DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Informe Técnico / Technical Report

Ref. No: DSIC-II/02/06 **Pages:** 214

Title: Soporte gráfico para trazabilidad en una herramienta de Gestión de modelos.

Author (s): Abel Gómez Llana, Artur Boronat Moll, José Á. Carsí Cubel.

Date: 19/01/2005

Key Words: Gestión de Modelos, Model Driven Engineering, Maude, Trazabilidad.

VºBº **Author (s):**

Leader of Reasearch Group

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Parte Primera:

Presentación del proyecto y Fundamentos.

1. PRESENTACIÓN.

La trazabilidad [BoC05] es una cuestión clave en entornos donde hay una cadena de procesos. Es estos casos, la información sobre cada paso de la cadena debe ser almacenada para poder ser consultada posteriormente. Por ejemplo, en el caso de la industria de la automoción, la trazabilidad hace posible la retirada de los vehículos del mercado en caso de fallos; en la industria alimentaria, contribuye a la seguridad de los alimentos; o, en el campo de la ingeniería del software, proporciona soporte para la validación de requisitos e incrementa la calidad del proceso de desarrollo de software.

En cualquier escenario en el campo de la Ingeniería del Software existe una manipulación de un artefacto software. La capacidad de describir y consultar las operaciones de manipulación que se han realizado sobre un determinado artefacto pueden ser relevantes para otras tareas relacionadas. Sin embargo, la gestión de la trazabilidad aún hoy, a menudo permanece en un segundo plano cuando se resuelven problemas en la Ingeniería del Software, y pocas son las herramientas que proporcionan un soporte completo para ésta.

En la iniciativa MDA (Model-Driven Architecture), todo artefacto software puede ser tratado como un modelo. Tareas típicas, como producción de código, integración de aplicaciones o interoperabilidad, son realizadas directamente a partir de modelos. Esto permite al usuario trabajar a nivel conceptual, haciendo más fácil la identificación de los elementos necesarios para automatizar estas tareas. Estas labores son usuales en diversos escenarios y son generalmente resueltas de manera *ad-hoc*.

Siguiendo esta aproximación dirigida por modelos, una nueva disciplina, llamada Gestión de Modelos, fue propuesta por Philip A. Bernstein (Microsoft Research) en [Ber00]. Esta disciplina considera los modelos como ciudadanos de primer orden, y proporciona una serie de operadores genéricos para aplicarlos a estos modelos: *Merge*, *Diff*, *ModelGen*, etc. Estos operadores, proporcionan una solución reusable a las tareas descritas anteriormente, de forma que el usuario manipula directamente los modelos, en lugar de trabajar con la representación interna de éstos a nivel de programación. Diversas aproximaciones a esta disciplina especifican operadores que están basados en correspondencias (*mappings*) para manipular modelos. Una correspondencia es una relación entre un elemento de un modelo dominio y un elemento de un modelo rango. Esto significa, que las correspondencias entre dos modelos deben ser explícitamente definidos para poder aplicarles un operador.

Usando la experiencia obtenida en la aplicación del formalismo proporcionado por las especificaciones algebraicas para resolver problemas en la ingeniería del software, se está trabajando en un framework llamado MOMENT [MOMENT]. En esta herramienta, los operadores de Gestión de Modelos han sido definidos algebraicamente, y 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.

Igualmente, en esta aproximación, las relaciones entre dos modelos se representan de forma implícita por medio de un morfismo de equivalencia que se define entre dos metamodelos desde un punto de vista más abstracto y reusable. Sin embargo, los *mappings* explícitos entre dos modelos son también útiles cuando no existe ninguna definición de este morfismo entre dos metamodelos.

Para dar el soporte formal a esta herramienta de gestión de modelos, se ha elegido un eficiente sistema de reescritura de términos, Maude, ya utilizado en diversos ámbitos formales.

En la Ingeniería del Software, y desde el punto de vista de la industria, la aplicación de formalismos para dar soporte a herramientas industriales no ha sido siempre bien vista. Es por esto que desde MOMENT se apuesta por la integración del entorno Maude, el motor de reescritura de términos, en un entorno de modelado industrial, como es el Eclipse Modeling Framework (EMF).

1.1. Objetivos.

El objetivo de este proyecto es la aportación de las herramientas necesarias para soportar sobre EMF, —el entorno de modelado visual empleado—, el soporte para trazabilidad implementado en el álgebra de MOMENT.

Este trabajo, no obstante, se ha desarrollado en el seno del proyecto MOMENT desde su fase inicial de desarrollo. Por ello en primer lugar, antes de abordar las cuestiones únicamente concernientes a la gestión de trazabilidad, se debe proporcionar la correspondiente integración del sistema de reescritura de términos en el entorno de trabajo, Eclipse.

Los principales objetivos de este proyecto son en primera instancia, por tanto, los siguientes:

- El primero y fundamental, es proporcionar una interfaz de programación Java para poder ejecutar el álgebra de MOMENT desde el entorno visual, pudiendo enviar comandos y consultar sus resultados.
- En segundo lugar, dado el aislamiento y peculiaridades de Maude (está diseñado para ejecutarse fundamentalmente en sistemas UNIX), se debe conseguir ejecutar el sistema en la mayor parte de sistemas operativos, especialmente dando soporte a sistemas Windows.
- En tercer lugar, dada la deficiente interfaz de programación proporcionada al usuario, (fundamentalmente en modo consola), se debe proporcionar una interfaz gráfica de programación en Maude que aumente la productividad del equipo de MOMENT en el desarrollo del *kernel* (álgebra ejecutable en Maude).

Todo este trabajo proporcionará las bases necesarias para continuar el desarrollo de MOMENT por parte de otros participantes del proyecto. En este sentido, no referimos, por ejemplo, a la programación del álgebra del *kernel*, el diseño

de puentes tecnológicos entre EMF y Maude [Ibo05] o la creación de la interfaz de diseño e invocación de operadores, entre otras cosas.

Una vez proporcionada la integración de Maude en Eclipse se procederá al diseño e implementación de las herramientas que den soporte para la trazabilidad en EMF.

En este caso, los objetivos son:

- Definir los metamodelos de trazabilidad que permitan capturar la información de las trazas.
- Proporcionar los mecanismos pertinentes para que el usuario pueda definir metamodelos de trazabilidad acordes a sus necesidades.
- Proveer de las herramientas necesarias para la visualización y consulta de modelos de trazabilidad.
- Proporcionar un mecanismo de creación automático de modelos de trazabilidad, que pueda conformar cualquier metamodelo de trazabilidad válido.
- Diseñar un editor que permita establecer manualmente las correspondencias entre modelos, dada la utilidad también de esta aproximación.

1.2. Descripción del documento.

El siguiente trabajo se organiza de la siguiente manera: El capítulo dos, presenta todos los fundamentos teóricos sobre los que se asienta MOMENT: *Model–Driven Engineering*, *Model–Driven Architecture* y *Meta–Object Facility*. También se presentan brevemente la aproximación de Gestión de Modelos y se hace una descripción de los espacios tecnológicos empleados por la herramienta: Maude y Eclipse.

El capítulo tres presenta el framework MOMENT, mostrando de forma somera cómo se comunican Maude y EMF. Esto sirve de ejemplo de motivación para el capítulo cuatro, donde se describe cómo se han implementado, y qué funcionalidad proporcionan las herramientas que integran Maude en Eclipse.

En el capítulo cinco se presenta de forma amplia las bases conceptuales del soporte para trazabilidad proporcionado por MOMENT. Igualmente en este capítulo se muestra un caso de estudio, que sirve para ilustrar la importancia de la gestión de la trazabilidad en una herramienta de gestión de modelos. De la misma forma, se muestra la utilidad de las herramientas implementadas.

El capítulo seis, por otra parte, describe de forma detallada la imlementación y funciones de las herramientas de soporte para trazabilidad de MOMENT. Por último el septimo capítulo expone las conclusiones obtenidas en este proyecto.

Los capítulos restantes recogen información complementaria a las secciones precedentes. El punto octavo contiene una relación de todas las referencias hechas a otros textos.

El primer anexo muestra de forma detallada el complejo proceso de compilación e instalación de Maude en un sistema Windows. El segundo, por otra parte, muestra el instalador *Maude for Windows*. Éste se ha implementado en este proyecto para configurar un sistema Windows para poder hacer uso de Maude como si de un programa Windows convencional se tratara.

El anexo tercero incluye el código fuente del *script* de creación del programa de instalación de *Maude for Windows*. El anexo IV contiene las gramáticas empleadas para descomponer comandos de full Maude; y el anexo V, recoge todos los modelos empleados en el caso de estudio de la sección quinta.

El último anexo muestra el proceso de creación paso a paso que se ha realizado para crear un plug-in de EMF con el código generado para el metamodelo de trazabilidad básico.

2. INTRODUCCIÓN.

2.1. *Model Driven Engineering* y *Model Driven Architecture.*

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

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

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

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

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

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

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

No obstante, esto conlleva un problema: el ciclo de vida de un sistema puede ser extremadamente largo (incluso más de treinta años [Fond04]). Depender en este caso de una única herramienta CASE (o un pequeño conjunto) es a menudo inaceptable, ya que pueden dejar de estar soportadas a lo largo del periodo. Una solución, es desarrollar según una serie de estándares que todas las herramientas CASE deberán usar a lo largo del ciclo completo de vida de los sistemas software.

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

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

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

2.1.1. Aplicación de un proceso MDE.

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

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

La aplicación de un proceso MDE se muestra en la Figura 1. El sistema en desarrollo es descrito en primer lugar por un modelo a un alto nivel de abstracción, esto es, ignorando cualquier tipo de dependencia de la plataforma. Este modelo es llamado *Computational Independent Model* (CIM) en la terminología MDA. Un buen candidato para el CIM son, por ejemplo, los casos de uso. Posteriormente, se deben realizar una serie de refinamientos interactivos con el objetivo de hacer el sistema más específico de la plataforma en cada paso. Por ejemplo, en el siguiente paso, el

sistema podría ser expresado de nuevo, pero de forma más precisa. En este caso se podría describir mediante diagramas de clases y diagramas de estados, para mostrar el comportamiento del sistema.

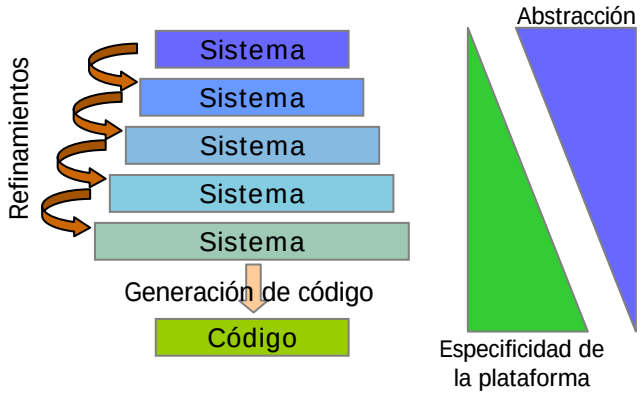

Figura 1: Aplicación de un proceso MDE.

En la terminología introducida por MDA, un modelo refinado se llamará Modelo Específico de Plataforma (PSM), y un modelo en el nivel de abstracción inmediatamente superior (que fue el modelo fuente para el paso de refinamiento correspondiente), será llamado Modelo Independiente de Plataforma (PIM).

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

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

2.2. Meta Object Facility.

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

MOF (Meta Object Facility) es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo. Mediante MOF pueden definir nuevos metamodelos, y por lo tanto nuevos vocabularios (de hecho se podría decir lenguajes, pero es conveniente no utilizar el término para evitar confusiones) con las mismas herramientas con que se definen modelos. Por otra parte, cabe preguntarse si existe un vocabulario de modelos superior que se utiliza para definir metamodelos.

La respuesta es que sí, a este metamodelo de metamodelos se le denomina metametamodelo. Pero como también es un modelo, ¿se podría seguir extendiendo esta pirámide de forma infinita?

En la práctica esto no tiene sentido, y los metamodelos y modelos se suelen organizar en una estructura de cuatro capas M3-M0 con la siguiente distribución:

- En el nivel M1 se sitúan los modelos, tal y como los hemos introducido aquí, descripciones abstractas de un sistema
- En la capa inmediatamente superior, denominada M2, se sitúan los metamodelos, «vocabularios para definir modelos».
- El nivel M3, que cierra la estructura por arriba, contiene el vocabulario base que permite definir metamodelos. Cabe resaltar que este nivel suele contener un único vocabulario, que caracteriza la aproximación de modelos escogida.

  Es imperativo que este vocabulario o metametamodelo esté definido utilizando como vocabulario a sí mismo, de ahí que se cierre la estructura.

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

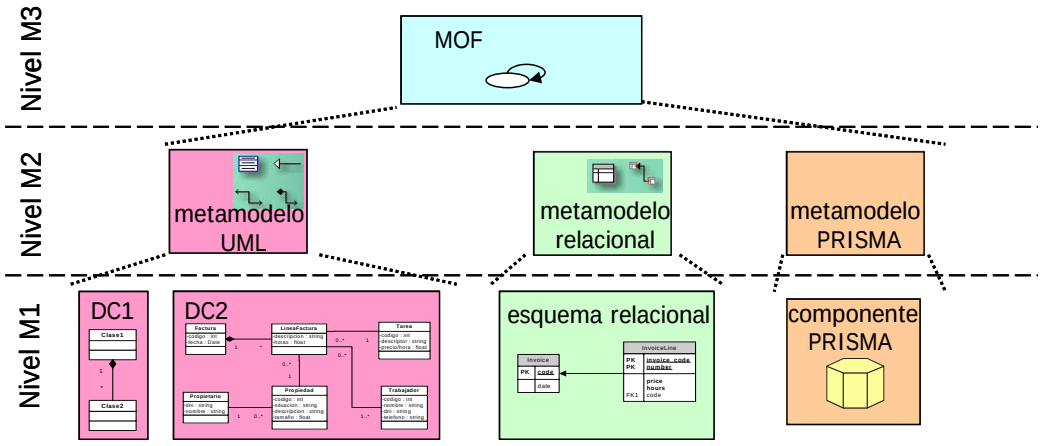

Figura 2: Arquitectura de niveles de MOF.

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

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

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

- UML, que proporciona un vocabulario para describir gran cantidad de sistemas.

  UML se caracteriza por ser un vocabulario independiente de dominio, si bien tiene sus raíces en el modelado orientado a objetos.

- CWM, un vocabulario específico para el dominio de los sistemas relacionados con la minería o explotación de datos
- QVT, un vocabulario que extiende OCL para expresar relaciones entre modelos.

2.3. Gestión de Modelos.

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

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

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

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

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

Evidentemente este esquema de trabajo es muy costoso y poco reutilizable, ya que generalmente las soluciones creadas no son lo suficientemente genéricas para ser aplicables a más de un metamodelo, y la interoperabilidad entre soluciones elaboradas por distintas partes es poco menos que imposible. Estas soluciones *ad-hoc* son costosas de implementar debido a la escasa ayuda proporcionada por los entornos de desarrollo actuales poco familiarizados con modelos, y costosas de rentabilizar, ya que continuamente aparecen nuevas aproximaciones o soluciones MDE y cuando, inevitablemente, se hace necesario cambiar de tecnología, resulta difícil reutilizar el trabajo realizado anteriormente.

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

productiva para trabajar con los procesos de transformación y refinamiento de modelos, de forma genérica y reutilizable.

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

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

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

2.3.1. Aproximaciones existentes.

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

2.4. MOMENT: Un marco de trabajo para Gestión de Modelos.

Basándose en la experiencia obtenida en la aplicación de formalismos de especificaciones algebraicas a la recuperación de sistemas legados [BoP04][BoCR05], se ha propuesto el desarrollo de una herramienta que dé soporte algebraico a los

operadores genéricos introducidos en la gestión de modelos. Esta herramienta se denomina MOMENT.

Con esta herramienta se pretende probar la falsedad del los mitos basados en la poca productividad de las herramientas formales y en el aumento del coste del proceso software debido a su uso. Como sistema formal, se ha elegido por tanto, un eficiente sistema de reescritura de términos, Maude, que ya ha sido empleado en muchos ámbitos formales [Mar02].

2.4.1. ¿Qué es Maude?

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

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

- Maude soporta de una manera sistemática y eficiente la reflexión lógica. Esto permite a Maude ser un lenguaje extremadamente potente y extensible, permitiéndole soportar un álgebra de operaciones de composición de módulos extensible.
- Maude es potente ya que puede modelar casi todo, desde el conjunto de los números naturales hasta un sistema biológico donde se programe el lenguaje Maude a sí mismo. Se dice que cualquier cosa que se pueda escribir, hablar o describir mediante el lenguaje humano, se puede expresar con instrucciones Maude [MaudeMan].
- Maude es simple. Su semántica está basada en los fundamentos de la teoría de clases, lo cual es bastante intuitivo y directo. Comparado con la mayoría de los lenguajes, Maude no tiene apenas sintaxis que memorizar.
- Maude está bien establecido. 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.
- Maude es desafiante. Puede llegar a solucionar lo más difícil ó complejo. Esto desafía al programador a ser astuto, en vez de resolver el problema afrontándolo con una serie de variables globales y funciones que a menudo son un caos de por sí.

Aunque Maude es un intérprete, su rendimiento es tal que puede ser usado en aplicaciones serias con un rendimiento competitivo y muchas ventajas sobre código convencional. Ilustramos esto con un ejemplo. Un componente, que se usaba para probar si una traza de eventos satisfacía cierta fórmula dada en lógica lineal temporal (LTL), fue escrito en Maude para un proyecto de la NASA [MaudeMan]. El componente tenía una prueba trivial de corrección, ocupaba apenas una página y fue desarrollado en unas horas. Este componente reemplazó un componente similar

escrito en Java que tenía aproximadamente 5000 líneas y que llevó más de un mes para ser desarrollado por un programador con experiencia. El código Java traducía 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.

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

2.4.2. Eclipse y Eclipse Modeling Framework.

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

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

2.4.2.1. ¿Qué es Eclipse?

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

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

2.4.2.2. Los proyectos.

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

i. El proyecto Eclipse.

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

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

ii. El proyecto de Herramientas.

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

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

iii. El proyecto de Tecnología.

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

2.4.2.3. La plataforma Eclipse.

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

i. Arquitectura de plug-ins.

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

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

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

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

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

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

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

iii. El framework UI.

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

2.4.2.4. ¿Qué es *Eclipse Modeling Framework*?

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

EMF es básica y simplemente un *framework* para describir un modelo y posteriormente poder generar otros elementos a partir de él. No obstante, hay una importante diferencia que hace que no sólo sea eso: EMF está realmente afinado e integrado para una programación eficiente.

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

Un modelo EMF es esencialmente el subconjunto de los diagramas de clases de UML. Esto es, un simple modelo de las clases o datos de la aplicación. Por esto, un amplio porcentaje de los beneficios del modelado pueden obtenerse en un entorno de desarrollo Java estándar. La correspondencia estre un modelo EMF y el código Java que lo implementa es sencilla y natural. En el apartado 6.1 presentaremos EMF y sus capacidades en mayor detalle.

2.5. Espacios Tecnológicos y puentes.

Como se ha observado, MOMENT se desarrolla en dos ámbitos tecnológicos completamente diferenciados. A un lado, se encuentra la parte de interfaz del usuario y modelado, implementada como un nuevo *framework* para Eclipse; y al otro lado, el motor de cálculo: el álgebra de MOMENT ejecutándose sobre Maude. A cada uno de estos ámbitos diferenciados se le denomina «espacio tecnológico».

El concepto de espacios tecnológicos fue introducido en [Kurt02] 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 [Béz05]. 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.

2.5.1. Puentes tecnológicos.

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

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

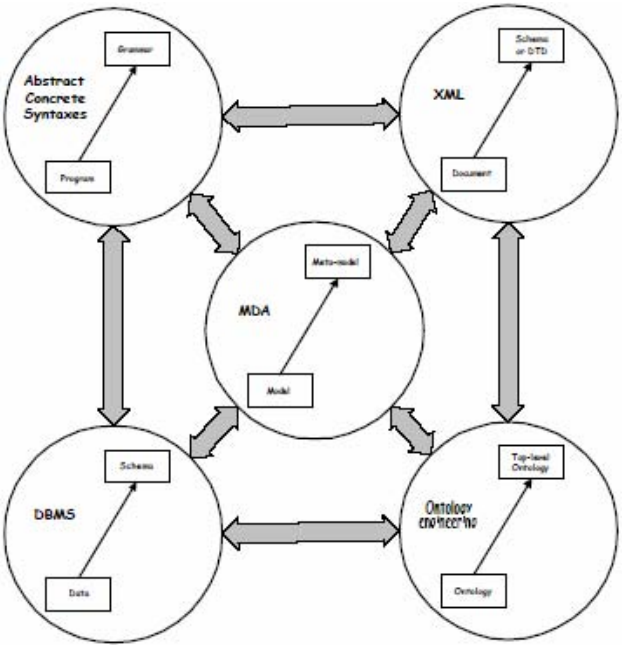

Figura 3: Cinco espacios tecnológicos y diversos puentes entre ellos. [Kurt02].

En MOMENT los operadores de gestión de modelos [Ber00] han sido especificados algebráicamente utilizando el formalismo Maude como se ha comentado anteriormente. El ET de Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc.

Este ET también puede ser visto como un paradigma de modelado, considerando el álgebra universal de Maude como el lenguaje de definición de

metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

MOMENT representa un modelo como una estructura de términos algebraicos, caracterizados por una especificación algebraica que proviene del metamodelo.

Para poder utilizar MOMENT desde la ingeniería de modelos será necesario disponer de unos puentes tecnológicos entre ambos espacios tecnológicos. Este problema es resuelto en [Ibo05]. En dicho proyecto se resuelve la definición y construcción de estos puentes tecnológicos entre el ET de Maude y el ET de EMF.

Estos puentes creados permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF. Se ha escogido EMF dentro del campo MDE por su interoperabilidad.

Se espera que EMF sea una puerta de entrada a otros entornos MDE, y que de esta manera el trabajo realizado sirva para habilitar de manera lo más completa posible la interoperabilidad de MOMENT con MDE.

Parte Segunda:

Integración de un sistema de Reescritura de Términos en Eclipse.

3. EJEMPLO DE MOTIVACIÓN: PRESENTACIÓN DEL FRAMEWORK MOMENT.

Ya se ha comentado en numerosas ocasiones que nuestra herramienta de gestión de modelos, llamada MOMENT (MOdel manageMENT), utiliza el entorno Maude desde un entorno de modelado industrial, como es *Eclipse Modeling Framework* (EMF) [EMF]. Esta integración de un método formal en una herramienta industrial de desarrollo de software, combina los esfuerzos que se están realizando sobre ambas herramientas en direcciones divergentes: en Maude sobre aspectos teóricos, y en EMF sobre su aplicación a la Ingeniería del Software.

3.1. Visión global del framework MOMENT.

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

Para ilustrar el funcionamiento de MOMENT, se indica un pequeño ejemplo de integración de esquemas XML. Para ello se ha definido una parte del metamodelo del lenguaje de definición XML (XSD), mostrado en notación UML en la Figura 4.

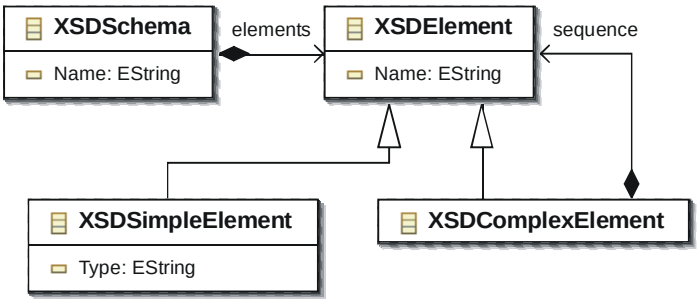

Figura 4: Parte del metamodelo XSD.

Utilizando el editor en forma de árbol que proporciona EMF, definimos los esquemas XML A y B en la Figura 5. Se aplica el operador *Merge* a ambos, obteniendo el esquema XML integrado C y dos modelos de trazas (mapAC y mapBC) que enlazan los elementos de los modelos de entrada con los elementos del modelo de salida. La invocación del operador es la siguiente: *<C, mapAC, mapBC> = Merge(A, B).*

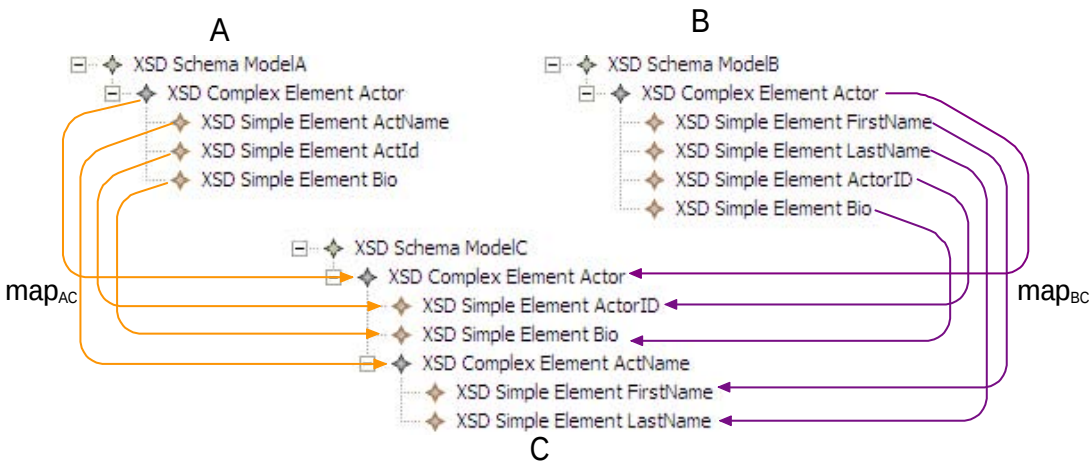

Figura 5: Aplicación del operador Merge.

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

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

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

Siguiendo un enfoque de Ingeniería de Modelos, para tratar con artefactos software utilizamos la terminología que define el estándar Meta-Object Facility de la iniciativa MDA. Este estándar, como se comentó en el apartado 2.2, presenta una arquitectura de cuatro capas de modelado que permite clasificar artefactos software con diferente propósito: M3 (metametamodelos), M2 (metamodelo), M1 (modelo), M0 (sistema real).

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

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

3.3. Proyecciones de artefactos software EMF sobre Maude.

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

El espacio tecnológico EMF se caracteriza por las facilidades que ofrece para representar una buena variedad de artefactos software como modelos y por su interoperabilidad con otras herramientas industriales de modelado

El espacio tecnológico Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET también puede ser visto como un paradigma de modelado, considerando el lenguaje Maude como el lenguaje de definición de metamodelos en el nivel M3. En el nivel M2, los metamodelos son los módulos que proporcionan especificaciones algebraicas Maude.

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

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

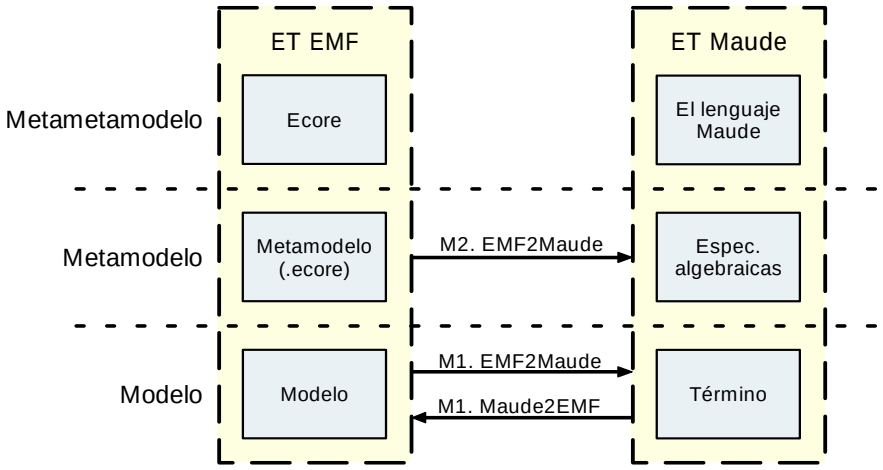

Figura 6: Enlaces entre el ET EMF y el ET Maude.

En la figura se muestran los enlaces que se han implementado en [Ibo05] para enlazar los espacios tecnológicos Maude y EMF. Se observa que el puente a nivel M2 es unidireccional. Éste sirve para añadir al álgebra de MOMENT la extensión correspondiente a un determinado metamodelo. En el nivel M1, por otra parte, el puente es bidireccional. En sentido EMF2Maude sirve para enviar a Maude la correspondiente representación como término de un modelo EMF dado. En el sentido inverso se emplea para obtener de nuevo la representación en EMF de un modelo que ha sido producido por Maude como resultado de una operación.

4. MAUDE DEVELOPMENT TOOLS: INTEGRACIÓN DE UN SISTEMA DE REESCRITURA DE TÉRMINOS (MAUDE) EN ECLIPSE.

El primer paso para hacer realidad la herramienta de Gestión de Modelos MOMENT sobre Eclipse es proporcionar los mecanismos necesarios para hacer posible una comunicación entre Java y Maude, de forma que se puedan establecer los puentes entre los dos espacios tecnológicos EMF y Maude.

Eclipse, como ya se reseñó anteriormente, proporciona un esquema de diseño modular y fácilmente extensible, por el cual, cualquier funcionalidad que éste proporciona está implementado como un plug-in. Por esto, para añadir este módulo de comunicación con un proceso Maude, deberemos diseñar uno o varios plug-ins que nos permitan encapsular esta funcionalidad.

Este esquema, a su vez, a parte de favorecer la mantenibilidad del software, ya que la integración con Maude, es completamente independiente de otros componentes de la herramienta MOMENT, favorece que pueda ser utilizado por cualquier otra persona al margen de este proyecto.

4.1. Plug-ins desarrollados.

Para dar soporte a estas necesidades se han desarrollado dos plug-ins, denominados respectivamente *Maude Daemon,* y *Maude SimpleGUI*. El primero de ellos proporcionará los mecanismos necesarios para arrancar Maude e interactuar con el proceso y el segundo, proporcionará un sencillo IDE para desarrollar programas en Maude. A continuación se describirán en mayor detalle.

4.1.1. Maude Daemon.

4.1.1.1. Descripción.

En esta sección se presenta una descripción a alto nivel del plug-in. Se presentará las necesidades a las que el sistema debe dar soporte, las funciones que debe realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo, como dependencias de otro software.

i. Funciones del plug-in.

Este plug-in deberá de proporcionar la siguiente funcionalidad:

- Integrar el sistema de reescritura de términos Maude como un plug-in para Eclipse, de forma modular. Para que pueda ser empleado de forma independiente del framework de MOMENT.
- Proporcionar una API para controlar el proceso de Maude durante la ejecución de un programa de Java. Debe proporcionar una manera de configurar maude, iniciarlo, enviar comandos y obtener su respuesta, y detener el proceso (de forma normal, o forzosa, si el proceso no responde).
- Deberá almacenar un fichero donde se registre toda la salida devuelta por el proceso Maude, para así poder observar a posteriori el comportamiento del sistema en caso de fallos.
- Deberá proporcionarse un soporte gráfico para configurar Maude integrado en las presencias del Eclipse. Esto es, especificar la ruta del ejecutable y el fichero que define Full Maude, así como el tipo de ejecución (Core o Full Maude), o en qué fichero se guardará el registro de la ejecución.
- El plug-in deberá ejecutarse en el mayor número de plataformas posible (dependiendo, obviamente, de si es posible ejecutar Maude en ellos o no).

ii. Restricciones.

Dada la implementación actual de Maude, sólo es posible ejecutar este sistema sobre equipos con diversas variantes de sistemas UNIX. Sin embargo, basándonos en otros trabajos realizados, como por ejemplo Maude Workstation [Muñoz04], podemos ejecutar Maude sobre un sistema Windows haciendo uso del entorno CygWin [Cygwin].

Para poder ejecutar Maude en Windows es necesario instalar el entorno CygWin y volver a compilar, con las librerías necesarias el proyecto desde el código fuente, como se indica en el Anexo I.

Para evitar este tedioso trabajo en un sistema Windows (para sistemas Linux ya existen versiones compiladas directamente en el sitio del Proyecto Maude), se ha desarrollado una programa de instalación que nos permite instalar Maude en nuestro sistema Windows como si se tratara de un programa cualquiera, copiando únicamente las librerías de CygWin y otros programas requeridos por el framework MOMENT de forma sencilla y ocupando el mínimo espacio. Para mayor detalle, consultar el [Ref].

iii. Dependencias.

Para la ejecución correctamente de este plug-in será necesario disponer de una instalación de Maude que funcione correctamente en el sistema.

Para instalar Maude en un sistema UNIX, en la página web de Maude [Maude] están disponibles las instrucciones para instalarlo de forma rápida. Bastará con descargarse los ficheros binarios de la web, y descomprimirlos en la carpeta deseada.

Así mismo, también son necesarios disponer del shell «bash» instalado en nuestro sistema, y el programa «Hill», para terminar de forma forzosa la ejecución del proceso, o mandar determinadas señales al proceso.

En un sistema Windows, como se ha comentado anteriormente, será necesario disponer de un ejecutable ya compilado en CygWin, así como que las librerías cygwin1.dll y cygncurses-8.dll estén accesibles en la ruta de búsqueda predeterminada del sistema (declaradas en la variable de entorno *PATH*). De la misma forma, se deberán disponer de los programas bash.exe y kill.exe disponibles en el directorio /bin de la instalación de CygWin, y también deberán estar accesibles mediante la variable *PATH*.

Como ya se ha comentado, y para simplificar este proceso de configuración de una instalación Maude sobre Windows, se ha implementado un instalador que realiza todos estos pasos de forma rápida con un sencillo asistente.

Para la compilación de este proyecto, igualmente, será necesario disponer instalado en el Eclipse el plug-in de ANTLR para Eclipse [ANTLR]. Éste nos permitirá generar las clases necesarias para analizar código Full Maude que sea enviado a Maude, subdividiendo un conjunto de comandos en trabajos simples.

4.1.1.2. Arquitectura del plug-in Maude Daemon.

i. Estructura del plug-in: dos APIs diferenciadas.

La estructura del plug-in está organizada en diferentes paquetes, agrupando las clases según su funcionalidad:

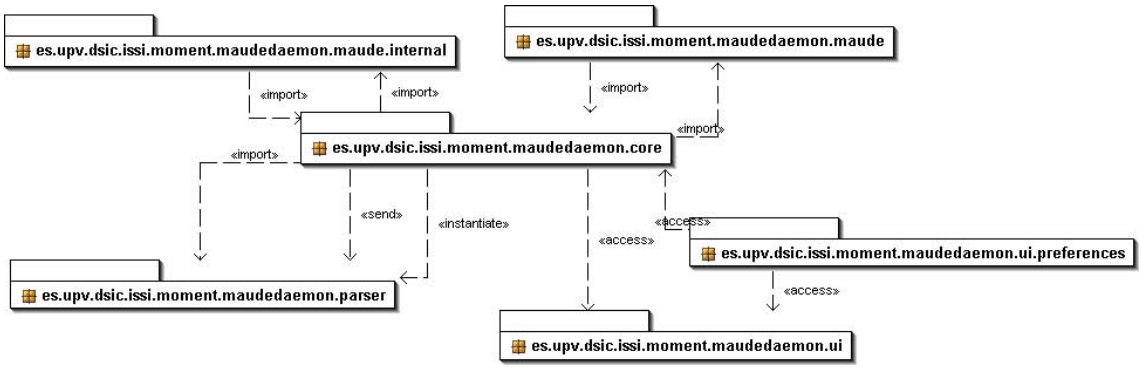

Figura 7: Diagrama de dependencias entre paquetes de Maude Daemon.

Como observamos en la Figura 7, contamos con seis paquetes. El paquete *core* (*es.upv.dsic.issi.moment.maudedaemon.core*) es el paquete principal, en él se definen las clases que permitirán crear e interactuar con el proceso Maude. En el paquete *maude*, se definen las interfaces que implementarán otras clases que intervienen con el proceso Maude, y en el paquete *maude.internal*, las clases que implementan de

estas interfaces. El paquete *parser* es generado a partir de una gramática por el plug-in *ANTLR*, y nos sirve de ayuda para analizar los comandos de Full Maude. Por último, los paquetes *ui*, y *ui.preferences*, agrupan las clases que implementan la interfaz al usuario del plug-in.

A continuación se muestra el diagrama de clases UML de las clases principales (estas son, las que se relacionan directamente con las clases *MaudeProcess*).

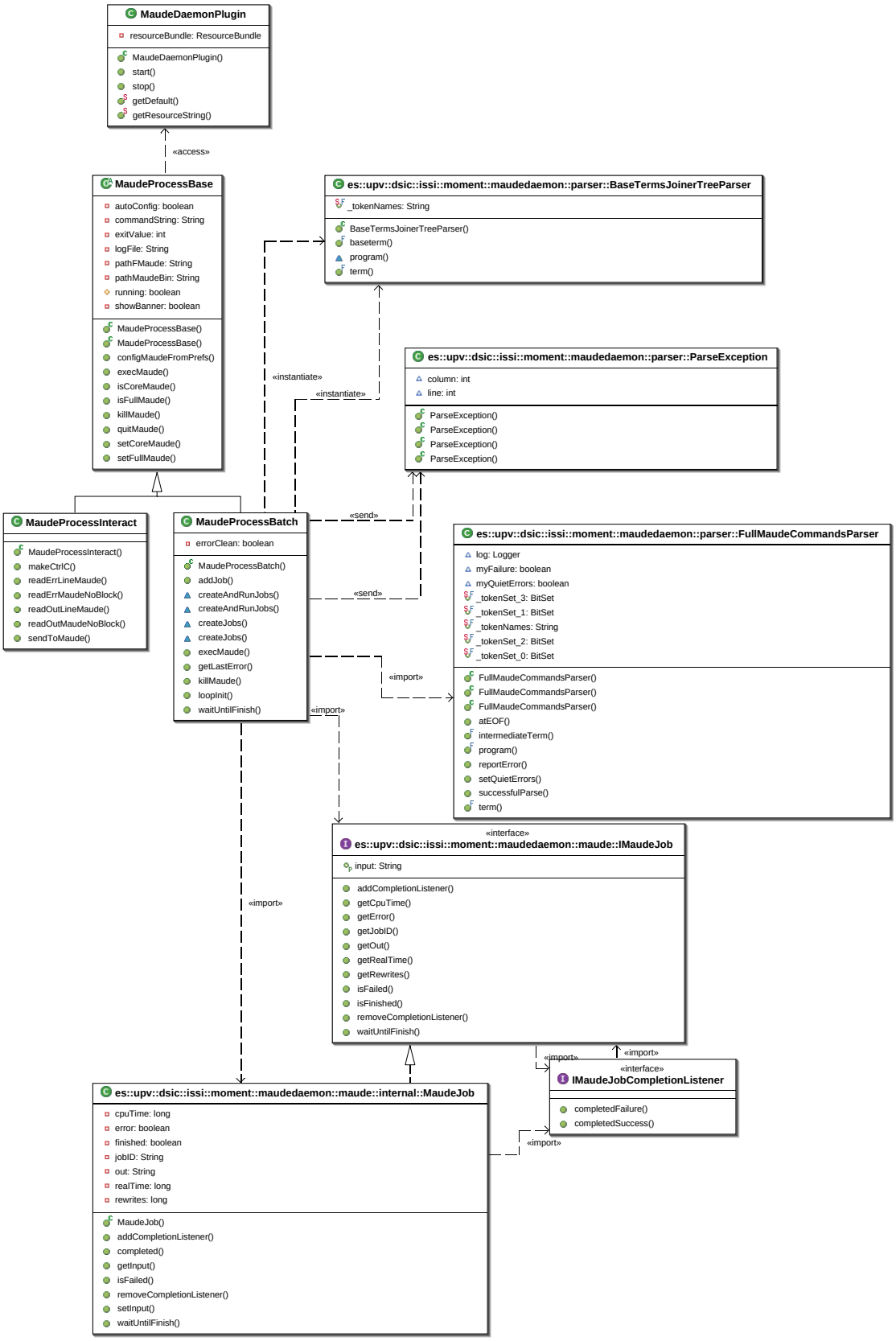

Figura 8: Diagrama de las clases directamente relacionadas con un proceso Maude.

Como se puede observar en la Figura 8, existe una clase base, denominada *MaudeProcessBase* de la que heredan dos clases: *MaudeProcessInteract* y *MaudeProcessBatch*. La clase *MaudeProcessBase* es la que implementa todo el

mecanismo de creación de un proceso Maude. Las clases hijas definirán los métodos para proporcionar las diferentes APIs para el programador, una para usar de forma interactiva, y otra como un proceso por lotes.

Ambas formas de interactuar sobre un proceso Maude son incompatibles entre sí, ya que en modo *batch*, un acceso mediante la API interactiva podría causar un fallo de sincronización, que provocaría que no se detectara el final de un trabajo, y el proceso se quedara bloqueado.

Es por esta razón por la que se ha optado por este diseño, en el que estas APIs se proporcionan en clases separadas, impidiendo que se pueda interactuar con un proceso Maude con ambas APIs de forma simultánea.

ii. Creación de un proceso Maude.

El mecanismo de creación de un proceso Maude está basado en el trabajo realizado por Alfredo Muñoz en la implementación de su plataforma Maude Workstation [Muñoz04].

Para lanzar un proceso externo al programa en ejecución, del que se dispone de un programa ejecutable en disco, se hace uso del método *exec(String cmd)* de la clase *java.lang.Runtime*, donde *cmd* es el comando externo a ejecutar. Esto devuelve una instancia de *java.lang.Process*, que es una instancia de la clase que permite controlar el proceso que hemos lanzado.

La clase *Process*, es la que proporcionará los métodos para interactuar con nuestro proceso lanzado. En particular, y los que nos resultarán más relevantes, son getInputStream(), getOutputStream() y getErrorStream(). Estos métodos devuelven los *Streams* correspondientes a la entrada estándar del proceso, a la salida estándar, y a la de error, respectivamente.

La lectura y escritura de forma adecuada de estos *Streams* es la que nos permitirá enviar comandos a Maude, y obtener sus respuestas o posibles errores.

El código, por tanto, que nos permite ejecutar un programa, y controlarlo es básicamente, el siguiente:

```
Process maudeProcess = runTime.exec(commandString);

BufferedWriter buffStdin = new BufferedWriter(
                              new OutputStreamWriter(maudeProcess.getOutputStream()));

BufferedReader buffStdout = new BufferedReader(
                              new InputStreamReader(maudeProcess.getInputStream()));

BufferedReader buffStderr = new BufferedReader(
                              new InputStreamReader(maudeProcess.getErrorStream()));
```

Ahora bien, ¿qué comando se debe ejecutar? Para el correcto funcionamiento del proceso Maude, en su versión para Linux, se debe ejecutar en el modo interactivo y a través del comando «*bash*» de Linux dentro de la aplicación. El comando es el siguiente:

```
"bash -c echo $$; exec ‘pathMaude’ -interactive"
```

El por qué de utilizar el *shell* «*bash*» para lanzar el proceso Maude es una cuestión que existe cuando se hace realiza el *CTRL–C* de Maude. Hacer un *CTRL–C* mientras se ejecuta Maude, supone que el proceso recibe una señal *SIGINT*, que provocará que Maude entre en modo traza.

Para hacer un *CTRL–C* de forma externa, se deberá por tanto mandar una señal SIGINT, lo que podemos conseguir mediante el comando UNIX «*kill –2 PID_del_proceso*», pero para esto, como observamos, necesitamos saber el PID de nuestro proceso Maude. Es por esto que se necesita ejecutar el *shell* «*bash*». Esto nos permite, mediante el «*echo $$*» obtener el PID del proceso, que podremos utilizar posteriormente para mandar señales al proceso.

Como se observa, Maude también se ejecuta con la opción «*–interactive*». Si esto no fuera así, al recibir una señal, el proceso Maude finalizaría por completo. Con este modificador se consigue que Maude actúe como es de esperar.

En el caso de que se use Maude versión Windows, compilado mediante CygWin, el comando adecuado será el que viene a continuación:

```
"cmd.exe /C bash -c \"echo $$; exec 'pathMaude' -interactive -no-tecla""
```

Como se observa, se sigue empleando el comando *bash* para ejecutar Maude. Esto es posible gracias a que se usa *CygWin* para dar un soporte similar a un sistema *UNIX*.

Un inconveniente que posee el proceso Maude con C*ygWin* es que, obviamente, gestiona las rutas al estilo *UNIX* "/maude/maude.exe" en vez de "c:\maude\maude.exe".

Existe un comando *CygWin*, llamado «*cygpath*», que permite obtener una ruta en formato *UNIX* para una ruta en formato Windows. Para evitar depender de un programa externo más, se ha simulado el comportamiento de este programa mediante la función «*private String getPathCygwin(String path)*».

En caso de que Maude, *CygWin*, y Eclipse no se encuentren en la misma unidad, cabe la posibilidad de que transformar una ruta al estilo «*c:\maude\maude.exe*» en «*/maude/maude.exe*» no funcione correctamente. Para tener acceso desde la raíz del sistema *CygWin*, a cualquier unidad de disco del sistema, existe el directorio «*/cygdrive/UNIDAD/*». De esta forma, la ruta anterior, se transformaría en «*/cygdrive/C/maude/maude.exe*», que funcionaría bien todos los casos, y permite que los programas se encuentren en cualquier unidad de disco del sistema.

Todos estos pasos se ejecutan mediante la invocación del método execMaude(). Si Maude está correctamente configurado en la ventana de preferencias de Eclipse basta con invocar este método para que todo funcione correctamente.

iii. Funcionamiento de las APIs.

Como ya se ha comentado, existen dos APIs diferenciadas, según se desee ejecutar Maude. En primer lugar, se comentará *MaudeProcessInteract*, ya que proporciona una API de más bajo nivel, y en segundo lugar, *MaudeProcessBatch*, que será la API utilizada por el framework MOMENT.

Estas dos clases, que heredan de *MaudeProcessBase*, implementan los métodos que tienen que ver con la entrada y salida del proceso Maude, y extienden aquellos que son necesarios para manejar correctamente los *buffers*.

a) MaudeProcessInteract.

El funcionamiento de esta API es sencillo. Básicamente, su objetivo es encapsular mediante métodos las lecturas y escrituras sobre los *buffers* del proceso Maude en llamadas a métodos, para simplificar la comunicación con el proceso Maude a ojos del programador. Sus principales métodos son:

```
public void sendToMaude(String txt)
public String readOutLineMaude()
public String readErrLineMaude()
public String readOutMaudeNoBlock(int maxLen)
public String readErrMaudeNoBlock(int maxLen)
```

El primero de ellos, nos sirve para mandar un texto a Maude y que se ejecute. Los cuatro siguientes corresponden a los métodos de lectura de la respuesta de Maude, con sus variantes bloqueantes y no bloqueantes. Cada vez que se invoca a cualquiera de estos cuatro últimos métodos, la cadena devuelta es también escrita a su vez en el fichero de registro de ejecución de Maude. Más adelante, se describirán estos métodos en mayor detalle.

Para ejecutar un comando, así pues, bastará con realizar un *sendToMaude(COMANDO),* y obtener su respuesta con una o sucesivas llamadas a *readOutMaudeNoBlock(NUM_CHARS)*, o cualquier otra variante del método de lectura. Cabe destacar que los métodos no bloqueantes pueden devolver una cadena vacía si Maude todavía no ha procesado por completo el comando enviado.

Por otra parte, la lectura por líneas debe realizarse cuidadosamente, ya que en caso de que ya se hayan leído todas las líneas correspondientes a una respuesta, y en el *buffer* de salida solo quede por leer el prompt de Maude (`Maude>`), realizar una lectura por línea (que es bloqueante), suspenderá la ejecución del programa que haya hecho la invocación del método, ya que dicha línea, no está completa.

Igualmente, el uso de estos métodos ha de realizarse con precaución, ya que descuidar la lectura de las salidas de Maude puede causar que los *buffers* del proceso se llenen y se bloquee del proceso. Se deja como tarea al programador la implementación de los hilos de ejecución que velen para que no se de esta situación. Como ya se ha señalado, esta es una API de bajo nivel destinada a la implementación de consolas virtuales de Maude, o para que el programador se diseñe su propia API de acceso a Maude. Para invocar comandos desde un programa Java en Maude, se ha diseñado la API de procesado por lotes.

b) MaudeProcessBatch.

b.1) Mecanismo de funcionamiento.

El funcionamiento de esta API es más elaborado que la anterior, pero no por ello más compleja en su uso para el programador. La idea fundamental es simple: todo comando o comandos que se deseen ejecutar en Maude es empaquetado en un trabajo (*job*) que se envía al proceso Maude. Un trabajo encapsula todos los datos que son relevantes para dicho conjunto de órdenes de Maude:

- Órdenes a ejecutar.
- Si ha sido procesado ya o no.

Y si las órdenes ya se han ejecutado:

- Respuesta de Maude.
- Respuesta por la salida estándar de Maude.
- Si ha fallado el comando o no.
- Número de reescrituras y tiempo de ejecución, si es aplicable.

Este método de interactuar con Maude es mucho más sencillo de cara a utilizarlo en un programa Java. La clase *MaudeProcessBatch* incluye los siguientes métodos, que nos permiten, a partir de un *stream* o una cadena de texto, crear uno o varios trabajos:

```
public List<IMaudeJob> createAndRunJobs (InputStream input) throws ParseException
public List<IMaudeJob> createAndRunJobs (String input) throws ParseException
public List<IMaudeJob> createJobs (String input) throws ParseException
public List<IMaudeJob> createJobs (InputStream input) throws ParseException
synchronized public void addJob(IMaudeJob job)
```

Los dos primeros únicamente crean los trabajos y también los añaden directamente en la cola de ejecución de Maude. Los dos siguientes métodos únicamente crean los correspondientes trabajos, y los devuelven en una lista. Cuando se desee, pueden ser también añadidos a la cola de ejecución de Maude usando el último método proporcionado.

El funcionamiento del proceso Maude es asíncrono, es decir, una vez mandados los trabajos a Maude, la ejecución del programa que hace uso de la API puede continuar normalmente sin quedarse detenida. Sin embargo, ya que puede resultar útil suspender la ejecución del programa hasta que se obtiene un determinado resultado, se han proporcionado sendos métodos para esperar la finalización de los trabajos: tanto la clase *MaudeProcessBatch* como *MaudeJob* proporcionan un método llamado *waitUntilFinish()*. El primer método, perteneciente a *MaudeProcessBatch,* suspende la ejecución hasta que todos los trabajos de la cola han sido procesados; el segundo, únicamente suspende el programa que lo invoca hasta que el trabajo al que pertenece el método termina.

Por otra parte, para terminar de comprender el funcionamiento interno de esta API, cabe comentar que la clase *MaudeProcessBatch*, contiene dos hilos de ejecución (*threads*) encargados de controlar la entrada, salida y salida de error del proceso.

Los trabajos son procesados en el orden de llegada a la cola. Esto es, se coge el primer trabajo de la cola, lo marca como trabajo activo, y uno de los hilos lanza las órdenes que contiene a Maude. El segundo hilo, lee de la salida estándar y la salida de error del proceso Maude y almacena los datos obtenidos en el trabajo. Cuando se detecta que se ha terminado de procesar, se marca el trabajo como terminado, se elimina de la cola, y se procede con el siguiente, si existe.

b.2) Creación de los trabajos.

