nombre del módulo - para poder incluirlo en la jerarquía de importación del módulo de operaciones auxiliares de las especificaciones algebraicas de los metamodelos implicados.

El diagrama de clases de la Figura 4.5 describe los elementos que intervienen en este proceso.

| | |
|---|---|
| `Operator` | Representa al operador compuesto. Conoce la definición axiomática y el nombre del operador. Mediante la asociación *`parameters`* conoce los operandos que acepta el operador |
| `Operand` | Representa un operando (o parámetro formal en un contexto funcional) de un operador. Conoce su nombre y su tipo. |
| `OperandTypes` | Tipo de datos enumeración que modela los diferentes tipos de operandos que un operador puede aceptar |

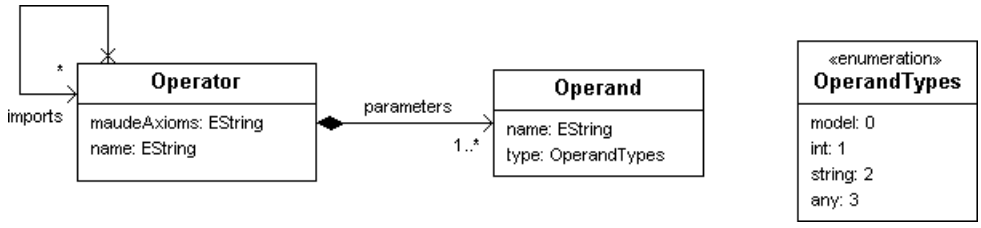



**Figura 4.5. Elementos que describen el concepto Operador Compuesto**

El problema de trabajar directamente con el módulo Maude que define a un operador compuesto (en efecto, con el fichero en lenguaje Maude que especifica el operador), es que para poder extraer los datos mencionados arriba sería necesario disponer de un analizador sintáctico del lenguaje Maude más o menos completo. Dado que no se dispone de ninguno, y la creación de uno desde cero supondría una gran inversión de esfuerzo, la solución adoptada se apoya en el editor para recoger toda esta información y guardarla en una instancia de la clase `Operator`, incluida la definición axiomática que se aloja en el atributo *`maudeAxioms`*. Cuando se necesite disponer de la expresión para cargarla en MOMENT, el módulo de la operación se genera mediante una plantilla de Velocity.

## Proyección de elementos en el ET Maude

Durante el proceso de invocación es necesario obtener la representación en Maude de varios elementos, por ejemplo tanto los metamodelos como los modelos implicados. Gracias a los puentes tecnológicos desarrollados entre EMF y Maude, apenas queda trabajo por realizar aparte de integrar dichos puentes en el modelo de análisis. La Figura 4.6, “Elementos que intervienen en la proyección a Maude” muestra los conceptos de análisis que modelan la interacción con el ET Maude.

`IMaudeable`

Interfaz para los conceptos que son capaces de proporcionar una representación de sí mismos en Maude, en forma de uno o más módulos. Esta representación se obtiene mediante el método

```
MaudeModule[] toMaudeModule();
```

Tanto `Operator` como `Metamodel` implementan esta interfaz

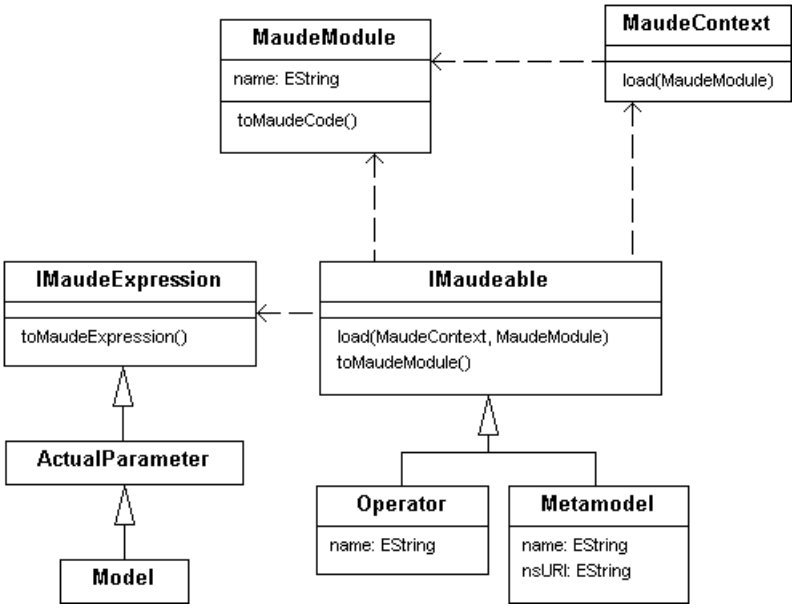



**Figura 4.6. Elementos que intervienen en la proyección a Maude**

`IMaudeExpression` — Interfaz para aquellos conceptos que constituyen una expresión en Maude, pero que no tienen la entidad suficiente para constituir un módulo por sí mismos. El ejemplo más claro son los modelos, que no son más que una expresión o parte de una expresión en el lenguaje Maude.

`Metamodel` — La implementación de `IMaudeable` utiliza el puente EMF->Maude de nivel M2 para obtener la representación algebraica de un metamodelo en Maude. Como resultado se obtienen los tres módulos (en forma de tres instancias de MaudeModule) que constituyen la especificación algebraica del metamodelo

`Model` — Representa a un modelo. Evidentemente, la implementación de `IMaudeExpression` utiliza el puente EMF -> Maude

| | |
|---|---|
| | de nivel M1 para obtener la representación algebraica de un modelo en forma de cadena. |
| `MaudeModule` | El método<br>`EString toMaudeCode();`<br>proporciona la representación textual en Maude de este módulo. |
| `MaudeContext` | Esta clase representa el punto de interacción con Maude. El método<br>`EBoolean load(MaudeModule x);`<br>permite cargar un módulo en Maude, representado por una instancia de MaudeModule. |

La clase `Model` se sirve del puente M2 desarrollado para implementar el método `toMaudeExpression`. Análogamente, la clase `Metamodel` utiliza el mismo puente a nivel M2 para implementar el método `toMaudeModule`, aunque en lugar de devolver una cadena devuelve una instancia de `MaudeModule`.

Nótese como `Operator` implementa la interfaz `IMaudeable`. Esto es coherente con lo que se comentaba en el apartado anterior: no se trabaja directamente con la especificación del operador compuesto en código Maude, sino con una abstracción de esta especificación en forma de instancia de `Operator`.

### Proceso de carga de módulos en Maude

De forma previa a la invocación es necesario cargar tanto la operación si es compleja como los metamodelos. En el caso de que la operación sea compleja, hay que ajustar la especificación algebraica de los metamodelos tal y como se comenta en la Sección 3.5.2.1, “Definición de operadores complejos en MOMENT” para conseguir que el módulo de operaciones auxiliares importe al módulo de la operación.

De esta manera, aunque en la Sección 3.5.2.1, “Definición de operadores complejos en MOMENT” se afirmaba que el nombre del módulo de la operación era

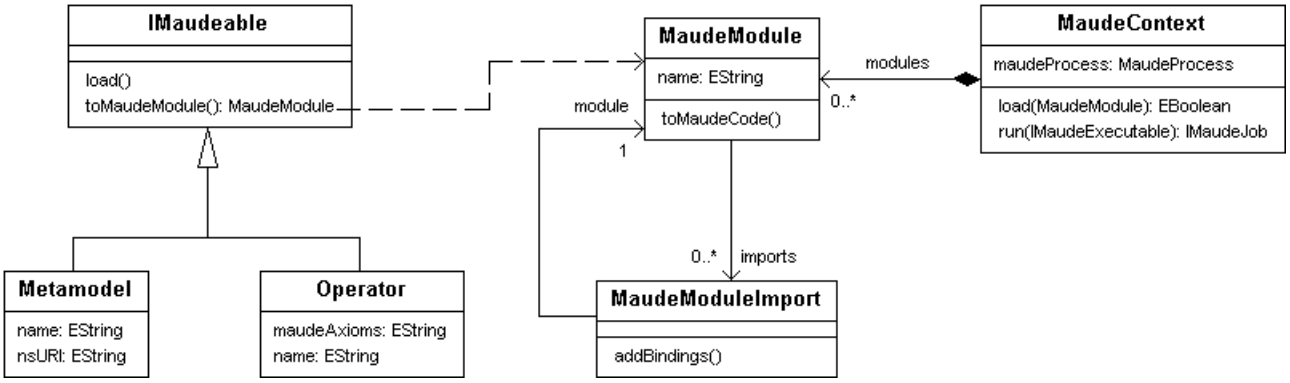



**Figura 4.7. Elementos del proceso de carga de módulos en Maude**

arbitrario, a efectos prácticos es necesario conocerlo para poder automatizar el proceso de carga. En el diagrama muestra los participantes en el proceso, el único elemento nuevo es `MaudeModuleImport`, la clase que modela la importación de un módulo por otro. La asociación *`module`* mantiene una referencia al módulo importado.

Suponiendo que se dispone de una instancia de `Metamodel` y una instancia de `Operator` que se quieren cargar para ejecutar una operación, para cumplir con lo expuesto en la Sección 3.5.2.1, "Definición de operadores complejos en MOMENT" se realizan los siguientes pasos:

1. Obtener los tres módulos que representan la especificación algebraica del metamodelo, así como el módulo para la operación, mediante la invocación del método `toMaudeModule` en cada instancia.

2. En el módulo de operaciones auxiliares de la especificación algebraica, manipular la instancia que representa al módulo de operaciones auxiliares para que en lugar de importar MOMENT-OP como ocurre por defecto, importe al módulo obtenido para el operador.

3. Cargar todos los módulos mediante el método `load` de una instancia de `MaudeContext`. El problema de obtener dicho contexto Maude (que además debe tener el kernel de MOMENT cargado) se abordará en el momento oportuno.

La clase `MaudeContext` encapsula toda la interacción con Maude, abstrayendo los detalles de la implementación subyacente. Cada instancia de esta clase se asocia con un proceso único de Maude, es decir, una instancia representa un contexto de trabajo con Maude, y diferentes instancias representarán diferentes contextos. La asociación *`modules`* mantiene una lista de los módulos cargados hasta el momento en el proceso de Maude asociado la instancia.

## Invocación de la operación

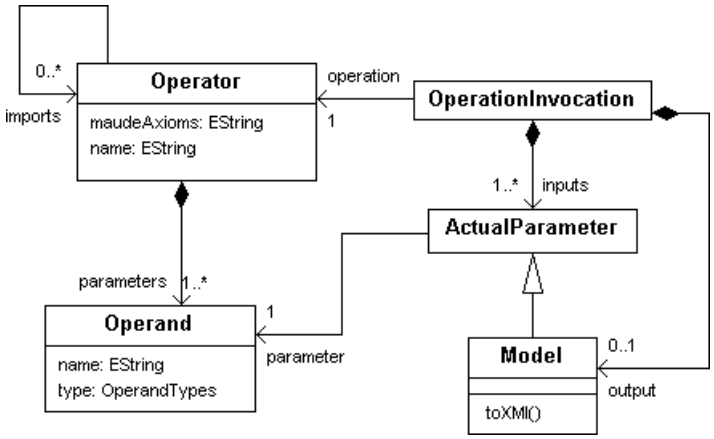



**Figura 4.8. Elementos que intervienen en la invocación**

Suponiendo que todos los módulos requeridos ya se encuentran cargados en Maude, el siguiente paso es invocar la operación. El diagrama muestra los elementos que modelan este proceso, en el cual aparecen algunas clases nuevas:

`IMaudeExecutable` — Interfaz que representa un tercer tipo de interacción con Maude: elementos o conceptos que pueden ser ejecutados en Maude.

OperationInvocation — Clase que modela una invocación. Conoce el operador a invocar y los parámetros de entrada, que pueden y suelen ser modelos. Gracias a que implementa `IMaudeExecutable`, puede ser ejecutado en Maude mediante el método `run` de `MaudeContext`.

La ejecución de una operación en Maude mediante `MaudeContext.run()` devuelve un objeto del tipo `IMaudeJob`. Esta clase pertenece a la librería MDT (Maude Development Toolkit) [15] y proporciona métodos para obtener el resultado de la operación (ver "MDT" más adelante).

### Proyección del resultado en EMF

Si la operación ha ido bien y no se han producido errores, se obtendrá un conjunto de términos algebraicos en código Maude que representan el modelo resultado de la operación. Para proyectarlos en EMF y poder proporcionarle el resultado al usuario no hay más que utilizar el puente Maude -> EMF a nivel M1.

## 4.3.2. Diseño de la solución

A partir de los modelos conceptuales presentados en el análisis del problema se ha construido el modelo de clases del prototipo. Para ello se han integrado todos los modelos parciales presentados y el resultado se ha refinado en varias etapas de un ciclo de desarrollo iterativo hasta obtener un modelo de clases que da un soporte satisfactorio al problema. El modelo presentado aquí pertenece al espacio del diseño de la solución. En él se consideran únicamente aspectos del problema, sin tener en cuenta otras cuestiones de la implementación como la integración en Eclipse o la interfaz gráfica

Mediante las facilidades de generación de código de EMF se ha obtenido una implementación en Java de la estructura de este modelo. No sólo eso, gracias a EMF se ha obtenido otra ventaja muy importante: la persistencia automática en XMI. Esta facilidad demuestra su utilidad en diversas ocasiones a lo largo de la solución:

- Para persistir en ficheros las definiciones de operadores compuestos, simplemente con serializar una instancia de `Operator`.