El principal problema que encontramos en la ejecución de los trabajos en Maude está relacionado con la forma de interactuar con el proceso, (leyendo directamente texto de su salida estándar e interpretándolo). Esto es, la complejidad de determinar cuando un trabajo se ha terminado de procesar y se está empezando a procesar el siguiente.

La mejor forma de detectar esta condición sería que en cada trabajo únicamente se incluya una única orden de Maude (definición de un único módulo, una única operación de reducción, etc.), pero es una restricción muy dura obligar que cada trabajo que genere el programador incluya únicamente una orden. Cabría la posibilidad de que el programador ordenara la creación de los trabajos de un conjunto de órdenes, y el método de creación de los trabajos, analizara el contenido de las órdenes, y las descompusiera en tantos trabajos sencillos como órdenes.

Esta sería la aproximación más deseable, pero el diseño de un analizador que permitiera esto para Core Maude no es una tarea trivial, ya que requeriría generar un analizador para el lenguaje Core Maude por completo.

Sin embargo, para código Full Maude es una tarea relativamente sencilla, ya que toda orden en Full Maude está encerrada entre paréntesis «(», «)».

Pese a que esto únicamente nos sirve para código Full Maude, y por tanto, no resuelve el problema de forma genérica en absoluto, se podrá adoptar para generar un sistema más robusto, y con mayor nivel de detalle a la hora de reportar un error, ya que podremos identificar el trabajo que ha fallado, en su caso, de toda la lista de órdenes que hayan sido mandadas.

- **Creación de los trabajos en Full Maude.**

Para diseñar la solución se ha abordado el problema como un caso de análisis de lenguaje.

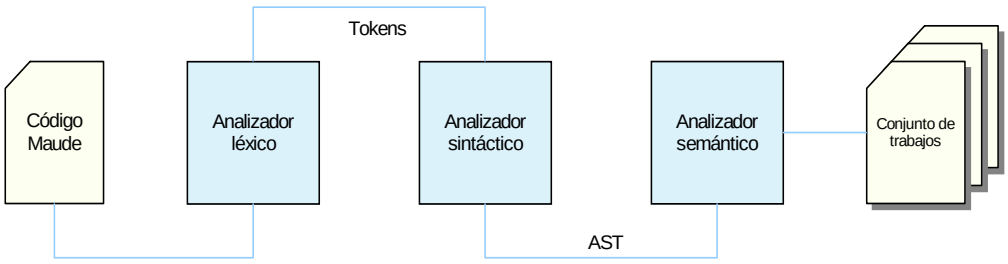

Figura 9: Proceso de análisis de un lenguaje.

En la figura se presenta un esquema del funcionamiento clásico de un analizador de lenguaje, muy aplicados en teoría de compiladores. 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 trabajos. 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 carácteres y los agrupa en tokens. Los tokens son los símbolos léxicos del lenguaje, también denominados lexemas. Se asemejan en cierta manera a las palabras en el lenguaje natural. Una vez identificados los tokens, son transmitidos al siguiente nivel de análisis.

  El programa que permite realizar este análisis es el analizador léxico, o simplemente lexer (o scanner).

- 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 comunmente pasadas, el AST es modificado, reestructurado, optimizado, etc. hasta conseguir el resultadofinal.

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

Una vez obtenido el AST final es transmitido al sistema siguiente, que en la bibliografía clásica suele ser un generador de código, para que ejecute las operaciones deseadas, para lo cual probablemente tenga que recorrer el árbol una vez más.

En nuestro caso obtendremos un AST con tantos hijos en su nodo raíz, como trabajos (bloques de código rodeados entre paréntesis) haya.

Para generar los analizadores léxico, sintáctico y semántico requeridos se ha empleado el generador de parsers ANTLR [ANTLR], 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. El conjunto de clases generadas las podemos encontrar en el paquete «*es.upv.dsic.issi.moment.maudedaemon.parser*».

- **Creación de trabajos en Core Maude.**

Una posible solución al problema de detectar el final de un trabajo es la ejecución de un determinado comando para el que podamos reconocer su respuesta, y que no produzca efectos colaterales en la ejecución de Maude.

Se ha optado en este caso, ejecutar un orden trivial: una reducción de un número natural. Pese a que es una orden sencilla, y que difícilmente vaya a ser ejecutada por nadie, se ha optado además, por ejecutar una reducción de un número al azar, que servirá de identificador del trabajo.

Así pues, todo trabajo, al ser creado, generará un número aleatorio entre 1 y 100.001, y lo almacenará como su identificador. Igualmente, añadirá a su lista de órdenes a ejecutar la orden «*red in NAT : ID_TRABAJO* . ». De esta forma, y sabiendo que todo trabajo terminará con una reducción de un número igual a su identificador, el hilo encargado de leer de la salida estándar de Maude podrá identificar el final de la ejecución de éste.

Una vez se ha terminado la ejecución de un trabajo, se marca como finalizado, y se procede a la ejecución de los métodos de post-procesado: éstos limpian el resultado devuelto por Maude eliminando información innecesaria, y comprobando posibles errores o advertencias.

Más adelante se entrará en mayor detalle en los métodos que proporciona cada clase y de su función.

4.1.1.3. Descripción detallada de los paquetes y clases.

i. es.upv.dsic.issi.moment.maudedaemon.core

El paquete core es el corazón del plug-in Maude Daemon. En él se definen las clases que permitirán declarar un proceso Maude como si de una clase se tratase, e interactuar con él por medio de una serie de métodos. A continuación se enumeran las clases e interfaces más interesantes declaradas en este paquete.

a) Interfaces

a.1) IMaudeJobCompletionListener

Esta interfaz declara los métodos que deben implementar los *listeners* asociados a un *MaudeJob*.

- **void completedFailure(IMaudeJob mj)**

Este es el método que será llamado desde *fireCompletionEvent()* en caso de que se haya producido algún fallo en la ejecución de un *MaudeJob*.

- **void completedSuccess(IMaudeJob mj)**

si no ha habido errores en la ejecución de un trabajo, se llamará a este método desde *fireCompletionEvent()*.

b) Clases:

b.1) MaudeDaemonPlugin

Esta clase es generada automáticamente mediante el asistente para crear un nuevo plug-in para Eclipse.

- **MaudeDaemonPlugin()**

Este es el método constructor, inicializa el campo «instancia compartida» (el propio plug-in).

- **static MaudeDaemonPlugin getDefault()**

Devuelve la instancia compartida.

- **java.util.ResourceBundle getResourceBundle()**

Devuelve el conjunto de recursos (resource bundle) del plug-in.

- **static java.lang.String getResourceString(java.lang.String key)**

Devuelve la cadena del paquete de recursos del plug-in o «key» si no se encuentra.

- **void start(org.osgi.framework.BundleContext context)**

Este método es llamado al iniciar el plug-in.

- **void stop(org.osgi.framework.BundleContext context)**

Este método es llamado al detenerse el plug-in.

b.2) MaudeProcessBase

Esta clase declara un proceso Maude básico. En ella se declaran los métodos para configurar un proceso Maude, iniciarlo y detenerlo; así como la inicialización de los buffers y fichero de log.

Es la clase base de la que después heredarán MaudeProcessBatch, y MaudeProcessInteract, que definirán los métodos para interactuar con Maude de forma interactiva, o como un proceso por lotes.

- **MaudeProcessBase()**

Este es uno de los constructors de MaudeProcessBase. En él se realñizan algunas inicializaciones, como el establecimiento del campo *runTime*.

- **MaudeProcessBase(java.lang.String[] commandString)**

Este constructor, además, nos permite establecer el comando que será ejecutado para lanzar el proceso Maude.

- **boolean configMaudeFromPrefs()**

Este método consulta la configuración que el usuario ha establecido para la ejecución de Maude en la hoja correspondiente de la ventana de preferencias de Eclipse. Según estos valores establece y configura las rutas y modo de ejecución según haya especificado el usuario.

- **private void createCommandToRun()**

A pesar de que este es un método privado, es interesante reseñarlo aquí, ya que es el que nos permite ejecutar Maude desde Java, tanto para un sistema Linux como para Windows.

En caso de que se pudiera ejecutar Maude en un sistema (seguramente UNIX), que no fuera Linux, el único cambio que se debería realizar en todo el plug-in sería en este método.

A continuación se muestra el segmento de código que permite crear el comando a ejecutar, tanto para Linux como para Windows:

```
/*
 * Platform dependent code
 */

if (System.getProperty("os.name").equals("Linux")) {

    // Linux System
    if (isCoreMaude())
        setCommandString(new String[] {"bash", "-c", "echo $$; exec '" + pathMaudeBin
        + "'" + (!showBanner ? " -no-banner " : "") + " -interactive "});

    else// Is Full Maude
        setCommandString(new String[] {"bash", "-c", "echo $$; exec '" + pathMaudeBin
        + "' '" + pathFMaude + "' "+ (!showBanner ? " -no-banner " : "")
        + " -interactive "});
} else {

    // We are running Windows
    String maude_comm = "bash -c \"echo $$; exec '"
        + getPathCygwin(pathMaudeBin) + "'"
        + (!showBanner ? " -no-banner " : "") + " -interactive -no-tecla \"";
    String fmaude_comm = "bash -c \"echo $$; exec '"
        + getPathCygwin(pathMaudeBin) + "' '" + getPathCygwin(pathFMaude) + "'"
        + (!showBanner ? " -no-banner " : "") + " -interactive -no-tecla \"";

    // Windows 95
    if (System.getProperty("os.name").equals("Windows 95")) {
        if (isCoreMaude())
            setCommandString(new String[]{"command.com", "/C", maude_comm});
        else
            // Is Full Maude
            setCommandString(new String[]{"command.com", "/C", fmaude_comm});
```

```
    // Windows NT/2000/XP
    } else {
        if (isCoreMaude())
            setCommandString(new String[]{"cmd.exe", "/C", maude_comm});
        else
            // Is Full Maude
            setCommandString(new String[]{"cmd.exe", "/C", fmaude_comm});
    }

}
```

Como observamos, para ejecutar Maude en un sistema Linux ejecutaremos un básicamente shell (*bash -c*) con argumentos «echo $$; exec '/path/a/maude.binario' –interactive –no–tecla» (a esta línea básica se le pueden añadir más argumentos, como la ruta del fichero Full Maude, o si deberá mostrar el banner inicial).

La ejecución de este comando nos proporcionará, en primer lugar el PID del proceso, que necesitaremos posteriormente para mandar una señal de CTRL-C, o para matar el proceso en caso de que no responda.

En el caso de un sistema Windows, el comando a ejecutar varía ligeramente. En caso de un sistema Windows 9X, el shell del sistema es «*command.com*», y de un sistema Windows NT o XP, «*cmd.exe*». El argumento «*/C*» es similar al «*-c*» de Linux, esto es, indicar al shell que deberá ejecutar las órdenes que se encuentren a continuación.

El comando a ejecutar a continuación es ya similar al ejecutado en Linux: «bash –c "echo $$; exec '/path/a/maude.binario' –interactive –no–tecla"», que como ya se ha indicado en numerosas ocasiones, funcionará correctamente si están instalados los ficheros de CygWin de la forma apropiada.

La implementación de este método está basada en el trabajo realizado por A. Muñoz [Muñoz04] en su implementación del entorno Maude WorkStation.

- **boolean execMaude()**

Este método causará que se ejecute Maude (según la línea de comando construida con *createCommandToRun()*) mediante el método *exec()* de la clase Java *java.lang.Runtime*. Si el flag de autoconfiguración está activo, se configurará Maude desde las preferencias, sino, los valores se tomarán según se haya configurado Maude desde la API proporcionada.

Igualmente, este método crea los correspondientes buffers de comunicación entre la API y el proceso Maude, y leerá el valor correspondiente al PID del proceso, para poder utilizarlo en ocasiones futuras.

- **java.lang.String[] getCommandString()**

Devuelve la cadena con el comando a ejecutar.

- **int getExitValue()**

Devuelve el valor de salida del comando ejecutado.

- **private java.lang.String getPathCygwin(java.lang.String path)**

Este método permite transformar un *path* en formato Windows a un *path* en formato UNIX, comprensible para las librerías de *CygWin*. Su respuesta es similar a la del comando *cygpath --unix* de *CygWin*, esto es, transformaría un path del estilo «d:\maude\maude.exe» a «/cygdrive/d/maude/maude.exe».

Se ha decidido hacer una implementación de este método, en lugar de usar una llamada a *cygpath* para depender del menor número de programas de *CygWin* posibles.

```
String result = "/cygdrive/" + path.charAt(0) +"/";
StringTokenizer stok = new StringTokenizer(path, "\\");
stok.nextToken();
while (stok.hasMoreTokens()) {
    result += stok.nextToken() + "/";

}
result = result.substring(0, result.length() - 1);
```

- **private boolean isConfigured()**

Este método comprueba que Maude está configurado, esto es, no existe ninguna ruta vacía.

- **boolean isCoreMaude()**

Devuelve true si Maude está configurado para ejecutar Core Maude, false en otro caso.

- **boolean isFullMaude()**

Devuelve true si Maude está configurado para ejecutar Full Maude, false en caso contrario.

- **boolean isRunning()**

Indica si Maude está en ejecución.

- **void killMaude()**

Este método cierra los *buffers* correspondientes y termina de forma forzosa la ejecución de Maude. Esto es útil en caso de que éste se haya bloqueado. Su funcionamiento consiste en ejecutar un «*kill –9*» mediante el método java.lang.Runtime.exec().

- **void quitMaude()**

Este método manda el comando «*quit*» a Maude para que éste termine su ejecución de forma voluntaria, y posteriormente, cierra los *buffers* de comunicación abiertos, así como el fichero de log.

- **protected void sendToLog(java.lang.String txt)**

Este método escribe el texto que se le pase como argumento en el fichero de registro establecido en la configuración de Maude.

- **void setAutoConfig(boolean value)**

Mediante la invocación de este método se puede activar o desactivar el *flan* de autoconfiguración de Maude.

- **private void setCommandString(java.lang.String[] commandString)**

Este método permite establecer el comando a ejecutar para lanza Maude.

- **void setCoreMaude()**

Con este método podemos establecer el modo de ejecución a Core Maude.

- **void setFullMaude()**

La invocación de este método establece el modo de ejecución a Full Maude.

- **void setLogFile(java.lang.String logFile)**

Este método permite establecer la ruta del fichero de registro de Maude.

- **void setPathFMaude(java.lang.String pathFMaude)**

La invocación de este método permite establecer la ruta del fichero donde se especifica Full Maude.

- **void setPathMaudeBin(java.lang.String pathMaudeBin)**

Este método permite establecer la ruta donde se encuentra el fichero ejecutable de Maude.

- **void setShowBanner(boolean showBanner)**

Este método permite cambiar el valor del flag «*showBanner*», que hará que se muestre o no el Banner de bienvenida al ejecutar Maude:

```
              \||||||||||||||||||/
            --- Welcome to Maude ---
              /||||||||||||||||||\
         Maude alpha86a built: Jul 14 2005 13:17:08
          Copyright 1997-2005 SRI International
            Tue Aug 2 21:16:48 2005

Maude>
```

b.3) MaudeProcessBatch

Esta clase es la que implementa los métodos para interactuar con Maude en forma de lotes. Es la que se usará en condiciones normales para lanzar comandos a Maude desde un programa Java, ya que proporciona una API de más alto nivel.

- **MaudeProcessBatch()**

Éste es el constructor de la clase. En él se crean e inician los hilos de ejecución que se encargarán de mandar los distintos trabajos a Maude así como de la obtención de su resultado.

- **void addJob(IMaudeJob job)**

Añade un MaudeJob dado a la cola de ejecución de Maude.

- **java.util.List<IMaudeJob> createAndRunJobs(java.io.InputStream input)**

Realiza la correspondiente llamada a *createJobs()*, para obtener una lista de trabajos a lanzar a Maude, y después los añade a la cola de ejecución.

- **java.util.List<IMaudeJob> createAndRunJobs(java.lang.String input)**

Este método crea un *InputStream* y realiza la correspondiente llamada a *createAndRunJobs(InputStream input).*

- **java.util.List<IMaudeJob> createJobs(java.io.InputStream input)**

Este método se encarga de generar una lista de trabajos que puedan ser mandados a Maude.

En caso de que se esté ejecutando Full Maude, el *stream* de entrada será analizado por un analizador léxico, que devolverá tantos trabajos como bloques entre paréntesis « (…) » existan en el *stream*. Esto se debe a la dificultad que existe en Maude para determinar cuándo se ha dejao de procesar un determinado trabajo y se está continuando con el siguiente, es por esto, que se descompone el *stream* en las unidades procesables mínimas.

El método invoca al analizador de la siguiente forma:

```
    if(isFullMaude())
        try {
            FullMaudeCommandsParser p = new FullMaudeCommandsParser(
                    new FullMaudeCommandsLexer(input));
            p.program();

            BaseTermsJoinerTreeParser bt = new BaseTermsJoinerTreeParser();
            List<String> commands = bt.program(p.getAST());
            for(String s : commands) {
                jobs.add(new MaudeJob(s));
            }

        [...]

    }
```

En el caso de Core Maude, el diseño de un analizador no es trivial, por lo que se optará por lanzar un comando cuya respuesta sea conocida, (por lo tanto detectable al leer la respuestas proporcionadas por Maude) y que no produzca efectos colaterales en la ejecución de Maude.

- **java.util.List<IMaudeJob> createJobs(java.lang.String input)**

Al igual que el método *createAndRunJobs(String input)*, este método genera un *InputStream*, para después realizar la correspondiente llamada a *createJobs(InputStream input).*

- **boolean execMaude()**

Este método extiende la funcionalidad de *execMaude()* de *MaudeProcessBase*. En primer lugar, realiza una llamada al método de la clase padre, y posteriormente, espera que Maude devuelva el control al usuario devolviendo el *prompt*, y vaciando la salida estándar del proceso.

- **protected void finalize()**

Este método termina de manera forzosa el proceso Maude realizando una llamada a *killMaude().*

- **IMaudeJob getLastError()**

Devuelve el último trabajo que produjo un error.

- **boolean isErrorClean()**

Devuelve el valor de *errorClean* (ocurrió algún error en la ejecución del trabajo, esto es, se interrumpió su proceso de forma anormal), y restaura el valor a *true.*

- **private void jobCompleted(IMaudeJob job)**

Actualiza, en caso de que el trabajo dado haya fallado, los valores de último trabajo que produjo un error, y la variable *errorClean*.

- **void killMaude()**

Este método extiende el método *killMaude()* de la clase padre, interrumpiendo antes de matar el proceso Maude los hilos encargados de la lectura y escritura de los *buffers* de entrada, salida estándar y salida de error.

- **void loopInit()**

Este método reinicia el bucle de Full Maude, vacía la cola de trabajos y reinicia los hilos de lectura/escritura.

- **void waitUntilFinish()**

La llamada a este método provoca que la ejecución del programa que lo invoca se suspenda hasta que hayan sido procesados todos los trabajos de la cola.

b.4) MaudeProcessInteract

Esta clase encapsula una API de bajo nivel que permite utilizar Maude de forma interactiva. Está diseñada para dar soporte al diseño de editores y emular la consola de Maude desde un programa en Java, de forma que se pueda mostrar en tiempo real los resultados que Maude devuelve.

Su objetivo es que sea el propio programador quien diseñe los hilos de ejecución que lean y escriban en los correspondientes *buffers* del proceso, según sus necesidades. Es por esto, que ha de ser utilizada con precaución, ya que descuidar la lectura de la salida estándar o la salida de error puede suponer que estos se llenen, y como consecuencia, el proceso Maude se suspenda.

- **MaudeProcessInteract()**

Éste es el constructor por defecto para la clase MaudeProcessInteract. Llama al constructor de la clase padre sin añadir ninguna funcionalidad adicional.

- **void makeCtrlC()**

Éste método provoca que se le mande una señal *SIGINT* (*interrupt*), mediante la ejecución de un «*kill -2*», de forma análoga a como se termina el proceso Maude de forma forzosa.

- **java.lang.String readErrLineMaude()**

Este método lee una línea de la salida de error del proceso Maude y la devuelve. Si no está disponible, se quedará bloqueado hasta que haya una línea disponible que leer.

- **java.lang.String readErrMaudeNoBlock(int maxLen)**

Este método lee los caracteres disponibles que haya en la salida de error del proceso Maude hasta un máximo dado (*maxLen*). En ningún caso, este método bloqueará la ejecución del programa que lo invoque.

- **java.lang.String readOutLineMaude()**

Este método lee una línea de la salida estándar del proceso Maude y la devuelve. Si no está disponible, se quedará bloqueado hasta que haya una línea disponible que leer.

- **java.lang.String readOutMaudeNoBlock(int maxLen)**

Este método lee los caracteres disponibles que haya en la salida estándar del proceso Maude hasta un máximo dado (*maxLen*). En ningún caso, este método bloqueará la ejecución del programa que lo invoque.

- **void sendToMaude(java.lang.String txt)**

Este método manda una cadena dada (*txt*) a la entrada estándar del proceso Maude, como si esta hubiera side escrita por teclado en su consola.

ii. es.upv.dsic.issi.moment.maudedaemon.maude

Este paquete encapsula las interfaces que se implementan distintas clases que conforman este plug-in.

a) Interfaces

a.1) IMaudeJob

Esta interfaz define los métodos que debe implementer un trabajo de Maude. Estos son:

- **public String getInput()**

Este método devolverá el código Maude que deberá ser procesado por Maude.

- **public void setInput(String input)**

Este método establece el código Maude que forma este trabajo y será procesado por Maude.

- **public String getOut()**

Este método devolverá el resultado «limpio» devuelto por Maude, esto es, sin el prompt, y otra información que no sea de interés.

- **public String getError()**

Este método devuelve el texto que Maude haya devuelto por su salida estándar de error (errores, alertas…).

- **public boolean isFailed()**

Este método determina y devuelve, tras analizar los resultados proporcionados tanto por la salida estándar como la salida de error, si ocurrió un error tras procesar el trabajo.

- **public boolean isFinished()**

La invocación de este método devuelve si el trabajo se ha terminado de procesar o no.

- **public void waitUntilFinish()**

La invocación de este método por parte de un programa provocará que éste se bloquee, esperando en ese punto hasta que la ejecución de dicho trabajo termine.

- **public long getCpuTime()**

Este método devolverá el tiempo de CPU consumido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el tiempo de la primera.

- **public long getRealTime()**

Este método devolverá el tiempo real consumido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el tiempo de la primera.

- **public long getRewrites()**

Este método devolverá el número de reescrituras producido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el número de reescrituras de la primera reducción.

- **public String getJobID()**

Este método debe devolver el identificador de un determinado trabajo.

- **public void addCompletionListener(IMaudeJobCompletionListener tracingCompletionHandler)**

Este método añadirá un *IMaudeJobCompletionListener* a la lista de *listeners* del trabajo.

- **public void removeCompletionListener(IMaudeJobCompletionListener tracingCompletionHandler)**

Este método eliminará de la lista de *listeners* del trabajo a *tracingCompetionHandler*.

iii. es.upv.dsic.issi.moment.maudedaemon.maude.internal

En este paquete se definen las clases que implementan las interfaces de «es.upv.dsic.issi.moment.maudedaemon.maude».

a) Clases

a.1) MaudeJob

- **public MaudeJob(String input)**

Este es el constructor de la clase. Al crear un objeto de ésta, se hace una llamada a *setInput(String input)*, y se genera un número aleatorio que será el identificador del trabajo.

- **public String getInput()**

Este método devuelve el código Maude que será procesado por Maude.

- **public void setInput(String input)**

Este método establece el código Maude que forma este trabajo y será procesado por Maude.

- **public String getOut()**

Este método devuelve el resultado devuelto por Maude tras post-procesarlo, para eliminar el prompt, y otra información que no es de interés.

- **public String getError()**

Este método devuelve el texto que Maude haya devuelto por su salida estándar de error (errores, alertas…).

- **public boolean isFailed()**

Este método determina y devuelve, tras analizar los resultados proporcionados tanto por la salida estándar como la salida de error, si ocurrió un error tras procesar el trabajo.

- **private synchronized void markFinished()**

Este método es el encargado de marcar el trabajo como terminado, indicarselo a todos los *listeners*, y post-procesar la salida devuelta por Maude.

- **public boolean isFinished()**

La invocación de este método devuelve si el trabajo se ha terminado de procesar o no.

- **public void waitUntilFinish()**

La invocación de este método por parte de un programa provocará que éste se bloquee, esperando en ese punto hasta que la ejecución de dicho trabajo termine.

- **public long getCpuTime()**

Este método devuelve el tiempo de CPU consumido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el tiempo de la primera.

- **public long getRealTime()**

Este método devuelve el tiempo real consumido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el tiempo de la primera.

- **public long getRewrites()**

Este método devuelve el número de reescrituras producido por la ejecución de un comando de reducción o similar. En caso de que en un mismo trabajo se hayan producido varias reducciones, indicará el número de reescrituras de la primera reducción.

- **private void fireCompletionEvent()**

Lanza un evento de que el trabajo se ha completado, para notificarlo a los *listeners* que observen este trabajo.

- **public String getJobID()**

Este método devuelve el identificador del trabajo que se ha generado al crearlo. Es consultado cuando se lee de la salida estándar de Maude para determinar si ha terminado ya o no.

- **public void addCompletionListener(IMaudeJobCompletionListener tracingCompletionHandler)**

Este método añade un *IMaudeJobCompletionListener* a la lista de *listeners* del trabajo.

- **public void removeCompletionListener(IMaudeJobCompletionListener tracingCompletionHandler)**

Este método eliminade la lista de *listeners* del trabajo a *tracingCompetion-Handler*.

iv. es.upv.dsic.issi.moment.maudedaemon.parser

Estas clases son generadas automáticamente a partir de las gramáticas de ANTLR que se muestran en el Anexo IV por lo que sólo comentaremos las clases de mayor interés.

Para consultar cómo se invocan estas clases se puede consultar en el punto 4.1.1.3 la descripción del método *java.util.List<IMaudeJob> createAndRunJobs(java.io.InputStream input)* comentado anteriormente.

a) Clases

a.1) FullMaudeCommandsLexer.

Esta clase corresponde con el analizador léxico de FullMaude. En su constructor recibirá la cadena de texto a analizar.

a.2) FullMaudeCommandsParser.

La clase *FullMaudeCommandsParser* recibe en su constructor una instancia de *FullMaudeCommandsLexer*, que a su vez habrá sido creada con el texto a analizar como entrada.

Mediante la llamada al método *program()* se provoca el inicio del análisis.

El método heredado *antlr.Parse.getAST()* devuelve el Árbol de sintaxis Abstracta (AST) generado.

a.3) BaseTermsJoinerTreeParser.

Esta clase es la encargada del último proceso de análisis.

- **public final List<String> program(AST _t) throws RecognitionException;**

El método *program(…)* recibe el *Árbol de Sintaxis Abstracta* y devuelve un vector de cadenas de caracteres. Cada cadena es un trabajo distinto (segmentos de código encerrados entre un par de paréntesis). Su invocación es, por ejemplo:

```
BaseTermsJoinerTreeParser bt = new BaseTermsJoinerTreeParser();
List<String> commands = bt.program(p.getAST());
```

v. es.upv.dsic.issi.moment.maudedaemon.ui

En este paquete encontramos las diferentes clases relacionadas con aspectos visuales del plug-in.

a) Clases

a.1) Messages

Esta clase encapsula el acceso al fichero de recursos donde se almacenan las cadenas de texto utilizadas en la interfaz del usuario del plug-in. Esta estructura facilita la traducción del plug-in a diferentes idiomas.

- **private Messages()**

Éste es el constructor de la clase.

- **public static String getString(String key)**

Devuelve la cadena correspondiente a la clave *key*.

vi. es.upv.dsic.issi.moment.maudedaemon.ui.preferences

En este paquete encontramos las definiciones de clases que definen la hoja que está integrada en la ventana de preferencias de Eclipse, y que permite configurar Maude.

a) Clases

a.1) MaudeDaemonPreferencePage

Esta clase representa la hoja de preferencias que se contribuye al diálogo de preferencias de Eclipse. Hereda de org.eclipse.jface.preference. FieldEditorPreferencePage, e implementa org.eclipse.ui.IWorkbenchPreferencePage.

- **public MaudeDaemonPreferencePage()**

Este es el constructor por defecto. Crea la página e inicializa ciertos valores, como el título.

- **public void createFieldEditors()**

Este método es el que establece el contenido de la hoja de preferencias. Crea los campos que compondrán la página. Cada campo sabe como salvarse y restaurar su contenido.

```
public void createFieldEditors() {
    addField(new FileFieldEditor(PreferenceConstants.P_PATHBIN,
        Messages.getString("MaudeDaemon.PREF_MAUDEBIN"), getFieldEditorParent()));
                                                                    //$NON-NLS-1$
    addField(new FileFieldEditor(PreferenceConstants.P_PATHFM,
        Messages.getString("MaudeDaemon.PREF_MAUDEFM"), getFieldEditorParent()));
                                                                    //$NON-NLS-1$
    addField(new DirectoryFieldEditor(PreferenceConstants.P_PATHLOG,
        Messages.getString("MaudeDaemon.PREF_LOGDIR"), getFieldEditorParent()));
                                                                    //$NON-NLS-1$
    addField(new StringFieldEditor(PreferenceConstants.P_LOGNAME,
        Messages.getString("MaudeDaemon.PREF_LOGFILE"), getFieldEditorParent()));
                                                                    //$NON-NLS-1$


    addField(new RadioGroupFieldEditor(
        PreferenceConstants.P_RUNTYPE,
        Messages.getString("MaudeDaemon.PREF_MODE"), //$NON-NLS-1$
        1,
        new String[][] {
            { Messages.getString("MaudeDaemon.PREF_RUNCORE"),
            Messages.getString("MaudeDaemon.PREF_RUNCORE_VALUE") }, //$NON-NLS-1$
                                                                    //$NON-NLS-2$
            { Messages.getString("MaudeDaemon.PREF_RUNFULL"),
            Messages.getString("MaudeDaemon.PREF_RUNFULL_VALUE") }  //$NON-NLS-1$
                                                                    //$NON-NLS-2$
        },
        getFieldEditorParent())
    );
}
```

En el ejemplo de código vemos como se establecen los distintos campos, ya sean para especificar un archivo, un directorio, una cadena o un grupo de botones de radio. Como se observa, se hace uso de la clase *Messages*, que permite centralizar en un único paquete la traducción de la interfaz.

- **public void init(IWorkbench workbench)**

Este método inicializa la hoja de preferencias para un determinado *Workbench*. Este método se llama automáticamente cuando la hoja de preferencias es creada e inicializada.

a.2) PreferenceConstants

Esta clase únicamente contiene las definiciones de constantes para las preferencias del plug-in como campos públicos.

a.3) PreferenceInitializer

Esta clase se utiliza para inicializar la hoja de preferencias la primera vez que se carga el plug-in. Hereda de *org.eclipse.core.runtime.preferences. AbstractPreferenceInitializer*, y su único método es:

- **public void initializeDefaultPreferences()**

Este método se invoca para inicializar la hoja de preferencias, esto es, la primera vez que se abre la hoja de preferencias de Maude, o en caso de que se presione el botón de restaurar las preferencias.

4.1.2. Maude SimpleGUI.

4.1.2.1. Descripción

Pese a que este plug-in no es directamente necesario para el framework MOMENT, ha sido desarrollado para aumentar la productividad en el desarrollo de otras partes de éste, principalmente el *kernel,* escrito en Maude, ya que los actuales entornos de programación en Maude son muy deficientes.

i. Funciones del sistema.

Este plug-in deberá proporcionar una interfaz de trabajo simple (ya que no es objetivo de este proyecto proporcionar un completo entorno de trabajo), pero que sea sencillo de manejar y proporcione la funcionalidad básica necesaria. Deberá proveer por tanto, los siguientes elementos:

- Un editor de texto que permita crear especificaciones en Maude, así como asistentes para la creación de nuevos ficheros Core Maude (extensión .maude) o Full Maude (.fm) integrados en los asistentes de Eclipse.
- Coloreado de sintaxis para facilitar la lectura de los programas Maude. Se deberán colorear las palabras claves del lenguaje Maude.
- El editor deberá proporcionar la capacidad de iniciar un proceso Maude (según esté configurado en las preferencias del plug-in Maude Daemon). Igualmente, deberá proporcionar la capacidad para detenerlo de forma normal o forzosa, en caso de que el proceso se quede bloqueado.
- Se podrá mandar como entrada al proceso Maude el contenido de los ficheros Maude que se encuentren en el entorno de trabajo, ya estén activos (abiertos en el editor) o no.

- Igualmente, se deberá proporcionar una interfaz que permite ver los resultados devueltos por Maude (ver la consola de Maude).
- En caso de que se esté ejecutando Full Maude, se debe de proporcionar la posibilidad de reiniciar el loop (loop init) de forma rápida.
- Deberán poderse utilizar todos los comandos de Maude, incluido el modo traza.

ii. Dependencias.

Para poder ejecutar este plug-in correctamente, obviamente, será necesario tener instalado el plug-in Maude Daemon, que es el que da soporte a la ejecución de Maude desde un programa Java. Se hará uso en este caso de la API interactiva, de más bajo nivel, pero necesaria para poder controlar el modo traza.

4.1.2.2. Arquitectura del plug-in Maude SimpleGUI.

i. Estructura de paquetes.

La estructura del plug-in está organizada en diferentes paquetes, agrupando las clases según su funcionalidad:

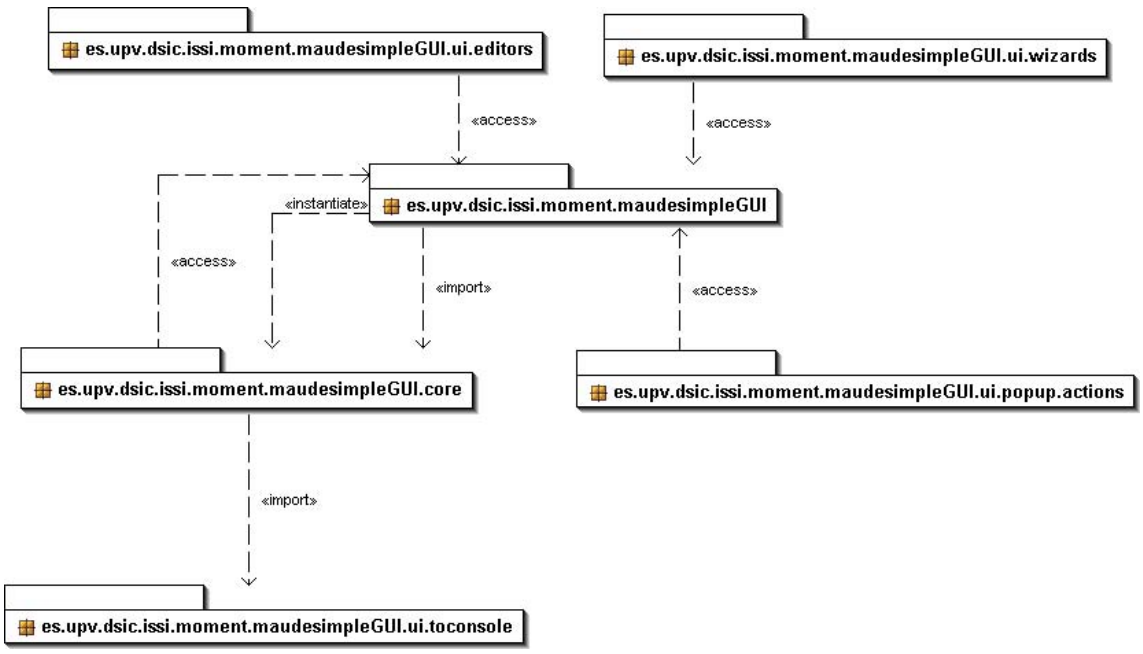

Figura 10: Diagrama de dependencias entre paquetes de Maude SimpleGUI.

El paquete principal es *es.upv.dsic.issi.moment.maudesimpleGUI*. En él se encuentran los datos que serán accedidos por todos los demás paquetes, principalmente, la instancia en ejecución de Maude. Igualmente, en este paquete se encuentra definida la clase *Message*, donde se encapsulan todos los posibles mensajes que muestre el plug-in al usuario.

En el paquete *core* está implementada la funcionalidad de Maude, y es el paquete que interactúa con el plug-in *Maude Daemon*. Proporciona una interfaz para acceder a Maude desde el resto de componentes del plug-in, y será la clase encargada de mostrar por consola (implementada en el paquete *ui.toconsole*) los mensajes devueltos por Maude.

El paquete *ui.popup.actions* implementa el menú contextual para enviar ficheros a Maude sin necesidad de ser abiertos en el editor, y en *ui.wizards*, se implementan los asistentes de creación de fichero nuevo.

ii. Un sencillo IDE para desarrollo de programas en Maude: Descripción y componentes.

a) Componentes no visuales.

Maude SimpleGUI es un entorno de desarrollo simple para la creación, prueba y ejecución de programas en Maude. Su núcleo básico y principal está formado por el editor, y la consola de Maude, que representan respectivamente, la entrada y la salida de Maude.

El funcionamiento del plug-in es sencillo: cuando éste se inicia se crea una instancia de la clase *Maude* (*es.upv.dsic.issi.moment.maudesimpleGUI. core.Maude*), que será el proceso Maude asociado a cualquier instancia abierta del editor.

La clase *Maude*¸ a su vez, creará un objeto *MaudeProcessInteract*, perteneciente al plug-in Maude Daemon (que debe estar también instalado en Eclipse); y una consola (*ConsoleOutputView*), que mostrará los resultados obtenidos de Maude.

Esta clase –*Maude*–, es la encargada de invocar los métodos del proceso Maude, así como de lanzar el hilo de ejecución dirigido a leer las respuestas e Maude, y mostrarlas por la consola. Igualmente, proporciona una serie de métodos, que serán los utilizados en el plug-in para interactuar con Maude, evitando invocar los propios de la clase *MaudeProcessInteract*.

b) Los asistentes.

Se han implementado dos asistentes, que nos permiten generar un fichero vacío con extensión «.maude» o «.fm», y al finalizar, los abren con el editor de código Maude.

Éstos están integrados en el menú para crear un nuevo documento de Eclipse como muestra la Figura 11.

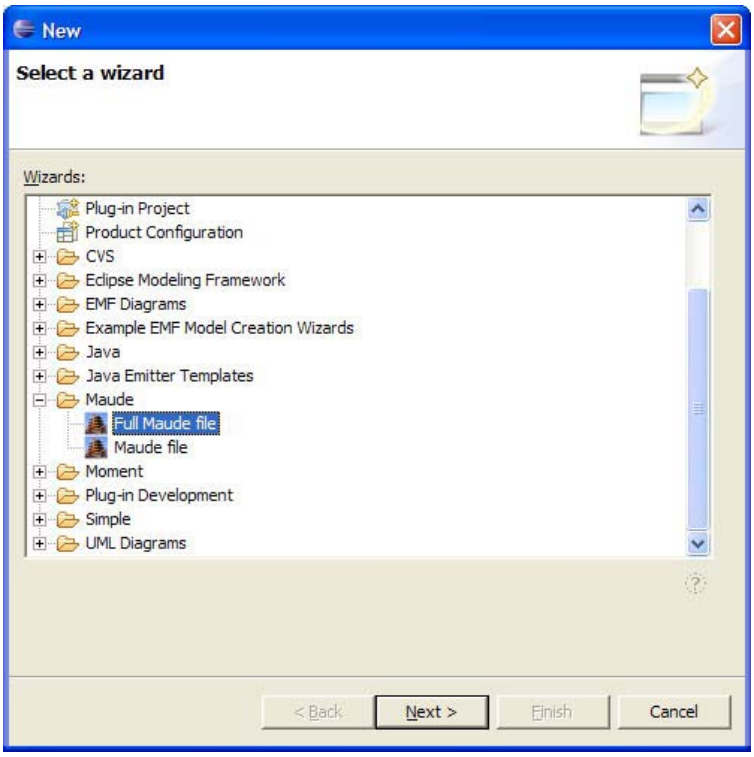

Figura 11: Asistente para nuevo documento de Eclipse.

Si seleccionamos generar un nuevo fichero de Full Maude, por ejemplo, aparecerá una nueva ventana que nos hará seleccionar en qué «contenedor» del espacio de trabajo deseamos salvar el fichero, así como que especifiquemos su nombre, como se muestra a continuación en la Figura 12.

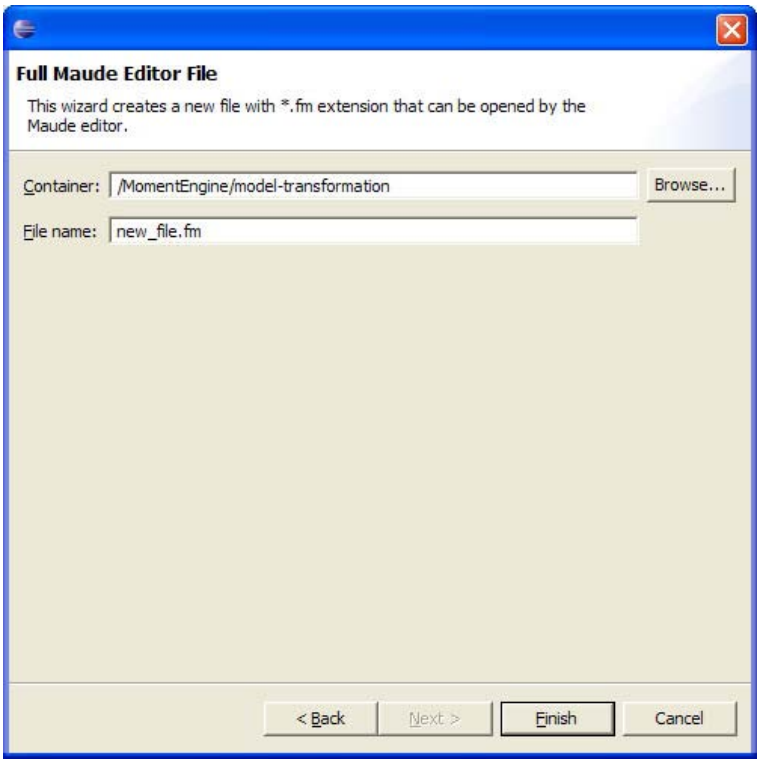

Figura 12: Asistente de creación de un nuevo fichero Full Maude.

Cuado pulsemos el botón «*Finish*», el asistente terminará, y se abrirá un documento en blanco en el editor de Maude.

c) El editor.

Una vez hemos creado en disco un fichero de Maude, el editor será el elemento que nos permita actuar con Maude de forma interactiva. Consta de una ventana con un editor de texto convencional, una barra de herramientas, y un menú en la barra de menús.

El editor (Figura 13) nos permite escribir código Maude, coloreando las palabras clave del lenguaje Maude. Además, las teclas de acceso rápido de los editores de Eclipse (por ejemplo, CTRL+ALT+↑) están completamente operativas para facilitar la escritura de programas.

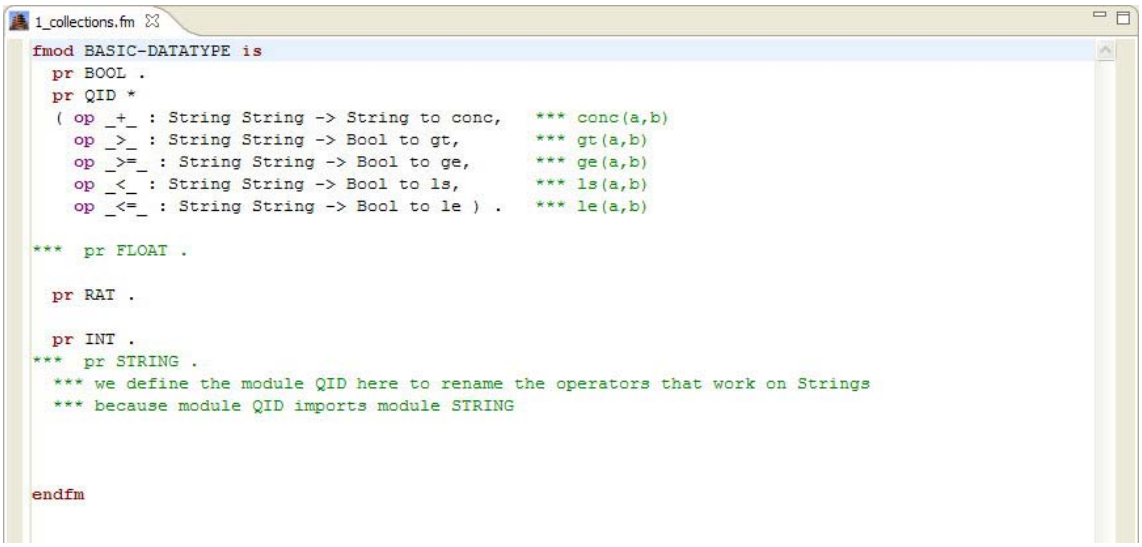

Figura 13: Vista del editor de Maude.

La interfaz que se proporciona al usuario para interactuar con el proceso Maude son el menú y la barra de herramientas. Este menú nos permite iniciar un proceso Maude (tal y como esté especificado en las preferencias del Maude Daemon), detenerlo, matarlo, reiniciar el loop de Full Maude, o activar el modo traza.

Pero sin duda, la opción más importante, es la que permite enviar el contenido del editor activo a Maude, para que lo procese.

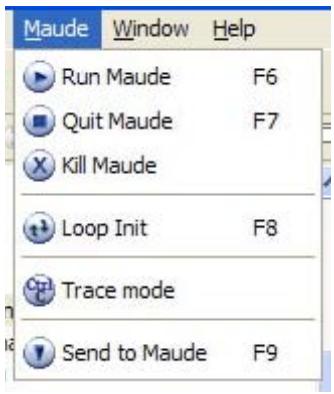

Figura 14: Menú para el control de Maude.

Como se puede observar en la Figura 14, se han habilitado accesos rápidos de teclado para las acciones más habituales. En la Figura 15 se muestra el aspecto de la barra de herramientas.

Figura 15: Barra de herramientas de control de Maude.

d) La consola de Maude.

La consola de Maude es el elemento que nos permite ver la respuesta devuelta por Maude. Muestra los mensajes en dos colores diferentes: negro y rojo. El *prompt* y las respuestas por la salida estándar en negro, y los mensajes de error (salida de error), en color rojo.

A continuación, la Figura 16 muestra un ejemplo de ejecución. En primer lugar, se ha creado el fichero «*trivial.maude*» con la ayuda del asistente, y una vez abierto en el editor, se ha escrito el comando trivial «*red true .*». Seguidamente se ha iniciado el proceso Maude (F6 o), y ha aparecido por la consola el *banner* de bienvenida. Por último, se ha enviado el comando para que se ejecute en Maude, pulsando F9 o .

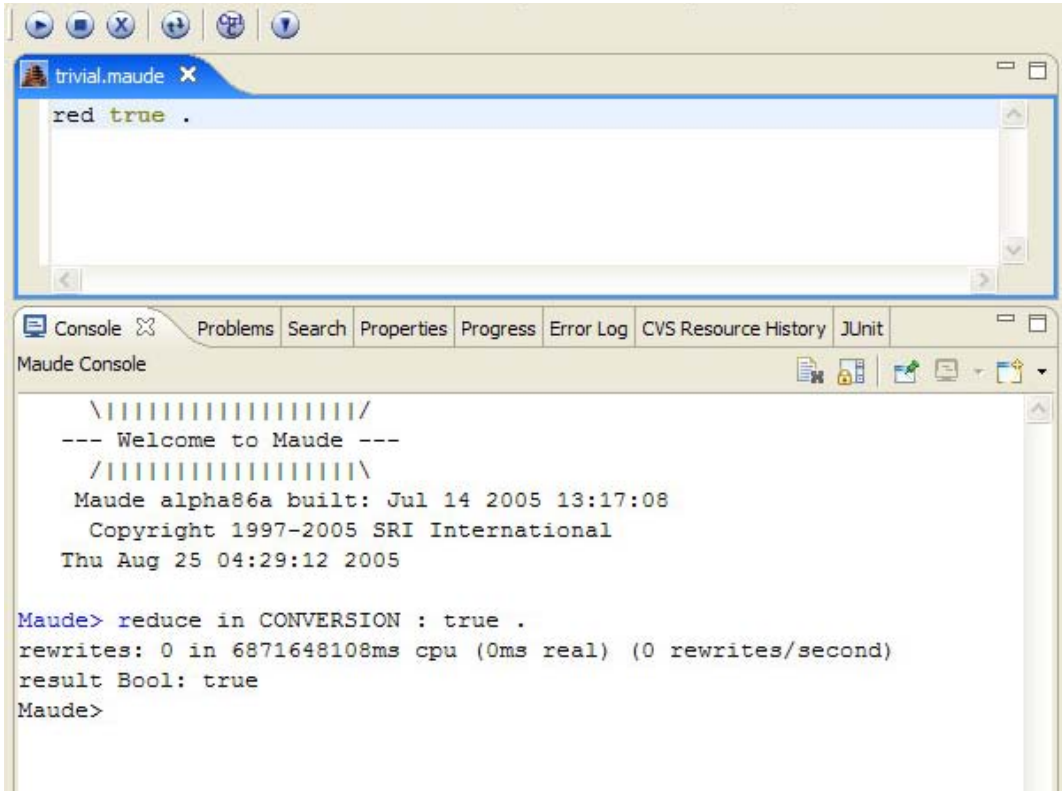

Figura 16: Ejemplo de comando enviado a Maude con el editor.

e) Envío directo de ficheros.

El envío directo de ficheros es una opción útil cuando se desea mandar numerosos ficheros a la vez a Maude. Bastará con seleccionarlos de la vista de carpetas de Eclipse, y pulsando el botón derecho del ratón sobre ellos, seleccionar «*Send to Maude*», del menú «*Maude*». Los archivos se enviarán a Maude en orden

alfabético, por lo que una buena opción es prefijarlos con un número, marcando el orden en que deben ser cargados.

Suponiendo que Maude ya está en ejecución (sino lanzará una advertencia de que no lo está), realizar esta acción sobre el archivo «trivial.maude» producirá el mismo efecto que ejecutar el comando «*send to Maude*» del editor. La Figura 17 nos muestra el menú contextual para el envío directo de ficheros.

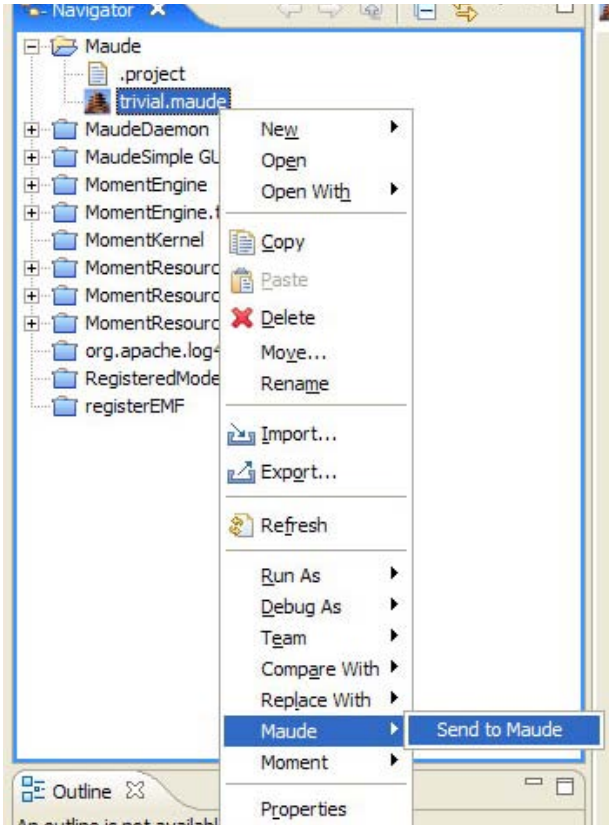

Figura 17: Menú contextual para el envío directo de ficheros a Maude.

4.1.2.3. Descripción detallada de paquetes y clases.

i. es.upv.dsic.issi.moment.maudesimpleGUI

a) Clases

a.1) MaudesimpleGUIPlugin

Esta es la clase principal de plug-in. Es generada automáticamente por el asistente de creación de un nuevo plug-in de Eclipse.

- **public MaudesimpleGUIPlugin()**

Este es el método constructor, inicializa el campo «instancia compartida» (el propio plug-in).

- **public static MaudesimpleGUIPlugin getDefault()**

Devuelve la instancia compartida.

- **public void start(BundleContext context)**

Este método se llama al iniciar el plug-in. En él se crea una instancia de la clase Maude (*es.upv.dsic.issi.moment.maudesimpleGUI.core.Maude*) que será la que se utilice con el editor.

- **public void stop(BundleContext context)**

Este método se ejecuta al finalizar ele ejecución del plug-in. En él se detienen y liberan algunos recursos. También, se detiene el proceso Maude si éste no estaba detenido, para que se salve a disco el *buffer* del fichero de registro.

- **public Maude getMaude()**

Devuelve la instancia de Maude que se ha creado para poder tener acceso a ella.

- **public ResourceBundle getResourceBundle()**

Devuelve el conjunto de recursos del plug-in.

- **public static String getResourceString(String key)**

Devuelve la cadena del paquete de recursos del plug-in o «key» si no se encuentra.

a.2) Messages

Esta clase encapsula el acceso al fichero de recursos (messages.properties) donde se almacenan las cadenas de texto utilizadas en la interfaz del usuario del plug-in. Esta estructura facilita la traducción del plug-in a diferentes idiomas.

- **private Messages()**

Éste es el constructor de la clase.

- **public static String getString(String key)**

Devuelve la cadena correspondiente a la clave *key*.

ii. es.upv.dsic.issi.moment.maudesimpleGUI.core

a) Clases

a.1) Maude

Esta es la clase que encapsula la interacción con el proceso Maude. Proporciona una interfaz muy similar a *MaudeProcessInteract*. En ella se incluye la implementación del hilo de ejecución encargado de la lectura de la salida estándar y la salida de error del proceso, y su impresión por la consola de Maude.

- **public Maude()**

Este es el constructor por defecto. En él se crea la instancia que se utilizará de *MaudeProcessInteract*. También se crea, y se inicia el hilo encargado de la lectura de los resultados de Maude para evitar que el proceso se bloquee.

- **public MaudeProcessInteract getMaudeProcess()**

Devuelve la instancia de *MaudeProcessInteract* por si es necesario acceder a algún método no implementado en la clase *Maude*.

- **public boolean isRunning()**

Realiza la correspondiente llamada a *maude.isRunning()* (*maude* es la instancia de *MaudeProcessInteract*). Devuelve *true* si Maude está en ejecución, y *false* en caso contrario.

- **public void killMaude()**

Realiza la llamada al método homónimo del objeto «*maude*». Finaliza de forma forzosa la ejecución de Maude.

- **public void makeCtrlC()**

Realiza la llamada al método homónimo del objeto «*maude*». Provoca el envío de la interrupción *SIGINT* al proceso, haciéndolo entrar en el modo traza.

- **public void quitMaude()**

Realiza la llamada al método *quitMaude()* del objeto «*maude*», provocando que Maude finalice de forma voluntaria.

- **public String readErrMaude(int max)**

Realiza una lectura no bloqueante sobre la salida de error de Maude hasta un máximo de «*max*» caracteres.

- **public String readOutMaude(int max)**

Realiza una lectura no bloqueante sobre la salida estándar de Maude hasta un máximo de «*max*» caracteres.

- **public void runMaude()**

Este método provoca la creación de la consola de Maude, así como de que se muestre automáticamente. Tras esto, realiza una llamada al método *maude.execMaude()*.

- **public void sendToMaude(String txt)**

La invocación de este método provoca que se envíe el texto «*txt*» a la entrada estándar del proceso Maude.

iii. es.upv.dsic.issi.moment.maudesimpleGUI.ui.editors

En este paquete se incluyen todas las clases que permiten crear el editor del plug-in, que nos permitirá desarrollar programas en Maude, con la ayuda del coloreado de sintaxis y completado de texto, aunque dada la sencilla sintaxis del lenguaje Maude. Esta última característica está deshabilitada en el editor final, aunque el código para conseguir esa funcionalidad está implementado. Para ver más detalladamente el proceso de creación de un editor se puede consultar el artículo de Elwin Ho [Ho03].

a) Clases

a.1) MaudeEditor

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

- **public MaudeEditor()**

Este es el constructor por defecto de la clase. En él se ejecuta el constructor de la clase padre, y se configura el editor para que incluya las funcionalidades adicionales implementadas (coloreado de sintaxis, completado de palabras…).

a.2) MaudeCompletionProcessor

Esta clase nos permite gestionar las capacidades de completado de texto del editor. Se pueden especificar las palabras que se propondrán cuando se invoque esta función de completado.

- **public MaudeCompletionProcessor()**

Este es el constructor por defecto de la clase.

- **public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int documentOffset)**

Éste es el método que se invocará para devolver toda la lista de palabras propuestas cuando el usuario vaya a utilizar el función de completado. Actualmente su código está comentado, y no devuelve ninguna propuesta.

- **public IContextInformation[] computeContextInformation(ITextViewer viewer, int documentOffset)**

Devuelve la información sobre posibles contextos basándose en la posición especificada en el documento correspondiente a la posición actual del cursor.

- **public char[] getCompletionProposalAutoActivationCharacters()**

Este método devuelve los caracteres que cuando sean pulsados por el usuario provocarán que se muestre automáticamente la lista de palabras propuestas.

- **public char[] getContextInformationAutoActivationCharacters()**

Este método devuelve los caracteres que cuando sean pulsados por el usuario provocarán que se muestre automáticamente la la información de contexto.

- **public IContextInformationValidator getContextInformationValidator()**

Devuelve un validador usado para determinar cuando la información de contexto debe ser desechada.

- **public String getErrorMessage()**

Devuelve la razón de porqué este asistente de contenido fue incapaz de devolver información de completado o de contexto.

a.3) MaudeEditorContributor

Esta clase es la encargada de instalar y desinstalar las acciones globales del editor. Será la que contribuirá a la interfaz con las barras de herramientas y los menús que aportarán al usuario la capacidad de interactuar con Maude, y mandarle el contenido de los editores.

- **public MaudeEditorContributor()**

Este es el constructor por defecto. Realiza la correspondiente llamada al constructor de la clase padre, *org.eclipse.ui.part.EditorActionBarContributor*, y ordena la creación de las acciones mediante la llamada al método *createActions()*.

- **private void createActions()**

Este método es sin duda el más importante de esta clase, ya que en él se especifican todas las acciones que podrán realizarse. Para cada acción se especifica su nombre, *tooltip*, iconos, tecla de acceso rápido, y órdenes a ejecutar. Las acciones implementadas son:

- **private Action actRunMaude.**

Esta acción implementa los pasos necesarios para ejecutar Maude. Si Maude no está en ejecución, realiza una llamada al método «*runMaude()*», del objeto «*maude*», creado en la clase «*MaudesimpleGUIPlugin*».

- **private Action actSendMaude.**

Esta acción envía el texto del editor activo a Maude si está en ejecución. Sino, mostrará un mensaje de advertencia, indicando que no hay ninguna instancia de Maude en ejecución.

- **private Action actLoopInitMaude.**

La acción *actLoopInitMaude* comprueba que hay en ejecución una instancia de Maude funcionando en modo Full Maude, y en ese caso, ejecuta el método *loopInit()*. Si no se da esta circunstancia, mostrará el pertinente mensaje de advertencia.

- **private Action actCtrlCMaude.**

Al igual que las acciones anteriores, ésta comprobará si existe una instancia de Maude en ejecución, y, en ese caso, mandará una señal *SIGINT* para afctivar el modo traza mediante el método *makeCtrlC()*.

- **private Action actQuitMaude.**

Esta acción provoca que Maude termine de forma voluntaria mediante la invocación del método quitMaude().

- **private Action actKillMaude.**

La acción *actKillMaude* provoca que Maude finalice de forma forzosa mediante la invocación del método *killMaude()*.

- **public void contributeToMenu(IMenuManager manager)**

Éste método se invoca para contribuir a la barra de menús con un nuevo menú. En su cuerpo se deben especificar qué acciones deben añadirse a éste.

- **public void contributeToToolBar(IToolBarManager manager)**

Éste método se invoca para contribuir a interfaz de usuario con una nueva barra de herramientas. En su cuerpo se deben especificar qué acciones deben añadirse a la barra.

- **protected IAction getAction(ITextEditor editor, String actionID)**

Devuelve la acción registrada con el identificador «*actionID*» en el editor «*editor*».

- **public void setActiveEditor(IEditorPart part)**

Este método establece el editor activo. Igualmente, añade las acciones comunes de un editor de texto (copiar, cortar, pegar, etc.).

a.4) MaudeRuleScanner

En la clase *MaudeRuleScanner*, se establecen las reglas para el formateado del texto del editor. Esto es lo que nos permitirá establecer diferentes colores y formatos para palabras clave, comentarios, etc. Hereda de la clase *org.eclipse.jface.text.rules.RuleBasedScanner*.

- **public MaudeRuleScanner()**

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

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

a.5) MaudeSourceViewerConfig

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

- **public MaudeSourceViewerConfig()**

Este es el constructor por defecto.

- **public IContentAssistant getContentAssistant(ISourceViewer sourceViewer)**

Este método devuelve un asistente de contenido. Lo crea, y establece algunos parámetros de configuración como la activación automática, o el retardo de aparición.

- **public IPresentationReconciler getPresentationReconciler(ISourceViewer sourceViewer)**

Devuelve el «reconciliador» de la presentación. Cada vez que el usuario cambia el documento, el «reconciliador» determina qué región de la presentación debe ser invalidada y como deber ser reparada.

Un daño es el texto que debe ser redibujado, y la reparación es el método utilizado para redibujar el área dañada. El proceso de mantener la presentación visual de un documento a medida que se realizan los cambios se reconoce como «reconciliado».

- **protected MaudeRuleScanner getTagScanner()**

Este método devuelve el *scanner* que define los atributos de texto según las reglas de coloreado. En caso de que no se haya creado todavía ninguna instancia de *MaudeRuleScanner*, la crea en este momento.

iv. es.upv.dsic.issi.moment.maudesimpleGUI.ui.popup.actions

Este paquete es creado mediante el asistente para creación de menús contextuales de la vista de carpetas. Permite crear acciones para realizar sobre ficheros.En este caso, nos permitirá procesar el contenido de uno o varios ficheros de Maude sin necesidad de ser abiertos en el editor. Esta característica es especialmente útil cuando se desean enviar numerosos ficheros a Maude, como por ejemplo, cuando se carga manualmente el *kernel* de MOMENT.

a) Clases

a.1) SendToMaude

Ésta es la única clase de que consta el paquete, y define el mecanismo por el cual se leerán los ficheros, y se enviarán a Maude. Implementa la interfaz *org.eclipse.ui.IObjectActionDelegate*.

- **public SendToMaude()**

Este es el constructor por defecto.

- **public void run(IAction action)**

Éste es el método principal de la clase, ya que en él se indica cómo deberán ser enviados los ficheros a Maude. Dado un conjunto de ficheros seleccionados en la vista de archivos de Eclipse, este método los abrirá por orden alfabético e irá enviando su contenido al proceso Maude activo.

En caso de que no haya ningún proceso activo, mostrará el correspondiente mensaje de advertencia.

- **public void selectionChanged(IAction action, ISelection selection)**

Este método es invocado cada vez que la selección en la vista de archivos varía, actualizando el campo «*sel*» de la clase, donde se almacena siempre el conjunto actualizado de ficheros.

v. es.upv.dsic.issi.moment.maudesimpleGUI.ui.toconsole

El paquete «*toconsole*» contiene la definición de la consola de Maude, por la cual se irá mostrando toda la respuesta de Maude, como si se hubiera ejecutado de forma normal.

a) Clases

a.1) ConsoleOutputView

Esta es la única clase que consta este paquete, e implementa la consola de Maude.

- **public ConsoleOutputView(String consoleName)**

Este es el constructor por defecto de la clase. En él se crea la consola, y los diferentes streams (negro, azul y rojo), que ayudarán a la lectura de los mensajes de la consola.

- **private MessageConsole findConsole(String name)**

Devuelve la consola cuyo nombre corresponda con «*name*».

- **public MessageConsoleStream getBlackStream()**

Devuelve el *stream* negro. En él se escribirá la salida normal de Maude.

- **public MessageConsoleStream getBlueStream()**

Devuelve el *stream* azul de la consola.

- **public MessageConsoleStream getRedStream()**

Devuelve el *stream* rojo de la consola. Este *stream* está pensado para escribir en él los mensajes de la salida de error de Maude.

- **public void showConsole()**

La invocación de este método provocará que se haga visible en el *Workbench* de Eclipse la vista de consola de Maude.

vi. es.upv.dsic.issi.moment.maudesimpleGUI.ui.wizards

El paquete *ui.wizards* incluye todas las clases que implementan los asistentes para crear nuevos ficheros de Maude y Full Maude. Estos asistentes han sido creados a partir del asistente sencillo de ejemplo incluido en el *Plug-in Development Environment* de Eclipse, que se puede generar de forma automática.

vii. Clases

a) FullMaudeNewWizard

Esta es la clase principal que se utilizará para ejecutar el asistente para generar un nuevo fichero de Full Maude. Hereda de la clase *org.eclipse.jface.wizard.Wizard*, e implementa la interfaz *org.eclipse.ui.INewWizard*.

- **public FullMaudeNewWizard()**

Este es el constructor por defecto. Invoca al constructor de la clase padre, y establece que se utilizará una barra de progreso.

- **public void addPages()**

Mediante el método *addPages()* se añaden las diferentes ventanas que formarán el asistente. En este caso, creará una instancia de la clase *FullMaudeNewWizardPage*, y la añadirá mediante el método *addPage()* de la clase padre.

- **private void doFinish(String containerName, String fileName, IProgressMonitor monitor)**

Éste es el método principal de la clase. Es donde se realiza todo el trabajo. Se crea el fichero en la ruta especificada y se fija su contenido inicial.

- **public void init(IWorkbench workbench, IStructuredSelection selection)**

Este método es invocado al crear el asistente. Recibe un conjunto de selección y se almacena en la clase. Ésta selección será utilizada posteriormente para saber cual es la carpeta activa, y se incializarán los diálogos según estos datos.

- **private InputStream openContentStream()**

Consultando este método obtendremos cual deberá ser el contenido inicial del fichero. En este caso, devuelve un *stream* vacío.

- **public boolean performFinish()**

Este método es llamado cuando se pulsa el botón «*Finish*» del asistente. En él suelen leerse los datos fijados en las distintas páginas del asistente, para así disponer de ellos en el momento en el que se ejecute el método *doFinish()*.

- **private void throwCoreException(String message)**

Este método lanza una excepción «*CoreException*» identificando que el plug-in que causó el error fue *es.upv.dsic.issi.moment.maudesimpleGUI*, e indicando el mensaje de error «*message*».

b) FullMaudeNewWizardPage

Esta clase implementa la única página de que consta el asistente de creación de nuevo fichero. Permite establecer el contenedor para el Nuevo archive, así como su nombre. Hereda de la clase o*rg.eclipse.jface.wizard. WizardPage*.

- **public FullMaudeNewWizardPage(ISelection selection)**

Éste es el constructor por defecto de la clase. Ejecuta el constructor de la clase padre, y configura algunos aspectos, como la selección, el título del asistente o su descripción.

- **public void createControl(Composite parent)**

El método *createControl* crea los controles de que constará la página, esto son, las diferentes cajas de texto o botones, así como su contenido inicial.

- **private void dialogChanged()**

Éste es el método encargado de la validación de los campos de la página. Si algún campo no cumple alguna regla, deberá invocarse a *updateStatus(String message)*, indicando en el mensaje la regla violada.

Si no se detecta ningún fallo, debe ejecutarse *updateStatus(null).*

- **public String getContainerName()**

Devuelve el nombre del contenedor donde se guardará el fichero.

- **public String getFileName()**

Devuelve el nombre que tendrá el fichero a crear.

- **private void handleBrowse()**

Este método crea un diálogo de selección de contenedor estándar para elegir un valor para el campo contenedor. Una vez se ha seleccionado y aceptado un contenedor en el diálogo, actualiza el campo editable de la página donde se indica el contenedor.

- **private void initialize()**

Comprueba si la actual selección del *Workbench* es un contenedor válido donde salvar el nuevo archivo.

- **private void updateStatus(String message)**

Este método establece el mensaje de error que indicará al usuario qué campos son inválidos. Cuando no haya campos incorrectos, se realizará una llamada a *setPageComplete(bolean complete)* de la clase padre, que establece si se puede avanzar/terminar el asistente.

c) MaudeNewWizard

Esta clase está creada con el asistente del PDE de Eclipse de igual forma que *FullMaudeNewWizard*, por lo que son aplicables los mismo métodos y descripciones que a la anterior.

d) MaudeNewWizardPage

Al igual que la clase anterior, también fue creada con al asistente del PDE de Eclipse. Sus métodos y forma de funcionamiento es, por tanto, similar que la clase *FullMaudeNewWizardPage*.

4.2. Trabajos relacionados: Maude Workstation.

El entorno de programación que se proporciona desde el proyecto Maude para su lenguaje es sin duda su principal debilidad, ya que se reduce a un terminal donde se escribirán los comandos.

Se han realizado diversos esfuerzos para mejorar la productividad en el desarrollo de programas en Maude, por ejemplo, existen módulos para integrar el uso de Maude en XEmacs, como los desarrollados por Kai Brünnler, Ellef Gjelstad [Brü05]. La desventaja de estas opciones es que no todo el mundo está familiarizado con los sistemas Linux y más aún, con un completo conocimiento de XEmacs.

Otras aproximaciones han pretendido proporcionar un entorno de desarrollo para Maude mucho más sencillo, y orientado a ejecutarse en sistemas de ventanas. Uno de estos intentos es Maude Workstation, implementado Alfredo Muñoz [Muñoz04].

Maude Workstation guarda ciertas similitudes con las «*Herramientas de Desarrollo de Maude*» (*Maude Development Tools*). Por ejemplo, ambos entornos están desarrollados en Java, por lo que pueden ejecutarse sin realizar cambio alguno en el software en numerosos sistemas, al contrario que como puede ocurrir con *XEmacs* con el modo Maude.

Por otra parte, tanto *Maude Workstation* como *Maude Development Tools* se ejecutan en sistemas Windows mediante el uso de *CygWin*, lanzando el proceso en Java, y capturando los *buffers* de entrada y salida.

Igualmente, ambos entornos proporcionan la interacción con Maude con los mismos comandos (iniciar Maude, detener Maude, matar el proceso, etc.).

A pesar de estas similitudes, también existen numerosas diferencias. Obviando que el editor Maude SimpleGUI es un editor simple, y que por consiguiente, no proporciona toda la funcionalidad que sí lo hace *Maude Workstation*, (una interfaz de usuario elaborada y con numerosas opciones, exploración del grafo módulos cargados, amplio soporte gráfico para modo traza, etc.), existen numerosas razones por las que es preferible el uso del plug-in de Eclipse.

En primer lugar el diseño modular de las *Maude Development Tools*, que diferencia en dos plug-ins independientes la API al programador, y el editor del usuario, permite una mayor mantenibilidad de código, que resultará en un producto de mayor calidad y extensible.

Por otra parte, esto aporta la ventaja de que cualquier otro plug-in que requiera interactuar con Maude ya dispondrá de una interfaz para hacerlo, cosa que Maude Workstation no proporciona, ya que la implementación de cómo se invoca a Maude no está encapsulada como un módulo independiente, sino que se entremezcla con cuestiones de interfaz de usuario.

Pero, al margen de cuestiones de reutilización del código, la principal ventaja obtenida con el editor Maude SimpleGUI, es la eficiencia. Realizando diversas pruebas en un Pentium 4 Hyper-Threading a 3GHz y 1 GB de memoria RAM, se ha comprobado que el envío de un único fichero de Maude (de aproximadamente 45KB) con la especificación de diversos módulos para cargar, tarda aproximadamente 15 segundos con Maude Workstation, siento el tiempo de carga inapreciable (menor que el segundo), con el plug-in desarrollado para Eclipse.

La imprecisión de estos datos se debe a que pruebas con ficheros mayores, (de los 50KB los 150 KB) han provocado que *Maude Workstation* deje de responder, mientras que con el editor de Eclipse se ha mantenido los tiempos de carga prácticamente inalterados, manteniéndose por debajo del segundo.

Por otra parte, en un entorno Windows, las *Maude Development Tools*¸ son mucho más flexibles y sencillas de instalar que *Maude Workstation*. Los plug-ins de Eclipse no tienen ninguna restricción sobre en qué unidades se encuentren los diferentes ejecutables y librerías necesarios para ejecutar; a diferencia de lo que ocurre con *Maude Workstation*. Esta facilidad de instalación, además, se ve incrementada con el instalador «*Maude for Windows*» que se ha creado en este proyecto.

Otra ventaja del uso de nuestra herramienta, es que gran número de sus componentes son extensiones sobre clases proporcionadas por la API de Eclipse. Las *Maude Development Tools* son mucho más robustas, y con menos *bugs* que *Maude Workstation* (por ejemplo, aquel que introduce numerosas líneas en blanco a lo largo de un fichero al copiar de un editor a otro).

A parte de todas estas cuestiones, no hemos de olvidar que Eclipse es sin duda, uno de los entornos de programación más extendidos que existen, con infinidad de desarrolladores trabajando en su mejora. Por lo que, tanto los posibles usuarios finales de este entorno, como los potenciales desarrolladores que lo mejoren, pueden ser numerosísimos. Así como que se mejora ampliamente la interoperabilidad entre Maude y otras tecnologías, como es el caso por ejemplo de la herramienta MOMENT.

Parte Tercera:

Soporte para trazabilidad en una Herramienta de gestión de Modelos.

5. SOPORTE PARA TRAZABILIDAD EN MOMENT.

5.1. Introducción.

La trazabilidad es una cuestión clave en entornos donde hay una cadena de procesos. Es estos casos, la información sobre cada paso de la cadena debe ser almacenada para poder ser consultada posteriormente. Por ejemplo, en el caso de la industria de la automoción, la trazabilidad hace posible la retirada de los vehículos del mercado en caso de fallos; en la industria alimentaria, contribuye a la seguridad de los alimentos; o, en el campo de la ingeniería del software, proporciona soporte para la validación de requisitos e incrementa la calidad del proceso de desarrollo de software.

En cualquier escenario en el campo de la Ingeniería del Software existe una manipulación de un artefacto software. La capacidad de describir y consultar las operaciones de manipulación que se han realizado sobre un determinado artefacto puede ser relevante para otras tareas relacionadas. Sin embargo, la trazabilidad aún hoy, a menudo permanece en un segundo plano cuando se resuelven problemas en la Ingeniería del Software, y pocas son las herramientas que proporcionan un soporte completo para ésta.

Como ya se ha indicado, en la iniciativa MDA, un artefacto software es visto como un modelo. Tareas típicas, como producción de código, integración de aplicaciones o interoperabilidad, son realizadas directamente con modelos. Esto permite al usuario trabajar a nivel conceptual, haciendo más fácil la identificación de los elementos necesarios para automatizar estas tareas. Estas labores son usuales en diversos escenarios y son generalmente resueltas de manera ad-hoc.

En la disciplina de la Gestión de Modelos, los modelos se consideran como ciudadanos de primer orden, y se proporcionan una serie de operadores genéricos ser aplicados sobre ellos: *Merge*, *Diff*, *ModelGen*, etc. Estos operadores, proporcionan una solución reusable a las tareas descritas anteriormente, de forma que el usuario manipula directamente los modelos, en lugar de trabajar con la representación interna de éstos a nivel de programación. Diversas aproximaciones a esta disciplina especifican operadores que están basados en *mappings* para manipular modelos. Un *mapping* es una relación entre un elemento de un modelo dominio y un elemento de un modelo rango que indica que ambos representan el mismo elemento en diferentes modelos. Esto significa, que los *mappings* entre dos modelos deben ser explícitamente definidos para poder aplicarles un operador.

En la herramienta de Gestión de Modelos MOMENT, las relaciones entre dos modelos se representan de forma implícita por medio de un morfismo de equivalencia que se define entre dos modelos desde un punto de vista más abstracto y reusable.

Sin embargo, los mappings explícitos entre dos modelos son también útiles cuando no existe ninguna definición de este morfismo entre dos metamodelos.

5.1.1. El problema de la trazabilidad en la Ingeniería de Requisitos.

En Ingeniería de requisitos, la Guía para la Especificación de Requisitos Software de IEEE [Dorf90] indica que una especificación de requisitos software es trazable si el origen de cada requerimiento es claro y si facilita la referencia de cada requisito en desarrollos futuros o mejoras de la documentación.

Basados en esta definición, Gotel y Finkelstein [Got94] describieron como trazabilidad de los requisitos cono la capacidad de describir y seguir la vida de un requerimiento, tanto hacia delante como hacia atrás, a los largo de todos los pasos de refinamiento en el proceso de desarrollo software. Consultas de este tipo permiten al usuario conocer qué refinamientos se han aplicado a un requerimiento (hacia adelante), y permiten la identificación de un requisito a partir de un artefacto software más específico, como por ejemplo, código (en la dirección inversa). Por lo tanto, la trazabilidad puede usarse para la validación de requisitos y para proporcionar soporte al mantenimiento del software. A su vez, también proporciona aprovechamiento económico, puesto que detalla cómo ha sido desarrollado el sistema y evitar el desarrollo redundante de ciertas partes.

Para conseguir un soporte para trazabilidad en el proceso de desarrollo software diversas tareas han de tenerse en cuenta [Ram01]:

1. *Definición de la traza*, para indicar los tipos de objetos de nuestro sistema que pueden ser trazados, y qué información se va a definir en una traza.

2. *Producción de la traza*, para indicar qué actividades, acciones, decisiones y eventos ocurridos a lo largo del desarrollo software generan trazas para un uso futuro.

3. *Extracción de la traza*, para indicar cómo las trazas generadas en la tarea anterior, pueden ser consultada para conseguir ciertos objetivos, como por ejemplo, validación de requisitos, o mantenimiento de software.

4. *Verificación de la traza*, para mantener la integridad del conjunto de objetos y trazas.

Existen diversas herramientas que proporcionan soporte para la trazabilidad de requerimiento [Vol]. Para proporcionar una gestión eficiente de la trazabilidad, estas herramientas deben resolver ciertos problemas que están presentes en el marco industrial: (a) la falta de unas pautas comunes que describan cómo definir un modelo de trazabilidad mediante un metamodelo bien definido, y como usarlo; como por ejemplo lo hace el estándar UML para el modelado orientado a objetos. Esto se debe a la naturaleza variable del uso y captura de la trazabilidad [Ram01], que varía de una organización a otra. (b) El conflicto entre usuarios finales y el sistema establecido [Got94], donde los proveedores de las trazas y los usuarios tienen diferentes objetivos y prioridades. (c) El uso de herramientas heterogéneas para definir y manipular los artefactos software implicados en el proceso de desarrollo

software. De esta manera, la cuestión de la interoperabilidad se presenta como una importante característica a tener en cuenta en una herramienta de gestión de la trazabilidad.

Por otra parte, estas herramientas están también implementadas de manera *ad-hoc* para el problema de la trazabilidad de requisitos, sin tener en cuenta que la misma funcionalidad puede ser utilizada en otros contextos.

5.1.2. El framework MOMENT.

La iniciativa MDA de OMG consiste en una familia de estándares que indican cómo definir y usar modelos para desarrollar aplicaciones software en el marco de MDE. La integración de aplicaciones y la interoperabilidad son dos objetivos de esta iniciativa, como se indica en la petición de proposiciones para el nuevo estándar Query/Views/Transformations [QVT-RFP]. No obstante, para conseguir la interoperabilidad entre aplicaciones los puentes construidos entre ellas son todavía ad-hoc.

Como ya se ha comentado anteriormente, se ha desarrollado el framework MOMENT, que proporciona una serie de operadores genéricos para manejar modelos por medio del *Eclipse Modeling Framework* (EMF).

5.1.3. Caso de estudio: propagación de cambios.

En este caso de estudio, utilizaremos un escenario de propagación de cambios que se asemeja al introducido en [Mel03]. Lo ilustramos mediante un ejemplo específico basado en el modelo de *Purchase Order* utilizado en [Bud03]. En este texto se presenta inicialmente un modelo UML simplificado (véase la Figura 26) para una aplicación que modele una orden de compra.

Para construir una nueva aplicación que almacene la información en una base de datos relacional, reusaremos la metainformación que describe el diagrama UML. Aplicando un mecanismo de transformación (paso 1), obtenemos la nueva base de datos relacional (RDB). El mecanismo de transformación también genera un conjunto de enlaces entre el nuevo esquema relacional generado (RDB) y el diagrama de clases origen para proporcionar soporte a la trazabilidad (mapUML2RDB).

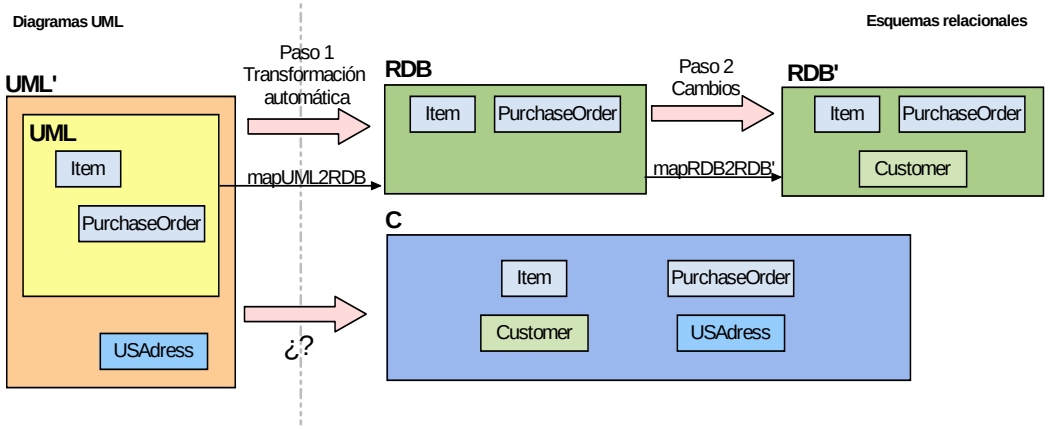

Figura 18: Ejemplo de propagación de cambios.

Tras obtener una base de datos relacional semáticamente equivalente al diagrama UML original, se continúa con el desarrollo del nuevo sistema. Esto puede implicar cambios en la aplicación y en la base de datos (paso 2), obteniendo el esquema relacional (RDB'). Estos cambios son trazados y almacenados por la herramienta que gestiona la manipulación del modelo, o directamente por el usuario (mapRDB2RDB').

Una vez se ha desarrollado el nuevo sistema, pueden producirse cambios en los requisitos del sistema, requiriendo modificaciones. Podemos tomar estas modificaciones, como la modificación realizada en [Bud03] sobre el modelo de la orden de compra (véase la Figura 30). Es más sencillo modificar el esquema UML que modificar la base de datos RDB. En este punto, la aplicación del mecanismo de transformación aplicado en el paso 1 descartaría los cambios aplicados de RDB a RDB'.

Una solución a este ejemplo de propagación de cambios puede ser realizado usando operadores de gestión de modelos. En esta aproximación, los enlaces de trazabilidad son utilizados para automatizar la propagación de cambios aplicada al esquema de la base de datos relacional para obtener el nuevo sistema C.

5.2. Soporte para trazabilidad en Gestión de Modelos.

Todas las definiciones de trazabilidad de los requerimientos comentadas en el punto 5.1.1 tienen una característica en común: una traza proporciona información acerca de una tarea que ha sido realizada en un artefacto software fuente en un proceso de desarrollo software, y lo relacione con el artefacto software resultante. El soporte para trazabilidad debe proporcionar dos mecanismos, en primer lugar, aquel que nos permite definir enlaces de trazabilidad y la funcionalidad de consulta que nos permite navegarlos. En esta sección definiremos trazabilidad en MDA desde el punto de vista de la gestión de modelos y se presentarán un conjunto de operadores que proporcionan soporte para trazabilidad en el framework MOMENT.

5.2.1. Gestión genérica de la trazabilidad.

Un proceso MDA consiste en una secuencia de operaciones realizadas sobre un conjunto de modelos. Estos modelos conforman un metamodelos y representan artefactos software específicos. Operaciones como integración o transformación de modelos pueden ser soportadas directamente por operadores de gestión de modelos simples. Otras operaciones, como la propagación de cambios, pueden ser especificadas como un operador complejo construido a base de otros operadores.

Cada operador simple realiza una manipulación sobre un conjunto de modelos de entrada. Para conseguir esto, el operador invoca una función que se encuentra definida a nivel de metamodelo. La semántica de esta función está definida axiomáticamente en lógica ecuacional, y cada uno de estos axiomas se denomina regla de manipulación. Para registrar la tarea realizada sobre un modelo, cada operador produce automáticamente un conjunto de enlaces entre los elementos del

modelo origen, y los del modelo resultante. Estos links se almacenan como modelos y son usados para proporcionar soporte para la trazabilidad.

Siguiendo la aproximación de gestión de modelos, definimos *Gestión genérica de la trazabilidad* como dos cuestiones fundamentales:

1. La definición de un metamodelo de trazabilidad, para indicar la información necesaria para enlazar los elementos de dos modelos diferentes que pueden pertenecer a dos metamodelos distintos en un contexto específico. El detalle del metamodelo depende del cómo se entienda la gestión de la trazabilidad en una sociedad específica. Por ejemplo, un metamodelo genérico de trazabilidad puede ser descrito para el campo de la ingeniería de requisitos, aunque parece más factible definir un modelo de trazabilidad para cada organización, o incluso cada proyecto.

2. Proporcionar un mecanismo para extraer información de un modelo de trazabilidad independientemente del metamodelo utilizado. Este mecanismo se compone de dos tipos de operadores:

    a. Operadores de consulta que proporcionan navegación hacia delante y hacia atrás a través de un modelo de trazabilidad.

    b. Operadores de gestión de la trazabilidad para manipular los modelos de trazabilidad para poder automatizar el razonamiento sobre los enlaces creados. Por ejemplo, el operador *Compose* permite encadenar enlaces de trazabilidad para convertir enlaces implícitos en explícitos; y el operador *Match* permite la inferencia de modelos de trazabilidad entre dos modelos. Es más, un modelo de trazabilidad puede ser manipulado por los operadores de gestión de modelos.

5.2.1.1. Definición del metamodelo de trazabilidad.

Para definir el metamodelo de trazabilidad el usuario puede usar la notación UML. Este trabajo es hecho por el usuario para un contexto de trabajo específico. MOMENT, por supuesto, debe de proporcionar un mecanismo que permita al usuario definir un modelo de trazabilidad acorde a sus necesidades.

En el marco de MOMENT se ha definido un metamodelo de trazabilidad que básicamente proporciona los constructores necesarios para relacionar elementos de un modelo dominio con elementos de un modelo destino, independientemente de sus metamodelos.

El mecanismo que se proporciona para que el usuario defina su metamodelo personalizado, está basado en el mecanismo de herencia de Ecore. Así pues, el metamodelo de trazabilidad básico será empleado como base común de todos los metamodelos de trazabilidad, y sobre él se construye todo soporte de trazabilidad de MOMENT.

El usuario deberá, por otra parte, heredar su metamodelo personalizado de éste. De esta forma podrá aplicar todas las modificaciones que desee a su metamodelo, hasta que permita capturar toda la información de interés.

i. Metamodelo de trazabilidad básico.

La adopción de un metamodelo básico, como se ha comentado, permite disponer de un metamodelo sobre el que se puede construir todo el soporte para trazabilidad de MOMENT de forma genérica, sin depender del metamodelo definido por cada usuario (pero sin tener que renunciar a la flexibilidad que ello proporciona).

Como se comentará más adelante, en el punto 5.3.1, la aplicación de un operador de MOMENT genera automáticamente uno o varios modelos de trazabilidad que describen las correspondencias entre los modelos dominio y los modelos rango. El metamodelo de trazabilidad básico será el que conformarán éstos modelos generados automáticamente.

Teniendo en cuenta que mediante la aplicación del mecanismo de proyección definido entre el espacio tecnológico EMF, y el espacio tecnológico Maude, obtenemos la especificación algebraica del metamodelo de trazabilidad, podemos especificar modelos de trazabilidad como conjuntos de elementos de forma que los operadores de MOMENT pueden ser empleados para manipularlos (*Merge, Cross, ModelGen…*).

Esto implica que dado un metamodelo de trazabilidad personalizado, el modelo generado automáticamente al realizar una operación podrá ser portado al metamodelo del usuario mediante operadores de Gestión de Modelos.

Por otra parte, la adopción de un metamodelo de trazabilidad básico nos permitirá generar con la ayuda de EMF un editor diseñado específicamente para visualizar, crear y editar los modelos de trazabilidad.

En la Figura 19 se muestra la parte del framework de MOMENT concerniente al soporte para trazabilidad.

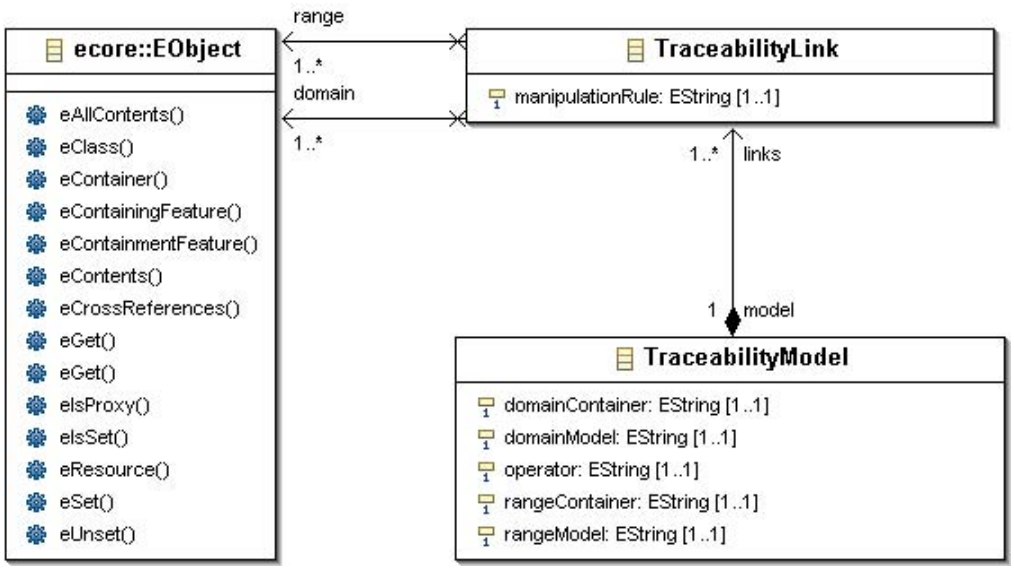

Figura 19: Metamodelo de trazabilidad básico del framework MOMENT.

En este metamodelo básico, la clase *TraceabilityModel* es el elemento raíz del paquete. Ésta nos permite definir modelos de trazabilidad. Una instancia de la clase *TraceabilityModel* contiene: información acerca del almacenamiento del modelo dominio y el modelo rango (*domainModel* y *rangeModel*); qué elemento de cada modelo es el raíz (mediante *domainContainer* y *rangeContainer*); el operador que se ha aplicado al modelo dominio; y los links que constituyen el modelo de trazabilidad.

La clase *TraceabilityLink* indica cómo definir una relación entre un conjunto de elementos del modelo dominio y un conjunto de elementos del modelo rango (mediante las referencias *domain* y *range*). Cada enlace está asociado al paso de la tarea de manipulación de modelos que lo ha producido (mediante el campo *manipulationRule*). En el metamodelo de la figura, el campo *Operator* representa un operador que se encuentra definido algebraicamente en MOMENT. El campo *manipulationRule* define la información necesaria para especificar un axioma para la función de manipulación usado por los operadores simples.

La clase *EObject*, por otra parte, se refiere a la clase org.eclipse.emf.ecore.EObject, que permite el acceso a cualquier elemento de un modelo EMF, por lo que cualquier modelo definido mediante EMF puede ser empleado. La ventaja que utilizar esta referencia aporta, sin embargo presenta una gran desventaja, inherente a cómo EMF representa estas referencias: mediante URIs (*Uniform Resource Identifier*).

a) El problema de las referencias en los modelos de trazabilidad.

En EMF todos los objetos tienen asociada una determinada URI que los identifican unívocamente de todos los demás. La URI de un determinado elemento perteneciente a un modelo Ecore está íntimamente relacionada con la representación canónica de éste (el formato persistente en *XMI*).

Veamos el problema en nuestro caso de estudio:

Para el modelo *PurchaseOrder* simplificado se genera el siguiente esquema relacional mediante el operador *ModelGen* (operador que nos permite realizar la transformación de modelos):

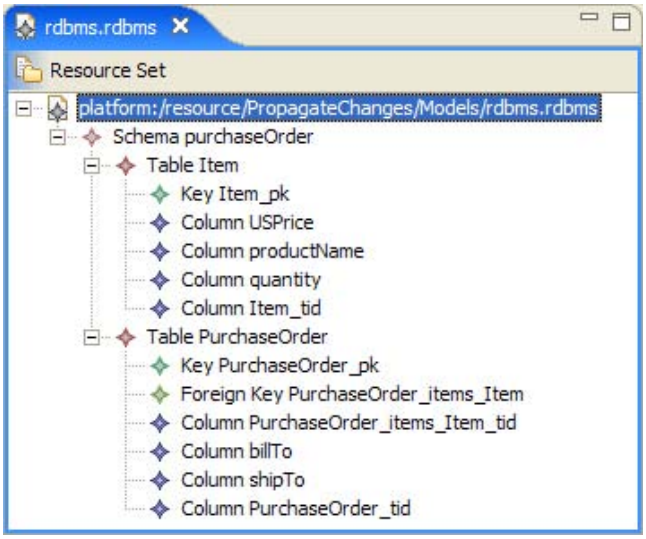

Figura 20: Modelo relacional generado para el modelo *Purchase Order* simplificado.

Para el modelo mostrado en la Figura 20, obtenemos la siguiente representación canónica:

```
<?xml version="1.0" encoding="ASCII"?>
<rdbms:Schema xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:rdbms="http:///es.upv.dsic.issi/moment/rdbms_mm.ecore" name="purchaseOrder"
kind="0">
  <tables name="Item" kind="0">
    <key name="Item_pk" kind="0" column="//@tables.0/@column.3"/>
    <column name="USPrice" kind="0"
        type="http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal"/>
    <column name="productName" kind="0"
        type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="quantity" kind="0"
        type="http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <column name="Item_tid" kind="0" type="NUMBER" key="//@tables.0/@key.0"/>
  </tables>
  <tables name="PurchaseOrder" kind="0">
    <key name="PurchaseOrder_pk" kind="0" column="//@tables.1/@column.3"/>
    <foreignKey name="PurchaseOrder_items_Item" kind="0"
        refersTo="//@tables.0/@key.0" column="//@tables.1/@column.0"/>
    <column name="PurchaseOrder_items_Item_tid" kind="0"
        type="NUMBER" foreignKey="//@tables.1/@foreignKey.0"/>
    <column name="billTo" kind="0"
        type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="shipTo" kind="0"
        type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="PurchaseOrder_tid" kind="0" type="NUMBER" key="//@tables.1/@key.0"/>
  </tables>
</rdbms:Schema>
```

Las URIs que identifican cada uno de los elementos de este modelo se muestran en la siguiente tabla:

PurchaseOrder	platform:/resource/PropagateChanges/Models/rdbms.rdbms#/
Item	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0
Item_pk	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@key.0
USPrice	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.0
productName	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.1
quantity	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.2
Item_tid	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.3
PurchaseOrder	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1
PurchaseOrder_pk	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@key.0
PurchaseOrder_items_Item	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@foreignKey.0
PurchaseOrder_items_Item_tid	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.0
billTo	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.1
shipTo	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.2
PurchaseOrder_tid	platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.3

Tabla 1: Identificadores (URIs) para los elementos del Modelo A.

Como se puede observar, estos identificadores se construyen a partir de la posición que cada elemento ocupa en el fichero XMI que se persistirá a disco.

Si examinamos el metamodelo de trazabilidad de la Figura 19, notaremos que los roles de *domain* y *range*, son referencias a *EObjects*. Estas referencias en el espacio tecnológico de EMF se establecen mediante URIs. En Maude, al generar el término que representa un modelo, se crea un identificador interno al kernel de MOMENT. Para modelos obtenidos directamente de la proyección de EMF a Maude, este identificador es igual a la URI de un determinado elemento. Estos identificadores son los que se emplearán para establecer las referencias entre modelos.

Cuando se realiza una operación sobre uno o varios modelos, y se obtiene un nuevo modelo como término del álgebra, los elementos que lo componen (procedentes

de los modelos fuente), conservan sus respectivos identificadores (en este caso, éstos ya no coinciden con la URI que éstos tendrán en su representación canónica). Será necesario por tanto proyectar el término de nuevo al espacio tecnológico EMF para conocer las URIs de sus elementos, como podemos observar en los modelos mostrados en el Anexo V.

Esto implica que en principio, un modelo de trazabilidad no puede ser creado por completo en el espacio tecnológico Maude, ya que, aunque sí se conocen las URIs de los elementos dominio de una determinada correspondencia (*TraceabilityLink*) no ocurre lo mismo para los elementos rango.

Por ello, será necesario un post-proceso que, dados el término que representa a un modelo devuelto por un operador, su representación canónica en EMF, y un modelo de trazabilidad (incompleto), sea capaz de completar el modelo de trazabilidad con las referencias correctas.

Para ello, se debe consultar el modelo de trazabilidad, acceder al rol rango de una correspondencia y obtener así uno o varios identificadores. Se debe entonces explorar el modelo devuelto directamente por el operador aplicado, y construir así la lista de elementos a los que corresponden dichos identificadores. Por último, se debe consultar la representación canónica, obteniendo la URI de cada elemento, y sustituyendo la antigua referencia de *rango*, en el *Link* del modelo de trazabilidad.

Realizar todas estas operaciones en Java no es un trabajo especialmente complejo. No obstante, teniendo en cuenta que disponemos de un motor como Maude, donde tareas de esta índole se pueden especificar con una enorme facilidad, es comprensible que se haya optado por esta segunda vía. La operación que nos permite realizar esto es:

```
RefreshUrisInTraceabilityModel (
    oldTraceabilityModel,                                    (1)
    1,                                                       (2)
    BuildOIDTable (oldModel, newModel, XMM)                  (3)
) .
```

Donde *oldTraceabilityModel* es el modelo de trazabilidad a actualizar, *oldModel* es el modelo devuelto por el operador, antes de proyectarse a EMF, y *newModel* es el modelo *oldModel*, proyectado a EMF, y luego de nuevo a Maude, por lo que el identificador de cada elemento ha sido actualizados a su URI. Por último *XMM* es el metamodelo que conforman *oldModel* y *newModel*.

Por tanto, el parámetro (1) es el modelo de trazabilidad a refrescar. El segundo parámetro (2), establece si se refrescarán los roles dominio (valor cero) o los rango (valor uno), y el parámetro (3), es una tabla de pares *oldOID–newOID*, que indican la relación entre un identificador viejo, y su correspondiente valor actualizado. Como se observa, esta tabla se genera automáticamente mediante *BuildOIDTable*.

ii. Metamodelo de trazabilidad personalizado de MOMENT.

Como se comentó inicialmente, el usuario debe hacer uso de un metamodelo de trazabilidad que hereda del metamodelo básico. La herramienta MOMENT proporciona directamente un metamodelo personalizado que hereda del básico, y puede ser empleado directamente por el usuario. La Figura 21 nos lo muestra:

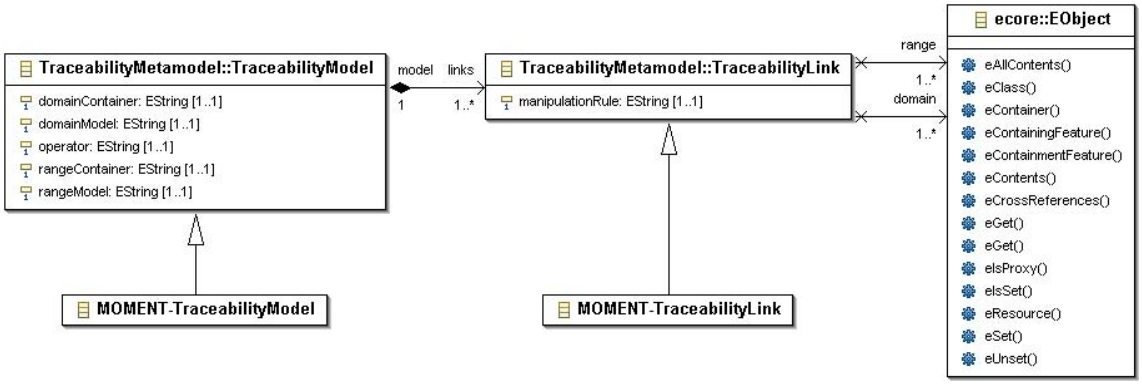

Figura 21: Metamodelo de trazabilidad personalizado de MOMENT.

En la figura, las clases TraceabilityMetamodel::TraceabilityModel y TraceabilityMetamodel::TraceabilityLink corresponden a las clases homónimas del metamodelo básico. Como se puede observar, las clases del metamodelo personalizado MOMENT-TraceabilityModel y MOMENT-TraceabilityLink heredan de las correspondientes clases del metamodelo básico.

El mecanismo para generar un nuevo metamodelo de trazabilidad personalizado por parte del usuario se realizada mediante un asistente. El metamodelo inicial generado será similar a éste, y podrá ser modificado mediante cualquier herramienta compatible con Ecore (el editor reflexivo de EMF, Omondo, etc.). Este mecanismo se comentará en mayor detalle más adelante.

5.2.1.2. Operadores de trazabilidad.

Una vez hemos mostrado la definición de un metamodelo de trazabilidad específico, explicaremos los operadores genéricos de trazabilidad que proporcionan en la herramienta MOMENT. Los operadores que proporcionan soporte para trazabilidad están definidos de forma genérica en una especificación algebraica parametrizada, llamada *MOMENT-TRAC(Y :: BASICTMM)*.