- Para almacenar un histórico de las invocaciones realizadas por el usuario, mediante serialización instancias de `OperationInvocation` que se almacenan en el espacio de trabajo de Eclipse y persisten entre sesiones de trabajo

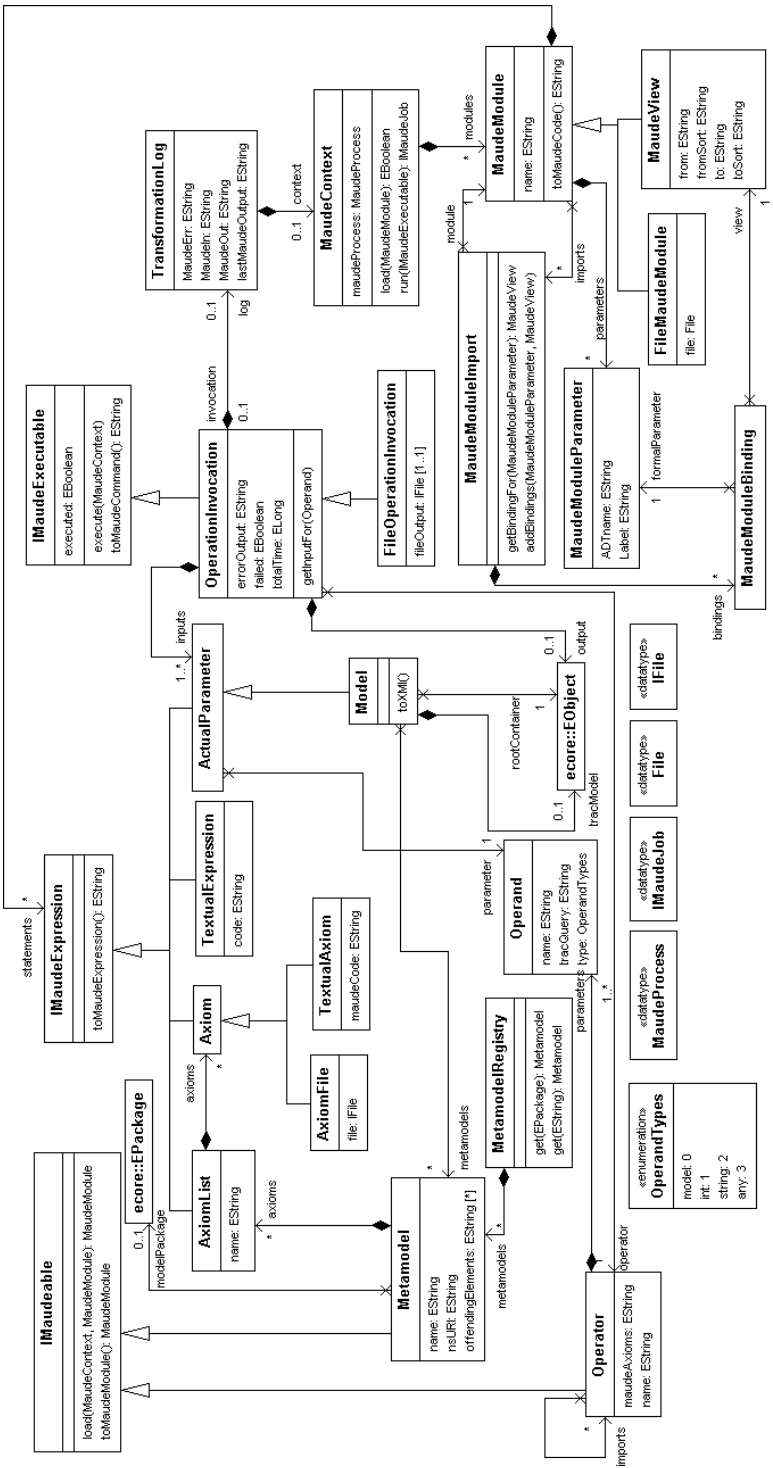



**Figura 4.9. Modelo completo de diseño**

- Para almacenar el fichero de configuración del kernel (ver Sección 4.3.1.2, "Proceso de carga del Kernel"), de forma similar a los dos casos anteriores.

Tras rellenar/completar el código generado e integrar los puentes se tiene una librería de clases que ofrece las facilidades necesarias para integrar MOMENT con EMF, pero aún no se ha obtenido un ejecutable ni interfaz gráfica. Ese es el objetivo de la integración en Eclipse. Los detalles de esta integración se discuten en el capítulo siguiente.

#### 4.3.2.1. Patrones de generación de código EMF

¿Qué tipo de código genera EMF? La primera cuestión que se observa es que una clase Ecore, es decir una `EClass`, da lugar a dos elementos en Java: una interfaz y una clase que implementa dicha interfaz. Por ejemplo, el código generado para la clase `MaudeModule` es una interfaz Java:

```
public MaudeModuleimplements EObject {
}
```

y la correspondiente clase que la implementa:

```
public MaudeModuleImpl extends EObjectImpl
    implements MaudeModule {
}
```

Esta separación entre interfaz e implementación es un patrón de código o "*idiom*" bastante extendido en Java. En EMF además se utiliza para permitir herencia múltiple.

Por otro lado, nótese que la interfaz extiende a `EObject` por una parte, y la clase extiende `EObjectImpl` por otra.`EObject` es el equivalente de EMF para`java.lang.object` en Java. `EObject` ofrece una serie de servicios básicos para todas las clases generadas por EMF relacionados con la reflexión/introspección, permitiendo acceder de forma genérica a las propiedades de la clase y al nivel meta.

El código generado para atributos y referencias de una clase tiene el aspecto que cabría esperar. El código generado además se encarga de mantener la integridad referencial en casos como asociaciones bidireccionales, agregaciones (que en EMF son siempre disjuntas), etc.

La idea principal a resaltar, sin entrar en más detalles sobre cómo genera EMF el código para cada característica de Ecore, es que el resultado es un código limpio, simple, eficiente y completo. EMF no se apoya en grandes clases base, su librería *runtime* es ligera. La idea es que el código generado se parezca lo máximo posible al código que se hubiera producido a mano.

Finalmente, es importante mencionar dos clases añadidas que se generan para un modelo: una factoría (*factory*) y un paquete (*package*). El modelo de programación EMF recomienda encarecidamente el uso del patrón de diseño *Abstract Factory*[8] para crear instancias. La clase *factory* generada implementa este patrón incluyendo un método *create* para cada clase del modelo.

En cuanto a la clase *package* generado, aquí se incluye la especificación e implementación del nivel meta del modelo, muy útil para realizar consultas de introspección y el acceso genérico a modelos.

Para más detalles sobre el código generado por EMF se recomienda acudir a [1] y [20].

#### 4.3.2.2. Integración de los puentes

Como se ha comentado anteriormente, los puentes desarrollados entre EMF y Maude han sido integrados en el modelo de clases A continuación se especifica los métodos dentro del esquema conceptual que realizan dichos puentes:

| | |
|---|---|
| EMF->Maude M2 | `EString Metamodel.toMaudeModule();` |
| EMF->Maude M1 | `EString Model.toMaudeExpression();` |
| Maude->EMF M1 | `Model MomentEngineFactory.createModel(String terms);` |

#### 4.3.2.3. Ejemplo de colaboración

La Figura 4.10, “Ejemplo de colaboración” muestra una colaboración típica entre objetos del modelo, en este caso para la invocación de una operación. El diagrama utiliza la notación estándar UML para diagramas de colaboración.

A la izquierda del cliente se muestra los datos que éste posee antes de iniciar el proceso: el operador y los modelos de entrada (input1).

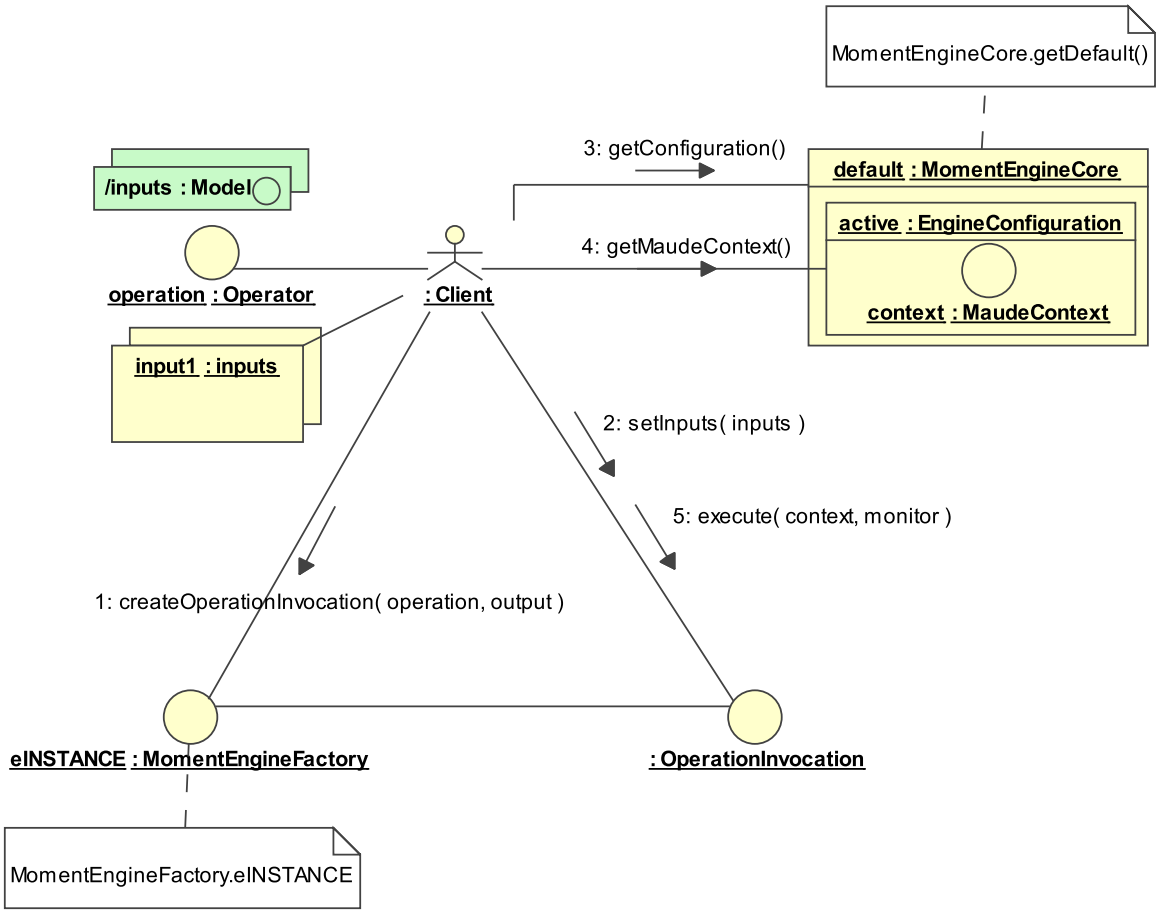



**Figura 4.10. Ejemplo de colaboración**

Por otro lado, en los mensajes 1 y 3 se puede ver que el cliente accede a dos instancias cuya procedencia no está clara. Tanto `MomentEngineFactory` como `MomentEngineCore` son dos casos del patrón de diseño *Singleton* [8]. Este patrón se aplica a una clase para conseguir las siguientes propiedades:

- Asegurar que sólo existe una instancia de la clase en todo el sistema, por ejemplo sólo debe existir una fábrica.
- Proporcionar un punto de acceso global a dicha instancia.

*MomentEngineFactory.eINSTANCE*, una variable de clase estática, y `MomentEngineCore.getDefault()`, un método público estático, son los puntos de acceso en este caso.

La clase `MomentEngineFactory`, generada parcialmente por EMF, contiene métodos para crear o importar instancias de todos los tipos del esquema conceptual. El método utilizado en el diagrama simplemente crea una nueva instancia de `OperationInvocation` asociada a una operación y a un fichero de salida determinados.

La clase `MomentEngineCore` es, como se detallará en el siguiente capítulo, el punto de entrada de la aplicación en Eclipse. En la colaboración se utiliza para obtener una instancia de `EngineConfiguration`. `EngineConfiguration` es una clase de soporte que almacena constantes comunes a todas las clases y otros parámetros que, en conjunto, conforman la configuración del prototipo. Aquí, por ejemplo, proporciona la instancia del contexto de Maude activo, asegurando que el contexto proporcionado tenga siempre el kernel de MOMENT precargado, y además que sólo hay un proceso Maude activo durante la ejecución de la aplicación. Esta y otras tareas de la misma índole son la responsabilidad de la clase `EngineConfiguration`.

El contexto Maude es necesario para poder realizar la invocación de la operación mediante el método *`OperationInvocation.execute()`*, cuya implementación se muestra a continuación:

```
public void execute (MaudeContext context) {
  EngineConfiguration engineConf = MomentEngineCore.getConfiguration();
  MaudeModule[] kernelImportsForOp = (MaudeModule[]) ArrayUtils.addAll(
   engineConf.getMomentKernelImportsForOp(),
   engineConf.getMomentKernelImportsForSp());
  MaudeModule opModule = operation.load(context, kernelImportsForOp);

  Set<Metamodel> metamodels = findMetamodels(this);
  for (Metamodel mm : uniqueMetamodels)
    mm.load(context, opModule));

  IMaudeJob invocationJob = context.run(this);

  if (!invocationJob.isFailed()) {
    String rawTerms = invocationJob.getOut();
    String processedTerms = postProcess(metamodels, rawTerms);
    Model result = MomentEngineFactory.eINSTANCE.createModel(processedTerms);
    setOutput(result);
}
```

#### 4.3.2.4. Interacción con Maude

El problema de la interacción con Maude se ha solucionado de forma muy sencilla gracias a la disponibilidad de una librería para invocar comandos Maude de forma programática desde Java. Esta librería es MDT(Maude Development Tools), y ha sido desarrollada en paralelo a este proyecto también desde el seno de MOMENT [15].

MDT proporciona una API que encapsula Maude en clases Java y simplifica en gran medida el problema de interacción que se planteaba en este proyecto. A continuación se ofrece una breve introducción a MDT.

**MDT**

Maude Development Tools está implementado como un plug-in para Eclipse que ofrece dos sistemas de interacción con Maude bien diferenciados. El que nos interesa aquí es el sistema denominado *Batch* en la referencia citada, que ofrece un modelo de interacción asíncrono.

La clase `MaudeProcessBatch` ofrece los métodos para configurar Maude, crear el proceso, enviar comandos y obtener los resultados. Estos son los métodos más relevantes:

```
public class MaudeProcessBatch {
 boolean execMaude() throws IOException;
 List<IMaudeJob> createAndRunJobs(String input) throws ParseException;
 void waitUntilFinish();
 void killMaude();
 ...
```

El método `createAndRunJobs` es el que permite enviar comandos a Maude. Los comandos se ejecutan de forma asíncrona, es decir, el método no se bloquea mientras Maude está trabajando sino que devuelve una lista de instancias de `IMaudeJob`. Esta lista contiene una instancia para cada módulo o comando contenido en el parámetro *`input`* [2].

La interfaz IMaudeJob permite comprobar si Maude ha completado el comando, si ha sido con éxito o no, obtener el resultado en forma de cadena y obtener los errores, entre otras cosas. En ocasiones puede ser interesante proceder de forma

[2] En el caso más común, se envían los comandos de uno en uno, y esta lista contiene un sólo elemento

síncrona y esperar hasta que Maude termine de procesar el comando. Para ello se proporciona el método `waitUntilFinish()`. Estos son los métodos más relevantes de `IMaudeJob`:

```
public interface IMaudeJob {
  public boolean isFinished();
 public void waitUntilFinish();
 public String getOut();
 public boolean isFailed();
 public String getError();
 ...
}
```

Estas dos clases ofrecen las primitivas necesarias para permitir la interacción con Maude en este prototipo. La clase `MaudeContext` del modelo de diseño se sirve de estos servicios para implementar los métodos que ofrece.

# Capítulo 5. Implementación

El siguiente paso es la integración de la herramienta en la plataforma Eclipse. A continuación se describe la estructura de plug-ins que conforman la herramienta. En segundo lugar se comentan los puntos de variabilidad de la implementación, ya que se estima que podrían resultar de interés en futuros trabajos.

# 5.1. Integración en Eclipse

**El Plugin en Eclipse**

Es conveniente para la comprensión de esta sección explicar, aunque sea a un nivel mínimo, el sistema de componentes utilizado por Eclipse. El componente unitario es el plug-in, compuesto básicamente por:

- *Puertos de entrada*, denominados en Eclipse *Extension Points*, sirven para permitir la interacción de otros *plug-ins* con éste
- *Conexiones a otros componentes*, denominadas *Extensions* en Eclipse, son una manera de interactuar con otros componentes
- *Binarios*, el código del *plug-in*, que es la otra manera de interactuar con otros componentes. Eclipse distingue dos tipos de interacción: como servidor, exportando una API, y como cliente, utilizando la API exportada por otros

Es importante recalcar que todo en Eclipse son *plug-ins*, así que una aplicación es cliente de los plug-ins que componen el núcleo de Eclipse, posiblemente de plug-ins de terceros, y no hay más. Cada *plug-in* encapsula un conjunto de funcionalidades, de manera que éste es el primer nivel de modularización a nivel de código fuente. Los *plug-ins* se agrupan en *features*, unidades de granularidad aún mayor cuya funcionalidad es proporcionar servicios añadidos, como la facilidad de actualizaciones automáticas o la instalación/desinstalación automática.

La aplicación está dividida en dos partes bien diferenciadas, implementada cada una como una *feature* de Eclipse. Recuérdese que la *feature* en Eclipse es una facilidad para agrupar varios *plug-ins* que constituyen un producto. Las *features* implementadas se denominan *MOMENT Engine* y *MOMENT Extensions*. La primera implementa la integración de MOMENT y EMF tal y como se ha visto hasta ahora, mientras que la segunda es una facilidad no destinada al usuario final y no es necesario instalarla. *MOMENT Extensions* expone directamente al usuario los puentes entre EMF y Maude, entre otras cosas, y que ha sido muy útil para facilitar el trabajo del equipo de MOMENT.

A continuación se describen ambas *features* en detalle.

## 5.1.1. Moment Engine

La *feature* Moment Engine agrupa el conjunto de plug-ins que constituyen el prototipo de integración de MOMENT en Eclipse.

### 5.1.1.1. Requisitos

La versión desarrollada aquí de Moment Engine requiere los siguientes componentes para funcionar:

- Eclipse 3.0 o superior
- Java 5 (1.5) o superior
- Eclipse EMF 2.0 o superior
- MDT (Maude Development Tools)

### 5.1.1.2. es.upv.dsic.issi.moment.engine.core

Este es el plug-in principal de todo el prototipo. Contiene todo el diseño desglosado en el apartado anterior, incluidos los puentes. Ahora bien, este plug-in no realiza integración alguna con Eclipse, sino que se limita a hacer accesible, como si fuera una librería de código, la implementación que se obtenía al final del diseño en el capítulo anterior.

En un contexto más amplio, se podría decir que la interacción entre los plug-ins que componen la *feature* Moment Engine sigue un comportamiento análogo al paradigma cliente-servidor. Este plug-in actúa de servidor exponiendo la API (Application Programming Interface) que permite trabajar con MOMENT y con los puentes, y el resto de plug-ins que componen la *feature* actúan como clientes, utilizando los servicios expuestos para proporcionar la funcionalidad del prototipo en Eclipse.

## Contribuciones del plug-in

### Extension Point axioms

El plug-in define un punto de extensión que permite, a cualquier otro plug-in que instancie esta extensión, asociar axiomas a un metamodelo determinado de forma estática. Por ejemplo, se utiliza para asociar los axiomas predeterminados al modelo Ecore.

## Código fuente

El plug-in consta de numerosas clases Java, repartidas en los paquetes que aparecen en la Figura 5.1, "Estructura de paquetes de MomentEngine.core" (alrededor de 12.000 líneas de código, aproximadamente la mitad de ellas generadas automáticamente).

### Estructura de paquetes

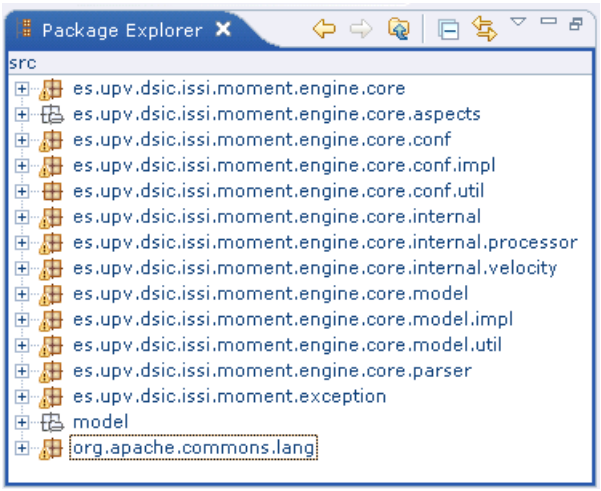



**Figura 5.1. Estructura de paquetes de MomentEngine.core**

A continuación se muestra un resumen de la estructura de paquetes del código. Para un nivel de detalle mayor, es recomendable acudir a los JavaDocs.

*es.upv.dsic.issi.moment.engine.core*
: Contiene sólo dos clases:

- `MomentEnginePlugin` - Punto de entrada del plug-in. Actúa además como *Service Locator* [1] o localizador central de servicios para el resto de clases. Esto quiere decir que hay una instancia de esta clase accesible siempre (de forma estándar mediante el método estático `getDefault`(), generado automáticamente por el asistente de Eclipse de creación de plug-ins ) y que esta instancia es capaz de proporcionar, bien sea de forma directa o indirecta, instancias de clases de servicio.

- `EngineConfiguration` - Esta clase centraliza varios servicios y parámetros de configuración utilizados por el resto de clases.

*es.upv.dsic.issi.moment.engine.core.model.**
Paquetes que implementan el modelo de diseño ilustrado anteriormente. Siguen la configuración habitual de EMF: el primero de los tres paquetes contiene las interfaces, el segundo contiene las implementaciones de éstas y el tercero contiene otras clases de ayuda menos relevantes. En estos tres paquetes se concentra la mayor parte de la lógica de diseño.

*es.upv.dsic.issi.moment.engine.core.internal.**
Clases de infraestructura o apoyo para diversas tareas de implementación.

*es.upv.dsic.issi.moment.engine.core.conf.**
Paquetes que contienen el código generado para el modelo de configuración del proceso de carga del Kernel de MOMENT (KernelConf).

*es.upv.dsic.issi.moment.engine.core.exception*
Paquete que contiene los distintos tipos de excepciones que la API puede producir.

*es.upv.dsic.issi.moment.engine.core.parser*
Contiene las clases generadas por ANTLR para el analizador de lenguaje.

[1] Service Locator es un patrón J2EE estándar o *Core J2EE Pattern*

### Otros artefactos software

**Anatomía de un plug-in Eclipse**

La estructura de un plug-in es simple. Dentro de la instalación de Eclipse hay una carpeta llamada `plug-ins`, que contiene los plug-ins instalados. Para instalar nuevos plug-ins sólo hay que copiarlos en esta carpeta.

Normalmente cada plug-in se almacena en una carpeta [2] . El nombre de la carpeta es el mismo que el del plug-in, seguido de un "_" y el número de versión del plug-in.

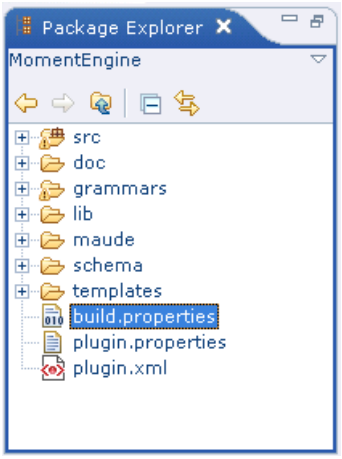
```
ECLIPSE
|
+---features
\---plug-ins
    +---es.upv.dsic.issi.moment.registerEMF_1.0.5
    |   |   plug-in.xml
    |   |   plug-in.properties
    |   |   registerEMF.jar
    |   |
    |   +---lib
    |
    ...
    +---es.upv.dsic.issi.moment.metamodels_1.0.6
    |
    ...
```

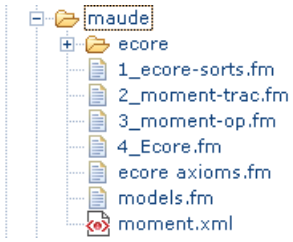
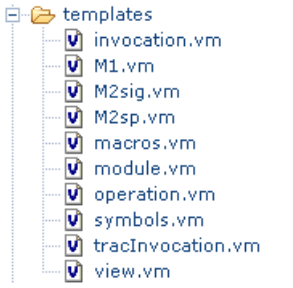
Dentro de esta carpeta se halla el contenido del plug-in, compuesto principalmente por un descriptor xml, el fichero "plugin.xml", y opcionalmente uno o varios ficheros .jar con el código del plug-in.

Además del código Java, este plug-in contiene diversos artefactos software que contribuyen al código fuente.

[2] A partir de la versión 3.1 de Eclipse, el método preferente es almacenar el plug-in en un jar en lugar de en una carpeta

**Tabla 5.1. Artefactos de código fuente**

| | |
|---|---|
|  | **Estructura de carpetas.**<br>• `src` - fuentes Java así como modelos EMF<br>• `grammars` - Gramáticas EBNF para ANTLR<br>• `maude` - Kernel de MOMENT<br>• `schema` - Definición del punto de extensión *axioms*<br>• `templates` - Plantillas de Velocity<br>• `plug-in.xml` - Descriptor del plug-in (Eclipse) |
|  | La carpeta `models` contiene los modelos EMF utilizados para generar parte del código.<br>• `MomentEngine.ecore` - modelo de clases de la solución<br>• `KernelConf.ecore` - submodelo para la configuración del kernel (ver Sección 4.3.1.2<br>Los correspondientes `.genmodel` son los modelos de configuración de generación de código. |
|  | La carpeta `grammars` contiene las gramáticas o descripciones EBNF a partir de las cuales ANTLR es capaz de generar, en código Java, los analizadores léxico, sintáctico y semánticos que son necesarios para el puente a nivel M1.<br>• `parser.g` - Gramática que describe los analizadores léxico y sintáctico (es tradicional definirlos en una misma unidad)<br>• `AttribsAndRefsPass.g` - *TreeWalker* que realiza la primera pasada de las descritas en la Sección 3.4.1 |

| | |
|---|---|
| | • `ReificationPass.g` - *TreeWalker* que realiza la segunda y última pasada de análisis semántico, produciendo ya el modelo EMF buscado en el proceso de análisis. |
|  | La carpeta `kernel` contiene los módulos que forman el kernel de MOMENT, así como la descripción del proceso de carga en el fichero `moment.xml`. Este fichero contiene una instancia de `kernelConf.Configuration`, tal y como está expuesto en Sección 4.3.1.2. |
|  | La carpeta `templates` contiene el conjunto de plantillas Velocity empleadas. |

#### 5.1.1.3. es.upv.dsic.issi.moment.engine.core.edit

Este plug-in contiene el código generado por EMF para auxiliar en la programación de interfaces gráficas basadas en MVC (Model-View-Controller).