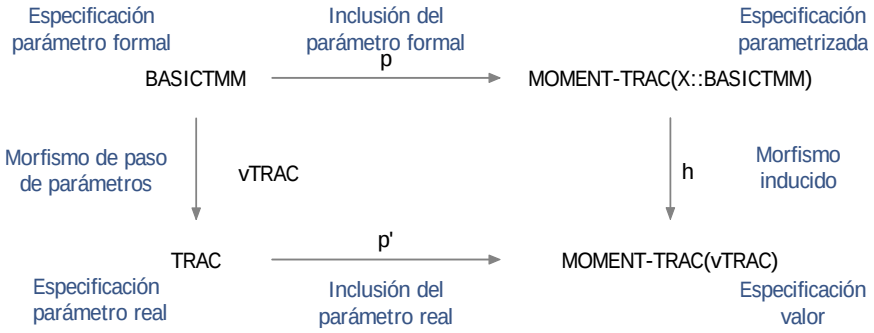

Figura 22: Diagrama de paso de parámetros para el módulo parametrizado MOMENT-TRAC(Y :: BASICTMM).

La Figura 22 muestra los elementos implicados en el mecanismo de paso de parámetros. *BASICTMM* (*BASIC Traceability MetaModel*) es la especificación algebraica del parámetro formal, denominado teoría en Maude. Esta teoría declara algunos operadores que garantizan la independencia entre la semántica de los operadores genéricos de trazabilidad y la semántica de un metamodelo específico.

Por ejemplo, un operador de este tipo es *GetDomain*. Éste obtiene el elemento dominio de un enlace de trazabilidad independientemente de la representación sintáctica del enlace. De esta forma, el parámetro formal se comporta como una interfaz a través de la que los operadores genéricos pueden acceder a los elementos de un modelo que conforma un metamodelo de trazabilidad específico.

TRAC es la especificación algebraica obtenida por el mecanismo de proyección de un metamodelo de trazabilidad específico. La especificación *TRAC* constituye el parámetro actual para el módulo *MOMENT-TRAC(Y :: BASICTMM)* y define la semántica del los operadores que se encuentran declarados únicamente en la teoría BASICTMM. La vista *vTRAC* Es el morfismo que relaciona los elementos de el parámetro formal BASICTMM con los elementos del parámetro actual *TRAC*.

La especificación algebraica parametrizada *MOMENT-TRAC(Y :: BASICTMM)* contiene la definición de los operadores de trazabilidad que son independientes del metamodelo de trazabilidad específico *TRAC*. La especificación del valor *TRAC(vTRAC)* resulta de la instanciación del módulo parametrizado con el metamodelo de trazabilidad específico *TRAC*.

Es esta figura «p» y «p'» son morfismos de inclusión que indican que la especificación del parámetro formal está incluida en la especificación parametrizada, y que la especificación del parámetro actual está incluida en la especificación valor, respectivamente. El morfismo «h» es el morfismo inducido, que relaciona los elementos del módulo parametrizado con los elementos de la especificación valor *MOMENT(vTRAC)*, usando el morfismo de paso de parámetros *vTRAC*.

Los operadores de trazabilidad definidos en el módulo parametrizado *MOMENT-TRAC(X :: BASICTMM)*, se clasifican en dos grupos: operadores que proporcionan soporte para la navegación, y operadores que realizan operaciones en modelos de trazabilidad.

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

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

   El operador *Domain* toma tres modelos como entrada: un modelo de trazabilidad (mapAB), un modelo dominio (A), y un modelo rango (B'). El operador navega los enlaces del modelo de trazabilidad y devuelve un submodelo de A (A'), como se muestra en la Figura 23.a.

   El operador *Range* también recibe tres entradas: un modelo de trazabilidad (mapAB), un modelo dominio (A'), y un modelo rango (B). Este operador realiza la operación inversa a la anterior: navega el los enlaces de trazabilidad que tienen elementos de A' como elementos

dominio y devuelve un submodelo del modelo rango B (B'), como se muestra en la Figura 23.b.

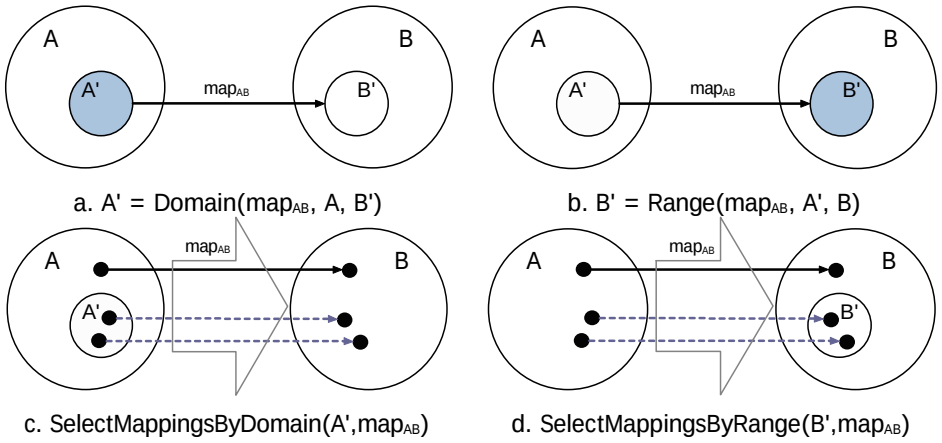

Figura 23: Operadores genéricos para navegación de las trazas.

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

   El operador *SelectMappingsByDomain* recibe dos modelos de entrada: un modelo dominio (A') y un modelo de trazabilidad (mapAB). El operador extrae los enlaces de trazabilidad del modelo «mapAB» que tienen elementos del modelo A' como elementos dominio y devuelve este submodelo. Los enlaces de trazabilidad que se añadir al modelo de trazabilidad de salida se muestran en la Figura 23.c con una línea punteada.

   El operador *SelectMappingsByRange* recibe dos modelos de entrada: un modelo rango (B') y un modelo de trazabilidad (mapAB). En este caso, el operador extrae los enlaces de trazabilidad del modelo «mapAB» que tienen elementos de B' como elementos rango, y devuelve este submodelo, como se muestra en la Figura 23.d.

5.3. Generación de modelos de trazabilidad.

Como ya se comentó someramente en el punto 5.1, una correspondencia (*mapping*), establece una relación entre un elemento de un modelo dominio, y un elemento de un modelo rango indicando que ambos representan el mismo elementos en modelos diferentes.

Diversas aproximaciones, como por ejemplo RONDO [RONDO], especifican operadores basados en estas correspondencias para tratar con modelos. Esto implica que las correspondencias entre ambos modelos, deben definirse de forma explícita para aplicarse sobre los modelos [Mel03].

Sin embargo, en MOMENT, esta relación se establece de forma implícita mediante un morfismo de equivalencia definido a nivel de metamodelo (nivel M2), y no de modelo (nivel M1), de forma más abstracta y reusable.

5.3.1. Generación automática.

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

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

A continuación, mostramos algunos ejemplos de operadores de Gestión de Modelos indicando sus entradas, salidas y semántica. Estos serán los operadores necesarios para resolver el ejemplo de propagación de cambios del punto 5.1.2:

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

2. *Diff*. Este operador realiza la diferencia entre dos modelos de entrada (A y B). La diferencia entre dos modelos (C) es el conjunto de elementos del modelo A que no corresponden a ningún elemento del modelo B. Este operador también devuelve dos modelos de enlaces (*mapAC* y *mapBC*) del mismo modo que los operadores *Cross* y *Merge.*

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

Como se observa en los ejemplos mencionados, toda aplicación de un operador sobre uno o varios modelos de entrada para producir los correspondientes modelos de

salida, producen de forma automática los modelos de correspondencias que relacionan las entradas (modelos dominio) con las salidas (modelos rango).

5.3.2. Generación manual.

Definir correspondencias entre modelos, no obstante, puede resultar útil. Es por esto, que a pesar de que la potencia de MOMENT reside en su alto nivel de abstracción y su genericidad, es posible definir también estas relaciones de forma explícita.

En nuestra herramienta, por tanto, será de especial utilidad definir las correspondencias de forma manual cuando no exista un morfismo de equivalencia definido. Para este fin se proporciona un editor que permite crear y modificar modelos de trazabilidad, como presentaremos en el punto 6.2.4.

En el caso de estudio, por ejemplo, puede emplearse para documentar (y reflejar en el correspondiente modelo de trazabilidad) las transformaciones o modificaciones realizadas sobre el modelo *RDB* para obtener el modelo *RDB"*.

5.4. Proceso del soporte para Trazabilidad en MOMENT.

Teniendo en cuenta el proceso descrito en el punto 5.1.1 para definir un modelo de trazabilidad, a continuación se indica cómo estas tareas se realizan en la herramienta MOMENT:

1. *Definición de la traza.* Los usuarios pueden definir su propio metamodelo en un contexto de trabajo específico. Es más, el metamodelo de trazabilidad por defecto del framework MOMENT puede ser utilizado en su lugar. Como se vio anteriormente, el metamodelo de trazabilidad puede ser definido usando notaciones gráficas bien conocidas, como UML, mediante herramientas compatibles con EMF.

2. *Producción de la traza.* Por defecto, el modelo de trazabilidad es generado automáticamente por un operador cuando éste realiza una manipulación sobre un conjunto de modelos de entrada.

   No obstante, los enlaces de trazabilidad pueden ser definidos manualmente mediante un editor generado a partir del metamodelo de trazabilidad siguiendo la cultura de desarrollo software de EMF. Es más, un modelo de trazabilidad entre dos modelos puede ser inferido automáticamente mediante el uso de heurísticas [Mad01] o conocimiento histórico [Mad03].

3. *Extracción de la traza.* El análisis del conocimiento proporcionado por un conjunto de modelos de trazabilidad puede ser útil para realizar otras tareas. Los operadores de trazabilidad se utilizan para manejar esta información en el framework MOMENT. Dichos operadores

constituyen una solución automática y reusable que proporciona soporte para trazabilidad en diversos escenarios en el campo de MDE. Por lo tanto, nuestro framework proporciona un soporte automática para este paso a pesar de que el usuario debe razonar acerca del conocimiento extraído.

4. *Verificación de la traza.* La consistencia de los modelos de trazabilidad puede conservarse automáticamente cuando cualquier información del modelo dominio o rango sea modificada, por medio de la aplicación de operadores de trazabilidad. Consideremos que tenemos un modelo dominio A, un modelo rango B, y un modelo de trazabilidad mapAB que ha sido definido entre ellos. Tres tipos de modificaciones pueden realizarse sobre modelos: (a) adición de elementos, (b) modificación de elementos existentes y (c) borrado de elementos.

   En el caso de añadir elementos a un modelo, el modelo de trazabilidad permanece consistente, puesto que no existe conexión alguna entre los nuevos elementos y los elementos del otro modelo, a pesar de que esta conexión sea definida explícitamente con posterioridad.

   En el caso de modificación o eliminación de elementos de un modelo, los enlaces pueden romperse cuando los elementos del modelo dominio o rango se eliminan. Este problema puede resolverse sencillamente utilizando operadores de trazabilidad. Si eliminamos elementos en el modelo A, obteniendo un modelo A', podemos aplicar el operador *SelectMappingsByDomain* para obtener el nuevo modelo de trazabilidad mapAB' que sea consistente (*mapAB' = SelectMappingsByDomain(A', mapAB)*).

5.5. Aplicación de MOMENT al caso de estudio.

5.5.1. Pasos a realizar.

El problema mostrado en el caso de estudio (véase apartado 5.1.3) puede ser simplificado como se muestra en la Figura 24, donde el modelo *mapUML2RDB'* puede ser fácilmente obtenido de los modelos *mapUML2RDB* y *mapRDB2RDB'* mediante el operador Compose. Por lo tanto, el problema puede enunciarte de la siguiente manera:

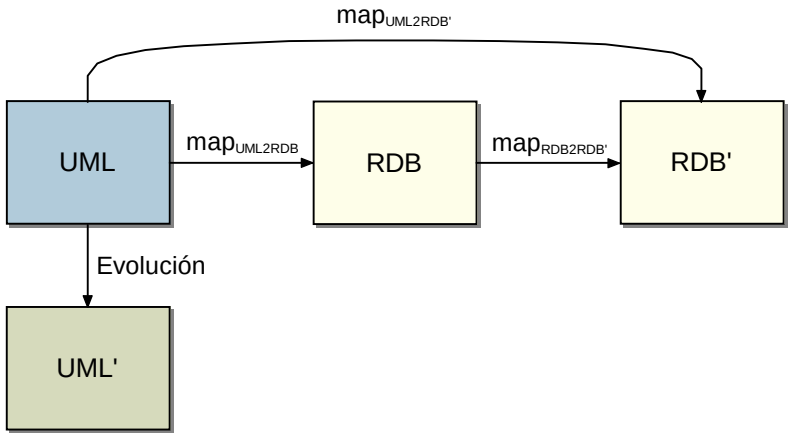

Figura 24: Esquematización del problema del caso de estudio.

«Dados los siguiente modelos: un diagrama de clases UML original (UML); un diagrama de clases UML (UML), que ha sido evolucionado de UML; una base de datos relacional RDB', que ha sido generada a partir del diagrama de clases UML y modificada posteriormente; y un modelo de trazabilidad entre UML y RDB' (mapUML2RDB'); deberemos obtener la base de datos relacional del diagrama de clases UML' que conserve los cambios realizados en RDB'.»

Éste problema puede ser resulto por el siguiente operador complejo:

```
operator PropagateChanges(UML, UML', RDB', map_UML2RDB') =
    <Unmodified, map_UML2Unmodified, map_UML'2Unmodified> = Cross(UML, UML')      (1)
    RDB'' = Range(map_UML2RDB', Unmodified, RDB')                                 (2)
    <newUML, map_UML'2newUML, map_Unmodified2newUML> = Diff(UML', Unmodified)     (3)
    <newRDB, map_newUML2newRDB> = ModelGenUML2RDB(newUML)                         (4)
    <C, map_RDB''2C, map_newRDB2C> = Merge(RDB'', newRDB)                         (5)

return (C)
```

Este operador está construido a partir de operadores simples del álgebra de MOMENT y los pasos seguidos en el *script* se representan en la Figura 25. Estos pasos son lso siguientes:

1. «*Unmodified*» es la parte del modelo UML que permanece sin modificar en el modelo UML'.

2. «*RDB''*» es el submodelo de RDB' que corresponde a la parte no modificada de UML'.

3. «*newUML*» es la parte de UML' que ha sido añadida al modelo UML.

4. «*newRDB*» es el esquema relacional obtenido de la traducción de newUML al metamodelo relacional.

5. «*C*» es el modelo final obtenido de la integración de las bases de datos obtenidas en los pasos 2 y 4.

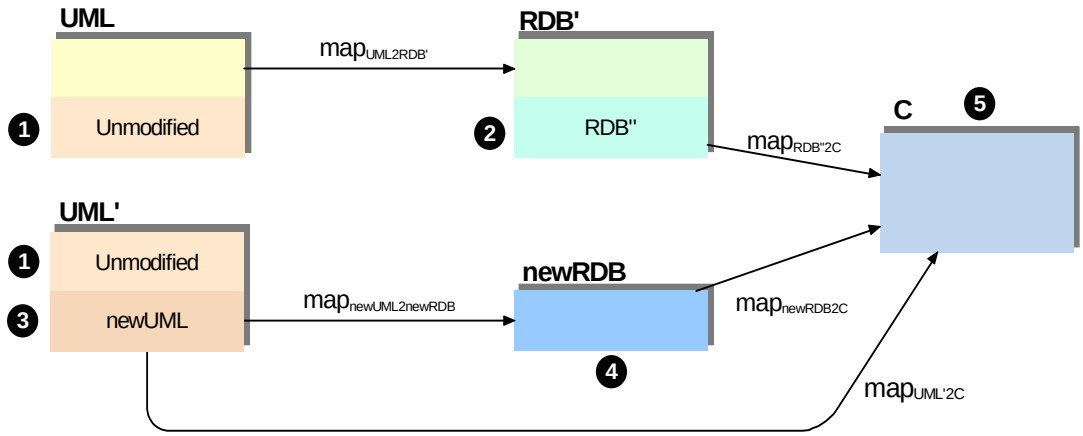

Figura 25: Solución al problema del caso de estudio.

Si deseamos añadir soporte para trazabilidad a éste operador para generar el modelo de trazabilidad que relacione el modelo UML' con el nuevo modelo (C), únicamente debemos añadir el siguiente paso tras el paso 5:

```
<mapUML'2C, mapmapUnmodified2C2mapUML'2C, mapmapnewUML2C2mapUML'2C> = Merge(
    Compose(Unmodified, SelectMappingsByDomain(Unmodified, mapUML2RDB'),
            RDB'', mapRDB''2C, C),
    Compose(newUML, mapnewUML2newRDB, newRDB, mapnewRDB2C, C)
  )
```

Este paso une dos modelos de trazabilidad: uno se define entre la parte no modificada de UML' y C, y el otro se define entre la parte nueva de UML' y C. Este paso une ambos modelos mediante el operador *Merge*, del mismo modo que otro par cualquiera de modelos pertenecientes al mismo metamodelo. El modelo mapUML'2C debe ser añadido como valor devuelto en el *script*.

El operador complejo resultante resuelve el problema de la propagación de cambios del caso de estudio de forma independiente de los metamodelos implicados de forma que puede ser aplicado a cualquier combinación de metamodelos, en lugar de usar los metamodelos UML y relacional.

5.5.2. Ejemplo de ejecución.

A continuación mostraremos paso por paso los modelos obtenidos, así como las operaciones a realizar de forma que se ilustre con detalle cómo se puede aplicar MOMENT al caso de estudio, así como la utilidad y el valor añadido por el soporte de trazabilidad proporcionado.

La Figura 26 muestra el modelo simplificado para una aplicación que gestione órdenes de compra introducido en [Bud03].

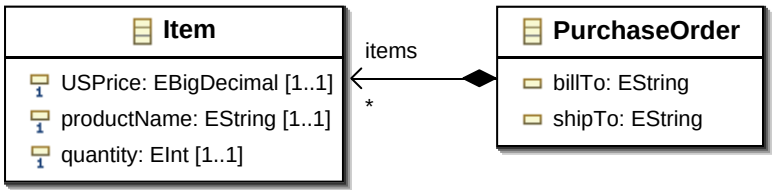

Figura 26: Modelo *Purchase Order* simplificado (UML).

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

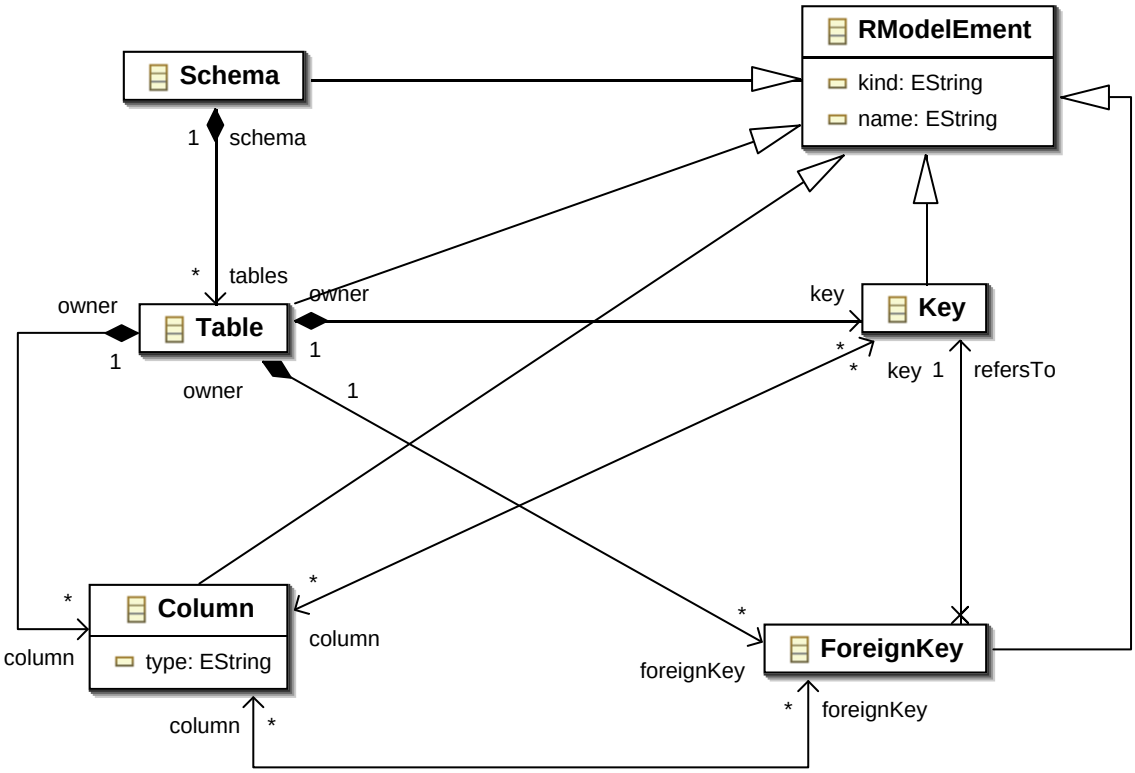

Figura 27: Metamodelo relacional simplificado.

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

A continuación la Figura 28 muestra el modelo *Ecore Purchase Order Simplificado*, su correspondiente modelo equivalente generado automáticamente para el metamodelo relacional, y el modelo de trazabilidad que los relaciona (en el punto 6.2.4 se describirá el editor de la figura con mayor detalle).

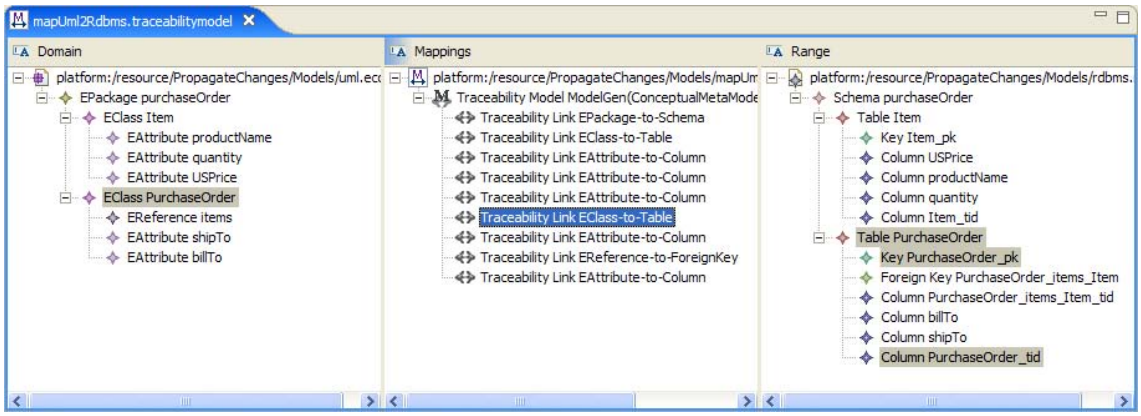

Figura 28: Vista de los modelos «*UML*», modelo de trazabilidad «*mapUML2RDB*», y «*RDB*», en el editor de trazabilidad de MOMENT.

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

En segundo lugar, se van a realizar modificaciones en la base de datos, añadiendo una tabla llamada *Customer*. Esta tabla almacenará los datos de clientes. Cada orden de compra se relacionará con una entrada de esta tabla, por lo que se deberán crear también las correspondientes columnas y claves ajenas en la tabla *PurchaseOrder*. El nuevo modelo relacional queda tal como se muestra en la Figura 29.

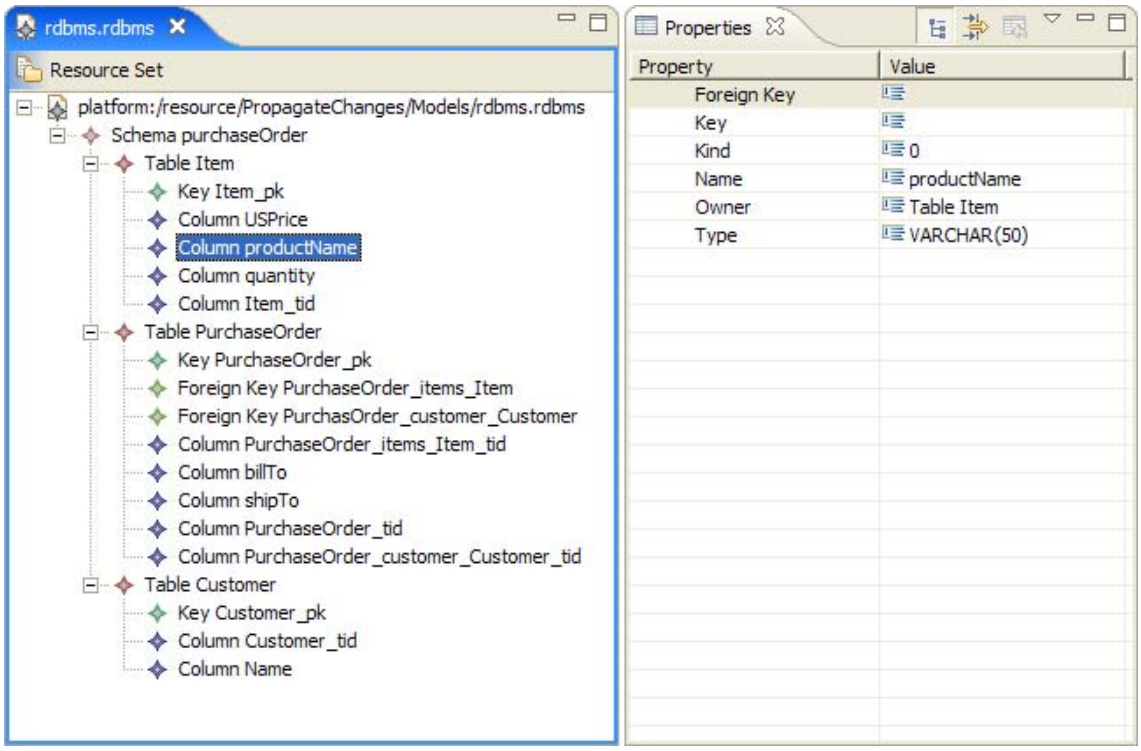

Figura 29: Modelo relacional *Purchase Order* modificado (RDB').

Tras generar esta primera base de datos, se modifican los requisitos del sistema, por lo que el diagrama UML para la orden de compra original se transforma en el modelo *Purchase Order* completo, mostrado en [Bud03]:

Figura 30: Modelo *Purchase Order* completo (UML').

Para propagar estos cambios deberemos aplicar el operador de propagación de cambios enunciado en el apartado 5.5.1. Mediante el operador *Cross* se obtiene la

parte común entre los modelos UML y UML' (la parte no modificada en UML'); y mediante el operador *Range*, se obtiene del modelo RDB' los elementos que corresponden a esta parte no modificada (RDB").

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

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

La invocación de este operador se realiza de la sigueiente manera:

```
PropagateChanges(umlModel, umlPrimaModel, rdbmsPrimaModel, mapUmlRdbmsModel)
```

Donde *umlModel* es el término que representa al modelo *Purchase Order simplificado* (*UML*), *umlPrimaModel* corresponde con el término de *Purchase Order Completo* (*UML'*), *rdbmsPrimaModel* es el término generado para el modelo relacional *Purchase Order modificado* (*RDB'*) y por último, *mapUmlRdbmsModel* es el modelo de trazabilidad entre UML y RDB'.

La ejecución produce el siguiente modelo:

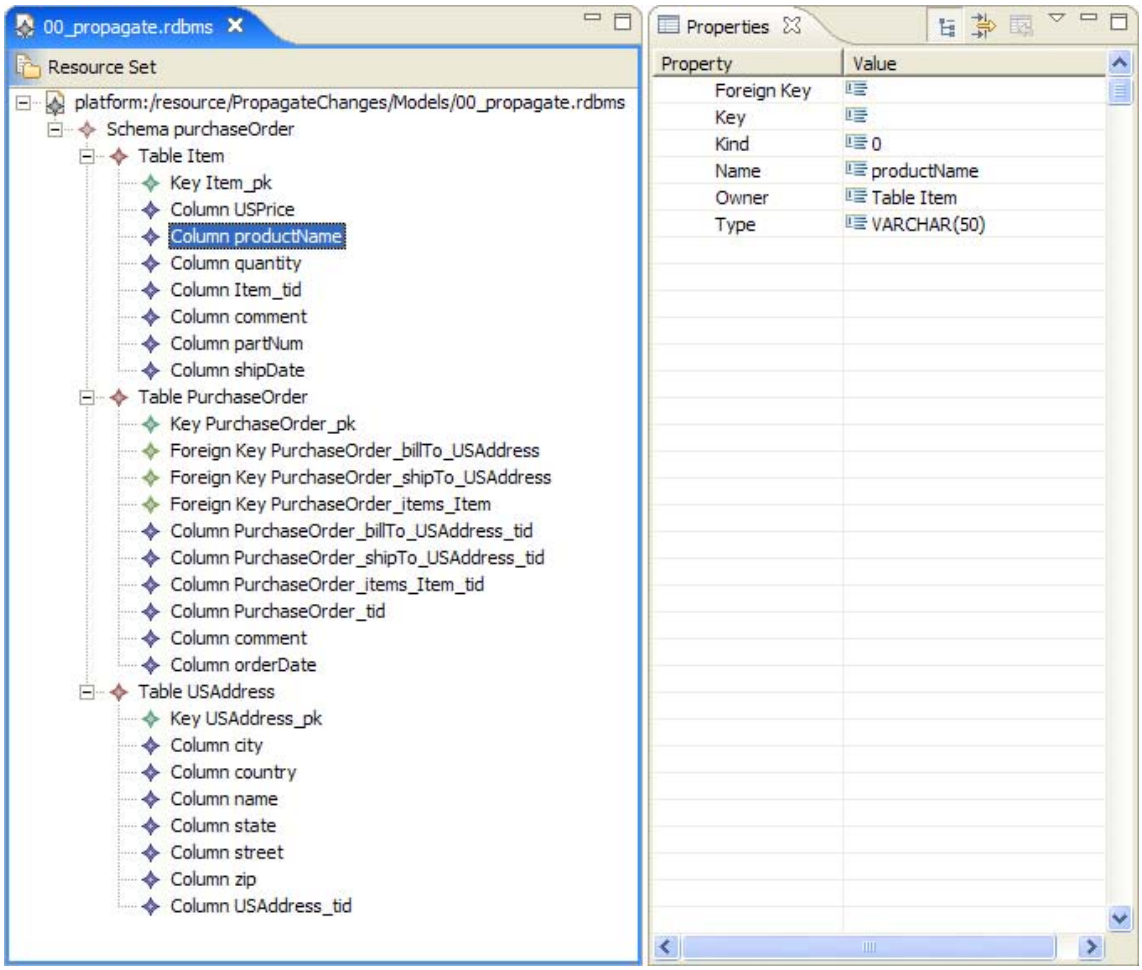

Figura 31: Modelo obtenido tras la aplicación del operador de propagación de cambios.

Se puede observar que el tipo de *productName* se ha mantenido en «*VARCHAR(50)*», en lugar de haber sido machacado de nuevo por el valor «*http://www.eclipse.org/emf/2002/Ecore#//EString*». Sin embargo se observa que la tabla *Customer* no se ha mantenido.

Esto se debe a que en el modelo de trazabilidad *mapUML2RDB'* (obtenido mediante la composición de *mapUML2RDB* y *mapUML2RDB"*) no existe información acerca del origen de esta tabla (ni de ninguno del resto de los elementos creados desde cero). Podría aplicarse la convención de que todo elemento nuevo creado tengo como elemento dominio en el correspondiente modelo de trazabilidad el elemento raíz del modelo dominio, de esta forma, los elementos nuevos creados sí que se mantendrían. Pero dado que este método es poco elegante, y excesivamente artificial, se puede emplear la potencia y la simplicidad de *MOMENT* para conservar estos elementos.

Los elementos que faltan en el modelo final son aquellos que aparecen únicamente en el modelo RDB', y no tienen ninguna relación con los elementos de UML. Esto se puede expresar mediante los operadores de MOMENT de la siguiente forma:

```
<addedElts, mapAC, mapBC> =
        Diff(rdbmsPrimaModel, Range(mapUmlRdbmsModel, umlModel, rdbmsPrimaModel))
```

Donde *addedElts* es el submodelo de RDB' con elementos añadidos a éste. Los modelos de trazabilidad se han denominado *mapAB* y *mapBC* por simplicidad de los nombres.

Así, por tanto, la aplicación del nuevo operador de propagación de cambios

```
operator
newPropagateChanges(umlModel, umlPrimaModel, rdbmsPrimaModel, mapUmlRdbmsModel) =
    incompleteModel =
        PropagateChanges(umlModel, umlPrimaModel, rdbmsPrimaModel, mapUmlRdbmsModel)

    <addedElts, mapAC, mapBC> =
        Diff(rdbmsPrimaModel, Range(mapUmlRdbmsModel, umlModel, rdbmsPrimaModel))

    C = Merge(incompleteModel,addedElts)

return (C)
```

generará el modelo representado en la Figura 32

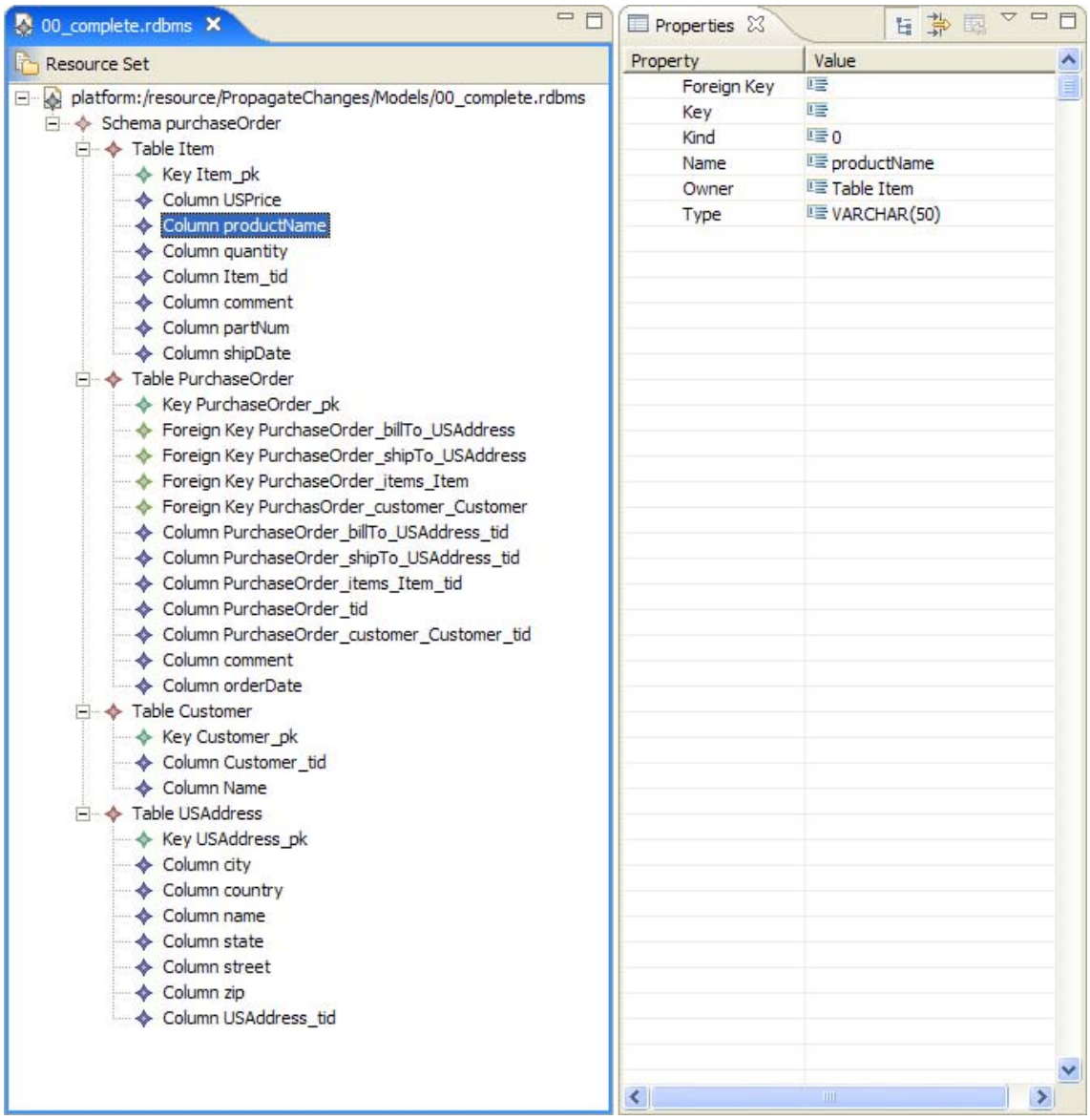

Figura 32: Modelo obtenido tras la aplicación de *newPropagateChanges*.

Se puede observar, ahora sí, cómo se incluyen las tablas, claves y columnas generadas manualmente en la base de datos, así como el cambio de tipo de la columna *productName*. También se observa como se han aplicado correctamente los cambio de los atributos *shipTo* y *billTo* del diagrama *UML* a referencias del diagrama *UML'*, con las respectivas columnas y claves ajenas.

6. MOMENT TRACEABILITY TOOLS: IMPLEMENTACIÓN DEL SOPORTE PARA TRAZABILIDAD EN MOMENT.

En el capítulo anterior hemos abordado de forma amplia el problema de la gestión de la trazabilidad en una herramienta de gestión de modelos, su utilidad y hemos ahondado de forma conceptual cómo se resuelve dentro de la herramienta MOMENT a través del caso de estudio.

En el próximo capítulo explicaremos cómo se han materializado todos estos aspectos en un conjunto de plug-ins para eclipse. Estos plug-ins se agrupan en la característica «*MOMENT Traceability Tools*».

Para la implementación de estas herramientas se ha hecho uso del *Eclipse Modeling Framework*. Este entorno de modelado nos permitirá definir los metamodelos de trazabilidad, generar un editor básico para éstos, así como los pertinentes asistentes para crear diversos modelos y metamodelos.

Para comprender mejor la estructura de plug-ins de MOMENT Traceability Tools, presentaremos en primer lugar Eclipse Modeling Framework, la metodología de trabajo, ventajas que nos proporciona y esquema de generación de código empleado.

Posteriormente, se presentarán los distintos plug-ins creados, en concreto, seis. El primero de ellos, *MOMENT Basic Traceability Metamodel* (*es.upv.dsi.issi.moment.traceability.basicmetamodel*), define el metamodelo básico de trazabilidad, para el que también se ha generado el correspondiente plug-in de soporte para la edición (*es.upv.dsic.issi.moment.traceability. basicmetamodel.edit*), *MOMENT Basic Traceabiliy Metamodel Edit Support*.

Otros dos plug-ins corresponden a la definición del metamodelo personalizado de MOMENT de trazabilidad, con su código para el soporte a la edición. (*MOMENT Custom Traceability Metamodel – es.upv.dsi.issi. moment.traceability.momentmetamodel y MOMENT Custom Traceability Metamodel Edit Support – es.upv.dsi.issi.moment.traceability.momentmetamodel.edit*)

Igualmente, se proporciona el asistente necesario para crear un nuevo metamodelo de trazabilidad de forma simple, listo para que el usuario lo adecue a sus gustos. Este plug-in se denomina *MOMENT New Traceability Metamodel Wizard* y su identificador es *es.upv.dsi.issi.moment.traceability. newmetamodel*.

Por último, el plug-in «*es.upv.dsic.issi.moment.traceability.editor*» provee a eclipse del pertinente editor de modelos de trazabilidad así como un asistente para generar un nuevo modelo de trazabilidad, listo para editar y generar manualmente el conjunto de correspondencias.

6.1. Eclipse Modeling Framework.

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

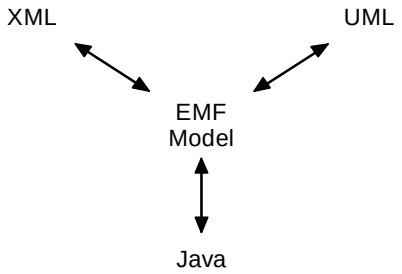

Figura 33: EMF unifica Java, XML, and UML.

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

6.1.1. Modelado en EMF.

EMF es básica y simplemente un *framework* para describir un modelo y posteriormente poder generar otros elementos a partir de él. No obstante, hay una importante diferencia que hace que no sólo sea eso: EMF está realmente afinado e integrado para una programación eficiente.

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

Un modelo EMF es esencialmente el subconjunto de los diagramas de clases de UML. Esto es, un simple modelo de las clases o datos de la aplicación. Por esto, un amplio porcentaje de los beneficios del modelado pueden obtenerse en un entorno de desarrollo Java estándar. La correspondencia entre un modelo EMF y su correspondiente código Java es natural y simple para que los programadores de Java lo comprendan.

6.1.2. Definiendo un modelo EMF.

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

referencias pueden también persistirse entre diferentes recursos (documentos), donde entra en juego resolución delegada y la carga por demanda.

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

6.1.2.1. El (Meta) modelo Ecore.

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

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

Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de metamodelos. Para ello, proporciona elementos útiles para describir conceptos y las relaciones entre ellos.

En la Figura 34 se muestra un subconjuto simplificado este modelo.

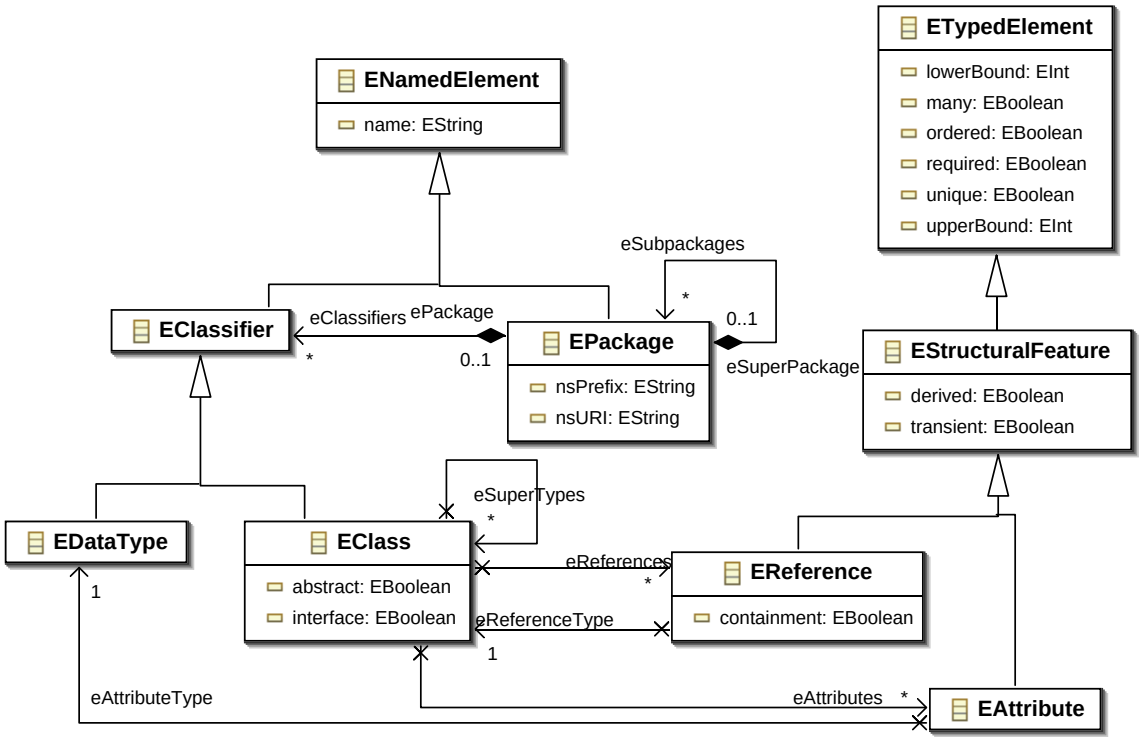

Figura 34: Subconjunto simplificado del modelo Ecore.

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

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

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

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

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

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

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

6.1.2.2. La creación de un modelo.

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

Si se comienza mediante interfaces Java, el generador de EMF introspeccionará el código y construirá el modelo *core*. Si por el contrario se comienza a partir de un esquema XML el modelo se construirá a partir de éste. En caso de que se comience con UML, existen 3 posibilidades:

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

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

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

6.1.2.3. Serialización en XMI.

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

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

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

6.1.3. Generando código.

La principal ventaja de EMF, como la del modelado en general es el aumento en la productividad que resulta de la generación automática de código. Dado un modelo Ecore que hemos definido, es posible obtener una implementación con unos pocos clicks. Todo lo que hay que hacer es crear un proyecto usando el Asistente para un nuevo proyecto EMF, que automáticamente lanza el generador y seleccionar Generar código del modelo desde un menú.

6.1.3.1. Clases generadas del modelo.

¿Qué tipo de código genera EMF? La primera cuestión que se observa es que una clase *Ecore* (*EClass*) corresponde de hecho con dos cosas en Java: una interfaz y

la correspondiente clase que la implementa. Por ejemplo, la *EClass* para *PurchaseOrder* corresponde a una interfaz *Java*:

```
public interface PurchaseOrder ...
```

y la correspondiente clase que la implementa:

```
public class PurchaseOrderImpl extends ... implements PurchaseOrder {
```

Esta separación entre interfaz/implementación es una decisión de implementación impuesta por EMF. La razón se debe, a parte de que se considera un patrón que debería seguir toda API de modelos, a que es necesaria para soportar herencia múltiple en Java.

Lo siguiente que se observa acerca de cada interfaz generada es que extiende directa o indirectamente de la interfaz base *EObject* de la siguiente manera:

```
public interface PurchaseOrder extends EObject {
```

EObject es el equivalente de EMF para *java.lang.Object*, esto es, es la clase base para todos los objetos modelados. Extendiendo de *EObject* se introducen tres comportamientos básicos:

1. *eClass()* devuelve el metaobjeto del objeto (una EClass).

2. *eContainer()* y *eResource* devuelve el objeto contedor y el recurso del objeto.

3. *eGet()*, *eSet()*, *eIsSet()* y *eUnset()* proporcionan una API para acceder al objeto de forma reflexiva. Éstos son ligeramente más ineficientes que los métodos generados (estos métodos invocan a los generados mediante una sentencia *switch()*), como por ejemplo, *getShipTo() o setShiptTo()*), pero abren el modelo a un acceso completamente genérico.

El primero y el tercer item son interesantes únicamente se se desea acceder de forma genérica al objeto en lugar de, o de forma adicional, a usar los accesos generados con comprobación de tipos.

EObject, a parte de estos métodos proporciona poca funcionalidad más. Sin embargo, cabe mencionar una más:

```
public interface EObject extends Notifier {
```

La interfaz *Notifier* es también bastante reducida, pero introduce, no obstante, importantes características a todo objeto modelado; notificaciones acerca del cambio del modelo como en *Observer Design Pattern* [Gam95]. Al igual que la persistencia de los objetos, la notificación es una característica muy importante de un objeto EMF.

Sobre los métodos generados, el patrón exacto que se utilizada para la implementación de una característica dada (esto es, un atributo o una referencia) depende del tipo y otras propiedades ajustables por el usuario. En general, las

características se implementan como es de esperar. Por ejemplo, el método *get()* para el atributo *shipTo* simplemente devuelve una variable de instancia de la siguiente manera:

```
public String getShipTo() {
    return shipTo;
}
```

El correspondiente método *set()* establece la misma variable, pero además, manda una notificación a cualquier observador interesado en el cambio de su estado:

```
public void setShipTo(String newShipTo) {
    String oldShipTo = shipTo;
    shipTo = newShipTo;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this,Notification.SET,
    POPackage.PURCHASE_ORDER__SHIP_TO,oldShipTo, shipTo));
}
```

Nótese que para hacer este método más eficiente cuando el objeto no tiene observadores, la llamada (relativamente costosa) a *eNotify()* se evita mediante la guarda *eNotificationRequired()*.

Para otro tipo de características (*features*), especialmente las referencias bidireccionales donde la integridad referencial se debe mantener, se generan patrones más complejos. Sin embargo, en todos los casos el código es lo más eficiente posible.

La idea principal a resaltar, sin entrar en profundidad sobre cómo EMF genera el código par aun determinado modelo, es que éste es limpio, simple y eficiente. EMF no se apoya en grandes clases base o código generado ineficiente. El *framework* EMF es ligero, como los objetos generados para un modelo. La idea es que el código que se ha generado se parezca lo más posible al código que se hubiera generado a mano. Pero, dado que ha sido generado, se tiene la certeza de que éste es correcto. Es un gran ahorro de tiempo, especialmente para el código que gestina las referencias que puede llegar a ser muy complejo, y complicado de implementar sin errores a mano.

Finalmente, también es interesante mencionar dos clases más que se generan para un modelo: una factoría (*factory*) y un paquete (*package*). La factoría generada incluye un método *create* para cada clase del modelo. El modelo de programación de EMF recomienda encarecidamente (pero no requiere) el uso de factorías para crear objetos. En lugar de usar el operador *new* para crear una orden de compra, se debería hacer lo siguiente:

```
PurchaseOrder aPurchaseOrder =
 POFactory.eINSTANCE.createPurchaseOrder();
```

El *package* generado proporciona los accesos correspondientes para todos los metadatos Ecore del modelo. El *package* además incluye los accesos convenientes para las *EClases*, *EAttributes* y *EReferences*.

6.1.3.2. Otros elementos generados.

Además de las interfaces y las clases descritas anteriormentes, el generador de EMF puede opcionalmente generar lo siguiente:

1. Una clase *adapter factory* para el modelo.

2. Una clase *switch* «de conveniencia» que implementa, mediante una sentencia *switch*, un mecanismo de *dispatching* basado en el tipo de objeto. La clase *adapter factory* utiliza esta clase en su implementación.

3. Un archivo de manifiesto de plug-in, para que el modelo pueda ser empleado como un plug-in par el Eclipse.

4. Un esquema XML para el modelo.

EMF, además de generar el código para el modelo, puede crear, mediante la extensión *EMF.Edit* generar las clases que permitan visualizar y editar el modelo, proporcionando soporte para deshacer cambios. Puede además, generar un editor para el modelo.

6.1.3.3. Regeneración y combinación.

El generador de EMF produce archivos destinados a ser una combinación de partes generadas y partes escritas a mano. Por esto, EMF utiliza los marcadores *@generated* en los comentarios de *Javadoc* de interfaces, clases, métodos y campos para identificar las partes generadas.

Todo método que no tenga esta etiqueta será dejado tal como está en cada proceso de regeneración.

6.1.3.4. El modelo generador (Generator Model).

La mayor parte de los datos necesitados pos el generador de EMF se almacenan en el modelo *core*. Como se ha mencionado anteriormente, las clases generadas, así como sus nombres, atributos y referencias se encuentran todas ahí. Existe, no obstante, más información que se debe proporcionar al generador, como dónde colocar el código generado, qué prefijo usar para la factoría generada o los nombres de los paquetes de las clases, que no se almacena en este modelo. Todos estos datos, que pueden ser ajustados por el usuario necesitan guardarse en algún lugar para disponer de ellos en caso de desear regenerar el modelo en un futuro.