**Dependencias**

- moment.engine.core

#### 5.1.1.4. es.upv.dsic.issi.moment.engine.launching

Este plug-in hace de puente para integrar en Eclipse a nivel programático los proceso de gestión de modelos provistos por `moment.engine.core`. Gracias a ello, es posible ejecutar operaciones de gestión de modelos en Eclipse de forma estándar y totalmente integrada.

Resaltar que este plug-in se limita a realizar la integración a nivel programático únicamente, sin proveer interfaces gráficas. Es el plug-in `moment.engine.ui` el que,

haciéndose cliente de este plug-in, facilita la interfaz gráfica. Esta es una forma habitual de proceder en Eclipse [9]

### Dependencias

- moment.engine.core

### Contribuciones del plug-in

#### org.eclipse.debug.core.launchConfigurationTypes

Este punto de extensión facilitado por Eclipse es un mecanismo configurable para añadir nuevos tipos de configuraciones de ejecución. El plug-in implementa este punto de extensión alrededor de la clase `OperationInvocation`. Es fácil ver que una instancia de esta clase contiene toda la información necesaria para ejecutar una operación de gestión de modelos.

## 5.1.1.5. es.upv.dsic.issi.moment.engine.ui

Este plug-in centraliza todas las contribuciones de interfaz gráfica de la *feature* MomentEngine. Se proporcionan el editor de invocaciones mostrado antes, un editor de operadores complejos, un asistente y varias contribuciones a menús contextuales.

El editor de operadores permite crear y editar operadores de forma sencilla. El operador es almacenado en una instancia de la clase Operation y persistido mediante XMI en un fichero con la extensión “mop” (MOMENT Operator). El editor está implementado mediante el *toolkit* gráfico “Eclipse Forms” que se incluye en la distribución estándar de Eclipse. En la Figura 5.2, “Editor de operadores complejos” se muestra una captura de pantalla del editor.

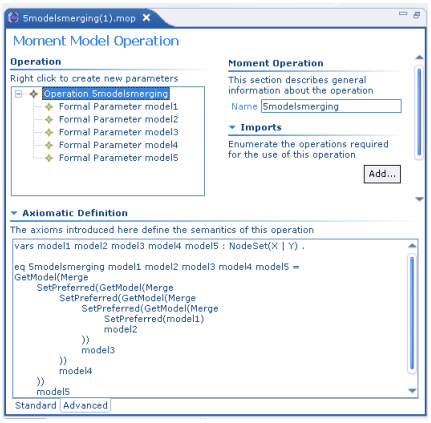



**Figura 5.2. Editor de operadores complejos**

El editor de invocaciones no tiene entidad propia, sino que está integrado en la metáfora de ejecución de operaciones de Eclipse. Aparece cuando el usuario expresa su intención de ejecutar una operación de gestión de modelos, por ejemplo mediante el menú contextual sobre un operador compuesto.

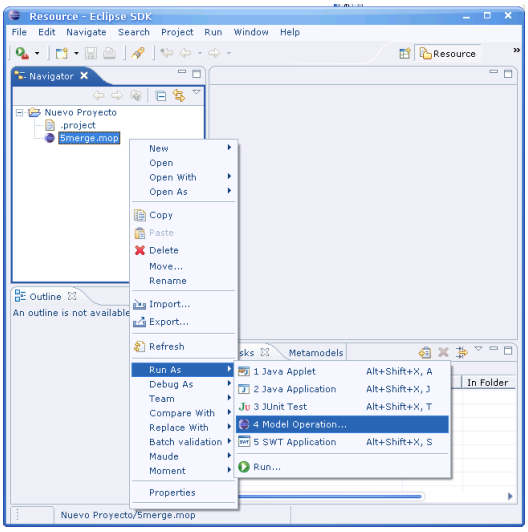



**Figura 5.3. Ejecutar operación de gestión de modelos**

## Dependencias

- moment.engine.core

- moment.engine.launching
- moment.engine.core.edit

## Contribuciones del plug-in

### org.eclipse.ui.editors

Este punto de extensión permite registrar nuevos editores a Eclipse, asociados a una extensión de fichero concreta. Este plug-in lo instancia para registrar el editor de operadores complejos con los ficheros con extensión “mop”.

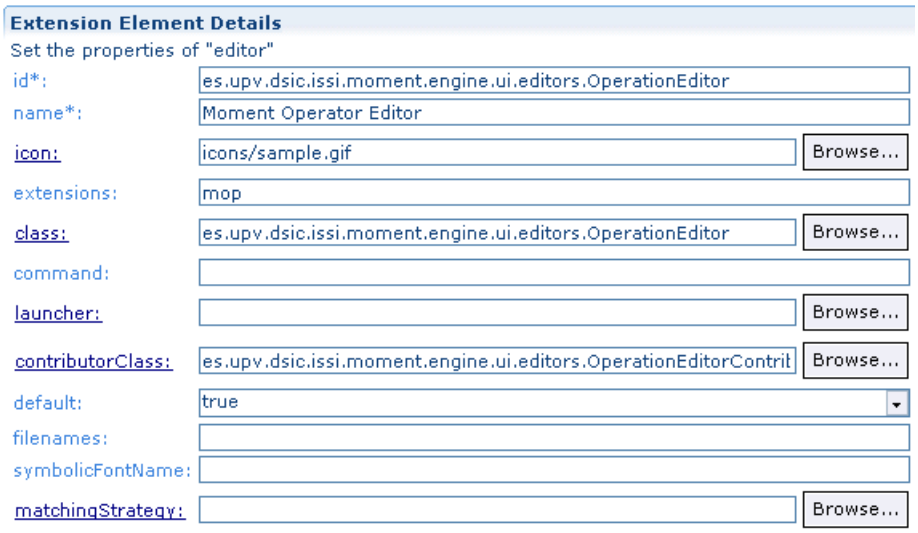



**Figura 5.4. Instanciación del punto de extensión *editors***

**Anatomía de un *Extension Point***

Cuando un plug-in quiere permitir que otros plug-ins extiendan o regulen porciones de su funcionalidad, lo hace declarando un punto de extensión, o *Extension Point* . Un punto de extensión establece un contrato, definido por un esquema XML especial y un conjunto de interfaces Java (para más información consultar la documentación del kit de desarrollo de Eclipse). Los plug-ins que quieran hacer uso de la extensión deben implementar este contrato, facilitando tanto los datos requeridos como implementaciones de las interfaces Java. La captura muestra el cuadro de diálogo mostrado por Eclipse para configurar la instanciación del punto de extensión *editors*. *editors* es el punto de extensión que permite crear nuevos editores en Eclipse. Los campos que aparecen en el cuadro de diálogo provienen del esquema XML que contiene la especificación del contrato del punto de extensión.Por ejemplo, el contrato requiere que se proporcione un nombre, un fichero con un icono, las extensiones de fichero que activan el editor, una clase java que lo implementa, etc. Asimismo, el contrato establece que la clase que implementa el editor (el campo *class*) tiene que implementar la interfaz del SDK de Eclipse `IEditorPart`:

```
package org.eclipse.ui;

public interface IEditorPart {
   public IEditorInput getEditorInput();
   public IEditorSite getEditorSite();
   public void init(IEditorSite site, IEditorInput input)
   public void createPartControl(Composite parent);
   public void setFocus();
   public void dispose();
   public void doSave(IProgressMonitor monitor);
   public void doSaveAs();
   ...
   ...
}
```

**org.eclipse.ui.newWizards**

Instancia este punto de extensión para proporcionar el asistente que permite crear un operador compuesto.

**org.eclipse.debug.ui.launchConfigurationTabGroups**

La instanciación de este punto de extensión registra el editor de invocaciones en la infraestructura de ejecuciones de Eclipse

**org.eclipse.debug.ui.launchShortcuts**

Esta instanciación hace aparecer la opción de ejecutar una operación de modelos en el menú contextual de un operador compuesto, tal y como se mostraba en la Figura 5.3, “Ejecutar operación de gestión de modelos”

## 5.1.2. Moment Extensions

Moment Extensions consiste en un conjunto de servicios añadidos que han demostrado ser útiles para el equipo de desarrollo de MOMENT, pero que sin embargo no presentarían utilidad evidente para un usuario final. Por ello la política adoptada es mantenerlos en una aplicación separada.

### 5.1.2.1. es.upv.dsic.issi.moment.ecore2maude.core

En primer lugar, `ecore2maude.core` proporciona una API muy simple, al estilo del patrón de diseño *Facade*[8], que permite utilizar los puentes EMF<->Maude de forma directa.

En segundo lugar, `ecore2maude.core` registra el puente M1 con la infraestructura de acceso a ficheros de EMF, lo que convierte al sublenguaje de términos algebraicos en una forma de codificación aceptada por EMF nativamente, lo cual pone a dicho sublenguaje al mismo nivel que XMI. Ha sido muy excitante poder serializar un modelo a un fichero de términos algebraicos y manipularlo como si fuera un modelo EMF con cualquier herramienta externa compatible con EMF, como por ejemplo el editor visual Eclipse UML de la compañía Omondo[3]. A pesar de ello, no era una funcionalidad requerida y no se invirtieron más esfuerzos en explorar otras posibilidades.

[3]http://www.omondo.com

### CVS

Este plug-in se encuentra en el CVS de MOMENT bajo el nombre de “MomentResource”

### Dependencias

- moment.engine.core

### Contribuciones del plug-in

#### org.eclipse.emf.ecore.extension_parser

La instanciación de este punto de extensión integra el sublenguaje de términos algebraicos como un formato de codificación de modelos en EMF

## 5.1.2.2. es.upv.dsic.issi.moment.ecore2maude.ui

Este plug-in utiliza los servicios ofrecidos por `moment.ecore2maude.ui` para proporcionar una interfaz gráfica que permite utilizar los puentes de forma manual.

Por ejemplo, si el usuario quiere obtener la especificación algebraica de un metamodelo, o la lista de términos algebraicos de un modelo, puede hacerlo utilizando un sencillo menú contextual que aparece en los ficheros que contienen modelos EMF (ver Figura 5.5, “Interfaz visual de los puentes”).

La operación inversa, esto es, recuperar el modelo EMF representado por un conjunto de términos algebraicos, se realiza de forma similar. Únicamente es necesario invocar el menú contextual de un fichero con extensión `.maude` o `.fm` (ver Figura 5.6, “Interfaz visual de los puentes (2)”). También es posible abrir estos ficheros con cualquier editor que acepte modelos EMF, como por ejemplo el editor por defecto en forma de árbol. Tan sólo es necesario registrar manualmente la extensión `.maude` o `.fm` con el editor deseado

Finalmente, este plug-in incluye otras contribuciones menos pulidas y de menor interés, como por ejemplo una contribución para visualizar el AST obtenido de un modelo como término algebraico (muy útil para depurar problemas).

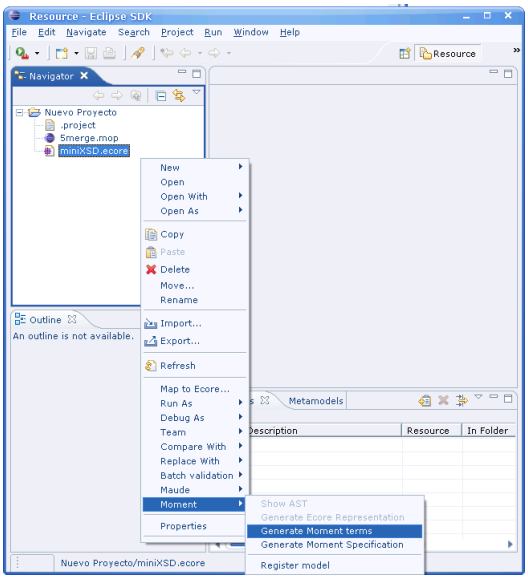



**Figura 5.5. Interfaz visual de los puentes**

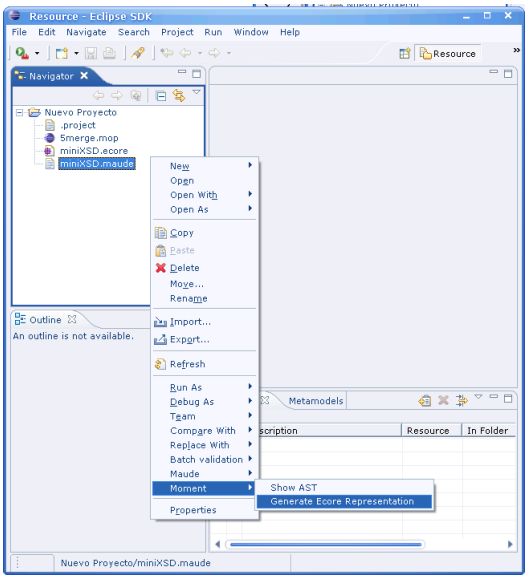



**Figura 5.6. Interfaz visual de los puentes (2)**

### 5.1.2.3. es.upv.dsic.issi.moment.registerEMF

Este sencillo plug-in expone una interfaz para el registro de EPackages de EMF.

La interfaz permite al usuario introducir EPackages en el registro mediante un menú contextual.

#### 5.1.2.4. es.upv.dsic.issi.moment.ecore2maude.metamodels

Este plug-in permite interactuar con el registro de metamodelos. Su funcionalidad incluye la posibilidad de ver, añadir y eliminar metamodelos del registro, así como registrar axiomas de usuario con un metamodelo. El registro de metamodelos se muestra en una *view* de Eclipse (ver Figura 5.7, “Registro de metamodelos”).

#### 5.1.2.5. es.upv.dsic.issi.moment.engine.conf.edit es.upv.dsic.issi.moment.engine.conf.editor

Plug-ins generados automáticamente por EMF que proporcionan un editor en forma de árbol para el fichero que contiene la configuración de carga del Kernel de MOMENT (ver Sección 4.3.1.2).

## 5.2. Detalles de la implementación

En este apartado se enumeran otros detalles de la implementación, que serán de interés principalmente para programadores que se propongan continuar con el desarrollo del prototipo.

### 5.2.1. Vista Conceptual

Este apartado aporta detalles de implementación sobre el modelo de clases de diseño mostrado en Sección 4.3.2, “Diseño de la solución”

Como ya se ha comentado, en este contexto un metamodelo está constituido por un `EPackage` que contiene un número de `EClassifiers`. La asociación *`modelPackage`* de `Metamodel` mantiene una referencia al `EPackage`.

La asociación *`axioms`* modela los axiomas específicos a un metamodelo.

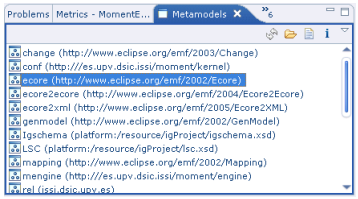



**Figura 5.7. Registro de metamodelos**

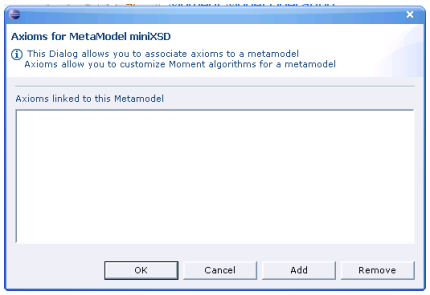



**Figura 5.8. Cuadro de diálogo para axiomas de usuario**

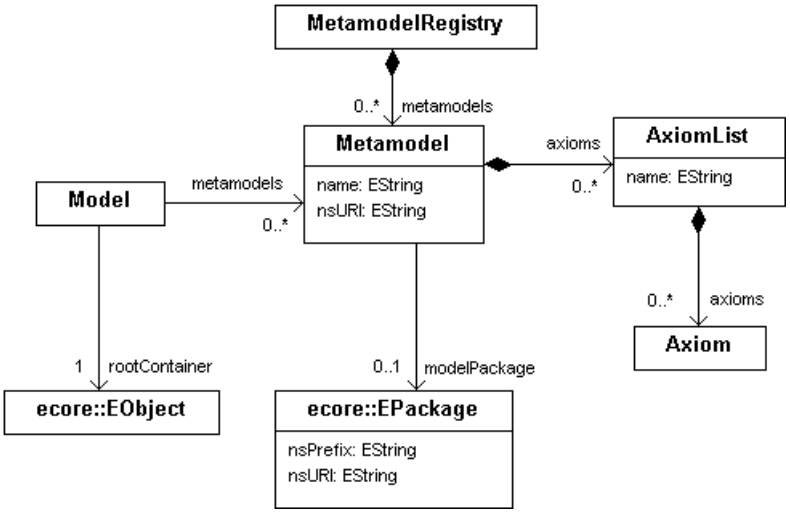



**Figura 5.9. Modelos y Metamodelos**

La figura muestra también la clase `Model`, que modela el concepto de modelo [4], cuya característica principal es la relación *`rootContainer`*, que mantiene una referencia a los contenidos del modelo

**¿Por qué rootContainer es un `EObject`?**

La clase Model en el esquema conceptual *modela*, valga la redundancia, el concepto de modelo. El atributo más importante es `rootContainer`, que es la referencia al modelo real, o más propiamente, al contenido del modelo real. EMF encapsula el contenido de un modelo como una instancia de la metaclase del modelo Ecore `EResource`, de ahí que surja la pregunta de por qué no utilizar esta clase para lo que buscamos, y utilizar en su lugar `EObject`.

La respuesta es que esa sería probablemente sea la manera óptima. Actualmente se está usando `EObject` en su lugar para aprovechar la facilidad de referencias externas provista por EMF que básicamente permite hacer referencias a elementos de modelos residiendo en otro fichero de forma transparente. Se puede hacer con una referencia a un `EObject`, pero no a un `EResource` porque EMF modela EResource como un `EDatatype` y no una clase [5].

Un problema de la solución actual se presenta al tratar con modelos en los que no hay un único contenedor raíz, es decir, una instancia que contenga todas las instancias del modelo. La alternativa de implementación más recomendable es almacenar en un atributo la URI de la `EResource` que contiene los objetos del modelo modelado.

El diagrama de clases de la Figura 5.10, “Implementación de la proyección de elementos a Maude” muestra la jerarquía completa de clases que tienen una proyección en forma de código Maude (recordar que en Figura 4.6, “Elementos que intervienen en la proyección a Maude” se mostraba este problema desde una perspectiva de análisis). La generación de estos artefactos de código se ha implementado mediante plantillas Velocity. Hay una plantilla para cada una de las clases que aparecen al final de la jerarquía de herencia. Algunas ya se han mencionado, como las que generan el código Maude de un modelo o de un metamodelo (ver Sección 4.1.2, “Diseño de la solución”). Las que faltan por mencionar son las siguientes:

[4] *Modelar* el concepto de modelo permite proporcionar métodos para integrar los proyectores Maude <->EMF, entre otras cosas.

[5] Para más detalles sobre esto es recomendable acudir a [1]

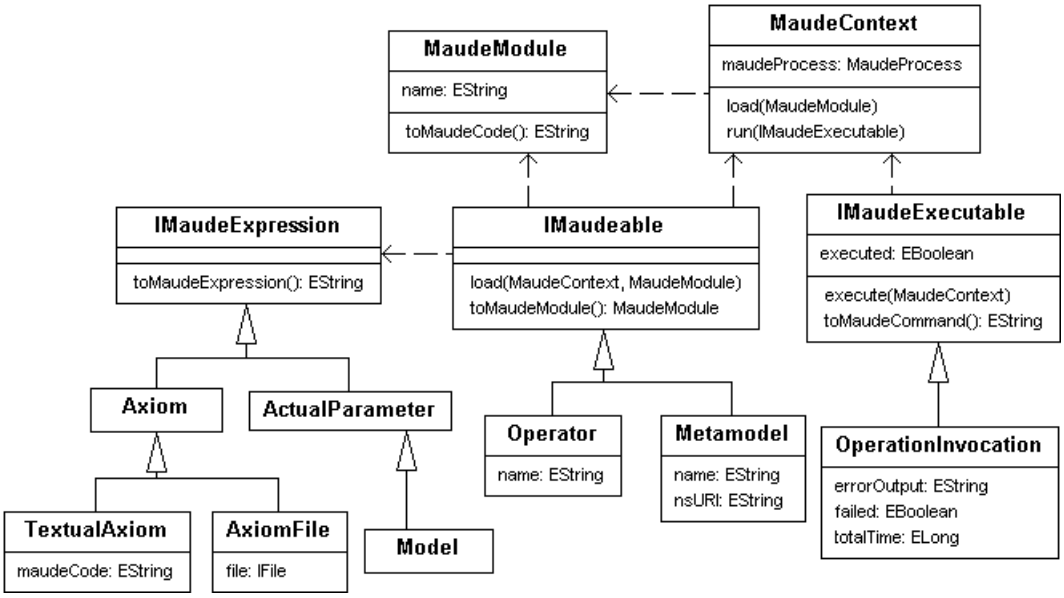



**Figura 5.10. Implementación de la proyección de elementos a Maude**

- `module.vm` - Genera la declaración de un modelo a partir de una instancia de `MaudeModule`.
- `operation.vm` - Genera la signatura y la definición axiomática de una operación a partir de una instancia de `Operator`.
- `invocation.vm` - Genera el comando que ejecuta una operación de MOMENT a partir de una instancia de `OperationInvocation`

Se puede ver que no hay ninguna plantilla para `Axiom`, y es que en esta versión del prototipo los axiomas de usuario no se generan, sino que deben ser introducidos directamente en código Maude. La especialización `TextualAxiom` permite introducir axiomas en forma de cadena, mientras que `AxiomFile` permite trabajar con axiomas almacenados en archivos.

Para evitar la duplicación de código entre las plantillas Velocity, hay dos plantillas de soporte:

- `macros.vm` - Contiene funciones utilizadas en varias plantillas.
- `symbols.vm` - Contiene constantes utilizadas en varias plantillas.

### 5.2.2. Composición por módulos

Se ha dado una vista conceptual del núcleo de la aplicación. A continuación se introduce una vista (Figura 5.11, "Vista de módulos por responsabilidades") de los módulos que componen la aplicación, utilizando para ello diagramas de clases UML donde los módulos están representados por paquetes.

Esta vista muestra de forma explícita los artefactos de implementación creados para los puentes Maude <-> EMF, así como las dependencias con los objetos del esquema conceptual que integran estos puentes en la aplicación

Adicionalmente a los puentes y al esquema conceptual expuesto anteriormente, en el gráfico aparecen un editor de operaciones y un editor de invocaciones, ambos integrados en la plataforma Eclipse.

La Figura 5.12, "Vista de módulos por capas" utiliza la metáfora de capas para ilustrar mejor las dependencias entre los módulos de la aplicación.

En esta figura aparecen varios librerías externas utilizadas en la implementación que no se habían mencionado antes:

| | |
|---|---|
| JUnit | Un *framework* o librería Java muy útil para escribir y mantener tests unitarios. En el proyecto se ha utilizado para crear y mantener una suite (parcial) de tests unitarios, tanto en la primera fase como en la segunda |
| Log4J | Un *framework* o librería Java que facilita capacidades de *logging* avanzadas. |
| Apache Jakarta Commons | Repositorio de librerías de uso general para Java. Facilitan el trabajo con Strings, con Colecciones, con E/S, etc. |

## 5.3. Escenarios de Variabilidad

Esta sección pretende ser una ayuda para futuros desarrolladores que trabajen en este proyecto. En ella se hace un estudio de los escenarios de cambios más

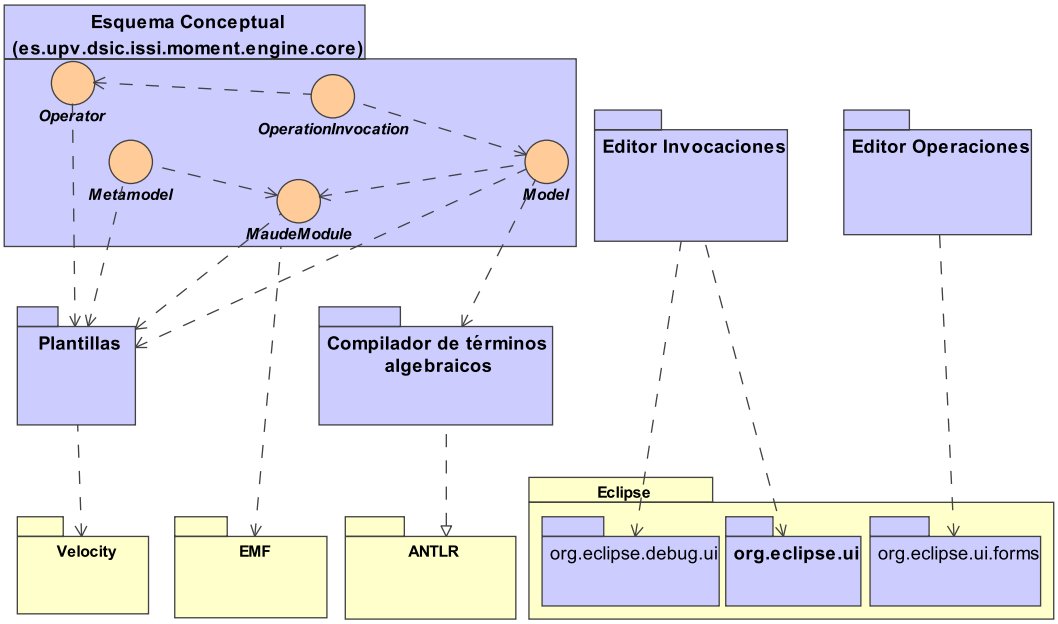



**Figura 5.11. Vista de módulos por responsabilidades**

probables identificados en el futuro próximo de la aplicación desarrollada, y se enumeran a grosso modo las modificaciones necesarias en los módulos que componen el prototipo. Quede claro que esta sección no pretende ser un manual infalible de como proceder en una situación así, lo que se persigue únicamente es facilitar la comprensión de la estructura del prototipo, y servir si acaso como chispa de arranque.

## 5.3.1. Cambios pequeños que afecten a la representación en Maude

Supongamos por ejemplo que se pasa a utilizar Full Maude[6] en lugar de Core Maude[7]. Sin entrar en más detalles, las diferencias más relevantes entre ambos son:

- la sintaxis utilizada en la declaración de módulos es ligeramente distinta.

[6]Full Maude es una extensión de Maude

[7]Core Maude es la denominación que recibe Maude con el propósito de distinguirlo de Full Maude. Cuando se nombraba Maude a lo largo de esta memoria, se hacía referencia a Core Maude en todo momento.