El generador de código de EMF utiliza un modelo generador para almacenar esta información. Al igual que Ecore, el modelo generador es un modelo EMF. De hecho, un modelo generador proporciona acceso a todos los datos necesitados para la generación, incluida la parte Ecore, «envolviendo» el correspondiente modelo *core*.

El significado de todo esto es que un generador EMF se ejecuta sobre un modelo generador, en lugar de un modelo *core*; es de hecho, un editor para modelos generadores. Cuando se emplea el generador de EMF se está editando un modelo

generador, que en indirectamente accede al modelo *core* para el cual se está obteniendo el código. Como se mostrará más adelante, en un proyecto de esto tipo se encuentran como mínimo dos ficheros: un archivo *.ecore* y uno *.genmodel*. El fichero *.ecore* es una serialización XMI del modelo core. El archivo *.genmodel* es un modelo generador serializado con referencias externes al documento hacia el archivo *.ecore*.

Separar el modelo generador del modelo *core* de esta maner tiene la ventaja de que el modelo Ecore actual puede permanecer puro e independiente de cualquier información que sólo es relevante para la generación de código. La desventaja de almacenar toda la información en el modelo *core* es que el modelo generador puede quedar desfasado si el modelo *core* cambia. Para tratar esta situación las clases del modelo generador incluyen métodos para «reconciliar» un modelo generado y su correspondiente modelo *core*. Empleando estos métodos, los dos archivos pueden mantenerse sincronizados automáticamente por el *framework* y el generador.

6.1.3.5. EMF dinámico.

Hasta ahora se ha considerado la potencia de EMF en la generación de implementaciones para modelos. A veces, simplemente se desea compartir objetos sin requerir que esté disponible una implementación generada. Una simple implementación interpretativa puede ser suficiente.

Una característica particularmente interesante de la API reflexive es que puede ser usada también para manipular instancias de clases dinámicas (no generadas). Por ejemplo, para el ejemplo sencillo de la orden de compra podemos generar un modelo *core* en tiempo de ejecución de la siguiente manera:

```
EPackage poPackage = EcoreFactory.eINSTANCE.createEPackage();

EClass purchaseOrderClass = EcoreFactory.eINSTANCE.createEClass();
purchaseOrderClass.setName("PurchaseOrder");
poPackage.getEClassifiers().add(purchaseOrderClass);

EClass itemClass = EcoreFactory.eINSTANCE.createEClass();
itemClass.setName("Item");
poPackage.getEClassifiers().add(itemClass);

EAttribute shipToAttribute =
  EcoreFactory.eINSTANCE.createEAttribute();
shipToAttribute.setName("shipTo");
shipToAttribute.setEType(EcorePackage.eINSTANCE.getEString());
purchaseOrderClass.getEAttributes().add(shipToAttribute);

// ...
```

Ahora tenemos en memoria un modelo *core*, para el que no hemos generado ninguna clase Java. En este momento, podemos crear una instancia de orden de compra e inicializarla utilizando las mismas llamadas reflexivas:

```
EFactory poFactory = poPackage.getEFactoryInstance();
EObject aPurchaseOrder = poFactory.create(purchaseOrderClass);
aPurchaseOrder.eSet(shipToAttribute, "Cno. Vera S/N");
```

Incluso, un escenario más interesante es una utilización mixta de las capacidades de generación. Ya que podemos generar código para ciertas clases, y generar ciertas clases de forma dinámica que hereden de ellas.

6.1.4. Edición de modelos con *EMF.Edit*.

Anteriormente se ha mostrado cómo EMF puede tomar la definición de un modelo y producir una implementación Java buena y fácilmente personalizable para él. Una vez se decide utilizar EMF para modelar una aplicación, se puede utilizar el framework EMF.Edit para construir editors y visores con numerosas funcionalidades. Se puede generar un editor que permita mostrar y editar (esto es, editar, copiar, pegar, arrastrar y soltar…) instancias de un modelo utilizando visores estándares de *JFace*, todos ellos con capacidades para deshacer y rehacer sin limitaciones. Alternativamente, se puede utilizar el soporte reflexivo en EMF.Edit para hacer el mismo tipo de ediciones de forma reflexiva, incluso con un modelo EMF no generado.

6.1.4.1. Generando el código de EMF.Edit

Anteriormente se ha mostrado cómo EMF puede tomar la definición de un modelo y generar implementaciones Java para él. Dada la misma definicón de un modelo, se puede utilizar además el soporte de generación de código para crear las clases necesarias para editar el modelo. El generador de código EMF.Edit no es una herramienta separada, sino que es otra característica del modelo generador.

En el caso del generador de código de EMF.Edit, el código no se almacena en el mismo proyecto que el modelo, como courría en el caso de la generación del código del modelo. El *framework* EMF.Edit se divide en dos plug-ins separados: la parte independiente de la interfaz de usuario y la parte dependiente de ésta. Por defecto, el código generado por EMF.Edit sigue este mismo patrón: «*Generate Edit Code*» generará un plug-in que contiene las clases de soporte para la edición independientes de la interfaz, y «*Generate Editor Code*» generará el resto en un plug-in separado que además dependerá de la interfaz de usuario de Eclipse. Se puede, sin embargo, anular esto y almacenar todo en un mismo plug-in si es lo que se desea.

i. Generación del código de edición.

Invocar la generación del código de edición en el generador de EMF creará un plug-in completo conteniendo la parte independiente de la interfaz, esto es lo siguiente:

1. Un conjunto de *item providers*, uno para cada clase en el modelo.

2. Una clase *item provider adapter factory* que creará los *item providers* generados. Esta extiende de la clase *adapter factory* generada para el modelo descrita en el punto 6.1.3.

3. Una clase *Plugin* que incluye los métodos para localizar los recursos de texto e iconos del plug-in.

4. Un archivo de manifiesto del plug-in, *plug-in.xml*, especificando las dependencias requeridas.

5. Un archivo de propiedades, *plugin.properties*, conteniendo las cadenas de texto externalizadas necesarias por las clases generadas y el *framework*.

6. Un directorio de iconos, uno para cada clase del modelo.

Lo más importante de todo esto es el conjunto de clases que implementan los *item providers*.

Se puede elegir no generar un *item provider* para una clase en caso de que nunca se vayan a mostrar instancias de éste o si no se necesita personalizarlo. En este caso se utilizará el *item provider* reflexivo de EMF.Edit (clase *ReflectiveItemProvider*).

ii. Generación del editor.

Generar el código del editor se emplea para obtener un plug-in de un editor completamente functional que permitirá ver instancias del modelo utilizando diversos visores comunes. Igualmente, permite añadir, eliminar, copiar, cortar y pegar objetos del modelo, o modificar los objetos en una hoja de propiedades estándar, y todo ello con soporte completo para deshacer/rehacer. En el plug-in del editor, por tanto, se generarán los siguientes elementos:

- Un editor integrado en el banco de trabajo de Eclipse.
- Un asistente para crear documentos con nuevas intancias del modelo.
- Una contribución a las barras de menús, de herramientas y menús contextuales.
- Una clase Plugin que incluye los métodos para localizar los recursos de cadenas de texto e iconos.
- Un archivo de manifiesto del plug-in, *plug-in.xml*, que especifica las dependencias requeridas y extensiones del editor, asistente, y puntos de extensión de las acciones del banco de trabajo.
- Un archivo de propiedades, *plugin.properties*, conteniendo las cadenas de texto externalizadas necesarias para las clases generadas y para el *framework*.
- Un directorio conteniendo iconos para el editos y el asistente de Nuevo modelo.

El editor generado es un editor multipágina. La vista de *Outline* muestra el archivo del modelo en un visor en árbol. Cada página del editor se sincroniza con ella, y muestra una forma diferente de visualizar el modelo. Las siguientes páginas se crean por defecto:

- «*Selection*» muestra un visor en árbol similar al que existe en la vista *Outline*.
- «*Parent*» es un árbol invertido que muestra el camino de los contenedores del elemento seleccionado en la vista de *Outline* hasta la raíz.

- «*List*» muestra una lista conteniendo los hijos de la selección de la vista *Outline*.
- «*Tree*» muestra otro visor en árbol, cuya raíz es la selección actual.
- «*Table*» muestra un visor en tabla conteniendo los hijos de la selección actual.
- «*TableTree*» es lo mismo, únicamente que empelando un visor en árbol tabular.

El asistente generado permite crear un documento con nueva instancia del modelo conteniendo únicamente un objeto raíz de uno de los tipos del modelo. La implementación por defecto proporciona una lista desplegable para seleccionar cual.

iii. Regeneración de los plug-ins de EMF.Edit.

Cuando se regenera un modelo sobre un proyecto ya existente, el generador de EMF soporta el mismo tipo de «*mezcla*» para código EMF.Edit que para el código del modelo. Para ello, se empleará de la misma manera la etiqueta «*@generated*»

Como acabamos de ver, EMF.Edit también genera tres tipos de contenidos no-Java: archivos de propiedades, iconor, y archivos de manifiesto. Los archivos generados de propiedades contienen las cadenas de caracteres traducidas (recursos) referenciadas por el código generado. Se pueden añadir manualmente nuevos recursos de texto, o editar los generados y después regenerar sin perder los cambios. Cualquier string nuevo que se genere será añadido, pero aquellos que no se utilicen, sean generados inicialmente o no, nunca se eliminarán, por lo que se deberá hacer esta tarea manualmente.

Cada icono generado por EMF.Edit es unicamente una versión coloreada del icono genérico. Se espera que los iconos generados sean reemplazados por unos diseñados especialmente para el modelo, y por lo tanto, el editor nunca sobrescribirá un icono existente.

El generador, además, nunca sobreescribirá los archivos de manifiesto. No existe soporte para la mezcla automática ya que raramente es necesaria. Si se ha modificado un archivo de manifiesto a mano (por ejemplo, para añadir un nuevo punto de extensión), y después se realizan modificaciones en el modelo que afectan al archivo *plugin.xml* generado; deberá renombrarse el archivo *plugin.xml* existente (a *plugin.tmp* por ejemplo), ejecutar el generador para generar el nuevo *plugin.xml*, y finalmente, realizar la composición entre ambos a mano.

6.2. Herramientas desarrolladas.

6.2.1. Soporte para el metamodelo de trazabilidad básico.

Para dar soporte al metamodelo de trazabilidad básico se han generado dos plug-ins mediante el generador de código de EMF. Éstos son «*MOMENT Basic Traceability Metamodel*» (*es.upv.dsic.issi.moment.traceability. basicmetamodel*) y

«*MOMENT Basic Traceability Metamodel Edit Support*» (*es.upv.dsic.issi.moment.traceability.basicmetamodel.edit*).

6.2.1.1. Descripción.

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

i. Funciones del plug-in.

Este plug-in deberá proporcionar la siguiente funcionalidad.

- Definir en Ecore un metamodelo básico de trazabilidad, que proporcione los constructores para tratar correspondencias entre dos modelos cualesquiera.
- Generar el código apropiado para manejar éstos modelos en EMF. Esto permitirá tratar con ellos en memoria, así cómo proyectar al espacio tecnológico Maude modelos de trazabilidad, y a su vez, parsearlos desde éste.
- Proveer de los mecanismos independientes de la interfaz para la edición de un modelo de trazabilidad, así como su personalización, que los distinga de otros modelos EMF.

ii. Restricciones.

Dada la implementación actual que se hace de EMF, las clases del metamodelo básico de trazabilidad no podrán ser abstractas (ni declararse como interfaces), ya que en este caso no se generará código Java alguno, no pudiendose dar soporte a las funciones antes mencionadas.

iii. Dependencias.

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

6.2.1.2. Diseño e implementación de los plug-ins.

i. Creación del proyecto EMF y generación de código.

a) Obtención de la implementación del metamodelo de trazabilidad.

Como se ha comentado anteriormente, la implementación del metamodelo de trazabilidad, que a su vez, permite que éste esté cargado automáticamente en la base

de datos de EMF se genera de forma automática. Para ello se debe crear en primera instancia un modelo *Ecore* que refleje el metamodelo de trazabilidad básico.

Este metamodelo se ha definido mediante el editor en árbol por defecto de EMF. A continuación, la Figura 35, muestra el aspecto final del metamodelo básico de trazabilidad, así como un pequeño detalle de la vista de propiedades.

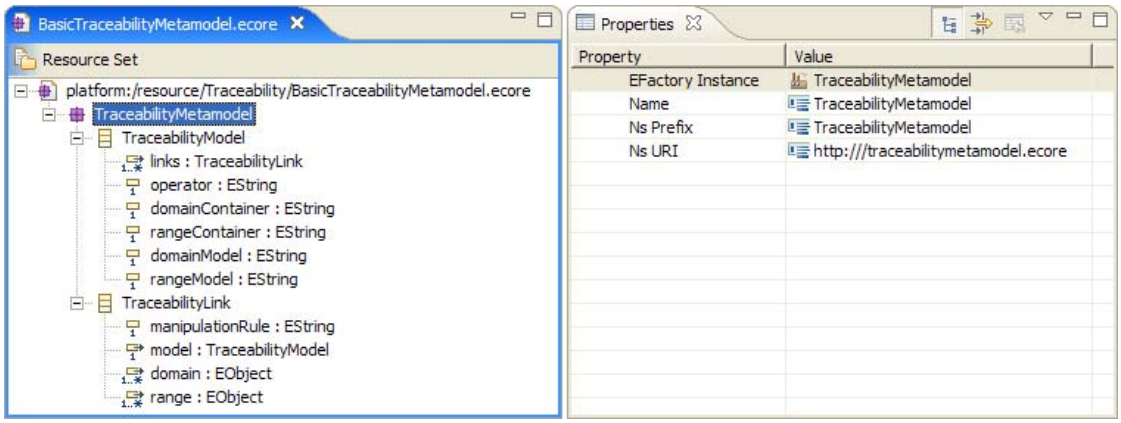

Figura 35: Metamodelo básico de trazabilidad en el editor en árbol de EMF.

Una vez creado el metamodelo, se puede hacer uso del asistente para crear un nuevo proyecto de EMF. Esto generará los dos ficheros necesarios para generarl el código para el modelo, el primero, el fichero *.ecore*, con el modelo propiamente dicho, y el segundo, el fichero generador *.genmodel*, con información acerca de cómo se generará el modelo. En el Anexo VI mostramos este proceso con más detalle.

La Figura 36 muestra el resultado de la ejecución del asistente: el modelo generador.

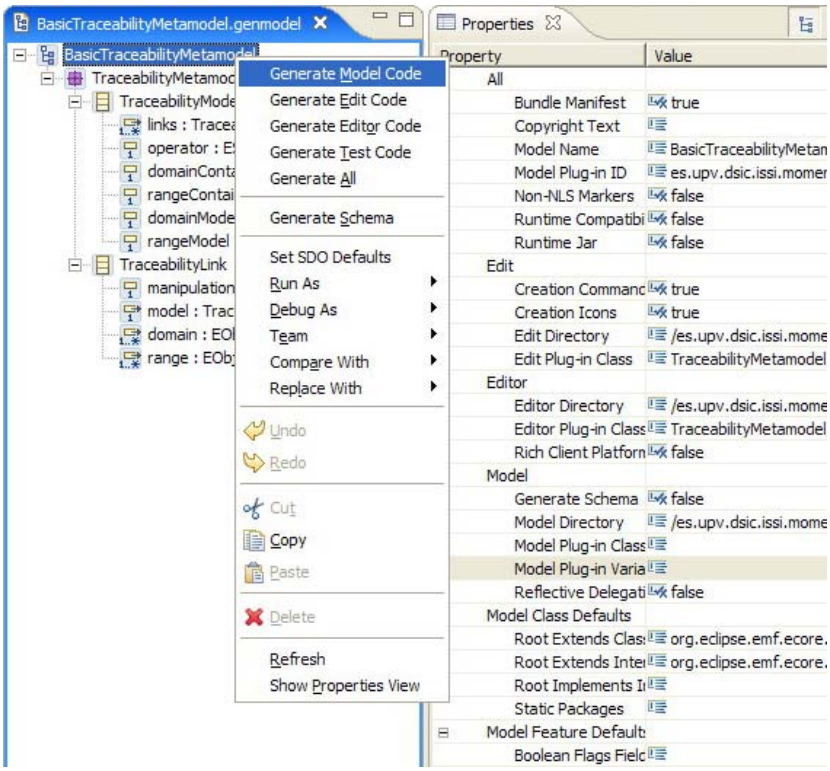

Figura 36: Vista del modelo generador y de las opciones de menú de generación de código.

Como se observa en la Figura 36, haciendo clic derecho sobre un elemento del modelo generador se pueden seleccionar las acciones de generación. Si el elemento sobre se pincha es el raíz, se generarán, a parte de los ficheros java pertinentes, otros archivos del proyecto, como el archivo de manifiesto, *plugin.xml*, etc.

Tras Ejecutar la acción de generar el código para el modelo el aspecto el proyecto del plug-in queda como muestra la Figura 37.

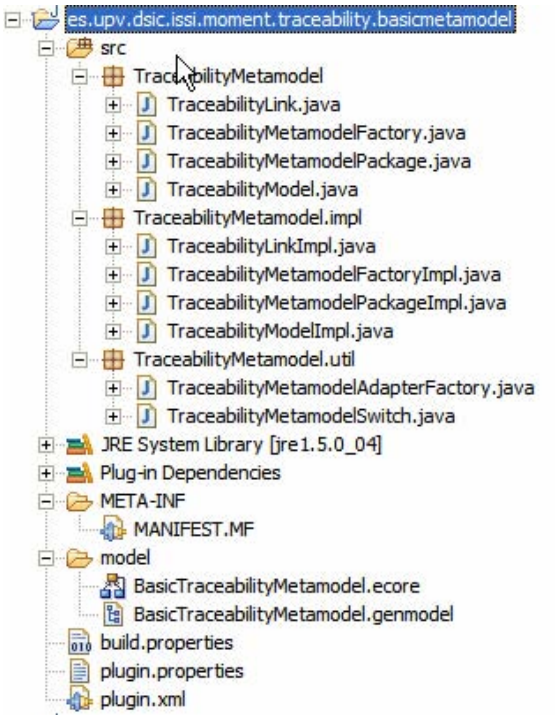

Figura 37: Proyecto del plug-in *MOMENT Basic Traceability Metamodel.*

Si pulsamos la generación del código de soporte para la edición, se creará en otro proyecto, y como un plug-in separado los archivos necesarios, como muestra la Figura 38.

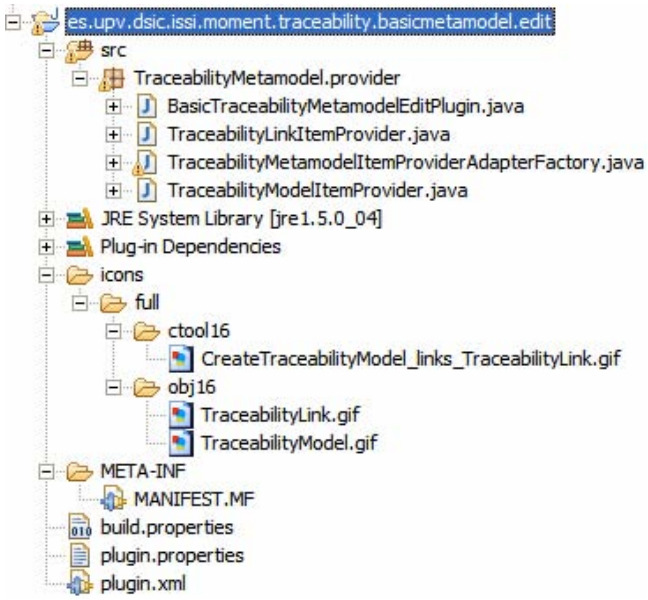

Figura 38: Proyecto del plug-in *MOMENT Basic Traceability Metamodel Edit Support.*

El código generado, así como los archivos de iconos, serán modificados para personalizar la representación de los modelos de trazabilidad.

ii. Estructura del código generado.

a) es.upv.dsic.issi.moment.traceability.basicmetamodel

El plug-in consta de tres paquetes como muestra la Figura 41.

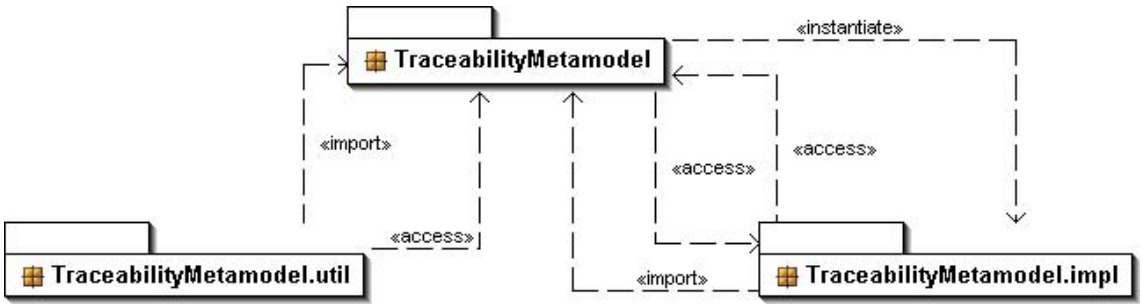

Figura 39: Diagrama de paquetes de *MOMENT Basic Traceability Metamodel*.

El paquete *TraceabilityMetamodel* define las interfaces que implementan las clases del paquete *TraceabilityMetamodel.impl*, tal y como se comentó en la sección 6.1.3 sobre cómo genera EMF el código para sus modelos. El paquete *TraceabilityMetamodel.util*, declara otras clases útiles para el tratamiento de los modelos, pero que no corresponden con elementos del modelo *core*.

La Figura 40 muestra un diagrama de clases simplificado con las dependencias entre algunas de las diferentes clases que proporcionan el código para dar soporte al metamodelo de trazabilidad.

Este diagrama permite refleja claramente el esquema de generación de código de EMF, los métodos que proporciona para el tratamiento de los modelos, y la jerarquía de herencia entre las distintas clases del modelo. No se entrará en mayores detalles de implementación, puesto que es código generado automáticamente por EMF.

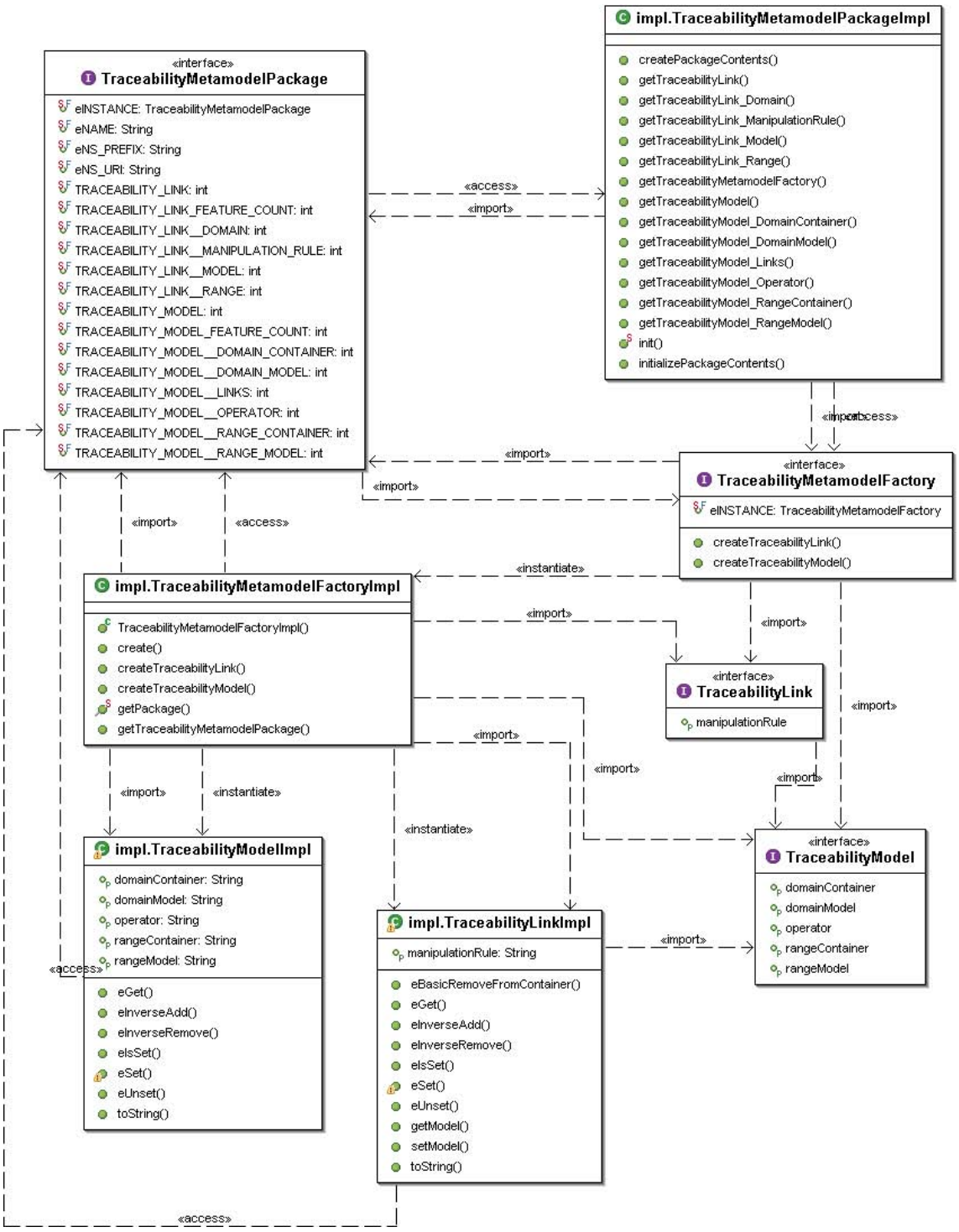

Figura 40: Diagrama de dependencias de clases simplificado de *MOMENT Basic Traceability Metamodel*.

b) es.upv.dsic.issi.moment.traceability.basicmetamodel.edit

El plug-in consta de únicamente de un paquete, *TraceabilityMetamodel.provider*. La Figura 41 muestra el diagrama de clases de este paquete.

Figura 41: Diagrama de clases del paquete *TraceabilityMetamodel.provider*.

La clase *BasicTraceabilityMetamodelEditPlugin* es la encargada de registrar el plug-in al iniciarse y controlar el ciclo de vida del plug-in. La clase *TraceabilityMetamodelItemProviderAdapterFactory* es la que se emplea para proporcionar las interfaces necesarias por los visores y hojas de propiedades.

Las clases *TraceabilityModelItemProvider* y *TraceabilityLinkItemProvider* se encargan de proporcionar los métodos para la edición y visualización de los elementos de un modelo de trazabilidad (modelo y correspondencias, respectivamente).

6.2.1.3. Funcionamiento de los plug-ins.

Estos plug-ins no proporcionan ninguna interacción con el usuario. «*es.upv.dsic.issi.moment.traceability.basicmentamodel*» únicamente permite que al estar instalado, el metamodelo de trazabilidad básico se encuentre registrado automáticamente en la base de datos de EMF como un modelo. De esta forma, se permite la visualización y creación de metamodelos de trazabilidad mediante editores reflexivos. De la misma manera, permite que el proyector de MOMENT del espacio tecnológico Maude a EMF pueda obtener la representación en Maude del modelo como un término del álgebra de MOMENT.

«*es.upv.dsic.issi.moment.traceability.basicmentamodel.edit*», por otra parte, proporciona el soporte para la edición de los modelos de trazabilidad. En este caso, establece la apariencia de los modelos en los editores de trazabilidad, indicando los iconos que se mostrarán los distintos elementos.

Igualmente, personaliza los menús contextuales para la creación de nuevos elementos de los modelos de trazabilidad. La razón se debe a que dado que un metamodelo de trazabilidad personalizado hereda del metamodelo básico, en caso de emplear un editor en árbol para editar el modelo, si se desea añadir un elemento *TraceabilityLink* al modelo en edición, el editor por defecto proporcionará tanto la clase *TraceabilityLink* del metamodelo básico, como del metamodelo personalizado.

El código de soporte para la edición del metamodelo básico de trazabilidad está modificado para que, en caso de que se haga generado código para el metamodelo personalizado, se modifique el comportamiento por defecto y no permita crear elementos del metamodelo básico siendo válidos únicamente los del metamodelo establecido por el usuario. No obstante, si el metamodelo personalizado es creado dinámicamente, no existe forma de evitar esto, ya que EMF emplea la API de acceso genérica en todo el proceso.

6.2.2. Soporte para el metamodelo de trazabilidad personalizado de MOMENT.

Para dar soporte al metamodelo de trazabilidad personalizado de MOMENT se han generado dos plug-ins mediante el generador de código de EMF, de formas similar al metamodelo básico de trazabilidad. Éstos son «*MOMENT Custom Traceability Metamodel*» (*es.upv.dsic.issi.moment. traceability.momentmetamodel*) y «*MOMENT Custom Traceability Metamodel Edit Support*» (*es.upv.dsic.issi.moment.traceability.momentmetamodel.edit*).

6.2.2.1. Descripción.

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

i. Funciones del plug-in.

Este plug-in deberá proporcionar la siguiente funcionalidad.

- Definir en Ecore un ejemplo de metamodelo personalizado de trazabilidad. Proporcionará un ejemplo de modelo básico personalizado, con dos clases, que hereden del metamodelo básico, tal como se describe en la Figura 21 (apartado 5.2.1.1).
- Generar el código apropiado para manejar estos metamodelo en EMF, y los metamodelos que lo conformen. Esto permitirá tratar con ellos en memoria, así cómo proyectarlos al espacio tecnológico Maude, y a su vez, parsearlos desde éste.
- Proveer de los mecanismos independientes de la interfaz para la edición de un modelo de trazabilidad, así como su personalización, que los distinga de otros modelos EMF.

ii. Restricciones.

El metamodelo personalizado debe heredar tal y como la Figura 21 describe. Igualmente, será necesario que el *EPackage* del metamodelo contenga una anotación cuyo nombre sea «*MOMENTTraceabilityMetamodel*», que permitirá identificar que un metamodelo de la base de datos de EMF es un metamodelo de trazabilidad básico.

La validez de un metamodelo de trazabilidad se comprueba de otras maneras más fiables, pero la inclusión de esta anotación permite hacer un primer filtrado de entre todos los modelos EMF mucho más eficiente.

iii. Dependencias.

Obviamente, puesto que heredan del metamodelo de trazabilidad básico para su correcto funcionamiento deberá estar instalado este plug-in (*es.upv.dsic.issi.moment.traceability.basicmetamodel*) en nuestra distribución de Eclipse.

A parte de que se encuentre instalado, si se desea editar, generar código o compilar el plug-in del metamodelo personalizado será necesario disponer del código fuente de *MOMENT Basic Traceability Metamodel* en el espacio de trabajo (el código fuente se incluye en el mismo plug-in del metamodelo básico, y se puede importar fácilmente en el *Workspace* con un sencillo asistente que Eclipse proporciona).

6.2.2.2. Diseño e implementación de los plug-ins.

i. Creación del proyecto EMF y generación de código.

a) Obtención de la implementación del metamodelo de trazabilidad.

Para crear este plug-in deberemos crear en primer lugar el modelo Ecore que herede del metamodelo básico de trazabilidad. Como se ha comentado en la sección de «Dependencias», es necesario disponer del código fuente del metamodelo básico, para incluir el modelo Ecore de éste y poder establecer las relaciones de herencia.

Esto se debe a que, desde el editor, éstas no se pueden establecer sobre modelos cargados en memoria (base de datos de EMF), aunque sí que se muestran si éstas existen. Sin embargo, de forma dinámica desde un programa Java sí que se permite, esta es la forma en la que generan los modelos de trazabilidad personalizados. Se ha implementado un asistente (véase el apartado 6.2.3: «Soporte para la creación automática de un nuevo metamodelo de trazabilidad personalizado.») que permite generar un modelo de trazabilidad personalizado y lo persiste en el correspondiente archivo *.ecore*.

Aunque se puede crear el metamodelo personalizado desde cero, para este caso se ha empleado el asistente arriba mencionado.

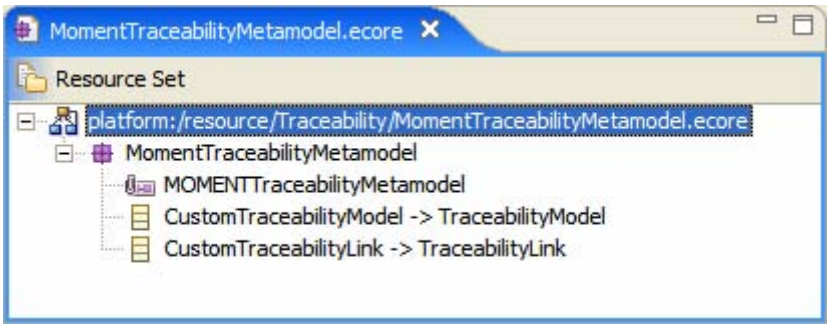

Figura 42: Metamodelo básico de trazabilidad en el editor en árbol de EMF.

Se puede observar que se genera también automáticamente la anotación que permitirá discriminar éste modelo *Ecore* como un metamodelo de trazabilidad válido.

Una vez creado el metamodelo, se puede hacer uso del asistente para crear un nuevo proyecto de EMF al igual que se hizo para el metamodelo básico. El proceso a seguir es muy similar, y está descrito en el Anexo VI. La principal diferencia, eso sí, radicará en la ventana del diálogo de selección de dependencias, en la que deberemos indicar que el modelo para el que estamos generando depende de otro proyecto, como se muestra en la Figura 43.

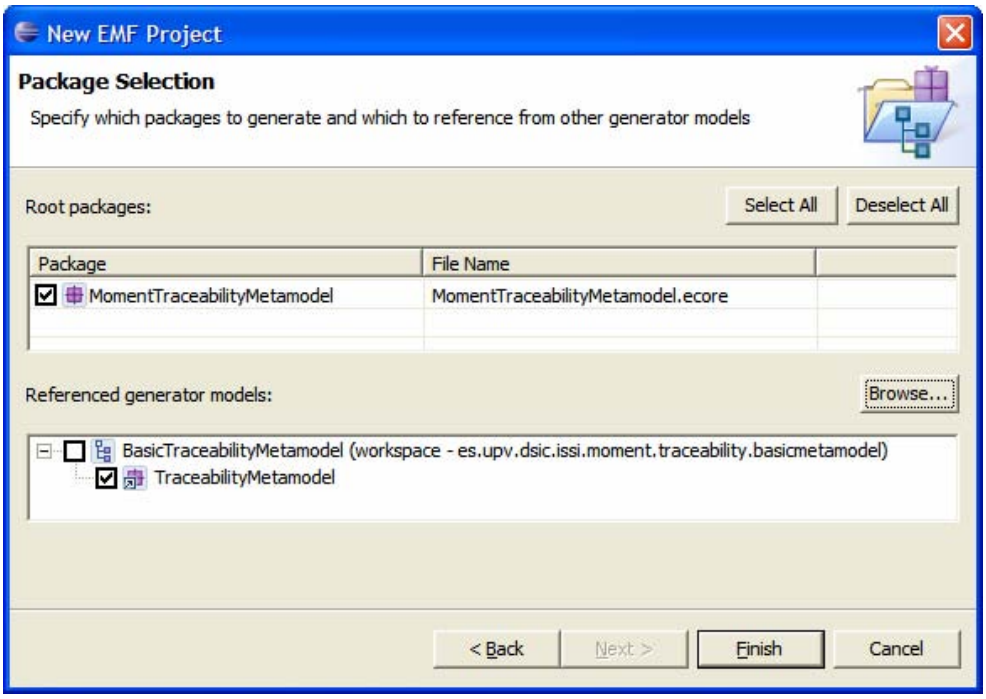

Figura 43: Asistente de nuevo proyecto EMF. Dependencias del modelo personalizado.

Mediante el botón «*Browse…*» se deberá seleccionar el archivo *BasicTraceabilityMetamodel.genmodel* de la carpeta *model* del plug-in *es.upv.dsic.issi.moment.traceability.basicmetamodel*.

Una vez finalizado el asistente, se dispondrá de un proyecto con una estructura similar al del metamodelo básico de trazabilidad.

ii. Estructura del código generado.

El código generado para estos plug-ins es muy similar al código del metamodelo básico de trazabilidad, al igual que las modificaciones realizadas sobre éste. Por lo que comentarlo y entrar en mayor detalle sobre él resulta irrelevante.

6.2.2.3. Funcionamiento de los plug-ins.

Estos plug-ins tampoco proporcionan interacción alguna con el usuario. El plug-in «*es.upv.dsic.issi.moment.traceability.momentmentamodel*» permite que siempre exista cargado en EMF un metamodelo de trazabilidad personalizado, de forma que si éste se ajusta a los requerimientos del usuario se evite la tarea de generar uno personalizado con la funcionalidad básica.

«*es.upv.dsic.issi.moment.traceability.momentmentamodel.edit*», por otra parte, proporciona el soporte para la edición de los modelos de trazabilidad personalizados de MOMENT, estableciendo la apariencia de los modelos en los editores de trazabilidad y sus iconos, al igual que el soporte para edición del metamodelo básico.

6.2.3. Soporte para la creación automática de un nuevo metamodelo de trazabilidad personalizado.

El soporte para la generación de un nuevo metamodelo de trazabilidad, pese a que se relaciona muy íntimamente con el plug-in «*es.upv.dsic.issi.moment.traceability.basicmetamodel*», se ha desarrollado como un único plug-in, denominado «*es.upv.dsic.issi.moment.traceability.newmetamodel*», separado para favorecer la modularidad, y mantendiendo a su vez la estructura de plug-ins por defecto de EMF.

Se ha partido de la implementación de un asistente de creación de nuevo fichero estándar del *Entorno de Desarrollo de Plug-ins* de Eclipse.

6.2.3.1. Descripción.

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

i. Funciones del plug-in.

Este plug-in deberá proporcionar la siguiente funcionalidad.

- Proporcionar soporte gráfico para la creación de un nuevo metamodelo de trazabilidad personalizado, que herede tal y como se muestra en la Figura 21 del metamodelo de trazabilidad básico, y listo para ser utilizado.

- El metamodelo de trazabilidad deberá incluir las anotaciones necesarias que lo identifiquen como un metamodelo válido de trazabilidad.
- El soporte gráfico deberá integrarse perfectamente en el conjunto de herramientas de MOMENT. Esto es, deberá encontrarse junto con el resto de asistentes que MOMENT contribuye a Eclipse.

ii. Restricciones.

Para su correcto funcionamiento, en el metamodelo personalizado que se genere se podrán modificar todos los parámetros que se deseen salvo tres: El nombre de la anotación «*MOMENTraceabilityMetamodel*», la relación de herencia de «*TraceabilityModel*», y la relación de herencia de «*TraceabilityLink*». El resto de parámetros, como la URI del metamodelo, prefijo, nombres de las clases, etc, pueden elegirse libremente.

iii. Dependencias.

Como es de esperar, deberá disponerse del plug-in «*es.upv.dsic.issi.moment.traceability.basicmetamodel*» instalado en la distribución de eclipse para el correcto funcionamiento de este asistente.

6.2.3.2. Arquitectura del plug-in.

La Figura 44 nos muestra la estructura de este plug-in. Se puede observar que consta de dos paquetes. El primero de ellos contiene una única clase, encargada de controlar el ciclo de vida del plug-in.

El segundo contiene dos clases, la primera (*NewTraceabilityMetamodelWizard*) engloba todo el proceso del asistente y en ella se implementan las acciones a realizar cuando finalice. La clase *NewTraceabilityMetamodelWizardPage* implementa la única ventana de que consta el asistente. Más adelante se comentarán estas clases en mayor detalle.

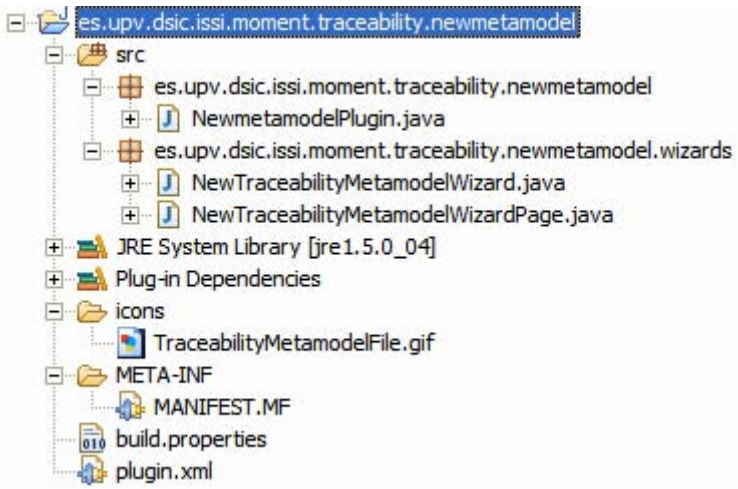

Figura 44: Estructura del plug-in *MOMENT New Traceability Metamodel*.

6.2.3.3. Funcionamiento.

El asistente se puede iniciar desde la ventana para crear un nuevo archivo de Eclipse, en la sección de MOMENT, bajo el título «*New traceability metamodel file*».

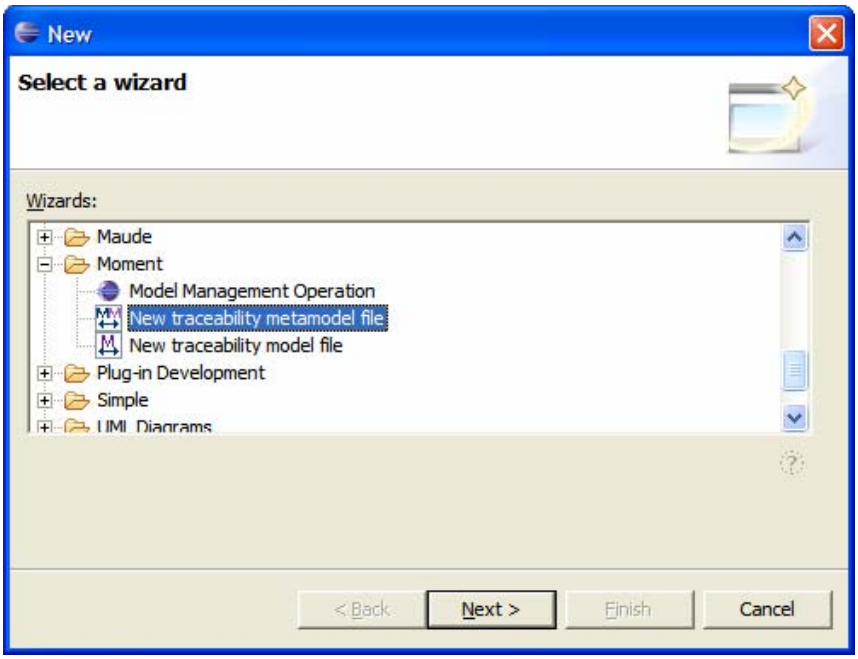

Figura 45: Asistente para crear un nuevo metamodelo de trazabilidad.

Seleccionanado esta opción se nos mostrará una ventana donde podremos determinar donde salvar el nuevo archivo, así como ajustar algunos valores iniciales del nuevo metamodelo, como su URI, nombre, o prefijo.

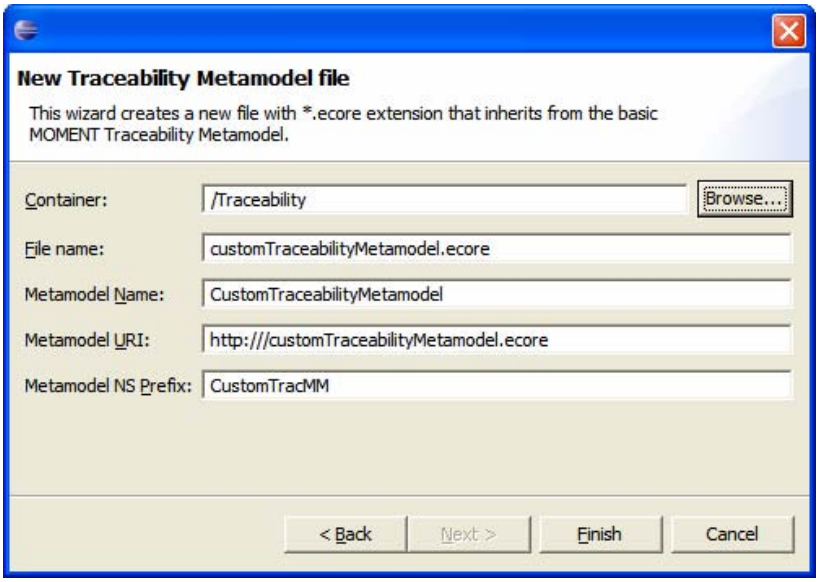

Figura 46: Establecimiento de valores iniciales del nuevo metamodelo.

Tras presionar el botón para finalizar el asistente el nuevo metamodeló se abrirá para su edición en el editor de EMF, de forma que el usuario podrá modificar el metamodelo a su gusto de la forma habitual:

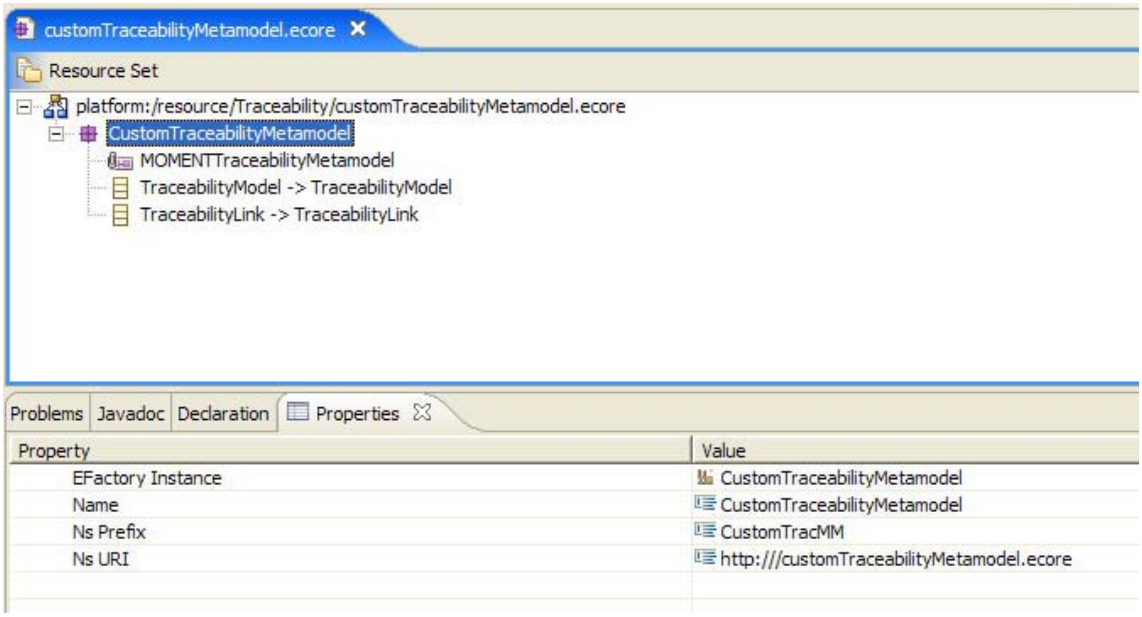

Figura 47: Vista inicial de un nuevo metamodelo de trazabilidad.

Se puede observar que un metamodelo inicialmente proporciona las mismas estructuras para guardar la información sobre una operación que el metamodelo básico.

Cabe reseñar también la anotación que encontramos en el metamodelo, citada ya en diversas ocasiones y que indentifica a este modelo *Ecore* como metamodelo de trazabilidad de MOMENT. Sirve para poder discriminar los metamodelos de trazabilidad válidos entre todos los metamodelos cargados en la base de datos de EMF de forma rápida.

6.2.3.4. Descripción detallada de paquetes y clases.

i. es.upv.dsic.issi.moment.traceability.newmetamodel.

Este paquete únicamente contiene la clase principal del plug-in, encargada de controlar los aspectos globales de éste.

a) Clases.

a.1) NewmetamodelPlugin.

Esta es la clase principal del plug-in. Se encarga de controlar el ciclo de vida del plug-in, así como de realizar las acciones necesarias al activarse y desactivarse.

- **public NewmetamodelPlugin()**

Éste es el constructor por defecto. Establece el valor de la instancia compartida.

- **static NewmetamodelPlugin getDefault()**

Este método estático devuelve la instancia compartida.

- **static org.eclipse.jface.resource.ImageDescriptor getImageDescriptor(String path)**

El método getImageDescriptor() devuelve un descriptor de imagen para la imagen que se encuentre en la ruta relativa al plug-in proporcionada como parámetro.

- **void start(org.osgi.framework.BundleContext context)**

Este método se llama automáticamente al inciarse el plug-in.

- **void stop(org.osgi.framework.BundleContext context)**

Este método se llama automáticamente cuando el plug-in se desactiva.

ii. es.upv.dsic.issi.moment.traceability.newmetamodel.wizards

En este paquete se encuentran las clases que implementan las ventanas del asistente y su funcionalidad.

a) Clases

a.1) NewTraceabilityMetamodelWizard

La clase *NewTraceabilityMetamodelWizard* se encarga de la gestión del asistente en general. Crea las diferentes páginas que lo compone, y al terminar éste realiza las acciones necesarias con los datos recogidos en las distintas ventanas. Extiende la clase *org.eclipse.jface.wizard.Wizard* e implementa la interfaz *INewWizard*.

- **NewTraceabilityMetamodelWizard()**

Este es el constructor por defecto de la clase. Llama al constructor de la clase padre e inicializa el monitor de progreso.

- **void addPages()**

En este método se realiza la creación e inclusión en el asistente de las diversas ventanas de diálogo que lo forman. En este caso, crea una instancia de

NewTraceabilityMetamodelWizardPage y la añade al asistente mediante el método de la clase padre *addPage(IWizardPage page).*

- **private void doFinish(String containerName, String fileName, org.eclipse.core.runtime.IProgressMonitor monitor)**

Este es el método que realiza las acciones. Recibe donde se va a salvar el fichero con el modelo de trazabilidad personalizado, así como su nombre. Dinámicamente crea su contenido y lo salva en la ubicación elegida.