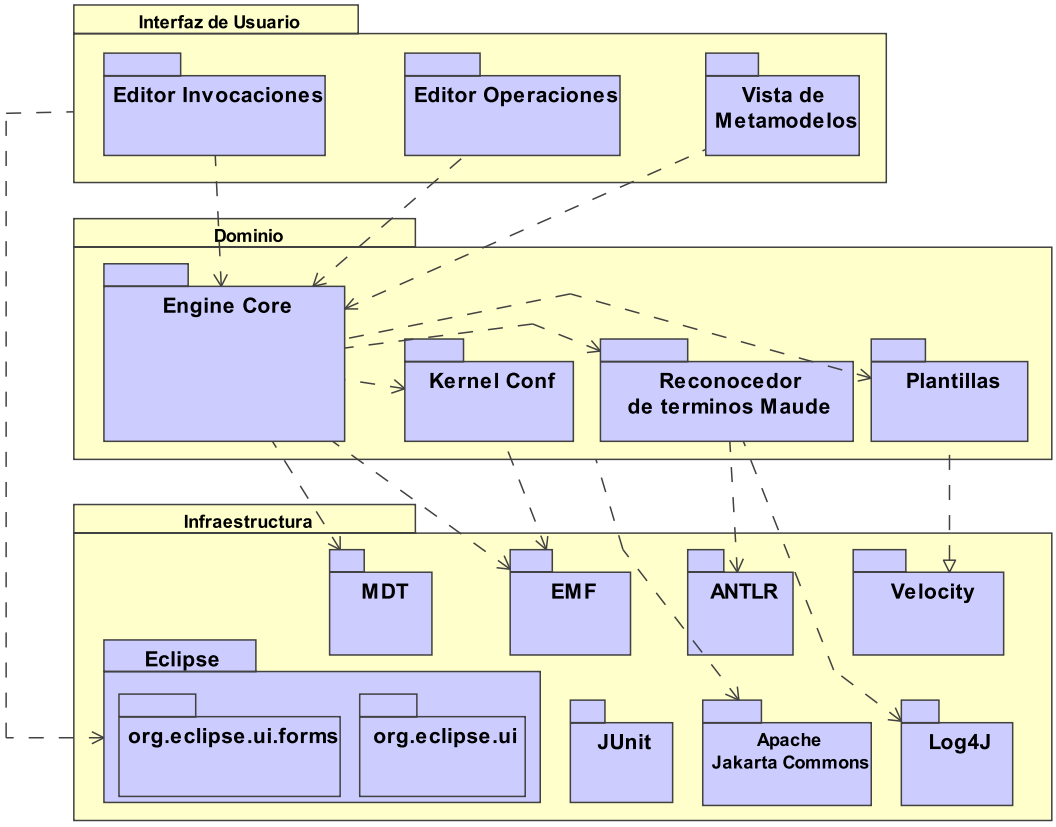



**Figura 5.12. Vista de módulos por capas**

- en Full Maude todos los comandos tienen que ir entre paréntesis. Esto afecta también a la declaración de los módulos (ya que tal declaración constituye un comando) y a la invocación de las operaciones de MOMENT.

Para introducir estos cambios en el prototipo, será necesario modificar las plantillas de Velocity que generan los artefactos mencionados:

- `invocation.vm` - la plantilla que genera el comando de invocación, para añadirle los paréntesis mencionados
- `module.vm` - la plantilla que produce la declaración de un módulo, para incorporar los paréntesis y las características sintácticas de Full Maude.

En general, cambios en la sintaxis de Maude se localizarían siempre en los puentes tecnológicos, principalmente en las plantillas de Velocity y posiblemente en el compilador de términos algebraicos que realiza el puente M1.

### 5.3.2. Cambios pequeños en la representación de los artefactos en MOMENT a nivel M2

Supongamos que se quiere añadir nuevas ecuaciones en el módulo de operaciones auxiliares. Los artefactos de implementación que habrá que modificar son:

- Las plantillas de Velocity relacionadas con este puente, en concreto la plantilla que genera el módulo de operaciones auxiliares (`m2sp.vm`)

En general, cualquier tipo de cambios en la representación de modelos a nivel M2 estará localizado en las plantillas correspondientes a ese nivel. Opcionalmente puede ser necesario dotar de nuevos métodos a la clase Java que colabora con este puente tecnológico cuando los cambios introduzcan complejidad, para evitar que las plantillas de Velocity se compliquen en exceso.

### 5.3.3. Cambios pequeños en la representación a nivel M1

Este tipo de situaciones se abordaría de manera similar a las del punto anterior, pero aquí además habrá que tener en cuenta el puente en dirección inversa, implementado por el compilador. Los cambios en el compilador se localizarían principalmente en el parser.

Los artefactos de implementación que habrá que modificar será:

- `parser.g` - la gramática de ANTLR que produce el componente del compilador que se encarga del análisis sintáctico y la construcción del AST.
- `m1.vm` - la plantilla de Velocity que implementa el puente M1.

## 5.3.4. Cambios en la estrategia de ejecución de operaciones

Se ha identificado que futuras versiones de MOMENT podrían introducir cambios en los procesos descritos en la Sección 4.3.1.3, “Proceso de ejecución de operaciones”. Por ejemplo,

- cambios en el orden de carga de los módulos implicados
- cambios en el formato de la instrucción que invoca la operación en Maude
- cambios en el formato del resultado o que requieran un post-procesado adicional

### 5.3.4.1. Cambios en la carga de los módulos

La carga de los módulos y otros aspectos del proceso de invocación se localizan en el método

```
OperationInvocationImpl.execute()
```

### 5.3.4.2. Cambios en el formato de la instrucción

La instrucción es generada a partir de una instancia de `OperationInvocation` mediante la plantilla `invocation.vm`. Por lo tanto este tipo de cambios se traducirán a modificaciones en dicha plantilla.

### 5.3.4.3. Postprocesado del resultado

En la versión actual ya se realizan tareas de postprocesado del resultado ofrecido por Maude tras una operación con MOMENT. Estas tareas están encapsuladas por medio del patrón de diseño *Composite*[8]. La configuración se realiza en el método:

`EngineConfiguration.getTextPostProcessor()`. Gracias al uso del patrón *Composite*, es muy fácil añadir nuevas etapas de postprocesado, constituidas por una clase Java que implementa la interfaz:

```
public interface TextProcessStrategy {
```

```
public String process(String s);
}
```

## 5.3.5. Sustitución de EMF por otra tecnología de modelos

Es conveniente dividir el problema en dos partes a considerar:

- La construcción de los puentes tecnológicos entre MOMENT y la nueva tecnología
- La integración de los puentes en el prototipo en el prototipo

### 5.3.5.1. Construcción de los puentes

La construcción de los puentes necesarios puede hacerse desde cero o reutilizar parte del trabajo realizado para EMF. En este caso, depende de la diferencia de conceptos que haya entre EMF/MOF y la tecnología de modelos adoptada.

Suponiendo que los cambios sean asequibles, y suponiendo también que exista una API para manipular los modelos de la tecnología adoptada de forma programática, debería ser posible construir un adaptador (ver patrón de diseño *Adapter*[8]) entre la API de EMF y la API destino. Mediante este adaptador, las modificaciones a los puentes existentes serían mínimas (o podrían llegar a ser inexistentes). Un requisito adicional es que exista interoperabilidad entre los lenguajes de programación de cada API.

En cuanto al compilador de términos algebraicos, si se mantiene la representación algebraica se podrán aprovechar todas las etapas del compilador menos la última. En cuanto a la última etapa, la que realiza la traducción a partir del AST obtenido en las anteriores, podría ser reutilizada si se construye el adaptador mencionado en el párrafo anterior.

### 5.3.5.2. Integración de los nuevos puentes

Debería bastar con sustituir los puentes antiguos por los nuevos, en los puntos de integración en el prototipo. Estos puntos, descritos en Sección 4.3.2.2, “Integración de los puentes”, no son configurables y se definen de forma estática.

Para permitir la creación de puentes por terceros y la coexistencia de varios puentes, sería interesante permitir su configuración de forma externa y dinámica, por ejemplo aprovechando la facilidad de *plug-ins* de Eclipse.

### 5.3.6. Sustitución de Eclipse por otra plataforma de integración

Este es otro escenario que implica un gran número de cambios. En este caso, los cambios no afectarán al esquema conceptual del prototipo, en efecto al núcleo del prototipo, sino que se concentrarán en los aspectos de interfaz de usuario.

La ventaja es que se podrá reutilizar la mayor parte del plug-in `es.upv.dsic.issi.moment.engine.core`, que contiene el núcleo del prototipo. Las dependencias de este código con Eclipse son mínimas, reduciéndose al manejo de errores y a la clase `Plugin`.

El resto de los plug-ins, que realizan la integración en Eclipse, deberán ser reconstruidos para la nueva plataforma de integración.

# Capítulo 6. Conclusiones

## 6.1. Resumen

En esta memoria se ha presentado un prototipo denominado Moment Engine que realiza la integración entre MOMENT, un sistema de gestión de modelos, y EMF, un entorno de trabajo con modelos.

Se han estudiado los patrones de conversión que permiten la interoperabilidad a nivel de modelos y metamodelos entre Moment y EMF, así como la gramática abstracta para el subconjunto relevante de la sintaxis de Maude.

En una segunda fase, se ha automatizado el funcionamiento de MOMENT simplificándolo al máximo, y se ha integrado la funcionalidad en el entorno de desarrollo Eclipse.

Los detalles de análisis, diseño e implementación de todo el prototipo han sido descritos. La relativa facilidad con que se ha llevado a cabo este trabajo ponen de manifiesto la calidad de EMF y la extensibilidad de la plataforma Eclipse.

## 6.2. Trabajos futuros

La plataforma de gestión de modelos MOMENT aún se encuentra en fase de desarrollo, y conforme vaya evolucionando y cambiando hasta llegar a un punto estable, este prototipo debe hacerlo con ella. Además, todavía quedan muchas tareas pendientes por realizar. A continuación se citan las más relevantes.

Una de las tareas pendientes más obvias es dar soporte a las operaciones extramodelo, así como a las operaciones simples. Dicho soporte debería permitir:

- La ejecución directa de operaciones simples de forma sencilla.
- La ejecución de operaciones entre modelos de distintos metamodelos.

A más largo plazo se quiere conseguir compatibilidad con el futuro estándar QVT para definir las transformaciones, esto es, la aplicación del operador ModelGen. Conseguir que MOMENT sea capaz de ejecutar especificaciones QVT posibilitará que los usuarios de QVT no tengan que aprender un lenguaje nuevo para definir las transformaciones en MOMENT, y además la adherencia a estándares es siempre un factor positivo.

Otro objetivo a medio o largo plazo es conseguir un editor con metáfora gráfica para introducir los axiomas de usuario. Recordemos que los axiomas de usuario se utilizan para definir las correspondencias o *mappings* de forma manual (MOMENT es capaz de inferir estos *mappings* automáticamente, ver [12]). Estas correspondencias serán luego tenidas en cuenta en la aplicación de los operadores de conjuntos, como Merge. Un editor gráfico permitirá, al igual que en el caso de QVT, que el usuario no tenga que aprender un lenguaje nuevo para definir estas correspondencias.

En el Apéndice B, *Trabajos futuros* se detallan otras ampliaciones o tareas pendientes que no se han citado aquí por ser más específicas y puntuales.

# Apéndice A. Manual de usuario de Moment Engine

## A.1. Instrucciones de Instalación

Para instalar Moment Engine siga los pasos siguientes:

1. Inicie Eclipse. Abra el menú “Help” y seleccione la opción “Software Update” > “Find and Install”.

2. Elija la opción “Search for new features to install” y presione el botón “Next”.

3. En el cuadro de diálogo “Update sites to visit” debe añadir la Update Site de MOMENT si no lo ha hecho ya. Para ello, haga clic en el botón "Add Update Site" o "New Remote Site".

4. Introduzca “MOMENT Update Site” en el campo “Name” y “ftp://moment.dsic.upv.es/moment/eclipse” en el campo “URL”. Presione “OK”.

5. De vuelta al cuadro de diálogo, marque la opción “MOMENT Update Site” y presione en el botón “Next” o “Finish”.

6. En el siguiente cuadro de diálogo, despliegue la rama “Moment” y seleccione la opción “Moment Engine”. Si no tiene instaladas las *Maude Development Tools*, selecciónelas también mediante la opción correspondiente[1]. Seguidamente presione “Next”, acepte la licencia y finalice la instalación siguiendo

[1] En cualquier caso será necesario tener instalado el sistema Maude, ya que MDT no lo incluye.

las instrucciones que le presente Eclipse. Deberá aceptar también el reinicio de Eclipse tras finalizar la instalación.