A continuación se muestra el código que permite generar el metamodelo personalizado de forma dinámica mediante la API de EMF. Se comprueba el uso mixto de la API genérica de EMF para crear las clases, y el código generado para establecer las relaciones de herencia.

```
// Obtenemos el metamodelo de trazabilidad básico
TraceabilityMetamodelPackageImpl
    theTraceabilityMetamodelPackage =
        (TraceabilityMetamodelPackageImpl)EPackage.Registry.INSTANCE.getEPackage(
            TraceabilityMetamodelPackage.eNS_URI
        );

// Obtención de una factoría genérica de Ecore.
EcoreFactory ecoreFactory = EcoreFactory.eINSTANCE;

// Creamos una clase Ecore, y establecemos el nombre y la relación de herencia.
EClass tracModelClass = ecoreFactory.createEClass();
tracModelClass.setName("CustomTraceabilityModel");
tracModelClass.getESuperTypes().
    add(theTraceabilityMetamodelPackage.getTraceabilityModel());

// Creamos otra clase de ecore, y establecemos su nombre y relación de herencia
EClass tracLinkClass = ecoreFactory.createEClass();
tracLinkClass.setName("CustomTraceabilityLink");
tracLinkClass.getESuperTypes().
    add(theTraceabilityMetamodelPackage.getTraceabilityLink());

// Creamos el paquete del metamodelo personalizado y añadimos los elementos
EPackage tracMetaModelMPackage = ecoreFactory.createEPackage();
tracMetaModelMPackage.setName(metamodelName);
tracMetaModelMPackage.setNsPrefix(metamodelNSPrefix);
tracMetaModelMPackage.setNsURI(metamodelURI);
tracMetaModelMPackage.getEClassifiers().add(tracModelClass);
tracMetaModelMPackage.getEClassifiers().add(tracLinkClass);

// Creamos y añadimos la anotación identificativa
EAnnotation eAnnotation = EcoreFactory.eINSTANCE.createEAnnotation();
eAnnotation.setSource("MOMENTTraceabilityMetamodel");
tracMetaModelMPackage.getEAnnotations().add(eAnnotation);

// Creamos el ResourceSet y salvamos el modelo a disco
ResourceSet resourceSet = new ResourceSetImpl();
Resource resS = resourceSet.createResource(
        URI.createPlatformResourceURI(file.getFullPath().toString()));
resS.getContents().add(tracMetaModelMPackage);
try {
    resS.save(null);
} catch (IOException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

- **void init(org.eclipse.ui.IWorkbench workbench, org.eclipse.jface.viewers.IStructuredSelection selection)**

Establece el conjunto de elementos seleccionados en el banco de trabajo en el momento de lanzar el asistente.

- **boolean performFinish()**

Este método se llama al pulsar el botón «*Finish*» del asistente. Se encarga de consultar todos los valores establecidos en las diferentes páginas, y de realizar la llamada al método *doFinish(…)*.

- **private void throwCoreException(String message)**

Este método, empleado por *doFinish(…)* es el encargado de lanzar la pertinente excepción, indicando que ha sido el este plug-in el que ha fallado con el mensaje especificado por *message*.

a.2) NewTraceabilityMetamodelWizardPage.

En esta clase que extiende a la clase *org.eclipse.jface.wizard.WizardPage*, establece la interfaz de usuario, que recogerá todos los parámetros configurables inicialmente para el metamodelo de trazabilidad personalizado.

- **NewTraceabilityMetamodelWizardPage(**
  org.eclipse.jface.viewers.ISelection selection)

Este es el constructor por defecto, llama el método constructor de la clase padre, y establece el título y la descripción de la ventana de diálogo.

- **void createControl(org.eclipse.swt.widgets.Composite parent)**

Este es el método que creará todos los botones, etiquetas y casillas editables que recogerán los datos a introducir por el usuario.

- **private void dialogChanged()**

Este método privado es el encargado de consultar todos los valores introducidos por el usuario y validarlos. Se ejecuta automáticamente cada vez que el contenido de un control varía. Debe realizar una llamada a *updateStatus("mensaje")* en caso de que encuentre un error. Si todos los valores son correctos, deberá ejecutar *updateStatus(null)*.

- **String getContainerName()**

Este método devuelve un string, con la ruta de la carpeta contenedora para el nuevo fichero.

- **String getFileName()**

El método *getFileName()* devuelve el nombre del archivo que contendrá el nuevo metamodelo.

- **String getMMName()**

Este método devuelve el nombre que el usuario ha especificado para el nuevo metamodelo personalizado.

- **String getMMNSPrefix()**

El método *getMMNSPrefix()* devuelve el prefijo introducido por el usuario para el nuevo metamodelo de trazabilidad.

- **String getMMURI()**

Este método devuelve la URI que se ha especificado en la casilla correspondiente, para el Nuevo metamodelo personalizado de trazabilidad.

- **private void handleBrowse()**

Este método crea una ventana de selección de contenedor estándar para establecer el Nuevo valor para el campo de texto etiquetado como «*Container:* ».

- **private void initialize()**

El método *initilize()* comprueba si la selección actual del banco de trabajo es un contenedor válido para ser usado. Si lo es, este será el contenedor que aparecerá seleccionado inicialmente al crear la ventana de selección de contenedor.

- **private void updateStatus(String message)**

Este método establece el mensaje de error a *message*. El mensaje de error se muestra en la parte superior de la ventana indicando la descripción de los fallos en los datos introducidos. Si existen errores (*message tiene valor distinto de null*) se realiza una llamada *setPageComplete(false)*, que impide que se avance en el asistente. Si no existen fallos, se ejecuta *setPageComplete(true)* permitiendo que el asistente continue (y finalice en este caso, ya que esta es la única página).

6.2.4. Editor de modelos de trazabilidad.

Este es el último plug-in desarrollado en el conjunto de las herramientas proporcionadas por MOMENT para gestión de la trazabilidad. Engloba el editor de trazabilidad así como el asistente necesario para crear un nuevo modelo de trazabilidad de forma automática.

6.2.4.1. Descripción.

En esta sección se presenta una descripción a alto nivel del plug-in. Se presentarán las necesidades a las que se deben dar soporte, las funciones que debe

realizar, los factores que restringirán su uso, y otras cuestiones que afecten al desarrollo del mismo.

i. Funciones del plug-in.

Este plug-in deberá proporcionar la siguiente funcionalidad.

- Proveer a la herramienta MOMENT para visualizar modelos de trazabilidad, junto con sus modelos dominio y rango.
- Permitir navegar los modelos de trazabilidad, mostrando automáticamente y de forma sencilla las relaciones entre los elementos dominio y rango de una o varias correspondencias (*TraceabilityLinks*).
- Proporcionar los mecanismos necesarios para modificar los modelos de trazabilidad, pudiendo crear enlaces de forma manual mediante una interfaz intuitiva.
- Se deben suministrar los mecanismos pertinentes para la creación de un nuevo modelo de trazabilidad, que conforme un metamodelo de trazabilidad personalizado válido de MOMENT.

  Estos metamodelos deben poder tener una estructura cualquiera (siempre que se cumplan las condiciones de herencia del metamodelo básico de trazabilidad) y podrán haberse registrado en EMF bajo cualquiera de las formas posibles (mediante código generado o de forma dinámica).

- Toda la funcionalidad proporcionada debe estar perfectamente integrada con el resto de herramientas que conforman el framework de MOMENT.

ii. Dependencias.

Para la correcta visualización de un modelo de trazabilidad deberá ser necesario que estén cargados en EMF tanto los metamodelos de trazabilidad (el básico, y el personalizado el cual conforma el modelo) como los metamodelos de los modelos dominio y rango. Es por esto, que será necesario como mínimo, que estén instalados los plug-ins «*es.upv.dsic.issi.moment. traceability.basicmetamodel*» y «*es.upv.dsic.issi.moment.traceability.basicmetamodel.edit*».

Igualmente sera necesario que esté instalado el soporte para el metamodelo de trazabilidad personalizado proporcionado por MOMENT. Esto son los plug-ins «es.upv.dsic.issi.moment.traceability.momentmetamodel» y «es.upv.dsic.issi.moment.traceability.momentmetamodel.edit».

Para el resto de metamodelos empleados (incluido el de trazabilidad personalizado, si éste no es el proporcionado por defecto por MOMENT), será tarea del usuario realizar los pasos necesarios necesarios para que éstos se encuentren cargados en EMF.

En este punto resulta especialmente útil la opción del menú contextual de MOMENT «*Register model*»; proporcionada por el plug-in «Register EMF model»

(*es.upv.dsic.issi.moment.registerEMF*), y requerido para el correcto funcionamiento de este plug-in. Se incluye en el conjunto de utilidades «*MOMENT Extensions*», y permite cargar en EMF un modelo *Ecore* cualquiera sin necesidad de generar el pertinente plug-in para su posterior instalación.

6.2.4.2. Arquitectura del plug-in.

Este plug-in consta de dos paquetes independientes entre si, «*es.upv.dsic.issi.moment.traceability.ui.editor*» y «*es.upv.dsic.issi.moment.traceability.ui.editor.wizards*».

i. es.upv.dsic.issi.moment.traceability.ui.editor

El primero de ellos contiene las clases que implementan el editor de trazabilidad. La Figura 48 nos muestra el diagrama de clases de este paquete. Las clases *TraceabilityEditorPlugin*, *TraceabilityEditor* y *TraceabilityActionBarContributor* han sido generadas automáticamente mediante el generador de código para un editor de EMF. El código de este editor es muy simple, y proporciona una funcionalidad similar al editor reflexivo por defecto de EMF.

La única diferencia con éste estriba en que en el constructor de la clase, para un modelo dado, se añaden a sus *Adapter Factories*, a parte de los genéricos de EMF, los generados en los plug-ins de soporte para la edición. No obstante, este segmento de código será modificado para añadir el soporte para la edición de modelos de trazabilidad que conforman el metamodelo básico así como el personalizado de MOMENT.

Son numerosas las modificaciones realizadas sobre este código. Entre ellas encontramos la eliminación de las páginas del editor innecesarias (inicialmente es un editor multipágina), así como la configuración visual de este, ya que se debe permitir la visualización de tres modelos simultáneamente en pantalla.

También se ha modificado el comportamiento del editor, modificando y añadiendo los métodos para controlar su comportamiento en los cambios de selección. Cuando se selecciona uno o varios elementos de cualquiera de los tres modelos, se navegan los dos restantes, y se resaltan los elementos de éstos relacionados con el/los elemenos seleccionados. En la sección 0 se entrará en mayor detalle.

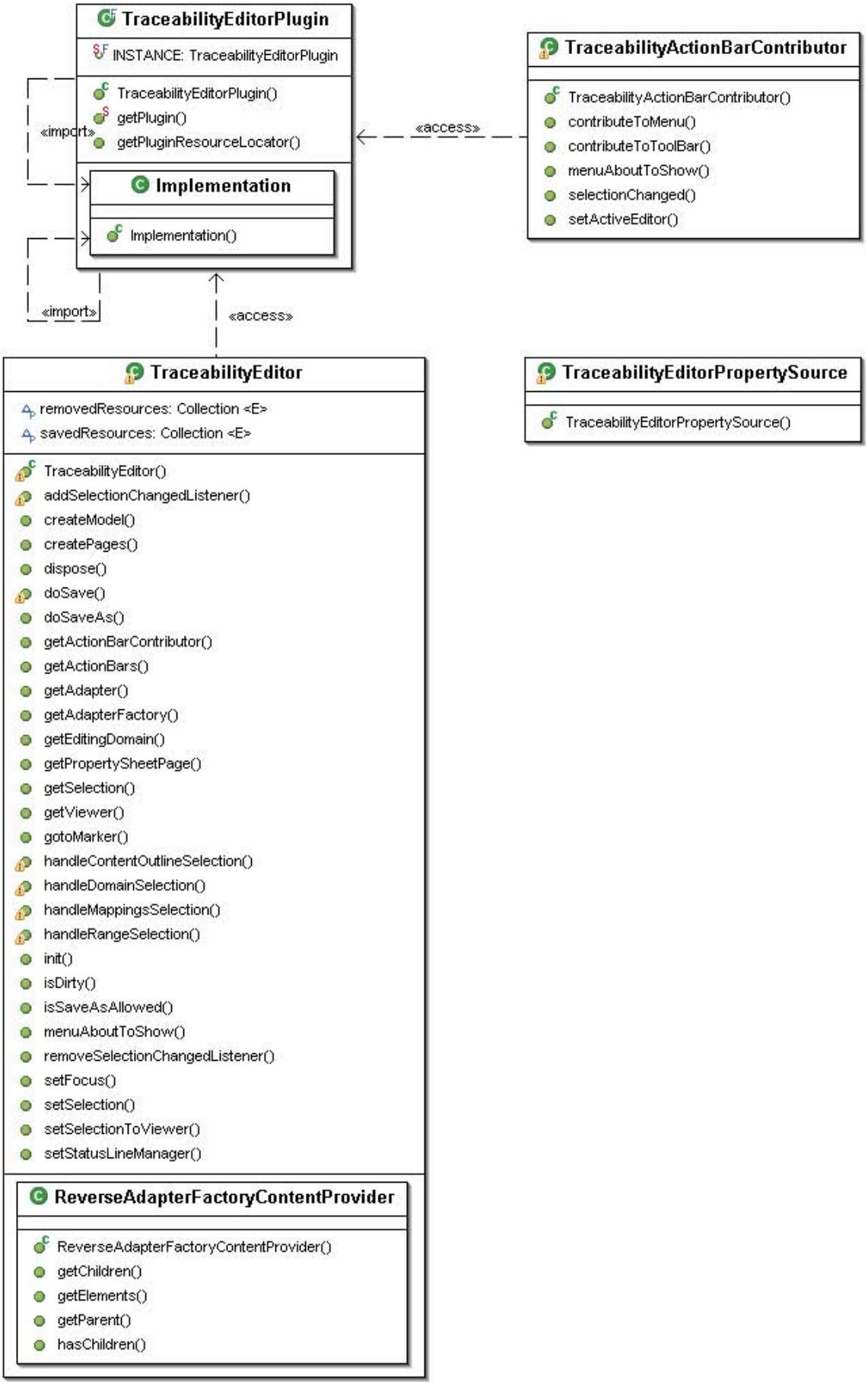

Figura 48: Diagrama de clases del editor de trazabilidad.

Por último, cabe mencionar la clase *TraceabilityEditorPropertySource*. Ésta extiende de *org.eclipse.emf.edit.ui.provider.AdapterFactoryContentProvider*. Es la clase encargada de gestionar los datos mostrados en la hoja de propiedades. Extiende la clase empleada por defecto, modificando su comportamiento. La necesidad de modificar el comportamiento que proporciona el editor inicialmente es sencilla.

Cuando se selecciona un elemento *TraceabilityLink*, en la hoja de propiedades se pueden establecer qué elementos son los elementos dominio y cuales son los elementos rango. Esto se hace mediante la selección de estos en una ventana de diálogo. Por defecto, ésta muestra todos los elementos de los modelos cargados en memoria, por lo que podrían añadirse a la lista de elementos dominio elementos del modelo rango, creando un modelo incoherente.

Extendiendo la clase *org.eclipse.emf.edit.ui.provider.AdapterFactory.ContentProvider* podemos realizar un filtrado de los elementos que se van a mostrar, permitiendo que solo se seleccionen aquellos válidos.

ii. es.upv.dsic.issi.moment.traceability.ui.editor.wizards

El segundo paquete, contiene las clases que definen el asistente de creación de un nuevo modelo. En la Figura 49 se muestra su diagrama de clases UML.

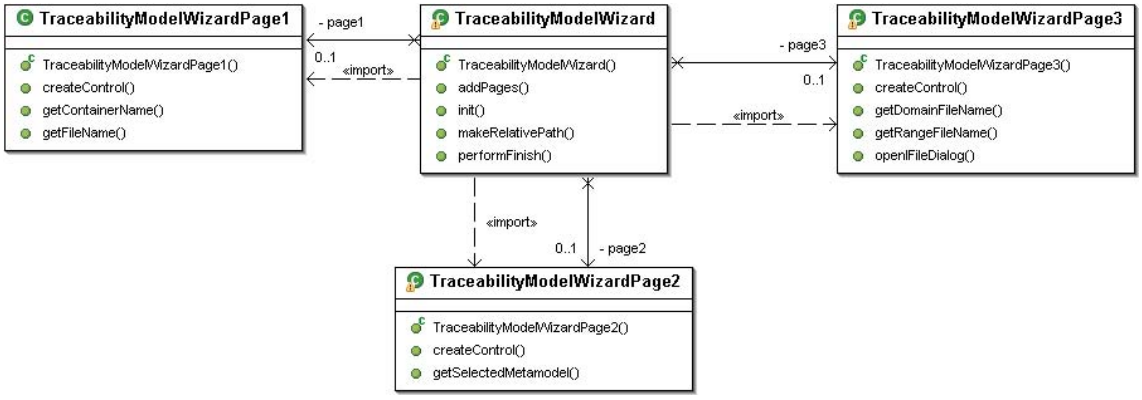

Figura 49: Diagrama de clases del asistente para nuevo modelo de trazabilidad.

Como se puede observar en la figura, el asistente consta de tres páginas. En la primera se obtienen los datos sobre el fichero nuevo a crear, que contendrá el modelo. Esta hoja del asistente está implementada de la forma habitual y no contiene código especialmente interesante que comentar.

La segunda, recoge el metamodelo de trazabilidad que conformará nuestro metamodelo. Para ello se muestra una tabla donde se muestran todos los metamodelos de trazabilidad. Para ello, se recogen todos los metamodelos cargados en la base de datos de EMF de la siguiente forma:

```
private void setTableContents() {

    EPackage.Registry registry = EPackage.Registry.INSTANCE;

    tableMetamodels.removeAll();

    for(Object key : registry.keySet().toArray()) {
        try {
            if (isValidTraceabilityMetamodel((EPackage)registry.
```

```
                                                    getEPackage((String) key))) {
                TableItem item = new TableItem(tableMetamodels, 0);
                item.setText(new String[] {
                        ((EPackage)registry.getEPackage((String) key)).getNsPrefix(),
                        ((EPackage)registry.getEPackage((String) key)).getNsURI()
                        });
            }
        } catch(Throwable t) {}
    }
}
```

Se puede comprobar que se comprueba si el metamodelo que se está consultando es un metamodelo de trazabilidad personalizado válido, mediante la llamada a *isValidTraceabilityMetamodel()*. Este método únicamente comprueba si existe la anotación que lo identifica como un metamodelo válido. Esto es mucho más rápido que comprobar si respetan la jerarquía de herencia impuesta por el metamodelo básico.

```
private boolean isValidTraceabilityMetamodel(EPackage pkg) {
    return (pkg.getEAnnotation("MOMENTTraceabilityMetamodel") != null);
}
```

Esta hoja, además, permite seleccionar un metamodelo personalizado que no está cargado en EMF y para el que no existe código generado. Una vez seleccionado, se carga en memoria, y se añade a la lista de metamodelos de trazabilidad disponibles.

La tercera, por último, establece la configuración inicial de nuestro modelo, esto es, los modelos dominio y rango. Para ello se ha implementado un diálogo de selección de fichero personalizado, ya que Eclipse no proporciona ninguno predefinido que se adecue a las necesidades.

Por último, como ocurre generalmente, la clase *TraceabilityModelWizard* es la encargada de lanzar el asistente, crear las páginas, y realizar el trabajo final de creación del modelo. Al igual que al crear un nuevo metamodelo de trazabilidad, para crear el modelo se hace uso de la API genérica de EMF.

A continuación mostraremos el código que permite generar el nuevo modelo. En primer lugar, se consulta el registro de modelos de EMF, y obtenemos el *EPackage* correspondiente al metamodelo de trazabilidad seleccionado.

```
EPackage.Registry registry = EPackage.Registry.INSTANCE;
EPackage pkg = ((EPackage) registry.getEPackage(metamodelURI));
```

A continuación, se consultan los elementos del modelo, en busca de aquel que hereda de la clase *TraceabilityModel*, del metamodelo básico de trazabilidad, y se guarda su valor en *tracMdl*. Esto es necesario puesto que deberemos crear un *EObject* de esta clase.

```
EClass tracMdl = null;

for (Iterator classifiers = pkg.getEClassifiers().iterator(); classifiers.hasNext(); ) {
    EClassifier eClassifier = (EClassifier)classifiers.next();
    if (eClassifier instanceof EClass) {
        EClass eClass = (EClass)eClassifier;
        if (!eClass.isAbstract() &&
                eClass.getEAllSuperTypes().contains(
                    (EClassImpl)TraceabilityMetamodelFactoryImpl.eINSTANCE.
                        createTraceabilityModel().eClass())) {
            tracMdl = eClass;                       }
    }
}
```

```
 if (tracMdl == null) {
    MessageDialog.openError(null,
            "Invalid Traceability Metamodel",
            "The selected custom traceability metamodel file" +
            "is invalid. It doesn't inherit from the Basic Traceability Metamodel.");
 }
```

Una vez se obtiene la clase del modelo de trazabilidad, se crea un nuevo modelo *Ecore*. Igualmente se crea un objeto de la clase *tracMdl* y se establecen los valores para los modelos dominio y rango. Finalmente, se añaden al nuevo modelo Ecore, y se salva en disco.

```
// Create a resource set
//
ResourceSet resourceSet = new ResourceSetImpl();

// Get the URI of the model file.
//
URI fileURI = URI.createPlatformResourceURI(file.getFullPath().toString());

// Create a resource for this file.
//
Resource res = resourceSet.createResource(fileURI);

// Add the initial model object to the contents.
//
EObject rootObject = EcoreUtil.create(tracMdl);

//ResourcesPlugin.getWorkspace().getRoot().
rootObject.eSet(tracMdl.getEStructuralFeature("domainModel"),
                makeRelativePath((Path)file.getFullPath(),
                new Path(domainModel)).toString());

rootObject.eSet(tracMdl.getEStructuralFeature("rangeModel"),
                makeRelativePath((Path)file.getFullPath(),
                new Path(rangeModel)).toString());

if (rootObject != null) {
    res.getContents().add(rootObject);
}

try {
    res.save(null);
} catch (IOException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

6.2.4.3. Funcionamiento del plug-in.

i. Creación de un nuevo modelo de trazabilidad.

A continuación vamos a mostrar el proceso de creación de un nuevo modelo de trazabilidad. Para ello, nos basaremos en los modelos del caso de estudio. Tal y como se comentó en la sección 5.5.2, en el ejemplo del caso de estudio, empleando el operador de propagación de cambios original (*PropagateChanges*), si no se incluyen en un modelo de trazabilidad los objetos creados al modificar manualmente la base de datos, estos cambios no se verán reflejados tras la ejecución del operador.

Ante esta situación se plantearon dos posibles soluciones. La primera de ellas, añadir estos elementos al modelo de trazabilidad de forma «*artificial*», indicando que estos elementos se han generado a partir del elemento raíz del modelo dominio. La segunda, es la empleada en el ejemplo, y consiste en crear un nuevo operador que incluye los elementos no contemplados en el modelo origen.

Para ilustrar el funcionamiento y la utilidad del asistente, y el editor, mostraremos cómo se debería haber adoptado la primera solución: documentar la generación de nuevos elementos.

a) Inicio del asistente de Nuevo modelo de trazabilidad.

El asistente de creación de un nuevo modelo de trazabilidad se puede iniciar mediante la combinación de teclas *CTRL+N* o mediante el menú «*File → New → Other*». Encontraremos la entrada en el grupo de asistentes de MOMENT, como muestra la Figura 50.

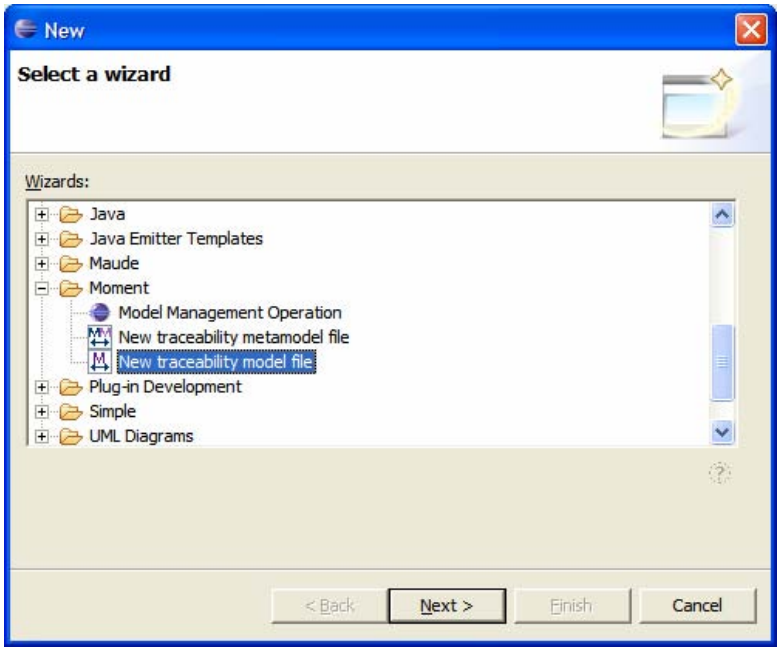

Figura 50: Inicio del asistente de creación de nuevo modelo de trazabilidad.

b) Datos del archivo del modelo de trazabilidad.

Tras iniciar el asistente, los primeros datos que se solicitarán son el contenedor que albergará el archivo, y su nombre.

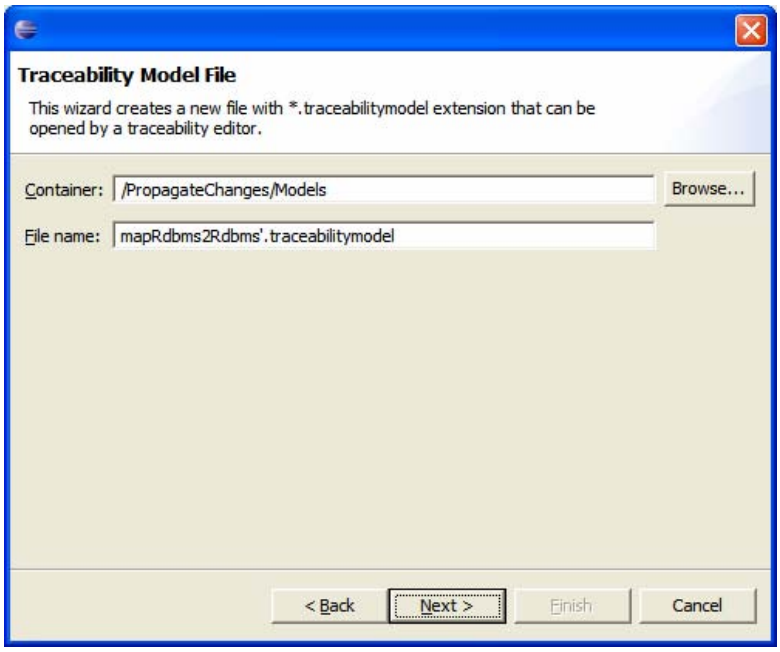

Figura 51: Asistente de creación de nuevo modelo de trazabilidad. Página 1.

c) Selección del metamodelo de trazabilidad personalizado.

A continuación, se muestra la pantalla de selección del metamodelo de trazabilidad personalizado que conformará nuestro modelo.

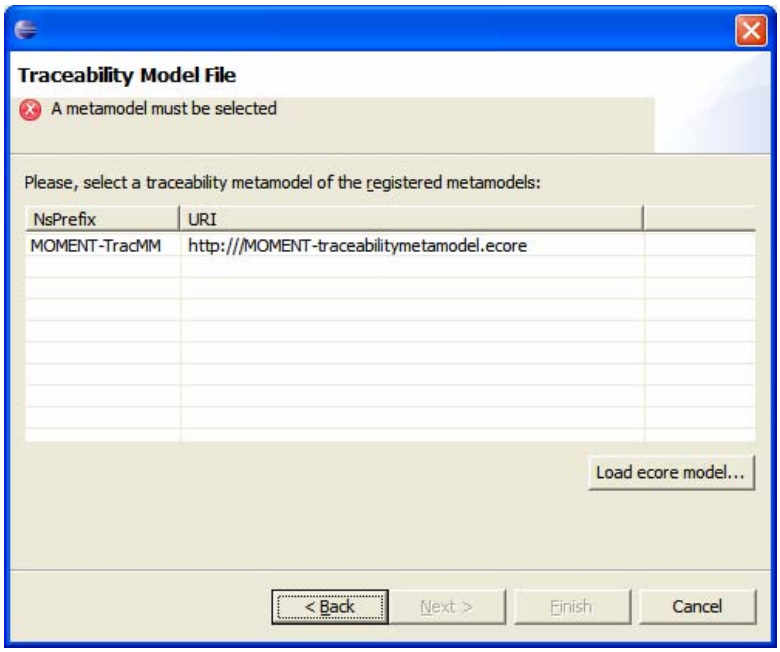

Figura 52: Asistente de creación de nuevo modelo de trazabilidad. Página 2.

Como se observa, inicialmente solo se muestra el metamodelo personalizado de MOMENT si el usuario no ha registrado anteriormente otro propio. En todo caso, si lo desea, mediante el botón «*Load ecore model…*» se abre un nuevo diálogo que permite seleccionar cualquier otro, si se dispone del archivo ecore correspondiente, como muestran las Figura 53 y Figura 54.

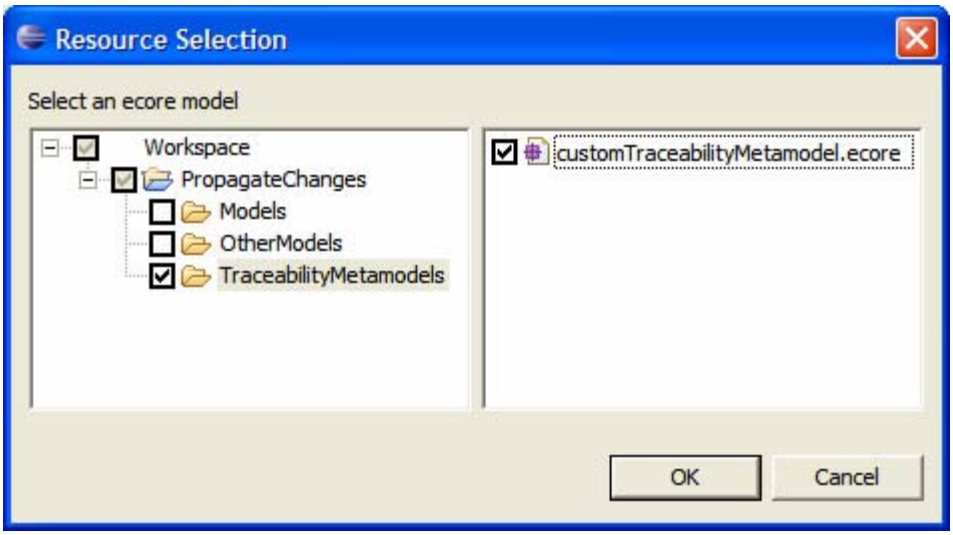

Figura 53: Selección de un metamodelo de trazabilidad personalizado.

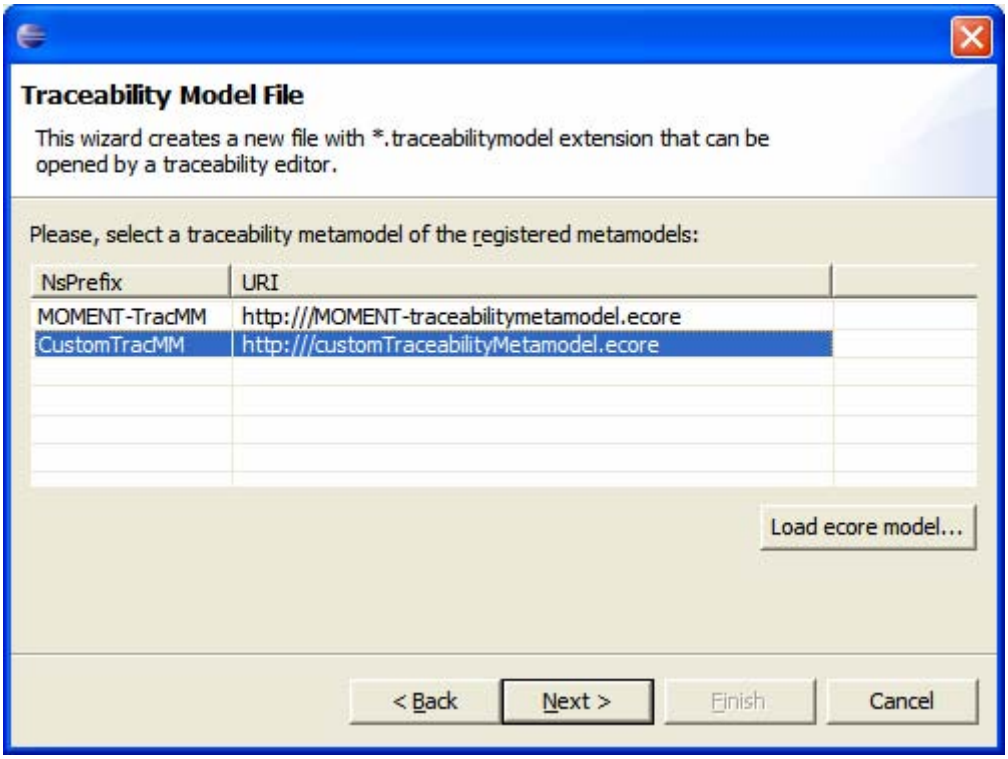

Figura 54: Nueva vista del asistente de creación de nuevo modelo de trazabilidad.

En nuestro caso, no obstante, optaremos por crear un modelo de trazabilidad que conforme el metamodelo personalizado de MOMENT, ya que para éste existe código generado y nos proporciona la funcionalidad suficiente.

d) Establecimiento de los modelos dominio y rango.

La última página del asistente nos permite seleccionar los archivos con los modelos dominio y rango, mediante una ventana de selección de archivos, como se muestra en la Figura 55.

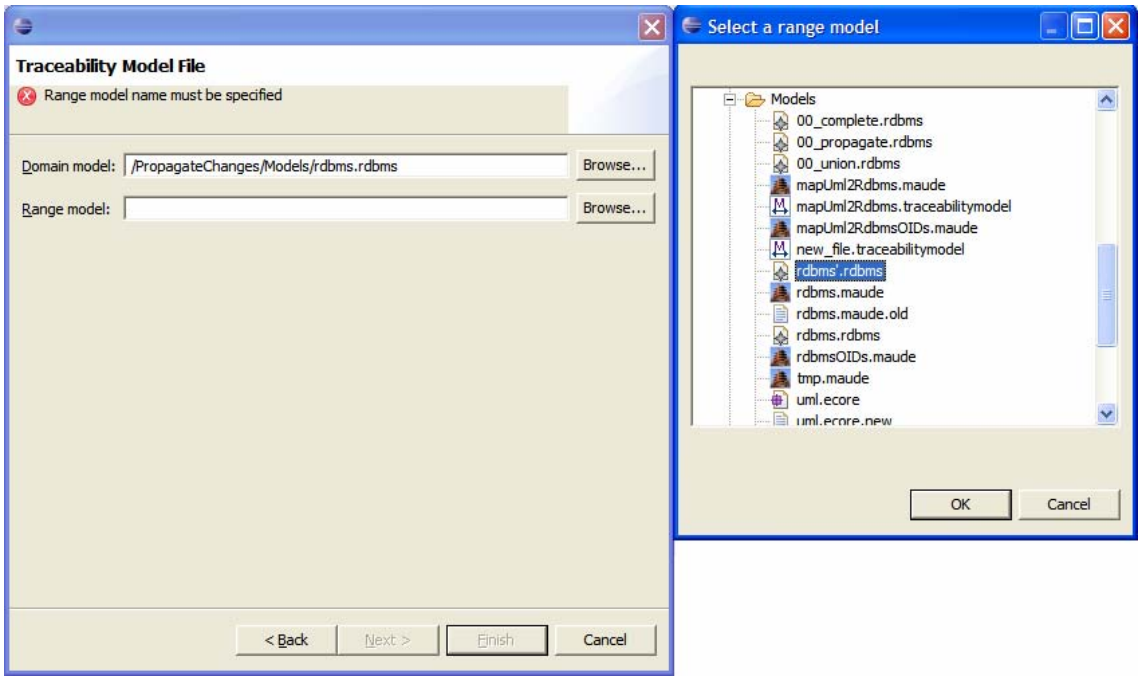

Figura 55: Asistente de creación de nuevo modelo de trazabilidad. Página 3.

e) Finalización del asistente

Tras indicar estos últimos datos, y presionar el botón de finalizar, se abre el editor de trazabilidad permitiendo modificar el resto de parámetros del modelo mediante la hoja de propiedades (nombre de los modelos dominio y rango, operador aplicado, etc), como se muestra a continuación.

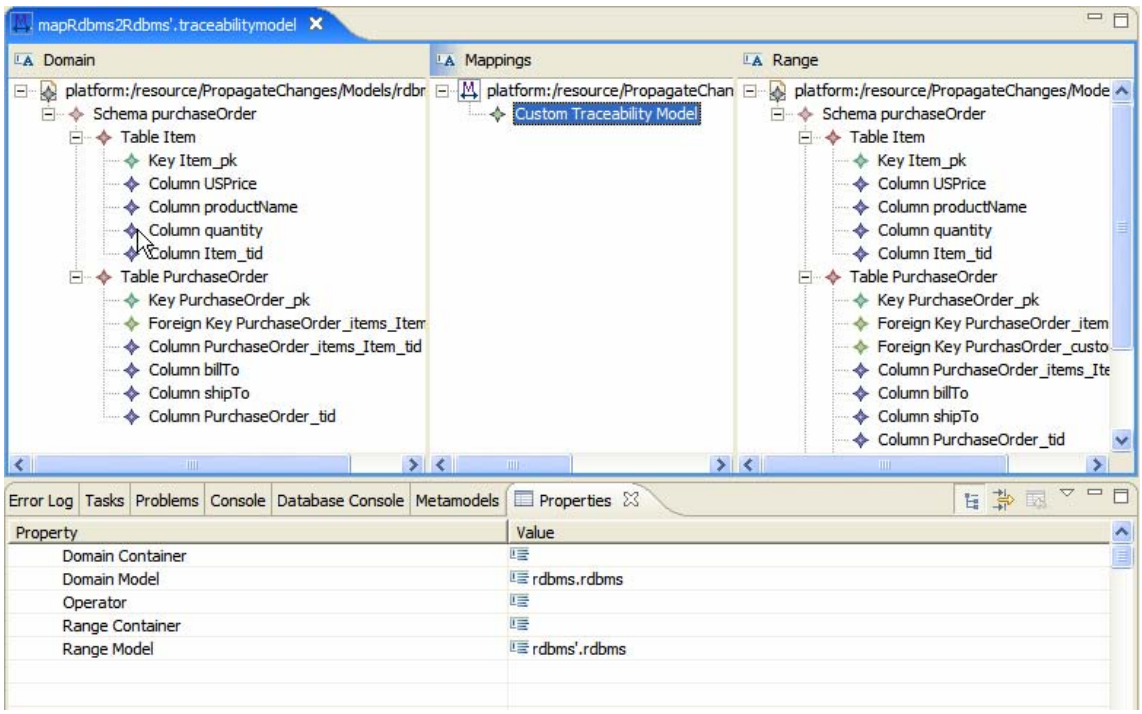

Figura 56: Vista del nuevo modelo de trazabilidad.

ii. Editor de modelos de trazabilidad.

El editor de modelos de trazabilidad permite visualizar, navegar y editar los modelos de trazabilidad. Su aspecto es parecido al editor proporcionado por el plug-in *Model Weaver* del proyecto ATL [ATL].

a) Edición de modelos de trazabilidad.

Para mostrar el funcionamiento del editor en la edición de modelos de trazabilidad, continuaremos con el modelo generado en la sección anterior. Utilizaremos el editor para crear los enlaces de trazabilidad que nos permitan documentar la transformación manual del modelo relacional RDB en RDB' (modelo *mapRDB2RDB'* comentado en el apartado 5.5: «*Aplicación de MOMENT al caso de estudio.*»).

Para crear una nueva correspondencia entre el modelo dominio y el modelo rango seleccionaremos «*Moment Traceability Link*» del submenú «*New Child*». Este submenú lo encontramos (teniendo seleccionado el elemento *Moment Traceability Model* del modelo trazabilidad) tanto en el menú de la barra de menús asociado al editor, como en el menú contextual que aparece al hacer clic derecho. La Figura 57 nos muestra cómo.

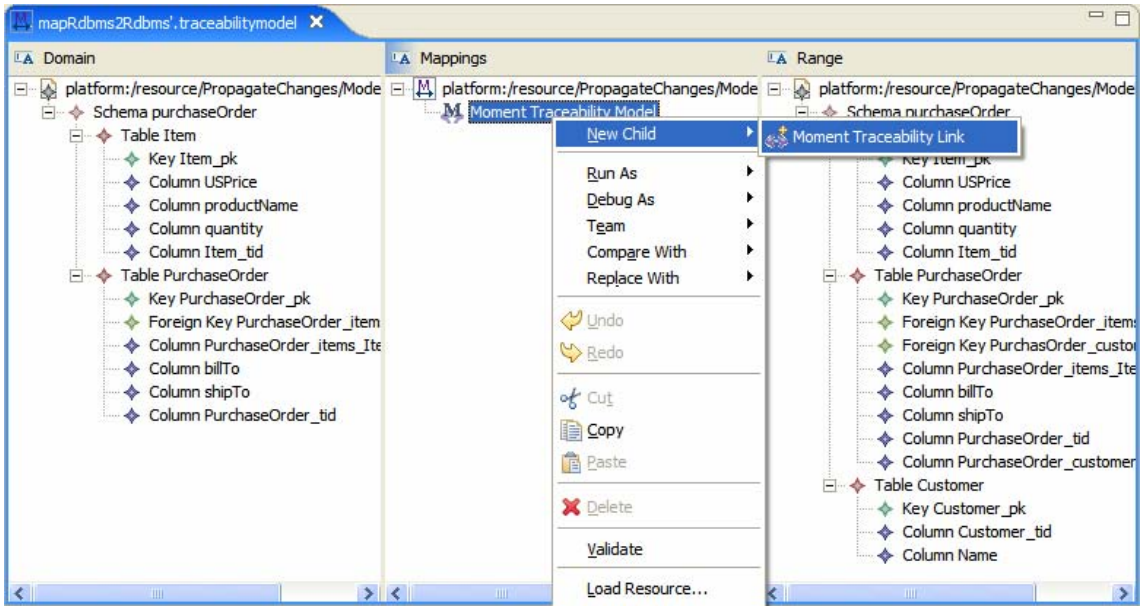

Figura 57: Inserción de un nuevo enlace de trazabilidad en un modelo.

En nuestro modelo de trazabilidad aparecerá un nuevo elemento «*Moment Traceability Link*». Seleccionándolo, podremos ajustar sus valores mediante la hoja de propiedades. En este caso, indicaremos la creación de la nueva tabla «*Customer*», esto comprende la tabla (elemento *Table*), así como su clave primaria (*Key*) y la columna asociada a la clave (*Column*).

Así puesta, esta correspondencia tendrá como dominio al elemento «*Schema purchaseOrder*», y como elementos rango a «*Table Customer*», «*Key Customer_pk*» y «*Column Customer_tid*».

La Figura 58 nos muestra la vista del editor junto a la hoja de propiedades que aparece cuando se selecciona un elemento *TraceabilityLink*.

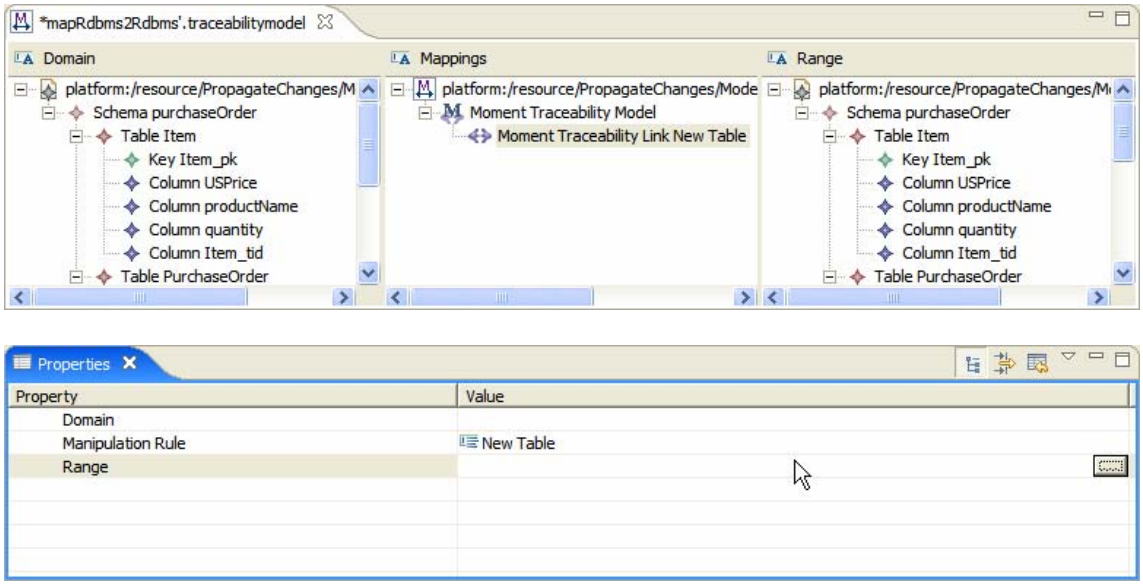

Figura 58: Muestra del elemento añadido, junto a la hoja de propiedades.

Como se puede observar, en este caso, ya hemos añadido como regla de manipulación el texto «*New Table*», que nos permite identificar en la vista de árbol las diferentes transformaciones realizadas.

También es de notar, que cuando se pinchar sobre la línea *Range* (o *Domain*), aparece uno botón a la derecha. Este botón nos permite abrir una ventana en la que podremos seleccionar los elementos del modelo rango (o dominio, según corresponda) que son afectados por la regla de manipulación seleccionada. En este caso, añadiremos los elementos comentados anteriormente:

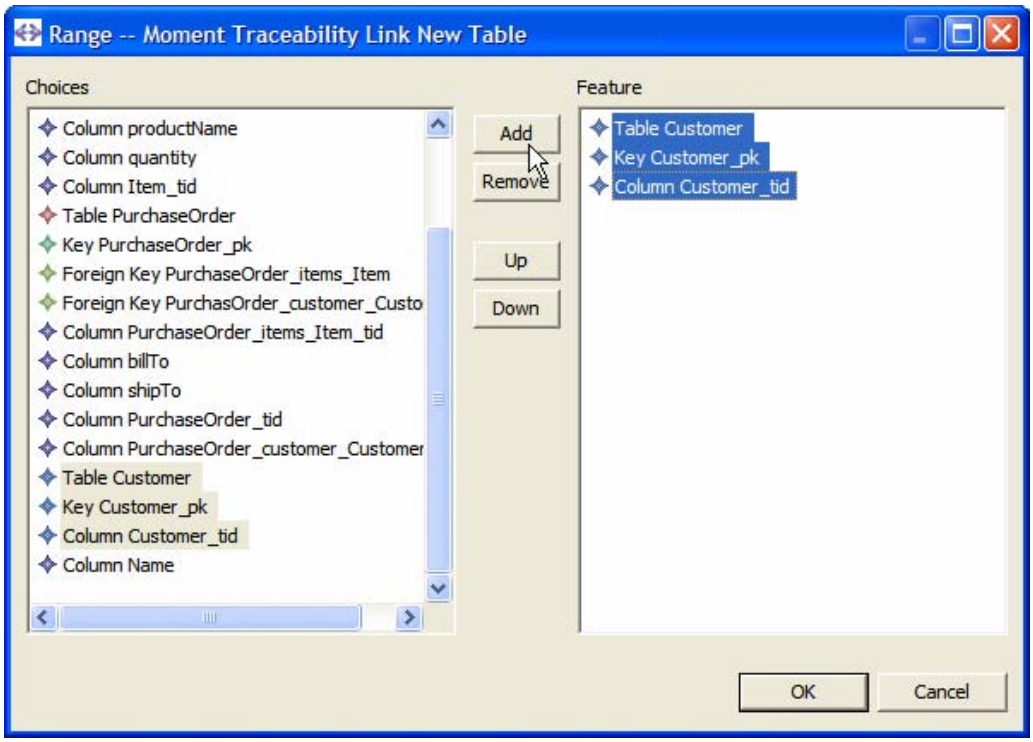

Figura 59: Selección de elementos rango.

La Figura 60 muestra el aspecto final de la hoja de propiedades, una vez establecidos los valores necesarios.

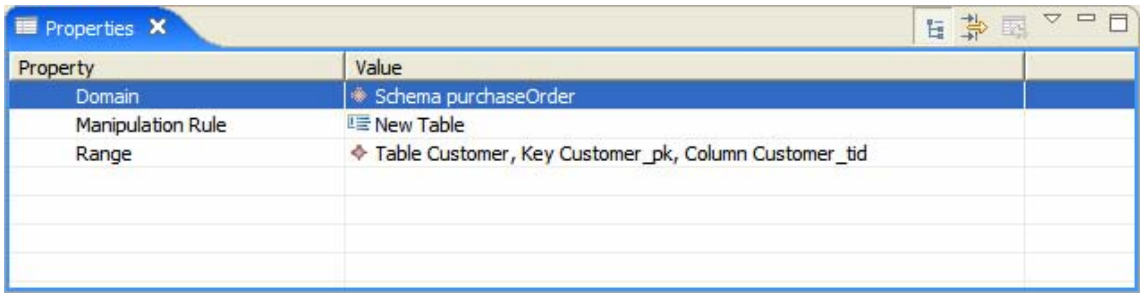

Figura 60: Vista de propiedades del elemento *Traceability Link* de ejemplo.

Para terminar de completar el modelo de trazabilidad que documente todos los cambios realizados, se proseguiría con el proceso de la misma manera que se ha mostrado.

b) Navegación de modelos de trazabilidad.

Para ilustrar las funciones de navegación del editor de trazabilidad emplearemos el modelo generado en la primera transformación del ejemplo de propagación de cambios de la sección 5.1.3, ya que resulta más descriptivo.

La ventana del editor consta de tres paneles: *Domain*, *Mappings* y *Range*. En el panel *Domain*, se muestra el modelo dominio (origen), en *Range*, el rango (destino) y en *Mappins* el modelo de correspondencias. En la Figura 61 se muestra una vista general del editor, mostrando el modelo de trazabilidad generado automáticamente tras la primera transformación del caso de estudio.

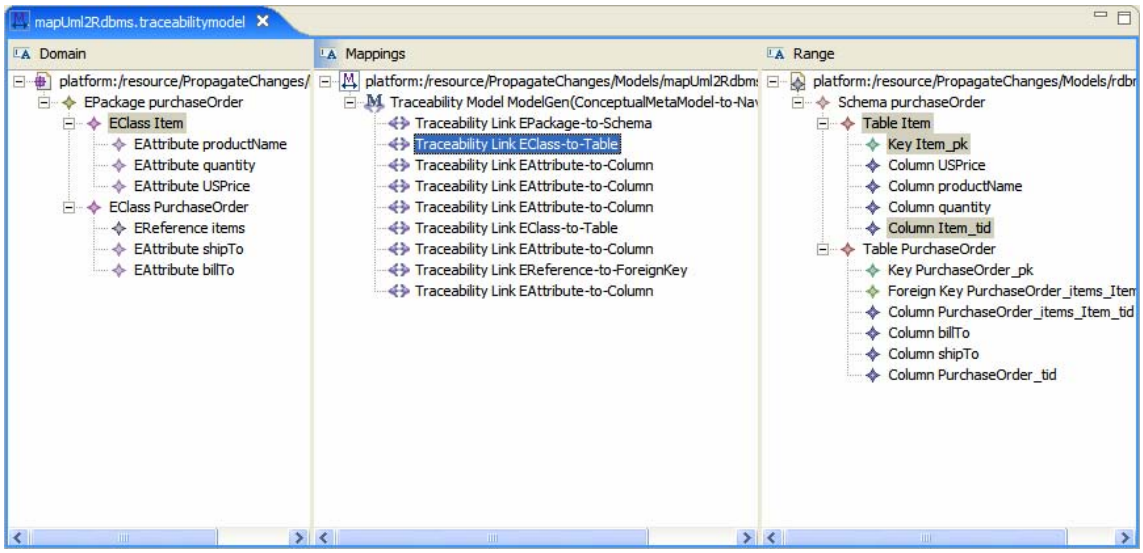

Figura 61: Vista general del editor de modelos de trazabilidad.