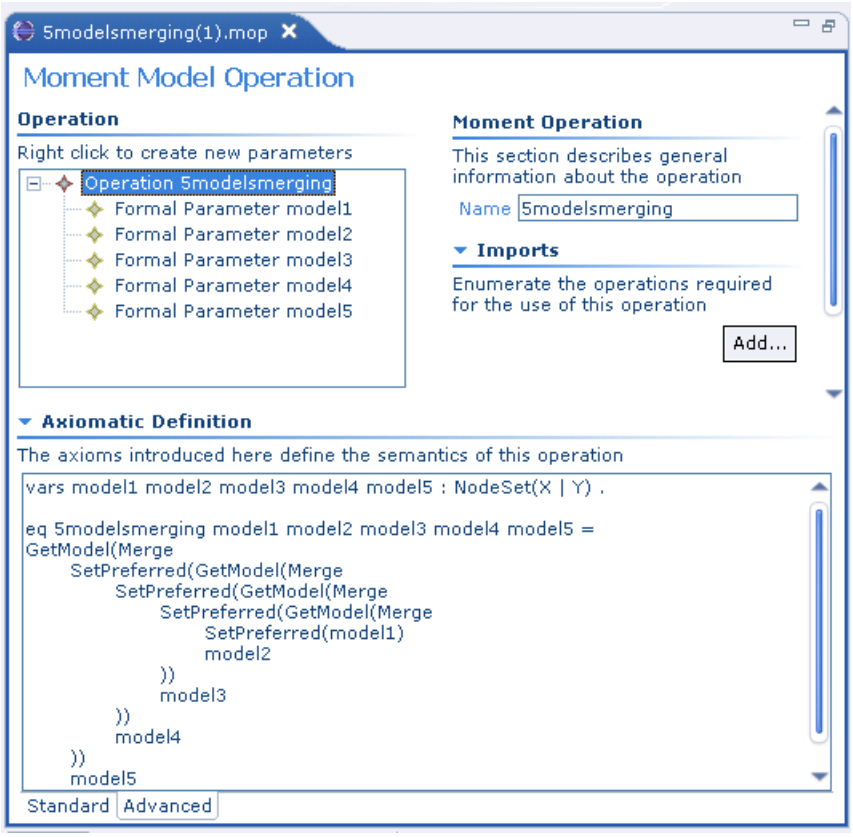
# A.2. Tareas comunes

## A.2.1. Creación de un operador complejo

El asistente “New” > “Moment” > “Model Management Operation” permite crear nuevos operadores complejos.

## A.2.2. Edición de operadores complejos

El editor de operadores complejos de Moment Engine permite definir el nombre, parámetros y dependencias de un operador complejo.

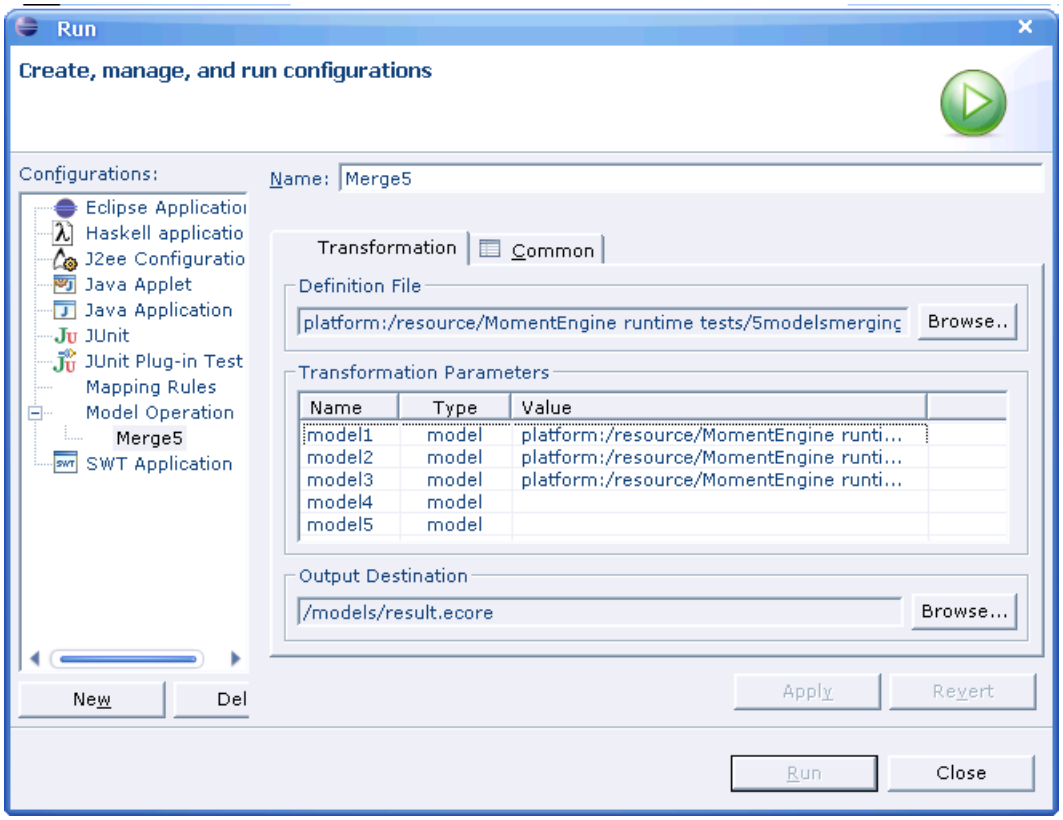


En el campo “Axiomatic Definition”, se deben introducir únicamente las ecuaciones que describen el comportamiento del operador, así como la declaración de variables y operaciones auxiliares. No es necesario introducir ni la signatura de la operación ni la declaración del módulo que la contiene.

En la ecuación axiomática (o ecuaciones) para el operador, el nombre de la operación debe ser el mismo que el introducido en el campo nombre.

## A.2.3. Ejecución de una operación compleja

Antes de ejecutar un operador complejo hay que definir una configuración de ejecución. Para ello se debe seleccionar el fichero que contiene el operador y en el menú “Run” de Eclipse, seleccionar “Run” > “Model Operation”. Eclipse mostrará el cuadro de diálogo que permite introducir los operandos (modelos, cadenas o números) que intervienen en la operación, así como la ruta del fichero para almacenar el resultado. Una vez todos los campos estén rellenados, para ejecutar la operación hay que presionar **Run**



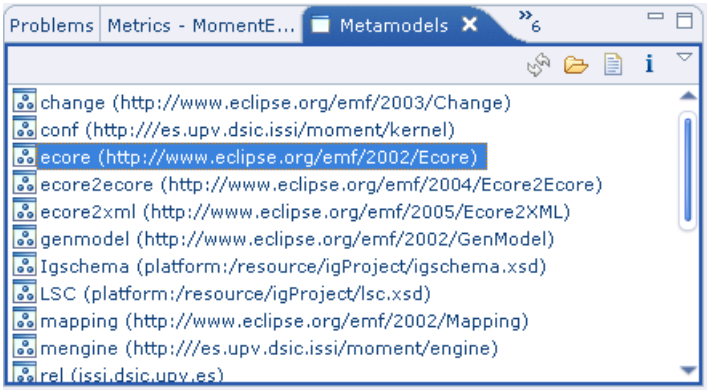
Tras la ejecución de una operación, si no han habido errores, el resultado se encontrará en la ruta especificada.

## A.2.4. Asignación de axiomas de usuario a un metamodelo

En algunos casos será necesario ajustar las reglas de equivalencia que se aplican en los operadores de conjuntos ofrecidos por MOMENT. Para hacerlo MOMENT permite al usuario declarar nuevas reglas de equivalencia en forma de axiomas. Para asignar un conjunto de axiomas a un metamodelo en Moment Engine se han de seguir los siguientes pasos:

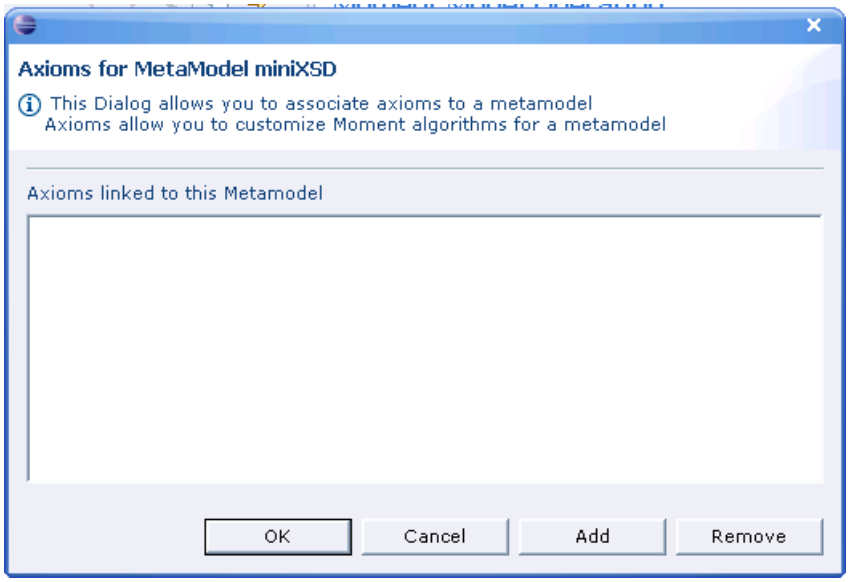
1. Introducir los axiomas en un fichero de texto, en código Maude. El fichero debe contener las ecuaciones axiomáticas y cualquier operación auxiliar. El fichero no debe contener ninguna declaración de módulos Maude.

2. Abrir la vista del registro de metamodelos mediante la siguiente secuencia de opciones de menú: "Window" > "Show View" > "Moment" > "Metamodels".




3. Hacer clic con el botón derecho sobre el metamodelo deseado y seleccionar la opción "Axioms".

4. Mediante la opción "Add", añadir los ficheros que contengan los axiomas.


Axioms for MetaModel miniXSD
This Dialog allows you to associate axioms to a metamodel
Axioms allow you to customize Moment algorithms for a metamodel
Axioms linked to this Metamodel
OK
Cancel
Add
Remove

# Apéndice B. Trabajos futuros

Este prototipo es sólo el primer paso hacia un resultado final satisfactorio. Muchas cosas se han quedado en el tintero, sin mencionar la maleta de fallos, errores y *bugs* que toda primera versión lleva implícita. En este apéndice se recogen ampliaciones de cara a la robustez de la aplicación.

## B.1. Mejoras en el puente M1

Sería conveniente mejorar los analizadores léxico y sintáctico del compilador principalmente para hacerlos más robustos.

- En el Lexer no se ha seguido ningún criterio de compatibilidad para las definiciones de los lexemas. Sería recomendable redefinirlos teniendo en cuenta las reglas de caracteres válidos que rigen tanto EMF como Maude. Por ejemplo, ¿qué símbolos pueden formar parte de una cadena en Maude? ¿Y en EMF? ¿Son los mismos? En este sentido se han tomado precauciones en la conversión de valores, pero esas precauciones no están plasmadas en el lexer.

- El no-terminal `moment_op` identifica el constructor del término, que como se vio en la Sección 3.3, "Puente EMF -> Maude a nivel M2" tiene la forma "pkg-clase". La regla que define este no-terminal es la siguiente:

  ```
  moment_op : pkg:CADENA separadorOp clase:CADENA
  protected separadorOp : GUION;
  ```

  Esta regla es problemática y fallaría en el caso de que o bien el nombre del paquete, o el de la clase, incluyesen un guión. Además, para que esta regla pueda funcionar, se ha declarado el guión como no válido en la regla del Lexer que define el lexema `CADENA`, lo que es sinónimo de problemas en

todos los contextos en los que se emplee este lexema, ya que tanto en EMF como en Maude el guión sí es válido en este contexto. Las soluciones posibles son:

- Dejar el lexema CADENA como está y reemplazar las apariciones de guiones en el nombre de una clase por otro símbolo para evitar la colisión.
- Habilitar el guión en el lexema CADENA y sustituir su uso aquí como separador por otro símbolo que no pueda provocar este problema, es decir, que sea ilegal en EMF pero que, lógicamente, no lo sea en Maude.
- Habilitar el guión en el lexema CADENA cambiando de estrategia sintáctica. En lugar de intentar separar entre prefijo y clase en el análisis sintáctico, relegar este paso a un refinamiento en una fase posterior de análisis semántico. El algoritmo en caso de que aparezca más de un guión es, obviamente, tomar el primero como separador.

- El no terminal `empty_set` que aparece en el parser, utilizado para identificar las palabras clave que definen conjuntos vacíos, está definida como:

```
protected empty_set : "empty" GUION ("set" | "bag" | "sequence" | "orderedset"
 );
```

Desafortunadamente no se ha tenido en cuenta que, a partir de esta definición, ANTLR identificado las siguientes palabras clave: empty, set, bag, orderedset, y sequence. Todas estas palabras son de uso común y pueden aparecer en cualquier modelo, con la consiguiente interpretación errónea del parser. Además, se podría haber evitado fácilmente definiendo las palabras clave como una sola cadena, por ejemplo empty-set, en lugar de separar por el guión, aunque para esto habría que solucionar previamente el problema comentado en el punto anterior (ya que de lo contrario no son cadenas válidas).

# Apéndice C. Correspondencia de tipos EMF <-> Ecore

| EMF | Maude |
|---|---|
| EString | String |
| EFloat | Float |
| EChar | Char |
| ECharacterObject | Char |
| EByte | Int |
| EByteObject | Int |
| EBigDecimal | Float |
| EBigInteger | Int |
| EFloatObject | Float |
| EShort | Int |
| EInt | Int |
| EInteger | Int |
| EBoolean | Bool |
| EBooleanObject | Bool |
| ELong | Int |
| ELongObject | Int |
| EJavaClass | String |
| EJavaObject | String |
| EEnumerator | String |

| **EMF** | **Maude** |
|---|---|
| EByteArray | String |
| EMap | String |
| EEList | String |
| EDate | String |

# Apéndice D. Proyecciones M2 y M1 completas del modelo miniXSD

En este apéndice se incluyen, a modo de referencia, las proyecciones completas del metamodelo miniXSD (ver Figura 3.3, "Metamodelo miniXSD") utilizado como ejemplo a lo largo de la memoria.