Como se puede observar, al hacer clic sobre un elemento del modelo de trazabilidad se resaltan automáticamente los elementos dominio y rango que este enlace relaciona. Igualmente, si se desea conocer qué elemento(s) del modelo dominio han generado cierto elemento del modelo rango, basta con seleccionarlo y se resaltará el elemento que lo ha originado, así como la correspondiente regla de transformación.

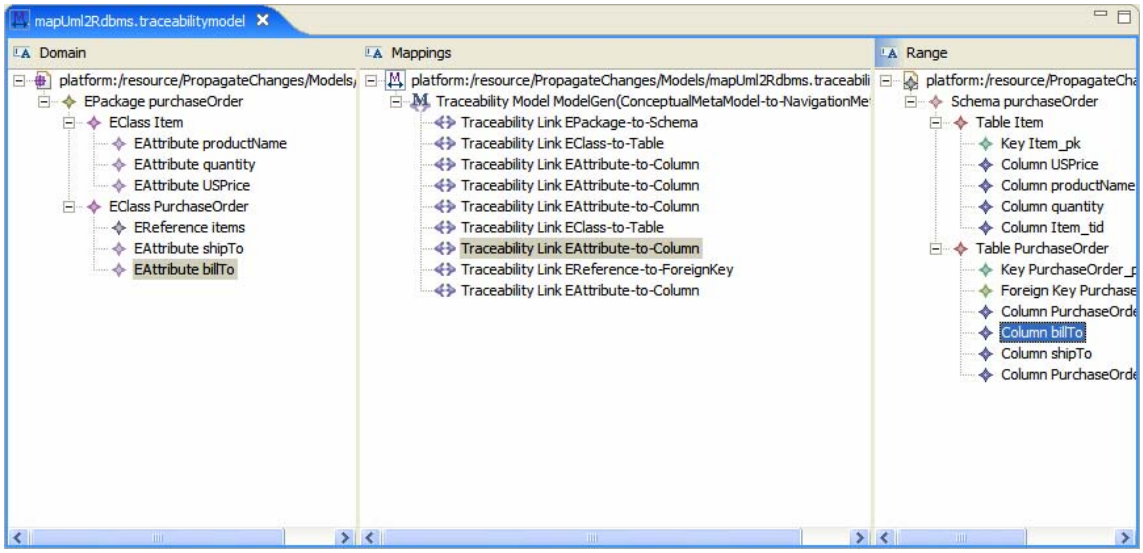

Figura 62: Consulta de un modelo de trazabilidad.

Se puede observar que dada una selección de elementos del marco *Domain*, los elementos que serán seleccionados en *Range*, serían los mismos que resultarían de la ejecución del operador *Domain* del álgebra de MOMENT. Por otra parte, los elementos que resultarían resaltados en el modelo de trazabilidad, serían los mismos que devolvería la aplicación del operador *SelectMappingsByDomain*.

Si la selección se realiza sobre elementos del modelo rango, ocurre de forma análoga, pero con los operadores inversos (estos son *Range* y *SelectMappingsByRange*).

En este caso, se ha realizado una implementación en Java sobre los métodos manejadores de las selecciones realizadas en las distintas vistas. Se ha implementado de esta manera por cuestiones de eficiencia, ya que es una tarea sencilla, y se evitar interactuar con Maude, lo que supondría la carga de los diferentes modelos, y la invocación de operadores.

Por otra parte, esto es posible ya que es posible emplear la API genérica de EMF, que nos permite consultar cualquier modelo independientemente del metamodelo que éste conforme.

6.2.4.4. Descripción detallada de paquetes y clases.

A continuación se realiza una relación de los dos paquetes que forman este plug-in, las clases que contienen, y sus métodos más interesantes.

i. es.upv.dsic.issi.moment.traceability.ui.editor

a) Clases

a.1) TraceabilityEditorPlugin

Esta es la clase central del plug-in del editor de modelos de trazabilidad. Extiende de la clase *org.eclipse.emf.common.EMFPlugin*, y tiene como clase anidada a la clase *Implementation*.

- **TraceabilityEditorPlugin()**

Este es el constructor por defecto. Llana al constructor de la clase padre, creando la instancia del plug-in EMF.

- **static TraceabilityEditorPlugin.Implementation getPlugin()**

Devuelve la instancia del plug-in de Eclipse.

a. org.eclipse.emf.common.util.

ResourceLocator getPluginResourceLocator()

Devuelve, al igual que le método anterior, la instancia compartida del plug-in.

a.2) TraceabilityEditorPlugin.Implementation

Esta es la implementación del Plug-in de Eclipse. Hereda de la clase *org.eclipse.emf.common.EMFPlugin.EclipsePlugin*.

- **TraceabilityEditorPlugin.Implementation()**

Crea la instancia compartida del plug-in.

a.3) TraceabilityEditor

Esta es la clase que comprende el editor reflexivo de trazabilidad. Inicialmente es creada automáticamente por el generador de código de EMF. Hereda de *org.eclipse.ui.part.MultiPageEditorPart*, e implementa las interfaces de la API de Eclipse *IEditingDomainProvider*, *ISelectionProvider*, *IMenuListener*, *IViewerProvider* e *IGotoMarker*.

Posteriormente ha sido modificada para soportar la visualización y edición de tres modelos simultáneamente.

- **TraceabilityEditor()**

Este es el constructor por defecto. Llama al constructor de la clase padre, y crea el editor. Además, inicializa diversos valores concernientes al editor, como los *Adapter Factories*, la pila de comandos a ejecutar, añade los observadores, y crea los dominios de edición de los modelos.

- **void addSelectionChangedListener(**
  org.eclipse.jface.viewers.ISelectionChangedListener listener)

Añade *listener* a la lista *listeners* que observan este editor.

- **protected void createContextMenuFor(**
  org.eclipse.jface.viewers.StructuredViewer viewer)

Este método crea un menu contextual para el visor *viewer*, y añade un listener, registrando este menú.

- **void createModel()**

Este es el menu que se llama para cargar los recursos (*resources*) en su dominio de edición. Esto es, en este método se carga el modelo de trazabilidad, y se consulta, para cargar igualmente en EMF los modelos dominio y rango.

- **void createPages()**

Este es el método llamado por el framwork para crear las páginas que forman el editor. En primer lugar, se realiza una llamada a *createModel()*, para cargar los modelos, y después se crea la página para el visor de tres paneles.

Posteriormente, se crean cada uno de los tres paneles y se establecen los modelos que mostrarán, sus *Adapter Factories*, manejadores de selección etc.

- **void dispose()**

Este método se encarga de eliminar todas las referencias a objetos del editor, para que el recolector de basura de *Java* libere la memoria.

- **void doSave(org.eclipse.core.runtime.IProgressMonitor progressMonitor)**

El método *doSave(…)* se encarga de salvar los cambios realizados en el modelo del editor en su correspondiente fichero.

- **void doSaveAs()**

Este método lanza un cuadro de diálogo de creación de nuevo fichero, y posteriormente realiza una llamada a «*doSaveAs(uri, editorInput)*».

- **protected void doSaveAs(org.eclipse.emf.common.util.URI uri, org.eclipse.ui.IEditorInput editorInput)**

Este método cambia el nombre de fichero asociado al modelo en edición, y posteriormente realiza la pertinente llamada a *saveAs()*.

- **protected void firePropertyChange(int action)**

Este método lanza la actualización de los distintos visores que intervienen en el proceso de edición para que sincronicen sus datos al realizar cualquier modificiación.

- **org.eclipse.emf.edit.ui.action.EditingDomainActionBarContributor getActionBarContributor()**

Devuelve la instancia que contribuye a las barras de acciones.

- **org.eclipse.ui.IActionBars getActionBars()**

Este método devuelve las barras de acciones a las que contribuye el editor.

- **java.lang.Object getAdapter(java.lang.Class key)**

Este es el método que determina qué interfaces se implementan.

- **org.eclipse.emf.common.notify.AdapterFactory getAdapterFactory()**

Devuelve el *Adapter Factory* del modelo de trazabilidad.

- **org.eclipse.ui.views.contentoutline.IContentOutlinePage getContentOutlinePage()**

Este método accede a la vista «*Outline*» del banco de trabajo de Eclipse. Si no se ha inicializado para nuestro editor, crea la ventana y la inicializa.

- **org.eclipse.emf.edit.domain.EditingDomain getEditingDomain()**

Devuelve el dominio de edición del modelo de trazabilidad.

- **org.eclipse.ui.views.properties.IPropertySheetPage getPropertySheetPage()**

Este método accede a la hoja de propiedades. En caso de que no se haya creado la hoja de propiedades para nuestro editor, lo crea, estableciendo como proveedor de datos una instancia de la clase *TraceabilityEditorPropertySource*.

- **org.eclipse.jface.viewers.ISelection getSelection()**

Devuelve la selección actual del editor.

- **private static String getString(String key)**

Devuelve la cadena de texto identificada por la clave *key* del archivo de recursos *plugin.properties* del plug-in.

- **private static java.lang.String getString(java.lang.String key, java.lang.Object s1)**

Devuelve la cadena de texto identificada por la clave *key* del archivo de recursos *plugin.properties* del plug-in, haciendo una sustitución.

- **org.eclipse.jface.viewers.Viewer getViewer()**

Este método devuelve el visor, como se require según la interfaz *IViewerProvider*.

- **void gotoMarker(org.eclipse.core.resources.IMarker marker)**

Establece la selección del editor según el marcador *marker*.

- **protected void handleActivate()**

Este método gestiona la activación del editor o sus vistas asociadas.

- **protected void handleChangedResources()**

Gestiona qué hacer con los recursos que han sufrido cambios al activarse.

- **void handleContentOutlineSelection(**

  org.eclipse.jface.viewers.ISelection selection)

Este método establece cómo se desea que afecte al resto de vistas un cambio en la selección en la ventana de «*Outline*». En este caso, selecciona en el visor del modelo de trazabilidad los mismos elementos que en esta ventana, pero sin afectar a la selección de los modelos dominio y rango.

- **protected boolean handleDirtyConflict()**

Este método muestra un diálogo que pregunta si los cambios no salvados deben guardarse.

- **void handleDomainSelection(org.eclipse.jface.viewers.ISelection selection)**

Este método establece como la selección en la vista de árbol del modelo dominio afectará al resto de visores. En este caso, seleccionará en el modelo de trazabilidad los elementos que devolvería la ejecución de *SelectMappingsByDomain()*, tomando como argumentos los elementos seleccionados.

Igualmente, para el modelo rango, seleccionará los elementos resultantes de la ejecución del operador *Range()* de forma análoga.

A continuación se muestra el código que consulta los diferentes modelos mediante la API genérica de EMF.

```
// List with the selected elements
ArrayList mappingsSelectionList = new ArrayList();
ArrayList rangeSelectionList = new ArrayList();
List eltsMappDom, eltsMappRange;

for(Iterator selectedElements = ((IStructuredSelection)selection).iterator();
selectedElements.hasNext();) {
    // Get the selected element.k
    //
    Object selectedElement = selectedElements.next();
    if (selectedElement instanceof EObject) {
        EObject eoselElem = (EObject)selectedElement;
        // Mappings Model........
        TreeIterator it_mps =
            ((Resource)mappingsEditingDomain.getResourceSet().
            getResources().get(0)).getAllContents();
        while (it_mps.hasNext()) {
            // For each element from the mappings model...
            EObject obj = (EObject) it_mps.next();
            if (obj.eClass().getEStructuralFeature("domain") != null) {
                // ... we check if contains in the domain property the selected one
                eltsMappDom = (List) obj.
                    eGet(obj.eClass().getEStructuralFeature("domain"));
                if (eltsMappDom.contains(mappingsEditingDomain.
                    getResourceSet().getEObject(EcoreUtil.getURI(eoselElem),true))) {
                    // we set the selection in the mappings model
                    mappingsSelectionList.add(obj);
                    // and we get the corresponding objects of
                    // the range model and we add them to the selection list
                    eltsMappRange = (List) obj.eGet(obj.eClass().
                                getEStructuralFeature("range"));
                    ListIterator it = eltsMappRange.listIterator();
```

```
                    while (it.hasNext()) {
                        rangeSelectionList.add(rangeEditingDomain.getResourceSet().
                        getEObject(EcoreUtil.getURI((EObject)it.next()),true));
                    }
                }

            }
        }
    }
}
// Set the selection to the widgets.
rangeViewer.setSelection(new StructuredSelection(rangeSelectionList));
mappingsViewer.setSelection(new StructuredSelection(mappingsSelectionList));
```

- **void handleMappingsSelection(**

  org.eclipse.jface.viewers.ISelection selection)

Aquí se establece cómo un cambio en la selección del modelo de trazabilidad afecta al resto de visores. Esto es, al modificar la selección de este visor, se seleccionarán los elementos relacionados por la selección tanto del modelo dominio como del rango.

- **void handleRangeSelection(org.eclipse.jface.viewers.ISelection selection)**

Este método establece como la selección en la vista de árbol del modelo rango afectará al resto de visores. En este caso, seleccionará en el modelo de trazabilidad los elementos que devolvería la ejecución de *SelectMappingsByRange()*, tomando como argumentos los elementos seleccionados.

Igualmente, para el modelo dominio, seleccionará los elementos resultantes de la ejecución del operador *Domain()* de forma análoga.

El código es fundamentalmente el mismo que el que gestiona la selección del modelo dominio, con la salvedad de que se intercambian los modelos origen y destino.

- **protected void hideTabs()**

El método *hideTabs()* oculta las pestañas inferiores del editor multipágina si éste tiene únicamente una.

- **void init(org.eclipse.ui.IEditorSite site,**

  org.eclipse.ui.IEditorInput editorInput)

Este método se llama al inciar el editor. Inicializa el editor, estableciendo su contenido, etc.

- **boolean isDirty()**

Este método comprueba si es existen cambios pendientes de guardar a disco en el modelo de trazabilidad que está siendo editado en el momento.

- **boolean isSaveAsAllowed()**

Este método devuelve siemrpe cierto, ya que actualmente no se soporta esta característica.

- **void menuAboutToShow(**
  org.eclipse.jface.action.IMenuManager menuManager)

Este método es el encargado de completer los menús contextuales con las acciones propias de nuestro editor de trazabilidad.

- **protected void pageChange(int pageIndex)**

Este método se emplea para registrar cual es la página activa del editor.

- **void removeSelectionChangedListener(**
  org.eclipse.jface.viewers.ISelectionChangedListener listener)

Elimina el *listener* dado de la lista de observadores del editor.

- **void setCurrentViewer(org.eclipse.jface.viewers.Viewer viewer)**

Establece el visor actual a *viewer*, actualizando los *listeners* correspondientes.

- **void setCurrentViewerPane(org.eclipse.emf.common.ui.ViewerPane**
  viewerPane)

Actualiza el valor de la variable que indica cuál es el panel actual.

- **void setFocus()**

Establece el foco en el editor de trazabilidad, activándolo.

- **void setSelection(org.eclipse.jface.viewers.ISelection selection)**

Este método establece la selección actual del editor a *selection*, y se lo notifica a los correspondientes *listeners*.

- **void setSelectionToViewer(Collection collection)**

Este método establece la selección del visor que esté activo.

- **void setStatusLineManager(org.eclipse.jface.viewers.ISelection selection)**

Este método se encarga de proporcionar al gestor de la linea de estado de Eclipse la fuente de datos adecuada a la selección.

a.4) TraceabilityActionBarContributor

Esta es la contribución que se hace a los menús de Eclipse por parte del editor de trazabilidad.

- **TraceabilityActionBarContributor()**

Este método es el constructor por defecto.

- **protected void addGlobalActions(org.eclipse.jface.action.IMenuManager menuManager)**

This inserts global actions before the "additions-end" separator

Este método inserta las acciones globales antes del separador «*additions-end*».

- **void contributeToMenu(org.eclipse.jface.action.IMenuManager menuManager)**

Por medio del este método se añade a la barra de menús un menú con diversas opciones del editor, así como el submenú para crear nuevos elementos del modelo de trazabilidad.

- **void contributeToToolBar(org.eclipse.jface.action.IToolBarManager toolBarManager)**

Éste es el método encargado de realizar la contribución a la barra de herramientas.

- **protected void depopulateManager(org.eclipse.jface.action.IContributionManager manager, java.util.Collection actions)**

Este método elimina del *manager* especificado todas las acciones contenidas en la colección *actions*.

- **protected java.util.Collection generateCreateChildActions(java.util.Collection descriptors, org.eclipse.jface.viewers.ISelection selection)**

Este método genera un objeto *org.eclipse.emf.edit.ui.action.CreateChildAction* para cada objeto en *descriptors* y devuelve la colección de estas acciones.

- **protected java.util.Collection generateCreateSiblingActions(java.util.Collection descriptors, org.eclipse.jface.viewers.ISelection selection)**

Este método genera un objeto *org.eclipse.emf.edit.ui.action.CreateSibling-Action* para cada objeto en *descriptors* y devuelve la colección de estas acciones.

- **void menuAboutToShow(org.eclipse.jface.action.IMenuManager menuManager)**

Este método puebla el menu contextual antes de que aparezca.This populates the pop-up menu before it appears

- **protected void populateManager(org.eclipse.jface.action.IContributionManager manager, java.util.Collection actions, java.lang.String contributionID)**

Este método pueble el *manager* especificado con *org.eclipse.jface.action. ActionContributionItem* basados en las acciones contenidas en *actions*, insertándolas detrás del ítem identificado por *contributionID*.

- **void selectionChanged(org.eclipse.jface.viewers.SelectionChangedEvent event)**

Este método gestiona la selección actual del editor, de forma que se determinen los hijos y hermanos que pueden añadírsele a un determinado objeto.

- **void setActiveEditor(org.eclipse.ui.IEditorPart part)**

Este método realiza las actualizaciones necesarias cuando el editor activo cambia.

a.5) TraceabilityEditorPropertySource

- **TraceabilityEditorPropertySource(org.eclipse.emf.common.notify.AdapterFactory adapterFactory,**

org.eclipse.emf.edit.domain.EditingDomain mappings,

org.eclipse.emf.edit.domain.EditingDomain domain,

org.eclipse.emf.edit.domain.EditingDomain range)

Este es el constructor por defecto, llama al constructor de la clase padre con el *Adapter Factory* proporcionado, e inicializa los dominios de edición.

- **protected org.eclipse.ui.views.properties.IPropertySource**

  createPropertySource(java.lang.Object object,

  org.eclipse.emf.edit.provider.IItemPropertySource itemPropertySource)

Este método crea un objeto que proporcionará los datos para rellenar la el valor de un elemento en la hoja de propiedades. En caso de que la selección sea el elemento *Dominio* o *Rango*, se crea un descriptor de propiedades personalizado, donde se filtran los elementos para que únicamente aparezcan los elementos válidos. A continuación se muestra el código que nos permite esto para el caso del elemento *Dominio*.

```
if (itemPropertyDescriptor.getDisplayName(object).equals("Domain")) {
    PropertyDescriptor p = new PropertyDescriptor(object,itemPropertyDescriptor) {
        public org.eclipse.jface.viewers.CellEditor createPropertyEditor(
                                org.eclipse.swt.widgets.Composite parent) {

            DialogCellEditor editor = new ExtendedDialogCellEditor(parent,
                                        getEditLabelProvider()) {

                protected Object openDialogBox(Control cellEditorWindow) {
                    ArrayList choiceOfValues = new ArrayList();

                    TreeIterator it = domainDomain.getResourceSet().getAllContents();
                    it.next();  // We skip the first element (the resource)...
                    while (it.hasNext()) {
                        choiceOfValues.add(it.next());
                    }

                    Object genericFeature = itemPropertyDescriptor.getFeature(object);
                    final EStructuralFeature feature =
                        (EStructuralFeature) genericFeature;

                    FeatureEditorDialog dialog = new FeatureEditorDialog(
                            cellEditorWindow.getShell(),
                            getEditLabelProvider(),
                            object,
                            feature.getEType(),
                            (List) ((IItemPropertySource) itemPropertyDescriptor.
                                getPropertyValue(object)).getEditableValue(object),
                            getDisplayName(),
                            choiceOfValues);
                    dialog.open();
                    return dialog.getResult();
                }
            };
            return editor;
        }
    };
    return p;
}
```

El caso para el elemento *Rango* es similar. Si no se da ninguno de estos casos, se crea el descriptor por defecto:

```
return super.createPropertyDescriptor(itemPropertyDescriptor);
```

ii. es.upv.dsic.issi.moment.traceability.ui.editor.wizards

a) Clases

a.1) TraceabilityModelWizard

La clase *TraceabilityModelWizard* se encarga de la gestión del asistente en general. Crea las diferentes páginas que lo compone, y al terminar éste realiza las acciones necesarias con los datos recogidos en las distintas ventanas. Extiende la clase *org.eclipse.jface.wizard.Wizard* e implementa la interfaz *INewWizard*.

- **TraceabilityModelWizard()**

Este es el constructor por defecto de la clase. Llama al constructor de la clase padre e inicializa el monitor de progreso.

- **void addPages()**

En este método se realiza la creación e inclusión en el asistente de las diversas ventanas de diálogo que lo forman. En este caso, crea una instancia de cada una de las páginas *TraceabilityModelWizardPage1, TraceabilityModelWizardPage2, TraceabilityModelWizardPage3* y las añade al asistente mediante el método de la clase padre *addPage(IWizardPage page).*

- **private void doFinish(String containerName, String fileName, String metamodelURI, String domainModel, String rangeModel, org.eclipse.core.runtime.IProgressMonitor monitor)**

Este es el método que realiza las acciones. Recibe donde se va a salvar el fichero con el modelo de trazabilidad, su nombre, metamodelo que conforma y modelos dominio y rango. Dinámicamente crea su contenido y lo salva en la ubicación elegida tal y como se explicó en la sección 6.2.4.2.

- **void init(org.eclipse.ui.IWorkbench workbench, org.eclipse.jface.viewers.IStructuredSelection selection)**

Establece el conjunto de elementos seleccionados en el banco de trabajo en el momento de lanzar el asistente.

- **org.eclipse.core.runtime.Path makeRelativePath(org.eclipse.core.runtime.Path referencePath, org.eclipse.core.runtime.Path targetPath)**

Devuelve una ruta para *targetPath* relative a la ruta *referencePath*.Returns a Path for targetPath relative to referencePath

- **boolean performFinish()**

Este método se llama al pulsar el botón «*Finish*» del asistente. Se encarga de consultar todos los valores establecidos en las diferentes páginas, y de realizar la llamada al método *doFinish(…)*.

- **private void throwCoreException(java.lang.String message)**

Este método, empleado por *doFinish(…)* es el encargado de lanzar la pertinente excepción, indicando que ha sido el este plug-in el que ha fallado con el mensaje especificado por *message*.

a.2) TraceabilityModelWizardPage1

En esta clase que extiende a la clase *org.eclipse.jface.wizard.WizardPage*, Crea la interfaz necesaria para que el usuario introduzca el contenedor y el archivo donde se salvará el modelo.

- **TraceabilityModelWizardPage1(org.eclipse.jface.viewers.ISelection selection)**

Este es el constructor por defecto, llama el método constructor de la clase padre, y establece el título y la descripción de la ventana de diálogo.

- **void createControl(org.eclipse.swt.widgets.Composite parent)**

Este es el método que creará todos los botones, etiquetas y casillas editables que recogerán los datos a introducir por el usuario.

- **private void dialogChanged()**

Este método privado es el encargado de consultar todos los valores introducidos por el usuario y validarlos. Se ejecuta automáticamente cada vez que el contenido de un control varía. Debe realizar una llamada a *updateStatus("mensaje")* en caso de que encuentre un error. Si todos los valores son correctos, deberá ejecutar *updateStatus(null)*.

- **String getContainerName()**

Este método devuelve un string, con la ruta de la carpeta contenedora para el nuevo fichero.

- **String getFileName()**

El método *getFileName()* devuelve el nombre del archivo que contendrá el nuevo modelo de trazabilidad.

- **private void handleBrowse()**

Este método crea una ventana de selección de contenedor estándar para establecer el nuevo valor para el campo de texto etiquetado como «*Container:* ».

- **private void initialize()**

El método *initilize()* comprueba si la selección actual del banco de trabajo es un contenedor válido para ser usado. Si lo es, este será el contenedor que aparecerá seleccionado inicialmente al crear la ventana de selección de contenedor.

- **private void updateStatus(String message)**

Este método establece el mensaje de error a *message*. El mensaje de error se muestra en la parte superior de la ventana indicando la descripción de los fallos en los datos introducidos. Si existen errores (*message tiene valor distinto de null*) se realiza una llamada *setPageComplete(false)*, que impide que se avance en el asistente. Si no existen fallos, se ejecuta *setPageComplete(true)* permitiendo que el asistente continue.

a.3) TraceabilityModelWizardPage2

En esta clase, que al igual que la anterior extiende a la clase *org.eclipse.jface.wizard.WizardPage*, establece los controles necesarios para que el usuario seleccione un metamodelo de trazabilidad.

- **TraceabilityModelWizardPage2(org.eclipse.jface.viewers.ISelection selection)**

Este es el constructor por defecto, llama el método constructor de la clase padre, y establece el título y la descripción de la ventana de diálogo.

- **void createControl(org.eclipse.swt.widgets.Composite parent)**

Este es el método que creará todos los botones, etiquetas, casillas editables y la tabla para que el usuario indique el metamodelo de trazabilidad.

- **private void dialogChanged()**

Este método privado es el encargado de comprobar que se ha seleccionado un metamodelo de trazabilidad, y que éste es válido. Se ejecuta automáticamente cada vez que el contenido de un control varía. Debe realizar una llamada a *updateStatus("mensaje")* en caso de que encuentre un error. Si todos los valores son correctos, deberá ejecutar *updateStatus(null).*

- **String getSelectedMetamodel()**

Este método devuelve la URI (como cadena de texto) que identifica al metamodelo de trazabilidad personalizado seleccionado.

- **private void handleBrowse()**

Este método crea una ventana de selección de recurso. Se emplea para seleccionar un modelo *Ecore* que contenga un metamodelo de trazabilidad personalizado.

- **private void initialize()**

El método *initilize()* comprueba si la selección actual del banco de trabajo es un contenedor válido para ser usado. Si lo es, este será el contenedor que aparecerá seleccionado inicialmente al crear la ventana de selección de contenedor.

- **private boolean**

  isValidTraceabilityMetamodel(org.eclipse.emf.ecore.EPackage pkg)

Este método comprueba si un modelo (*pkg*) *Ecore* es un metamodelo de trazabilidad válido, comprobando si contiene o no la anotación identificativa «*MOMENTTraceabilityMetamodel*».

- **private void setTableContents()**

Este método se encarga de poblar la tabla de metamodelo de trazabilidad, consultando todos los modelos cargados en EMF y añadiendo aquellos que sean válidos.

- **private void updateStatus(String message)**

Este método establece el mensaje de error a *message*. El mensaje de error se muestra en la parte superior de la ventana indicando la descripción de los fallos en los datos introducidos. Si existen errores (*message tiene valor distinto de null*) se realiza una llamada *setPageComplete(false)*, que impide que se avance en el asistente. Si no existen fallos, se ejecuta *setPageComplete(true)* permitiendo que el asistente continue.

a.4) TraceabilityModelWizardPage3

Esta es la última página del asistente para la creación de un nuevo modelo de trazabilidad. Permite seleccionar los modelo dominio y rango.

- **TraceabilityModelWizardPage3(org.eclipse.jface.viewers.ISelection selection)**

Este es el constructor por defecto, llama el método constructor de la clase padre, y establece el título y la descripción de la ventana de diálogo.

- **void createControl(org.eclipse.swt.widgets.Composite parent)**

Este es el método que creará todos los botones, etiquetas y casillas editables que recogerán las rutas donde se encuentran los modelos.

- **private void dialogChanged()**

Este método privado es el encargado de comprobar que se seleccionan dos archivos existentes en el especio de trabajo. Se ejecuta automáticamente cada vez que el contenido de un control varía. Debe realizar una llamada a *updateStatus("mensaje")* en caso de que encuentre un error. Si todos los valores son correctos, deberá ejecutar *updateStatus(null).*

- **String getDomainFileName()**

El método *getDomainFileName()* devuelve la ruta del archivo que contiene el modelo dominio.

- **String getRangeFileName()**

Este método devuelve la ruta del fichero que contiene el modelo rango.

- **private void handleBrowseDomain()**

Este método crea un diálogo de selección de fichero personalizado (mediante la llamada a *openIFileDialog*), y rellena con el valor devuelto el contenido del campo etiquetado como «*Domain model:* ».

- **private void handleBrowseRange()**

El método *handleBrowsRange()*, al igual que el anterior, realiza una llamada a *openIFileDialog*, y rellena con el valor devuelto el contenido del campo etiquetado como «*Range model:* ».

- **private void initialize()**

El método *initilize()* comprueba si la selección actual del banco de trabajo es un contenedor válido para ser usado. Si lo es, este será el contenedor que aparecerá seleccionado inicialmente al crear la ventana de selección de contenedor.

- **private void updateStatus(String message)**

Este método funciona de la misma manera que en el resto de páginas. En este caso, al tratarse del la última página, si todo es correcto, causa la finalización del asistente.

6.3. Trabajos relacionados.

En en campo de la Gestión de Modelos, las herramientas no suelen tratar directamente con las cuestiones de trazabilidad. Usualmente, trabajan con modelos de *mappings*, que definen relaciones de equivalencia entre los elementos de dos modelos de forma que un operador de gestión de modelos puede ser definido de forma genérica. RONDO es un buen ejemplo de esta aproximación.

6.3.1. RONDO.

RONDO, como se introdujo en el apartado 2.3.1, es la plataforma de Gestión de Modelos basada en los trabajos de P. Bernstein [Ber03]. En esta herramienta, por ejemplo, el operador *Merge* (que permite la integración de dos modelos), recibe como entradas dos modelos (*A* y *B*) y un modelo de *mappings* entre ellos (map_{AB}); y produce el modelo combinado *C*, y dos nuevos modelos de correspondencias (map_{AC} y map_{BC}): $<C, map_{AC}, map_{BC}> = Merge(A, B, map_{AB})$.

Para la creación de estos modelos de correspondencias se proporciona una sencilla interfaz. La Figura 63 muestra el editor simple de correspondencias implementado en Java para la plataforma RONDO. Éstos serán los mappings que se emplearán en la operación *Merge* en su ejemplo de propagación de cambios. Para mayor detalle se puede consultar [RONDO].

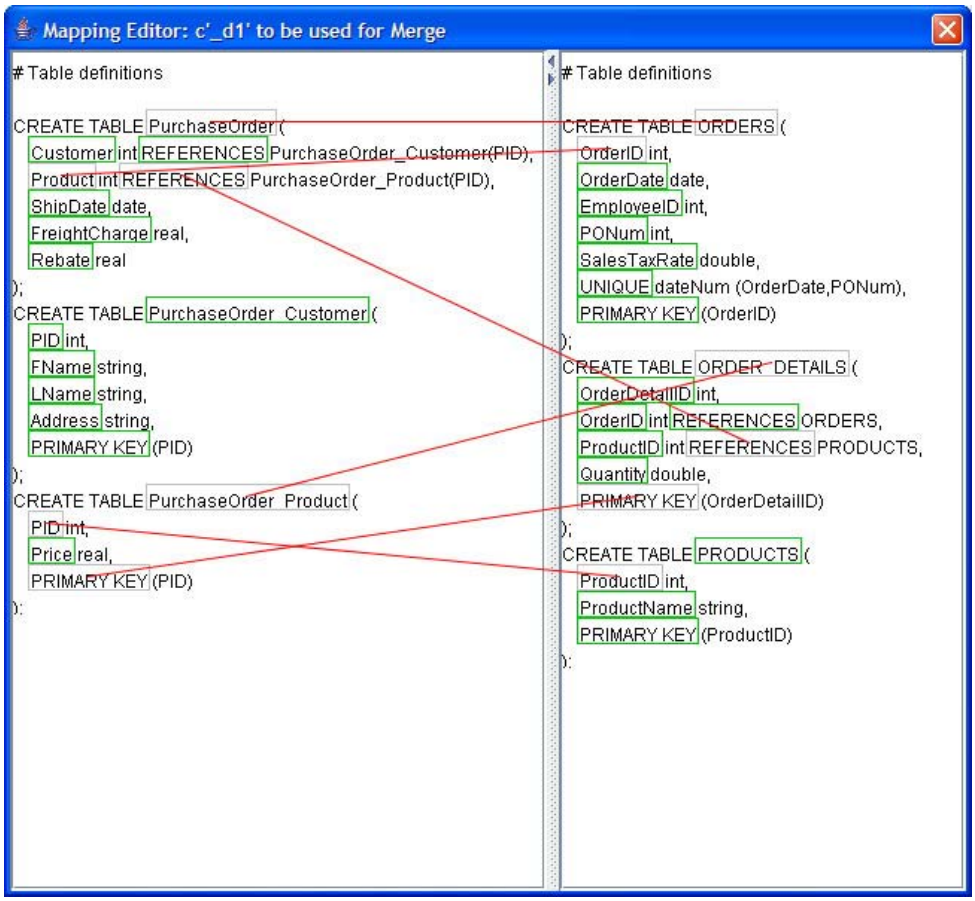

Figura 63: Vista del editor de correspondencias de RONDO.

En MOMENT, los modelos de *mappings* se introducen como modelos de trazabilidad. Esto se debe a que los operadores no se apoyan en ellos para aplicarse a un conjunto de modelos. En MOMENT, las relaciones de trazabilidad entre los elementos de dos modelos, que se necesitan para aplicar un operador, se definen entre los elementos de sus correspondientes metamodelos de forma axiomática con los correspondientes operadores. La colección de relaciones de equivalencia entre dos metamodelos constituye un morfismo que puede ser reusado por todos los operadores del álgebra de MOMENT. Esto permite una especificación más clara de los operadores complejos. En MOMENT, el operador Merge es de la siguiente manera: $<C, map_{AC}, map_{BC}> = Merge(A, B)$. Los modelos de correspondencias son producidos por la aplicación de un operador simple a un conjunto de modelos, y almacena la información acerca de la tarea de manipulación realizada sobre un modelo. Por ello, se tratan estos modelos de correspondencias desde el punto de vista de la trazabilidad.

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

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

6.3.2. *ATLAS Model Weaver.*

La herramienta AMW (*ATLAS Model Weaver*) es una herramienta que permite la definición de modelos de correspondencias (llamados *weaving models*) entre modelos EMF en la arquitectura de gestión de modelos ATLAS (AMMA – *ATLAS Model Management Architecture*). AMW proporciona un metamodelo de *weavings* básico que puede ser extendido para permitir la definición de *mappings* complejos.

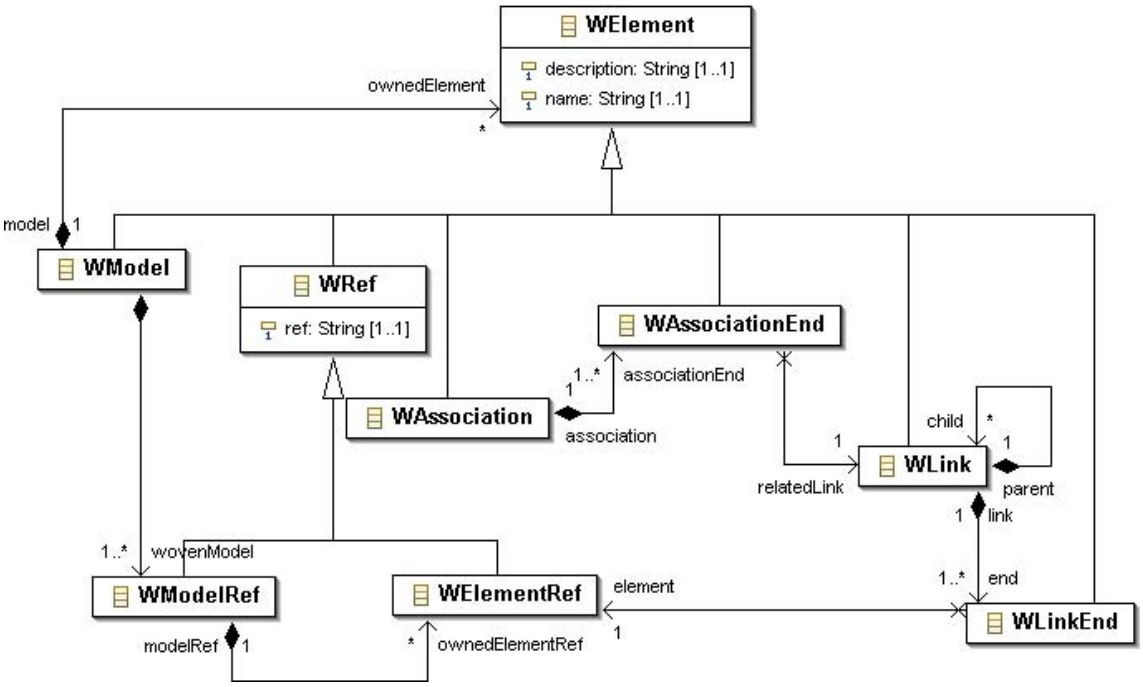

Figura 64: Metamodelo de *weavings* de AMW.

Estos *mappings* los define generalmente el usuario, aunque pueden ser inferidos mediante heurísticas, como en [Mad01]. En MOMENT, estos *mappings* son generados por los operadores de gestión de modelos automáticamente en un modelo de trazabilidad, y pueden ser manipulados por otros operadores. Además, en MOMENT también se soporta la extensión del metamodelo de trazabilidad.

Por otra parte, como se puede observar, el metamodelo de *weavings* de AMW es más complejo que el metamodelo básico proporcionado por MOMENT. No obstante, a pesar de la simplicidad de éste, se ha comprobado que permite tratar con operadores complejos satisfactoriamente.

La interfaz que proporciona AMW para la edición de estos modelos de *weavings* es muy similar a la proporcionada por MOMENT, pero más compleja y algo menos intuitiva.

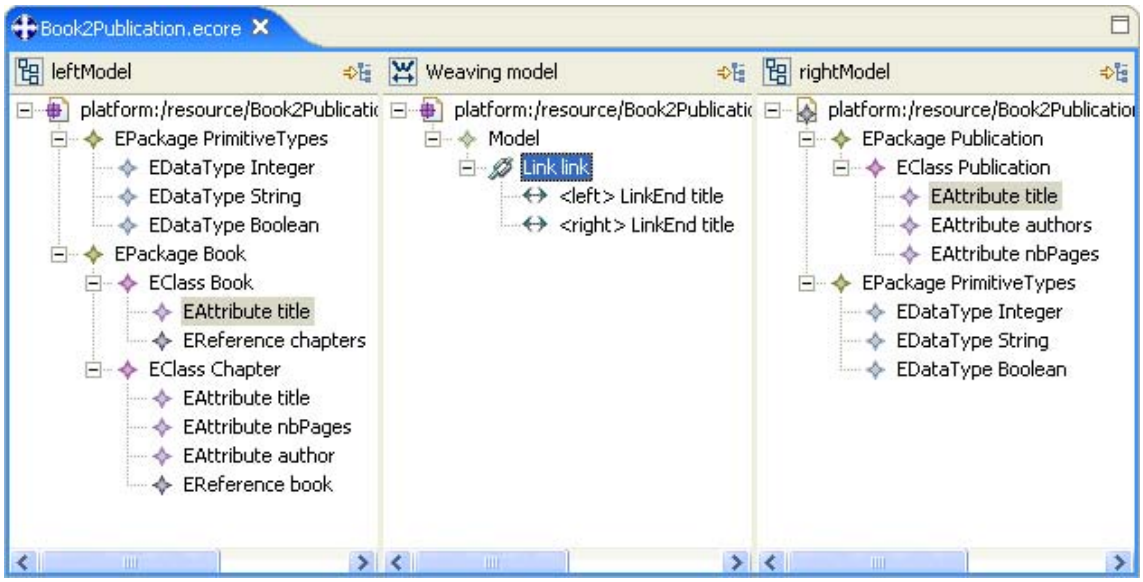

Figura 65: Aspecto del editor de modelos de *weavings* de AMW.

Como se observa en la Figura 65, el editor también consta en este caso de tres paneles. En el ejemplo se ha creado un *Link* manualmente entre los dos atributos *title* de ambios modelos. De las posibles opciones de navegación posibles, este editor únicamente proporciona el resaltado de los elementos relacionados entre ambos modelos, izquierdo y derecho, para un *Link* determinado. No se resaltan, en este editor, los elementos del modelo derecho (denominado «rango» en la terminología de MOMENT) relacionados con los elementos que se seleccionen en la parte izquierda (dominio) mediante una o varias correspondencias (o viceversa).

6.3.3. *Model Transformation Framework.*

Model Transformation Framework (MTF) es un conjunto de herramientas desarrollado por IBM que permiten hacer comparaciones, comprobar la consistencia y ejecutar transformaciones entre modelos EMF [MTF]. En este sentido, el punto que nos interesa de MTF es su capacidad para salvar modelos de trazabilidad que reflejen las operaciones realizadas en un paso de transformación, ya que no se trata de una herramienta de Gestión de Modelos.

El visor de modelos de correspondencias presenta el aspecto mostrado en la Figura 66.

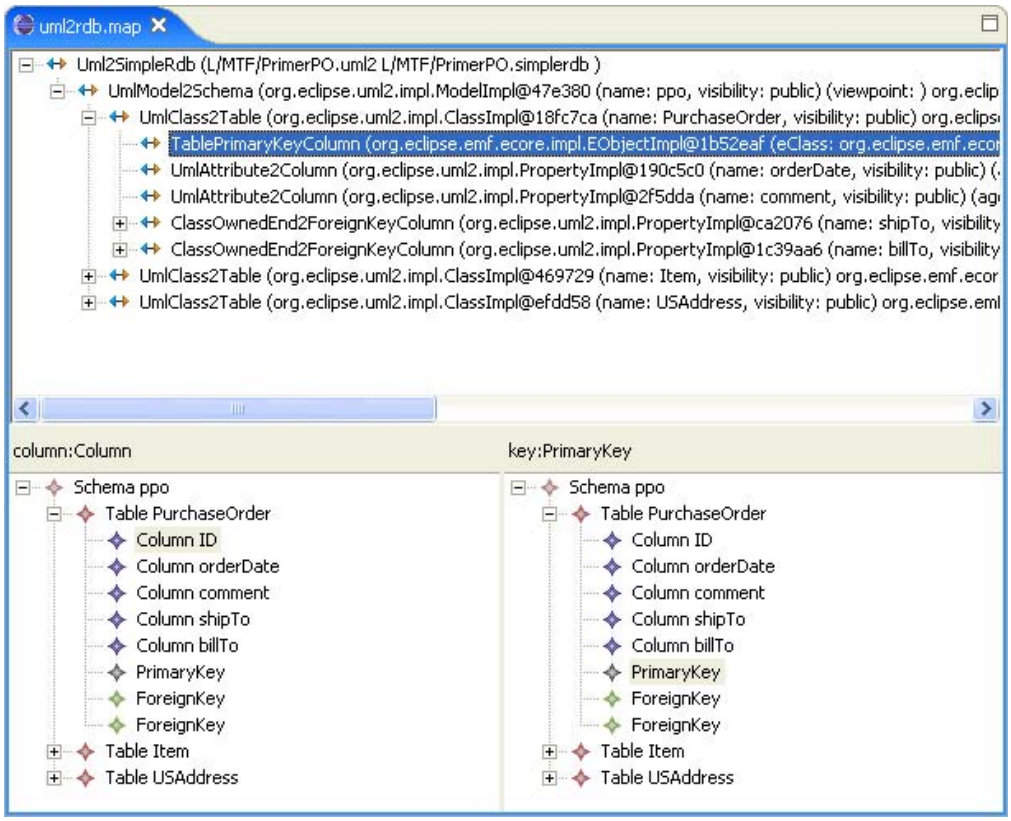

Figura 66: Aspecto del visor de *mappings* proporcionado por MTF.

En este sentido MTF presenta una interfaz más completa para mostrar modelos de trazabilidad. La figura miestra un ejemplo de transformación del modelo *PurchaseOrder* UML al metamodelo relacional bastante similar al planteado en el caso de estudio. Para ello, se ha creado el correspondiente *script* que realiza esta transformación de UML a relacional. Este ejemplo se puede encontrar en [MTFex].

En este caso, la debilidad que presenta MTF frente a MOMENT es precisamente, que no se trata de una herramienta de Gestión de Modelos que proporcione una serie de operadores básicos. En este caso, preparar un escenario de propagación de cambios no sería una tarea sencilla, comparable a la composición de un operador complejo, tal y como se hace en MOMENT. Igualmente en este caso, los modelos de trazabilidad únicamente sirven para inspeccionar el proceso de transformación realizado, y no pueden editarse ni emplearse como entrada para un proceso de transformación posterior.

Parte Cuarta:

Conclusiones e informaciones complementarias.

7. CONCLUSIONES.

En este trabajo se han implementado los mecanismos necesarios para integrar una herramienta formal en un entorno industrial; y, por otra parte, se han propocionado los mecanismos pertinentes para gestionar de forma gráfica las capacidades de trazabilidad de un sistema de gestión de modelos.

Uno de los principales objetivos de MOMENT, el cual consideramos que se ha alcanzado, era probar la falsedad de la poca productividad de los sistemas formales en la ingeniería del software. En pocos meses se ha conseguido desarrollar un primer prototipo funcional con el que se han resuelto diversos casos de ejemplo. Podemos concluir que MOMENT es:

- Genérico. Ya que como se ha mostrado con diversos ejemplos a los largo de éste documento, MOMENT puede operar con modelos pertenecientes a diversos metamodelos, como son los esquemas XSD (sección 1), UML o relacional (sección 5.5).
- Correcto. Dadas las propiedades que un sistema formal nos proporciona, se tiene la certeza de que los modelos generados mediante MOMENT son correctos sin necesidad de pasos de validación posteriores a las operaciones.
- Eficiente. Ya que se hace uso de Maude, un sistema de alto rendimiento, habiéndose ejecutado las transformaciones de los ejemplos descritos en pocos segundos.
- Potente. Dado el álgebra genérica que proporciona MOMENT es posible construir operadores complejos que satifagan nuestras necesidades en tiempo record. Por ejemplo, dado el caso de estudio de propagación de cambios, basta con añadir una simple invocación más para obtener los resultados deseados. Esto desde otra herramienta de transformación presumiblemente nos hubiera supuesto la modificación de numerosos líneas de un largo código.

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

Igualmente consideramos que se han cumplido los objetivos planteados para las herramientas de desarrollo de Maude. Se ha conseguido portar Maude a un sistema Windows, y proporcionando un sistema de instalación sencillísimo, algo impensable hasta el momento.

Se ha proporcionado de un entorno gráfico altamente eficiente comparado con las escasas soluciones que se han proporcionado al respecto. Por otra parte, dado que se ha elegido Eclipse como entorno gráfico para estas herramientas de desarrollo de Maude, se abre ahora un amplio abanico de posibles usuarios de este sistema formal, dadas las mejoras que se han proporcionado con estos plug-ins, así como por la popularidad de este entorno.

Igualmente, el carácter de código abierto con que se han desarrollado estas herramientas, así como su modularidad, pueden permitir que las *Maude Development Tools* se popularicen y sean desarrolladas por mayor número de personas (incluso desvinculándose del proyecto MOMENT), incrementando notablemente sus prestaciones, que en la actualidad son las básicas.

Por último queda suficientemente probada la necesidad de proporcionar gestión para la trazabilidad en una herramienta de gestión de modelos. Esto queda patente en la potencia de que dota al sistema como se observa en el ejemplo de propagación de cambios.

Se ha conseguido integrar en EMF este soporte, proporcionando los mecanismos necesarios para la definición de metamodelos personalizados de forma extremadamente simple.

En este sentido, la potencia de MOMENT es la que nos permite que, aunque en el álgebra se emplee únicamente un modelo de trazabilidad, el usuario pueda trabajar con cualquier metamodelo que cumpla unas características mínimas. Simplemente basta realizar un paso de transformación de modelos mediante el operador *ModelGen*, y el usuario obtendrá el modelo de trazabilidad conforme a su metamodelo específico.

7.1. Trabajos futuros.

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

En cuanto a las herramientas de desarrollo de Maude, el principal trabajo que deberá ser realizado en un futuro es la mejora en la tolerancia a fallos en el código. Esto podría ser abordado mediante el diseño de un analizador sintáctico que validara las expresiones antes de lanzarlas a Maude. Igualmente, esto permitiría un mayor detalle a la hora de generar un conjunto de trabajos a partir de un segmento de código incluso en Core Maude.

La implementación actual, para un segmento de código *Core Maude* genera un único trabajo (a diferencia de como ocurre con *Full Maude*). Mediante este analizador se podría abordar la tarea de descomposición en tareas simple. No obstante, debería evaluarse el impacto sobre la eficiencia que ello pudiera tener, así como la dificultad de mantenimiento del código al producirse variaciones en la especificación de la gramática del lenguaje Core Maude.

Por otra parte, otro interesante trabajo, aunque costoso y no relacionado específicamente con MOMENT sería la mejora del editor de Maude, añadiendo soporte para la integración de mecanismos de *debug* en el mismo, al igual que hacen otros numerosos IDE basados en Eclipse.

En cuanto al soporte para trazabilidad en un futuro debe darse soporte a una cuestión muy importante: el refresco de referencias de los modelos de trazabilidad. En la actualidad MOMENT proporciona una interfaz de creación de operadores

implementada como un editor de Eclipse, así como una interfaz de ejecución de estos operadores.

Estas interfaces de usuario, nacidas en el seno del proyecto [Ibo05], funcionan con el *kernel* de MOMENT implementado en Full Maude. Actualmente estas interfaces de usuario ya han quedado obsoletas y se está portando y extendiendo el *kernel* de MOMENT en Core Maude con soporte para parametrización (actualmente *Core Maude alpha 86a*). Los operadores de trazabilidad, igualmente, están implementados en este nuevo *kernel*.

En este sentido, actualmente no existe soporte gráfico para la ejecución automática de operadores desde Eclipse sobre esta última versión del álgebra. Una vez se hayan actualizado los correspondientes editores y módulos de ejecución de operadores, se deberá integrar el mecanismo de refresco de URIs de los modelos de trazabilidad en estos módulos. Esto es, porque como se recordará, debía ejecutarse manualmente una operación de refresco de URIs tomando como parámetros el modelo rango devuelto por el operador, el mismo modelo, una vez cargado sobre EMF y el modelo de trazabilidad erróneo.

8. BIBLIOGRAFÍA.

[ANTLR] ANTLR plug-in for Eclipse. http://antlreclipse.sourceforge.net/

[ATL] Bézivin, J., Valduriez. P., Jouault, F. «The ATL home page». http://www.sciences.univ-nantes.fr/lina/atl/

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

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

[Béz05] Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F. «An M3Neutral infrastructure for bridging model engineering and ontology engineering». INTEROP-ESA'05, Geneve, Switzerland. 2005.