En primer lugar se facilita la codificación XMI del metamodelo, para poder recrearlo de forma exacta. En segundo lugar se muestra la especificación algebraica (nivel M2) que representa en MOMENT a miniXSD como un metamodelo. En tercer lugar se muestra el conjunto de términos (nivel M1) que representa a miniXSD en MOMENT como un modelo del metamodelo Ecore.

**Nota**

En el soporte de datos físico que acompaña a esta memoria se pueden encontrar estos ejemplos, en la ruta `apendice/miniXSD`.

Adicionalmente se ha incluido como referencia la proyección del modelo Ecore en los dos niveles, en la ruta `apendice/ecore`.

## D.1. Codificación en XMI de miniXSD producida por EMF

```
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="miniXSD"
    nsURI="http://es.upv.dsic/issi/moment/miniXSD" nsPrefix="mXSD">
  <eClassifiers xsi:type="ecore:EClass" name="Schema">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name"
```

```
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="elements" upperBound="-1"

        eType="#//Element" containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Element" abstract="true">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="Name"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Simple" eSuperTypes="#//Element">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="Type"
eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Complex" eSuperTypes="#//Element">
    <eStructuralFeatures xsi:type="ecore:EReference" name="sequence" lowerBound="1"

        upperBound="-1" eType="#//Element" containment="true"/>
  </eClassifiers>
</ecore:EPackage>
```

# D.2. miniXSD visto como metamodelo

```
fmod sigminiXSD is
  pr DATATYPE .

  sorts mXSD-Schema mXSD-Element mXSD-Simple mXSD-Complex miniXSDNode .
  subsorts mXSD-Schema mXSD-Element < miniXSDNode .
  subsorts mXSD-Simple mXSD-Complex < mXSD-Element .

*** op mXSD-Schema: Qid, Name, elements,
  op `(mXSD-Schema___`) : Qid String OrderedSet {QID}  -> mXSD-Schema [ctor] .

*** op mXSD-Simple: Qid, Name, Type,
  op `(mXSD-Simple___`) : Qid String String  -> mXSD-Simple [ctor] .

*** op mXSD-Complex: Qid, Name, sequence,
  op `(mXSD-Complex___`) : Qid String OrderedSet {QID}  -> mXSD-Complex [ctor] .

endfm


view vminiXSD from TRIV to sigminiXSD is
  sort Elt to miniXSDNode .
endv


fmod spminiXSD is
  pr MOMENT-OP{vminiXSD} .
  pr specore .

  vars Name1 Type1 : String .
  vars elements1 sequence1 : OrderedSet{QID} .
  var OID1 : Qid .
  vars Model Set : Set{ vminiXSD } .
  var OSet : OrderedSet{QID} .
```

```
  var StringValue : String .
  var BoolValue : Bool .
  var IntValue : Int .
  var IdValue : Qid .
  var DT : ecoreNode .

  eq (mXSD-Schema OID1  Name1  elements1) :: OID = OID1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: OID <-- IdValue = (mXSD-Schema IdValue
  Name1  elements1) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Schema OID1  Name1  elements1) :: Name = Name1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: Name <-- StringValue = (mXSD-Schema
OID1 StringValue elements1) .
  op elements : -> Fun{vminiXSD} [ctor] .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements = elements1 .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements ( Model ) = (Model -> select
 ( hasId ; ? elements1 ; empty-set )) -> sortedBy (firstThan ; ? elements1 ;
empty-set) .
  eq (mXSD-Schema OID1  Name1  elements1) :: elements <-- OSet = (mXSD-Schema OID1
 Name1 OSet) .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: OID = OID1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: OID <-- IdValue = (mXSD-Simple IdValue
Name1 Type1 ) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Name = Name1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Name <-- StringValue = (mXSD-Simple OID1
 StringValue Type1) .
  op Type : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Type = Type1 .
  eq (mXSD-Simple OID1  Name1 Type1 ) :: Type <-- StringValue = (mXSD-Simple OID1
 Name1 StringValue) .
  eq (mXSD-Complex OID1  Name1  sequence1) :: OID = OID1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: OID <-- IdValue = (mXSD-Complex
IdValue  Name1  sequence1) .
  op Name : -> StringFun{vminiXSD} [ctor] .
  eq (mXSD-Complex OID1  Name1  sequence1) :: Name = Name1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: Name <-- StringValue = (mXSD-Complex
 OID1 StringValue sequence1) .
  op sequence : -> Fun{vminiXSD} [ctor] .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence = sequence1 .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence ( Model ) = (Model -> select
 ( hasId ; ? sequence1 ; empty-set )) -> sortedBy (firstThan ; ? sequence1 ;
empty-set) .
  eq (mXSD-Complex OID1  Name1  sequence1) :: sequence <-- OSet = (mXSD-Complex
OID1 Name1 OSet) .

endfm
```

# D.3. miniXSD visto como modelo

```
*** Generated by Moment
***$ ecore - "http://www.eclipse.org/emf/2002/Ecore"

Set {
  (ecore-EPackage 'platform:/miniXSD.ecore#/ "miniXSD"
```

```
"http://es.upv.dsic/issi/moment/miniXSD" "mXSD" empty-orderedset OrderedSet {'#//
 } OrderedSet { 'platform:/miniXSD.ecore#//Schema  ::
'platform:/miniXSD.ecore#//Element  :: 'platform:/miniXSD.ecore#//Simple  ::
'platform:/miniXSD.ecore#//Complex } empty-orderedset empty-orderedset ),
  (ecore-EClass 'platform:/miniXSD.ecore#//Schema "Schema" "" "0" "0" false false
 empty-orderedset OrderedSet {'platform:/miniXSD.ecore#/ } empty-orderedset
empty-orderedset OrderedSet { 'platform:/miniXSD.ecore#//Schema/Name } OrderedSet
 { 'platform:/miniXSD.ecore#//Schema/elements } OrderedSet {
'platform:/miniXSD.ecore#//Schema/elements } OrderedSet {
'platform:/miniXSD.ecore#//Schema/Name } OrderedSet {
'platform:/miniXSD.ecore#//Schema/elements } empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Schema/Name  :: 'platform:/miniXSD.ecore#//Schema/elements
 } empty-orderedset empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Schema/Name  :: 'platform:/miniXSD.ecore#//Schema/elements
 } ),
  (ecore-EAttribute 'platform:/miniXSD.ecore#//Schema/Name "Name" true true 0 1
false false true false false "" "0" false false false empty-orderedset OrderedSet
 {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/miniXSD.ecore#//Schema } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } ),
  (ecore-EReference 'platform:/miniXSD.ecore#//Schema/elements "elements" true true
 0 -1 true false true false false "" "0" false false true false true empty-orderedset
 OrderedSet {'platform:/miniXSD.ecore#//Element } OrderedSet
{'platform:/miniXSD.ecore#//Schema } empty-orderedset OrderedSet
{'platform:/miniXSD.ecore#//Element } ),
  (ecore-EClass 'platform:/miniXSD.ecore#//Element "Element" "" "0" "0" true false
 empty-orderedset OrderedSet {'platform:/miniXSD.ecore#/ } empty-orderedset
empty-orderedset OrderedSet { 'platform:/miniXSD.ecore#//Element/Name }
empty-orderedset empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Element/Name } empty-orderedset empty-orderedset
OrderedSet { 'platform:/miniXSD.ecore#//Element/Name } empty-orderedset
empty-orderedset OrderedSet { 'platform:/miniXSD.ecore#//Element/Name } ),
  (ecore-EAttribute 'platform:/miniXSD.ecore#//Element/Name "Name" true true 0 1
false false true false false "" "0" false false false empty-orderedset OrderedSet
 {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/miniXSD.ecore#//Element } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } ),
  (ecore-EClass 'platform:/miniXSD.ecore#//Simple "Simple" "" "0" "0" false false
 empty-orderedset OrderedSet {'platform:/miniXSD.ecore#/ } OrderedSet {
'platform:/miniXSD.ecore#//Element } empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Element/Name  :: 'platform:/miniXSD.ecore#//Simple/Type
 } empty-orderedset empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Simple/Type } empty-orderedset empty-orderedset OrderedSet
 { 'platform:/miniXSD.ecore#//Element/Name  :: 'platform:/miniXSD.ecore#//Simple/Type
 } OrderedSet { 'platform:/miniXSD.ecore#//Element } empty-orderedset OrderedSet
{ 'platform:/miniXSD.ecore#//Simple/Type } ),
  (ecore-EAttribute 'platform:/miniXSD.ecore#//Simple/Type "Type" true true 0 1
false false true false false "" "0" false false false empty-orderedset OrderedSet
 {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/miniXSD.ecore#//Simple } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } ),
  (ecore-EClass 'platform:/miniXSD.ecore#//Complex "Complex" "" "0" "0" false false
 empty-orderedset OrderedSet {'platform:/miniXSD.ecore#/ } OrderedSet {
'platform:/miniXSD.ecore#//Element } empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Element/Name } OrderedSet {
'platform:/miniXSD.ecore#//Complex/sequence } OrderedSet {
'platform:/miniXSD.ecore#//Complex/sequence } empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Complex/sequence } empty-orderedset OrderedSet {
```

```
'platform:/miniXSD.ecore#//Element/Name  ::
'platform:/miniXSD.ecore#//Complex/sequence } OrderedSet {
'platform:/miniXSD.ecore#//Element } empty-orderedset OrderedSet {
'platform:/miniXSD.ecore#//Complex/sequence } ),
  (ecore-EReference 'platform:/miniXSD.ecore#//Complex/sequence "sequence" true
true 1 -1 true true true false false "" "0" false false true false true
empty-orderedset OrderedSet {'platform:/miniXSD.ecore#//Element } OrderedSet
{'platform:/miniXSD.ecore#//Complex } empty-orderedset OrderedSet
{'platform:/miniXSD.ecore#//Element } )
}
```

# Bibliografía

## Libros


[1] *Eclipse Modeling Framework: A Developer's Guide*. 0-13-142542-0. Addison Wesley. Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, y Timothy J. Grose.

[2] *MDA Distilled: Principles of Model-Driven Architecture*. 0-201-78891-8. Addison Wesley. Stephen D. Mellor, Kendall Scott, Axel Uhl, y Dirk Weise.

[3] *MDA Explained: The Model Driven Architecture*. Anneke Kleppe, Jos Warmer, y Wim Bast. Addison Wesley. 0-321-19442-X.

[4] *The Maude Manual*. José Meseguer, Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, y Carolyn Talcott.

[5] *The Object Constraint Language, 2nd edition*. Jos Warmer y Anneke Kleppe. 0321179366.

[6] *Beginning XSLT*. 1590592603. Apress. Jeni Tennison.

[7] *Building Parsers With Java*. Steve John Metsker. Addison Wesley. 0201719622.

[8] *Design Patterns: elements of reusable OO Software*. 0201633612. Addison Wesley. Erich Gamma, Richard Helm, John Vlissides, y Ralph Johnson.

[9] *Contributing to Eclipse*. 0-321-20575-8. Addison Wesley. Kent Beck. Erich Gamma.

[10] *Fundamentals of Algebraic Specification, vol.1*. H. Ehrig. B. Mahr. Springer-Verlag.

# Publicaciones

[11] *Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework*. JISBD. 2005. Artur Boronat Moll. José Iborra. José A. Carsí Cubel. Isidro Ramos. Abel Gómez.

[12] *Generic Model Merging applied to Class Diagram Integration*. Artur Boronat Moll, José A. Carsí Cubel, Isidro Ramos, y Patricio Letelier.

[13] *Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine*. Artur Boronat Moll, José A. Carsí Cubel, y Isidro Ramos. 2005/03. IEEE Computer Society. 9th European Conference on Software Maintenance and Reengineering.

[14] *Technological Spaces: an Initial Appraisal*. Ivan Kurtev, Jean Beziven, y Mehmet Aksit.

[15] *Soporte gráfico para trazabilidad en una herramienta de gestión de modelos*. Abel Gómez. Memoria PFC - UPV (Valencia).

[16] *Applying Model Management to Classical Meta Data Problems*. Philip A. Bernstein. Proceedings of the 2003 CIDR Conference.

[17] *Guía práctica de ANTLR*. Enrique J. García Cota. José A. Troyano Jiménez. ETS de Ingeniería Informática de la Universidad de Sevilla.

# Sitios Web

[18] *The Maude Programming Language*. http://maude.cs.uiuc.edu.

[19] *The Eclipse Platform*. http://www.eclipse.org.

[20] *The Eclipse Modeling Framework*. http://www.eclipse.org/emf.

[21] *The Velocity Project*. http://jakarta.apache.org/velocity.

[22] *The Atlas Transformation Language Project*. http://www.sciences.univ-nantes.fr/lina/atl/.

[23] *The IBM Model Transformation Framework*. http://www.alphaworks.ibm.com/tech/mtf.

[24] *The ANTLR Parser Generator*. http://www.antlr.org.

[25] *The OMELET Project*. http://www.eclipse.org/omelet.

[26] *The Generative Model Transformer Project*. http://www.eclipse.org/gmt.

[27] *The XML Schema Working Group*. http://www.w3.org/XML/Schema.

[28] *The UML2 Eclipse Project*. http://www.eclipse.org/uml2.

[29] *The XML Schema Infoset Model Eclipse Project*. http://www.eclipse.org/xsd.

[30] *IBM Integrated Ontology Development Toolkit*. http://www.alphaworks.ibm.com/tech/semanticstk.