[BoC05] Boronat, A., Carsí, J.Á., Ramos, I. «Automatic Support for Traceability in a Generic Model Management Framework.». ECMDA-FA, Noviembre 2005.

[BoCR05] Boronat, A., Carsí, J.Á., Ramos, I. «Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine». IEEE Computer Society Press. 9th European Conference on Software Maintenance and Reengineering. Manchester, Reino Unido. 2005.

[BoI05] Boronat A., Iborra J., Carsí J. Á., Ramos I., Gómez A. «Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework». JISBD'05. Septiembre 2005.

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

[Brü05] Brünnler, K. Software Section. Abril 2005. http://www.iam.unibe.ch/~kai/

[Bud03] Budinsky F., Steinberg D., Merks E., Ellersick R., Grose T. J. «Eclipse Modeling Framework, A Developer's Guide». Addison Wesley, 2003.

[BuDDy] «BuDDy: A Binary Decision Diagram library».

http://buddy.sourceforge.net/

[Cygwin] RedHat, Inc. «Cygwin Information and Installation». http://www.cygwin.com.

[Dorf90] Dorfman, M., Thayer, R. «Guide to software requirements specification». IEEE Standards, Guidelines and Examples on System and Software Requirements Engineering, 1990.

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

[EMF] Eclipse Tools. EMF home. http://www.eclipse.org/emf/

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

[Gam95] E.Gamma, R.Helm, R.Johnson, J.Vlissides. «Design Patterns: Elements of Reusable Object-Oriented Software». Addison-Wesley, Reading, MA, 1995.

[GMP] «GMP: GNU Multiple Precision Arithmetic Library». Free Software Foundation. http://www.swox.com/gmp/

[Got94] Gotel, O. C. Z., Finkelstein, A. C. W. «An Analysis of the Requirements Traceability Problem». ICRE '94., 1994.

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

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

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

[KiCH05] «KiCHik» (Lead NSIS developer). NSIS Wiki: Path Manipulation. Julio 2005. http://nsis.sourceforge.net/wiki/Path_Manipulation

[Kurt02] Kurtev, I., Bézivin, J., Aksit, M. «Technological Spaces: An Initial Appraisal.» Int. Federated Conf. (DOA, ODBASE, CoopIS). Irvine, 2002.

[Mad01] Madhavan, J., P.A. Bernstein, E. Rahm. «Generic Schema Matching using Cupid». VLDB 2001.

[Mad03] Madhavan, J., Bernstein, P. A., Chen, K., Halevy, A.Y., Shenoy, P. «Corpus-based Schema Matching». IJCAI. 2003.

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

[Maude] Department of Computer Science, University of Illinois, Urbana-Champaign. The Maude System. http://maude.cs.uiuc.edu/.

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

[MaudeW] Álvarez J.M. «Maude on Microsoft Windows». http://maude.cs.uiuc.edu/download/windows.html

[Mel03] Melnik, S., Rahm E., Bernstein P. A., «Rondo: A Programming Platform for Generic Model Management». SIGMOD 2003. Extended version in Web Semantics, Volume 1, Number 1.

[MOMENT] Gómez, A. MOMENT Website. «MOMENT: A framework for MOdel manageMENT.». Junio 2005. http://moment.dsic.upv.es

[MTF] Griffin, C. Model Transformation Framework. Julio 2005. http://www.alphaworks.ibm.com/tech/mtf

[MTFex] Demathieu, S., Griffin, C., Sendall, S. «Model Transformation with the IBM Model Transformation Framework». Mayo 2005. http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/

[Muñoz04] Muñoz A., «Maude Workstation: Entorno de programación para el lenguaje de especificación algebraica Maude». Proyecto final de carrera. Universidad de Málaga. Julio 2004.

[NSIS] Nullsoft Scriptable Install System. http://nsis.sourceforge.net/

[QVT-RFP] Object Management Group. «Request for Proposal: MOF 2.0 Query / Views /Transformations RFP». 2002. ad/2002-04-10.

[Ram01] Ramesh, B., Jarke, M. «Toward reference models for requirements

traceability». Software Engineering, 27(1):58–93, 2001.

[RONDO] Melnik, S. «Rondo: A Programming Platform for Model Management», Junio 2003.
http://www-db.stanford.edu/~melnik/mm/rondo/

[Tecla] Shepherd, M., «The Tecla command-line editing library». Octubre 2004.
http://www.astro.caltech.edu/~mcs/tecla/

[Vol] Volere: Requirement Resources. Requirements tools. Marzo 2005.
http://www.volere.co.uk/tools.htm

ANEXO I. INSTALACIÓN DE MAUDE SOBRE UN SISTEMA WINDOWS.

Maude es un sistema diseñado inicialmente para ser ejecutado sobre sistemas UNIX, pero es posible ejecutarlo sobre un sistema Windows mediante el uso de CygWin. Para ello ha de compilarse el código fuente de Maude sobre CygWin con la ayuda de diversas librerías. A continuación se muestra el proceso de instalación compilación y configuración de forma manual de Maude, basándonos en el trabajo realizado por José María Álvarez Palomo [MaudeW].

No obstante, dada la complejidad de este proceso, y a que no siempre se tiene éxito en la compilación (por problemas de compatibilidad entre el código fuente y la versión del compilador, o las librerías), se recomienda encarecidamente el uso del instalador *Maude for Windows* descrito en el capítulo 1 y que se puede descargar de [MOMENT], que simplifica de forma sorprendente el proceso de instalación.

I.1. Instalación de CygWin.

En primer lugar, se debe (a) descargar el instalador de CygWin disponible en [Cygwin] y (b) ejecutarlo. Se recomienda (c) desactivar los posibles antivirus activos en el sitema. Igualmente, se recomienda la (d) instalación desde internet, y para (e) todos los usuarios, (f) en «c:\cygwin» (u otra carpeta sencilla y sin espacios en la ruta.

Tras elegir el servidor «espejo» más cercano, se deberán seleccionar los paquetes a instalar. A parte de los predeterminados, se debe comprobar que están seleccionados los siguientes:

Bash:	bash
Devel:	gcc, g++ (3.3.x), gdb, bison, flex, make, libncurses-devel
Doc:	cygwin-doc
Libs:	libiconv

Tabla 2: Paquetes adicionales de la instalación de CygWin.

Nótese que se indica la versión 3.3.x del componente g++. Instalar una versión diferente puede provocar errores en la compilación, imposibilitando obtener un ejecutable final.

El resto del proceso de instalación es automático. Al finalizar, aparecerá un icono para ejecutar CygWin en el escritorio, y existirá un entorno similar a un sistema UNIX en «c:\cygwin».

I.2. Compilación de Maude.

Para compilar Maude, en primer lugar hemos de preparar las librerías adicionales requeridas. Estas son las siguientes:

- *BuDDy*. Ésta es una librería para BDD (Binary Decision Diagram). Se puede encontrar más información en [BuDDy].
- Tecla. La librería Tecla proporciona facilidades de edición para la línea de comandos. Puede descargarse de [Tecla].
- GNU MP. GNU MP (GMP) es una librería que proporciona soporte para aritmética de precisión arbitraria. La página de este proyecto es [GMP].

Al margen de los sitios oficiales donde se pueden descargas estas librerías, están disponibles algunas versiones anteriores directamente en la página del proyecto Maude [MaudeW]. Las instrucciones que se muestran a continuación se refieren a las versiones disponibles en la dirección web de Maude.

I.2.1. Compilación de BuDDy 2.2.

Para compilar esta librería, una vez descargado el fichero *.tar* se deberá descomprimir al disco duro, y tras entrar en el directorio de buddy, ejecutar las siguientes acciones:

- Modificar el archivo «*config*» (como se indica en el fichero INSTALL de Maude) descomentando la línea.

```
# BUDDYUINT64 = -DBUDDYUINT64="long long"
```

- Ejecutar:

```
bash-3.00$ make
bash-3.00$ mkdir /usr/local/include
bash-3.00$ make install
```

(Las cabeceras y las librerías se copiarán a /usr/local/include y /usr/local/lib respectivamente).

- Ejecutar:

```
bash-3.00$ make examples
```

(Este paso puede obviarse, a parte de que puede que el código deba modificarse por cuestiones de compatibilidad).

I.2.2. Compilación de Tecla.

Tras descomprimir y acceder al directorio de «*libtecla*», deben ejecutarse las siguientes acciones:

- Ejecutar:

```
bash-3.00$./configure
```

- Modificar el archivo «*Makefile*» (como se indica en el fichero INSTALL de Maude), en su línea 89.

```
CFLAGS "-O2 -D_POSIX_C_SOURCE=1"
```

- Ejecutar:

```
bash-3.00$ make TARGETS=normal TARGET_LIBS=static install
```

- El último paso del proceso de compilación fallará, porque el proceso «*make*» esperará un ejecutable sin extensión (esto es, «*enhance*» en lugar de «*enhance.exe*»)

  Para solucionar el problema se deberá mover el fichero «enhance.exe» y cambiar sus permisos de forma manual de la siguiente forma:

```
bash-3.00$ mv enhance.exe /usr/local/bin/enhance.exe && chmod ugo+rx
/usr/local/bin/enhance.exe
```

I.2.3. Compilación de GMP.

Aunque CygWin incluye una versión de *libgmp*, ésta no incluye soporte para C++, por lo que deberemos compilarla también por separado. Para ello, descargaremos el fichero de internet, y los descomprimiremos en disco. Tras acceder a la carpeta de *libgmp*, realizaremos las siguientes acciones.

- Ejecutar:

```
bash-3.00$./configure --enable-cxx --disable-shared

bash-3.00$ make

bash-3.00$ make install
```

Todas las cabeceras y librerías se copiarán en /usr/local/include y /usr/local/lib respectivamente.

I.2.4. Compilación de Maude.

Finalmente, podemos comenzar la compilación de Maude. Tras, como anteriormente, descargar, descomprimir, y acceder al directorio de Maude, deberemos ejecutar los siguientes comandos para iniciar la configuración y compilación:

- Ejecutar:

```
bash-3.00$ mkdir Build

bash-3.00$ cd Build

bash-3.00$../configure CPPFLAGS="-I/usr/include -I/usr/local/include" LDFLAGS="-
L/usr/lib -L/usr/local/lib" GMP_LIBS="/usr/local/lib/libgmpxx.a /usr/local/lib/libgmp.a"

bash-3.00$ make

bash-3.00$ make install
```

El fichero ejecutable resultante tras todo este proceso puede ser muy voluminoso. Para reducir su tamaño aproximadamente al 10% o 15%, ejecutaremos:

```
strip -s maude.exe
```

I.3. Instalación de Maude

Para instalar y ejecutar correctamente Maude deberemos copiar el archivo «*maude.exe*» a un directorio incluido en la variable de entorno PATH (si tras la compilación todavía no lo está). Se puede modificar el fichero «*.bashrc*» del directorio «*home*» para añadir un nuevo directorio a *PATH*.

Se deberá incluir la variable *MAUDE_LIB* en el fichero «*.bashrc*» (*export MAUDE_LIB=~:/usr/local/share/*).

Igualmente, se deberán copiar los archivos *fm.maude*, *model-checker.maude* y *prelude.maude* en un directorio incluido en la variable *MAUDE_LIB*.

Para ejecutar Maude desde la línea de comandos de Windows, deberá crearse la variable de entorno *MAUDE_LIB* con el valor «*/usr/local/share*» (o el directorio donde se encuentren los archivos *prelude*, *model-checker* y *full-maude*).

ANEXO II. MAUDE FOR WINDOWS.

La instalación de Maude sobre un sistema Windows puede ser tediosa como ya se ha comentado en numerosas ocasiones, por ello, se ha realizado un programa de instalación. Si se disponen de versiones ya compiladas de Maude es posible evitar todos los pasos indicados en el Anexo I.

Éste programa permite copiar todos los archivos necesarios en una carpeta del sistema, así como establecer los valores apropiados para las variables del sistema. Igualmente, se proporciona un programa de desinstalación, que devuelve el sistema a su estado anterior, sin dejar ningún rastro del entorno *Maude for Windows*.

Para generar el instalador se ha empleado el software NSIS (Nullsoft Scriptable Install System) [NSIS], bajo licencia de código abierto. Esta herramienta permite generar instaladores completamente configurables mediante un lenguaje de *script*. En el Anexo III se muestra el código fuente de este *script*.

II.1. Programa de instalación.

II.1.1. Tareas realizadas.

El programa de instalación generado se ejecutará en español, inglés o alemán, según la configuración del sistema que lo ejecute o según la selección del usuario. Las acciones que realiza son las siguientes:

1. Copia los archivos ya compilados sobre *CygWin* necesarios para ejecutar Maude, y otros ficheros adicionales. Éstos son:

maude.exe	Fichero ejecutable de Maude.
maude.ico	Icono del programa.
prelude.maude	Fichero prelude.
fm212f.maude	Definición de Full Maude.
model-checker.maude socket.maude	Otros ficheros de Maude.
license.txt	Licencia de Maude (GPL).
fullmaude.bat	Archivo de proceso por lotes para ejecutar Full Maude.
coremaude.bat	Archivo de proceso por lotes para ejecutar Core Maude.
readme.txt	Fichero Léame de «*Maude for Windows*».

Tabla 3: Archivos de Maude que se instalarán con *Maude for Windows*.

   Igualmente, crea los respectivos accesos directos en el menú inicio para ejecutar Core Maude (coremaude.bat) o Full Maude

(fullmaude.bat); así como para el archivo léame de *Maude for Windows*.

2. Copia los archivos requeridos de *CygWin* en la carpeta «cygbin», dentro de la carpeta de instalación seleccionada para *Maude for Windows*.

bash.exe	Ejecutable del *shell* «*bash*».
kill.exe	Ejecutable del programa *kill*.
cygwin1.dll cygncurses-8.dll	Librerías de *CygWin* requeridas.

Tabla 4: Archivos requeridos de CygWin para ejecutar Maude.

3. Añade la carpeta «*INSTDIR\cygbin*» (donde *INSTDIR* es la carpeta elegida para instalar Maude) a la variable de entorno *PATH*. Para realizar esto, se ha hecho uso del *script* «*AddToPath.nsh*» [KiCH05], que simplifica el proceso de restauración de esta variable de entorno en la desinstalación.

4. También se generan sendos enlaces en el menú «Inicio» para acceder directamente tanto a la página web del proyecto Maude, como al proyecto MOMENT.

5. Se creará el directorio «*\tmp*» en la unidad de disco en la que se haya instalado Maude. Esto se hace para evitar que al ejecutar Maude sobre CygWin se advierta que esta carpeta no existe.

6. Por último, se genera el programa de desinstalación, que permite eliminar todo rastro de este software una vez se ejecute.

Cabe mencionar, que los cuatro primeros pasos son opcionales (aunque todos recomendados), por si ya existiera alguna instalación anterior de Maude, CygWin, o cualquier otra circunstancia que considere el usuario.

II.1.2. Ejecución de la instalación.

A continuación se muestra todo el proceso de instalación de Maude sobre un sistema Windows, incluyendo capturas de pantallas de todos los pasos que realiza el instalador.

La versión que instalaremos es «*Maude alpha86a for Windows*», versión *alpha*de la futura versión 2.2 de Maude. Al ejecutar (con privilegios de administrador) el fichero «*MaudeFW_alpha86a.exe*» aparecerá en pantalla el diálogo de selección de idioma de la instalación. El idioma que aparezca seleccionado por defecto corresponderá con el del sistema operativo, si éste está disponible.

Figura 67: Maude for Windows Installer. Diálogo de selección de idioma.

Tras seleccionar el idioma de la instalación, se mostrará la pantalla de bienvenida, y, tras pasarla pulsando el botón siguiente, deberemos aceptar los términos de la licencia GNU GPL bajo la que se distribuye Maude.

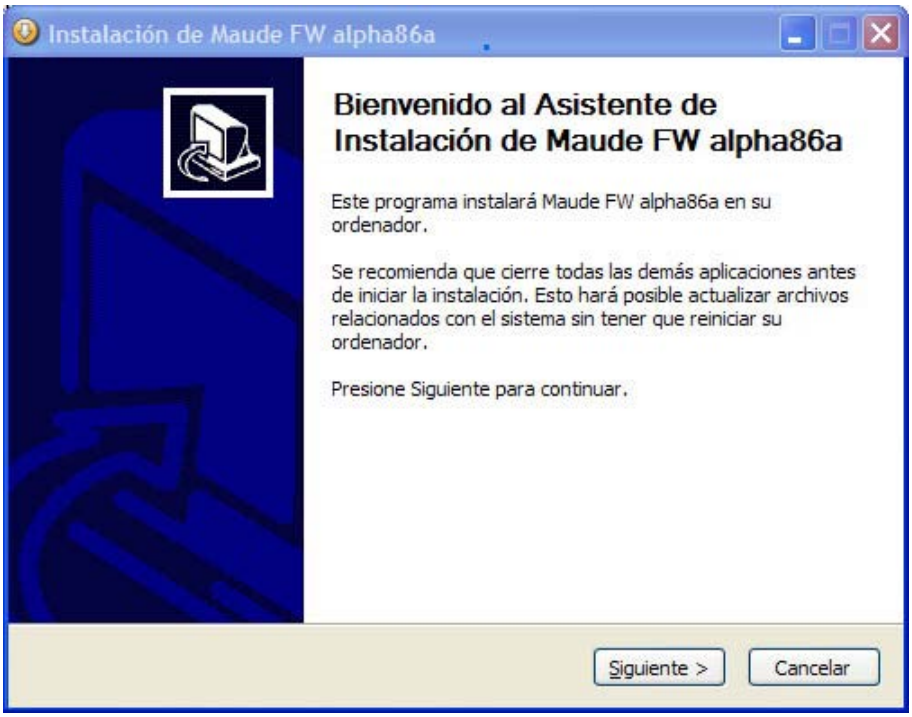

Figura 68: Maude for Windows Installer. Diálogo de bienvenida.

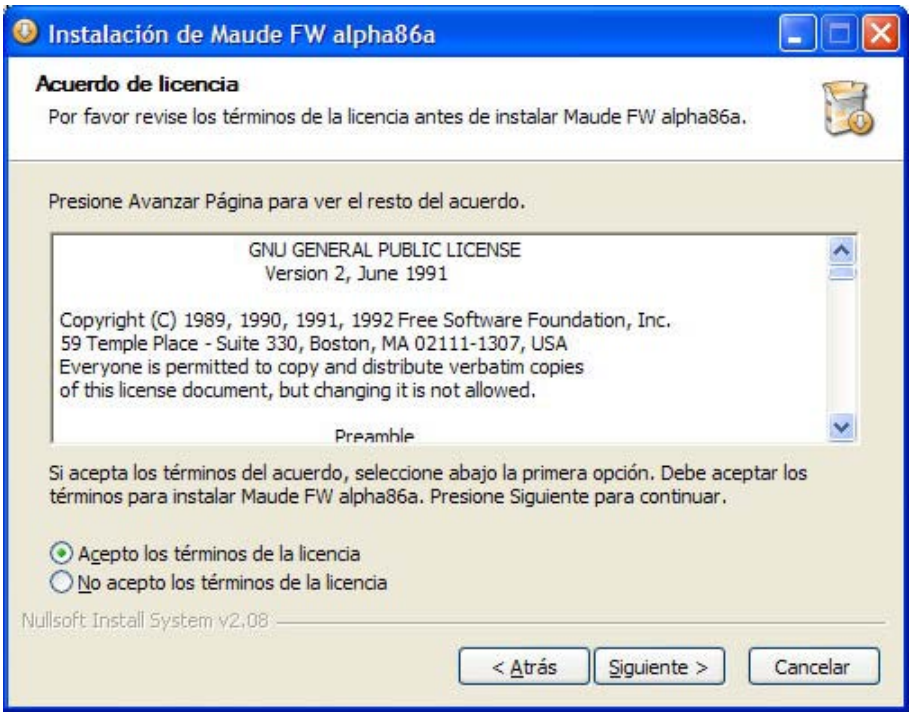

Figura 69: Maude for Windows Installer. Aceptación de la licencia de Maude.

Aceptada la licencia, deberemos seleccionar los componentes que se desean instalar. Se recomienda instalarlos todos para evitar tener que hacer configuraciones a posteriori de forma manual.

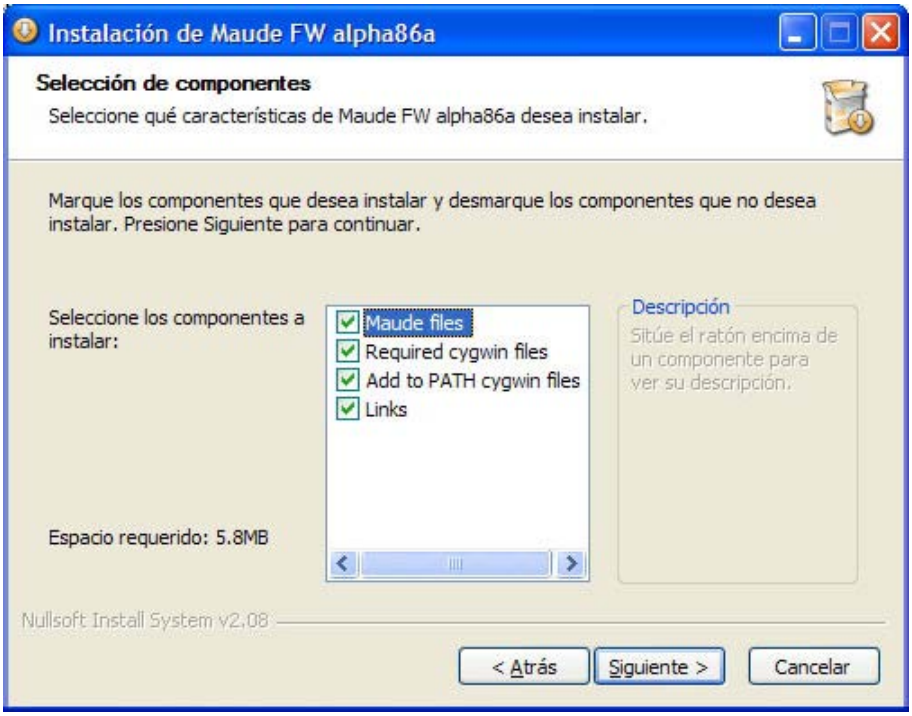

Figura 70: Maude for Windows Installer. Selección de componentes a instalar.

Seleccionaremos también, la carpeta de instalación donde salvaremos Maude. Es indiferente tanto para ejecutar Maude en modo consola, como para utilizarlo junto a las *Maude Development Tools* que la ruta contenga espacios.

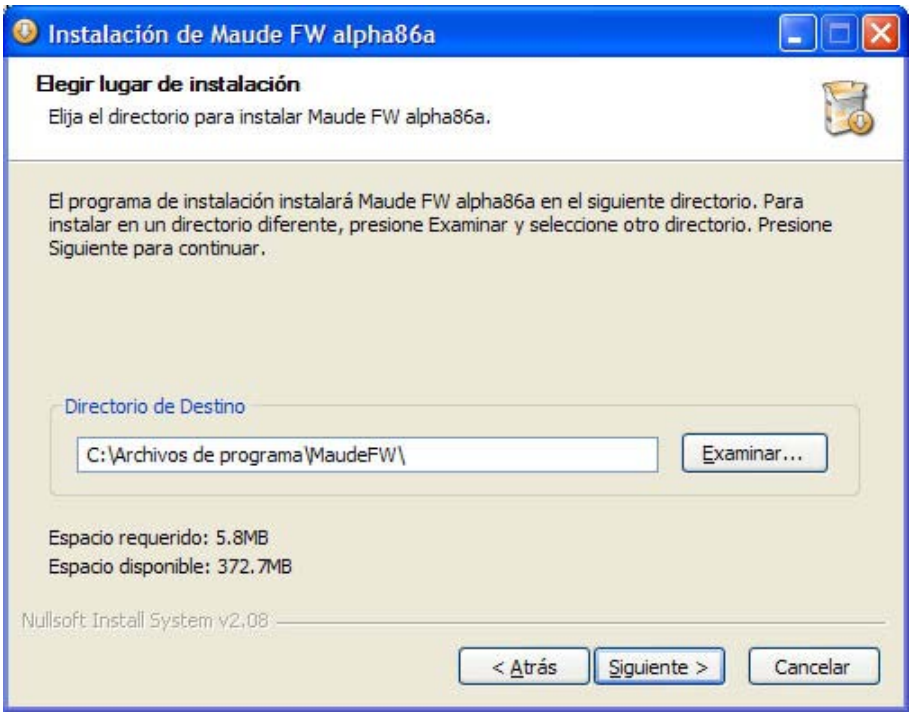

Figura 71: Maude for Windows Installer. Selección de directorio de instalación.

Igualmente, deberemos seleccionar el nombre del nuevo grupo que se creará en el menú inicio para almacenar los accesos directos a los nuevos programas.

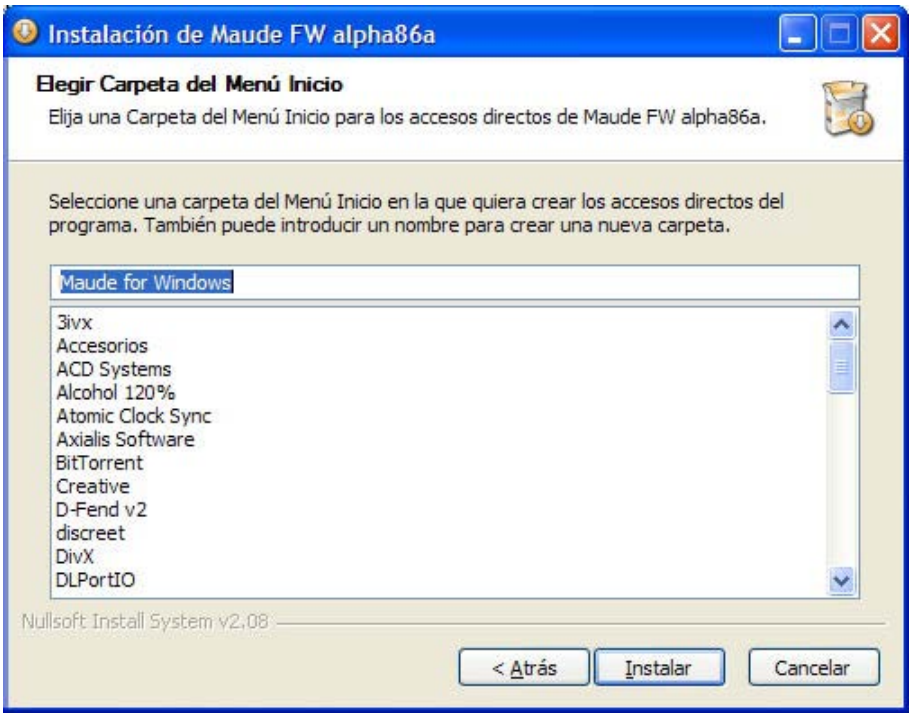

Figura 72: Maude for Windows Installer. Selección del grupo en el *Menú Inicio*.

Una vez se ha configurado la instalación tras todos estos pasos, comenzará el proceso de copia de archivos y configuración del sistema. Si se dispone de un sistema XP o NT no será necesario reiniciar el equipo para que los cambios en la variable de entorno *PATH* se hagan efectivos.

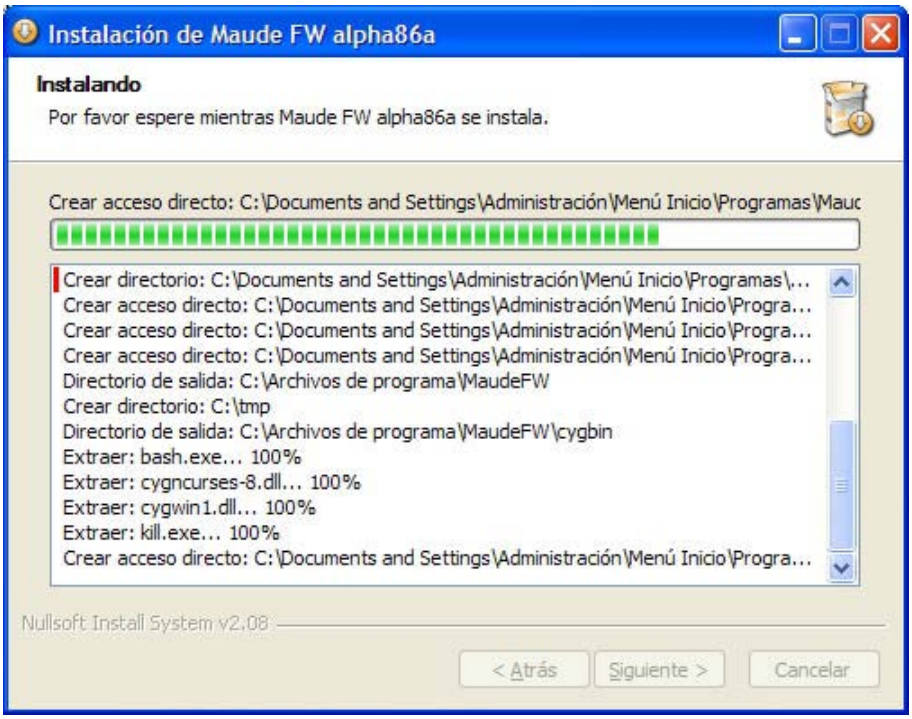

Figura 73: Maude for Windows Installer. Proceso de copia de archivos.

Una vez la instalación ha concluido, se da la opción de mostrar el archivo «LéaMe.txt», que contiene información sobre las modificaciones realizadas en el sistema, así como de otra información adicional sobre *Maude for Windows*.

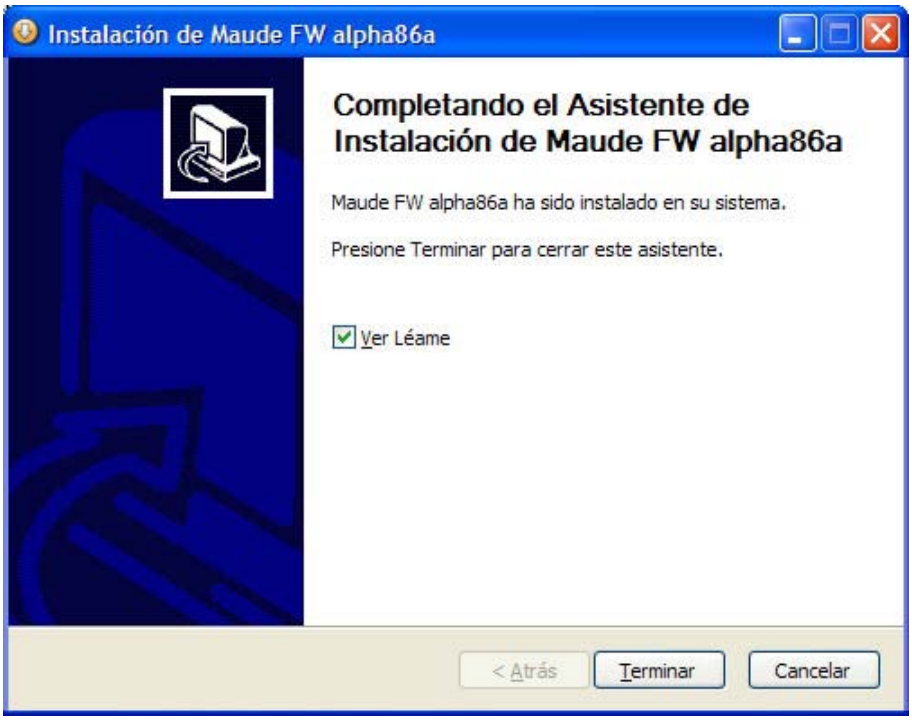

Figura 74: Maude for Windows Installer. Finalización de la instalación.

II.2. Ejecución de Maude for Windows.

La ejecución de *Maude for Windows* se realiza de forma análoga a cualquier programa Windows. Como se muestra en la Figura 75, para iniciar Core Maude o Full Maude, bastará con pinchar sobre el acceso directo en el menú Inicio.

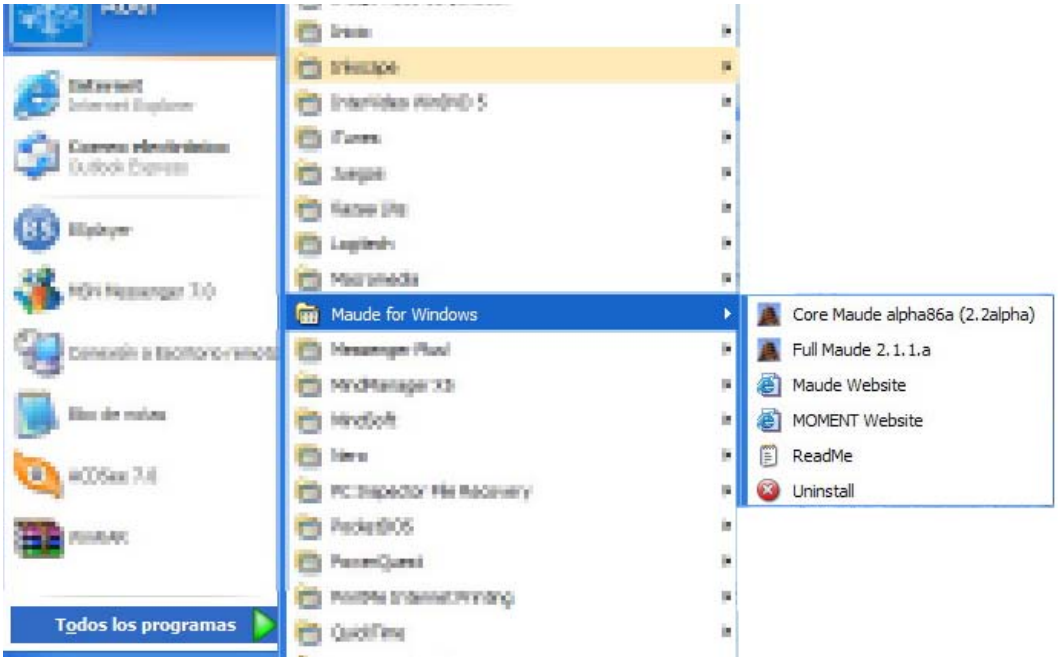

Figura 75: Maude for Windows. Ejecución de Maude.

Esto provocará que se abra una consola, en la que se podrá trabajar con Maude como es habitual, como muestra la Figura 76.

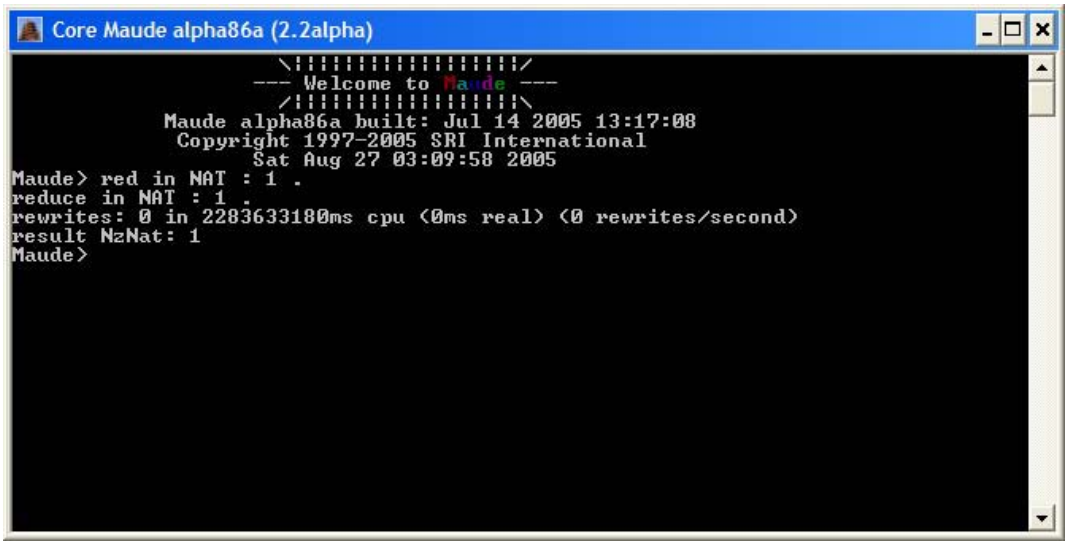

Figura 76: Maude for Windows. Ejemplo de ejecución.

II.3. Programa de desinstalación.

Una vez ejecutado el programa de instalación, se habrá generado automáticamente un programa de desinstalación. Éste permitirá eliminar

automáticamente del sistema la carpeta donde se haya guardado Maude, el grupo del menú inicio, así como eliminar el directorio «*cygbin*» de la variable de entorno *PATH*.

La eliminación de archivos se hará de forma selectiva, esto es, que todo aquel fichero que no proceda de una instalación de *Maude for Windows* no será eliminado a pesar de que se encuentre en la carpeta de instalación.

El instalador, a su vez, da la opción de eliminar por completo la carpeta «*\tmp*» que fue creada durante el proceso de instalación.

II.3.1. Ejecución de la desinstalación.

Tras e jecutar el acceso directo de la desinstalación (o mediante «*Agregar o quitara programas*» de Windows) aparecerá un mensaje que pedirá la confirmación de la acción.

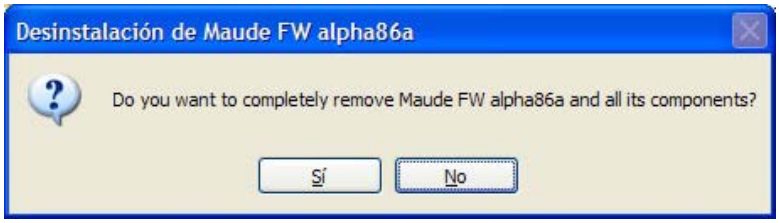

Figura 77: Maude for Windows Uninstaller. Confirmación de desinstalación.

Si confirmamos que deseamos desinstalar el software comenzará el proceso de eliminación de archivos. Cuando éste finalice se preguntará al usuario si desea eliminar también la carpeta «*\tmp*», antes de responder afirmativamente ¡asegúrese de que no contiene ninguna información que le interese!

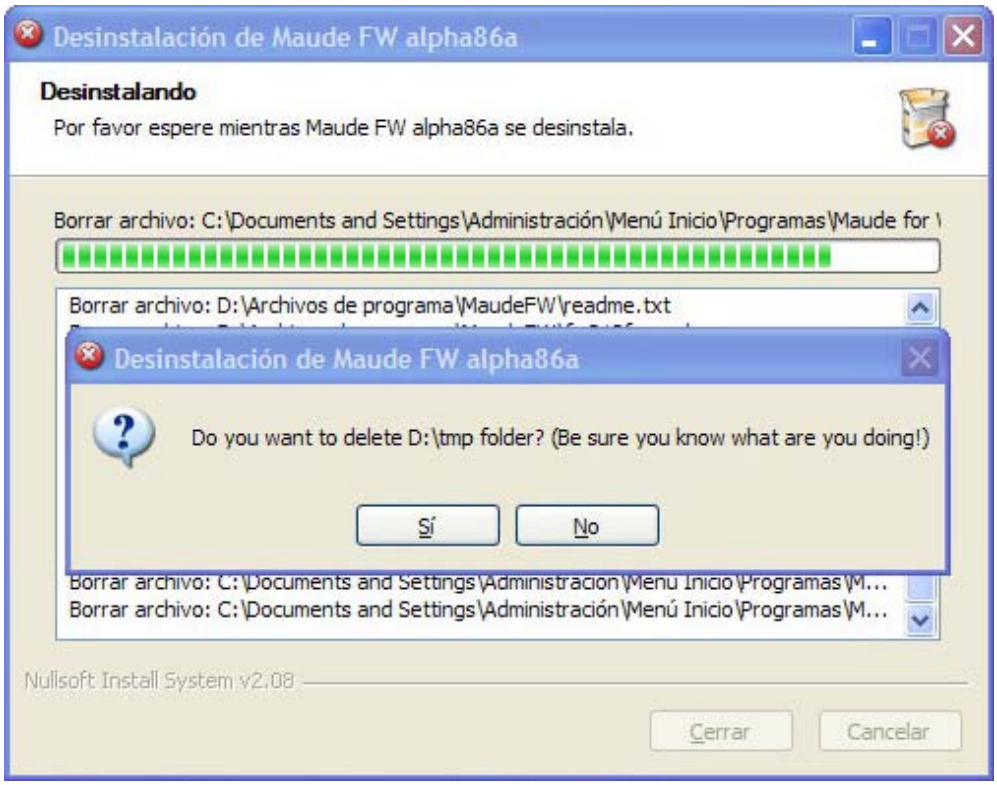

Figura 78: Maude for Windows Uninstaller. Proceso de desinstalación.

Tras la eliminación de todos los archivos, y la actualización de la variable de entorno *PATH*, se mostrará un mensaje informando de que la instalación a finalizado.

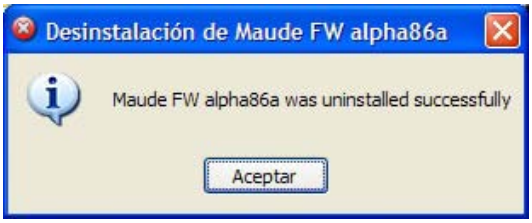

Figura 79: Maude for Windows Uninstaller. Desinstalación finalizada.

ANEXO III. CÓDIGO FUENTE DE MAUDE FOR WINDOWS.

A continuación se incluye el código fuente que, al ser compilado, genera el programa de instalación. Este script también incluye los pasos para que se genere el correspondiente programa de desinstalación una vez se ha instalado *Maude for Windows* en un sistema.

```
; Maude for Windows Installer script
; Gómez Llana, Abel. 2005. agomez@dsic.upv.es

!define ALL_USERS
!include "AddToPath.nsh"

; HM NIS Edit Wizard helper defines
!define PRODUCT_NAME "Maude FW"
!define PRODUCT_VERSION "alpha86a"
!define PRODUCT_PUBLISHER "ISSI"
!define PRODUCT_WEB_SITE "http://moment.dsic.upv.es:8080"
!define PRODUCT_DIR_REGKEY "Software\Microsoft\Windows\CurrentVersion\App
Paths\maude.exe"
!define PRODUCT_UNINST_KEY
"Software\Microsoft\Windows\CurrentVersion\Uninstall\${PRODUCT_NAME}"
!define PRODUCT_UNINST_ROOT_KEY "HKLM"
!define PRODUCT_STARTMENU_REGVAL "NSIS:StartMenuDir"
!define MAUDE_WEB_SITE "http://maude.cs.uiuc.edu"

; MUI 1.67 compatible ------
!include "MUI.nsh"

; MUI Settings
!define MUI_ABORTWARNING
!define MUI_ICON "${NSISDIR}\Contrib\Graphics\Icons\orange-install.ico"
!define MUI_UNICON "${NSISDIR}\Contrib\Graphics\Icons\orange-uninstall.ico"

; Welcome page
!insertmacro MUI_PAGE_WELCOME
; License page
!define MUI_LICENSEPAGE_RADIOBUTTONS
!insertmacro MUI_PAGE_LICENSE "license.txt"
; Components page
!insertmacro MUI_PAGE_COMPONENTS
; Directory page
!insertmacro MUI_PAGE_DIRECTORY
; Start menu page
var ICONS_GROUP
!define MUI_STARTMENUPAGE_NODISABLE
!define MUI_STARTMENUPAGE_DEFAULTFOLDER "Maude for Windows"
!define MUI_STARTMENUPAGE_REGISTRY_ROOT "${PRODUCT_UNINST_ROOT_KEY}"
!define MUI_STARTMENUPAGE_REGISTRY_KEY "${PRODUCT_UNINST_KEY}"
!define MUI_STARTMENUPAGE_REGISTRY_VALUENAME "${PRODUCT_STARTMENU_REGVAL}"
!insertmacro MUI_PAGE_STARTMENU Application $ICONS_GROUP
; Instfiles page
!insertmacro MUI_PAGE_INSTFILES
; Finish page
;!define MUI_FINISHPAGE_RUN "$INSTDIR\coremaude.bat"
!define MUI_FINISHPAGE_SHOWREADME "$INSTDIR\readme.txt"
!insertmacro MUI_PAGE_FINISH

; Uninstaller pages
!insertmacro MUI_UNPAGE_INSTFILES

; Language files
!insertmacro MUI_LANGUAGE "Spanish"
!insertmacro MUI_LANGUAGE "German"
!insertmacro MUI_LANGUAGE "English"

; MUI end ------

Name "${PRODUCT_NAME} ${PRODUCT_VERSION}"
OutFile "MaudeFW_${PRODUCT_VERSION}.exe"
```

```
InstallDir "$PROGRAMFILES\MaudeFW"
InstallDirRegKey HKLM "${PRODUCT_DIR_REGKEY}" ""
ShowInstDetails show
ShowUnInstDetails show

Function .onInit
  !insertmacro MUI_LANGDLL_DISPLAY
FunctionEnd

Section "Maude files" SEC01
  SetOverwrite on
  SetOutPath "$INSTDIR"
  File "fm212f.maude"
  File "license.txt"
  File "maude.exe"
  File "maude.ico"
  File "model-checker.maude"
  File "prelude.maude"
  File "socket.maude"
  File "fullmaude.bat"
  File "coremaude.bat"
  File "readme.txt"

; Shortcuts
  !insertmacro MUI_STARTMENU_WRITE_BEGIN Application
  CreateDirectory "$SMPROGRAMS\$ICONS_GROUP"
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\Full Maude 2.1.1.a.lnk"
"$INSTDIR\fullmaude.bat" "" "$INSTDIR\maude.ico"
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\Core Maude alpha86a (2.2alpha).lnk"
"$INSTDIR\coremaude.bat" "" "$INSTDIR\maude.ico"
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\ReadMe.lnk" "$INSTDIR\readme.txt"
  !insertmacro MUI_STARTMENU_WRITE_END
SectionEnd

var INSTDRIVE
Section "Required cygwin files" SEC02
  SetOutPath "$INSTDIR"
  SetOverwrite on
  StrCpy $INSTDRIVE $INSTDIR 2
  CreateDirectory "$INSTDRIVE\tmp"

  SetOutPath "$INSTDIR\cygbin"
  File "cygbin\bash.exe"
  File "cygbin\cygncurses-8.dll"
  File "cygbin\cygwin1.dll"
  File "cygbin\kill.exe"

; Shortcuts
  !insertmacro MUI_STARTMENU_WRITE_BEGIN Application
  !insertmacro MUI_STARTMENU_WRITE_END
SectionEnd

Section "Add to PATH cygwin files" SEC03
  Push "$INSTDIR\cygbin"
  Call AddToPath

; Shortcuts
  !insertmacro MUI_STARTMENU_WRITE_BEGIN Application
  !insertmacro MUI_STARTMENU_WRITE_END
SectionEnd

Section "Links" SEC04

; Shortcuts
  !insertmacro MUI_STARTMENU_WRITE_BEGIN Application
  WriteIniStr "$INSTDIR\${PRODUCT_NAME}.url" "InternetShortcut" "URL"
"${PRODUCT_WEB_SITE}"
  WriteIniStr "$INSTDIR\Maude.url" "InternetShortcut" "URL" "${MAUDE_WEB_SITE}"
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\MOMENT Website.lnk"
"$INSTDIR\${PRODUCT_NAME}.url"
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\Maude Website.lnk" "$INSTDIR\Maude.url"
  !insertmacro MUI_STARTMENU_WRITE_END
SectionEnd

Section -AdditionalIcons
  !insertmacro MUI_STARTMENU_WRITE_BEGIN Application
  CreateShortCut "$SMPROGRAMS\$ICONS_GROUP\Uninstall.lnk" "$INSTDIR\uninst.exe"
  !insertmacro MUI_STARTMENU_WRITE_END
SectionEnd

Section -Post
  WriteUninstaller "$INSTDIR\uninst.exe"
  WriteRegStr HKLM "${PRODUCT_DIR_REGKEY}" "" "$INSTDIR\maude.exe"
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "DisplayName"
"$(^Name)"
```

```
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "UninstallString"
"$INSTDIR\uninst.exe"
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "DisplayIcon"
"$INSTDIR\maude.exe"
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "DisplayVersion"
"${PRODUCT_VERSION}"
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "URLInfoAbout"
"${PRODUCT_WEB_SITE}"
  WriteRegStr ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}" "Publisher"
"${PRODUCT_PUBLISHER}"
SectionEnd

; Section descriptions
!insertmacro MUI_FUNCTION_DESCRIPTION_BEGIN
  !insertmacro MUI_DESCRIPTION_TEXT ${SEC01} "Maude required files."
  !insertmacro MUI_DESCRIPTION_TEXT ${SEC02} "Additional cygwin files (required if
cygwin isn't installed). A /tmp folder will be created too."
  !insertmacro MUI_DESCRIPTION_TEXT ${SEC03} "Add cygwin folder to the environment
variable 'PATH' (required if not added yet)."
  !insertmacro MUI_DESCRIPTION_TEXT ${SEC04} "Shortcuts to the Maude and the MOMENT
websites."
!insertmacro MUI_FUNCTION_DESCRIPTION_END


Function un.onUninstSuccess
  HideWindow
  MessageBox MB_ICONINFORMATION|MB_OK "$(^Name) was uninstalled successfully"
FunctionEnd

Function un.onInit
  MessageBox MB_ICONQUESTION|MB_YESNO|MB_DEFBUTTON2 "Do you want to completely remove
$(^Name) and all its components?" IDYES +2
  Abort
FunctionEnd

Section Uninstall
  Push $INSTDIR\cygbin
  Call un.RemoveFromPath

  !insertmacro MUI_STARTMENU_GETFOLDER "Application" $ICONS_GROUP
  Delete "$INSTDIR\${PRODUCT_NAME}.url"
  Delete "$INSTDIR\Maude.url"
  Delete "$INSTDIR\uninst.exe"
  Delete "$INSTDIR\coremaude.bat"
  Delete "$INSTDIR\fullmaude.bat"
  Delete "$INSTDIR\socket.maude"
  Delete "$INSTDIR\prelude.maude"
  Delete "$INSTDIR\model-checker.maude"
  Delete "$INSTDIR\maude.ico"
  Delete "$INSTDIR\maude.exe"
  Delete "$INSTDIR\license.txt"
  Delete "$INSTDIR\readme.txt"
  Delete "$INSTDIR\fm212f.maude"
  Delete "$INSTDIR\cygbin\kill.exe"
  Delete "$INSTDIR\cygbin\bash.exe"
  Delete "$INSTDIR\cygbin\cygwin1.dll"
  Delete "$INSTDIR\cygbin\cygncurses-8.dll"

  Delete "$SMPROGRAMS\$ICONS_GROUP\Uninstall.lnk"
  Delete "$SMPROGRAMS\$ICONS_GROUP\ReadMe.lnk"
  Delete "$SMPROGRAMS\$ICONS_GROUP\Maude Website.lnk"
  Delete "$SMPROGRAMS\$ICONS_GROUP\MOMENT Website.lnk"
  Delete "$SMPROGRAMS\$ICONS_GROUP\Core Maude alpha86a (2.2alpha).lnk"
  Delete "$SMPROGRAMS\$ICONS_GROUP\Full Maude 2.1.1.a.lnk"

  RMDir "$SMPROGRAMS\$ICONS_GROUP"
  RMDir "$INSTDIR\cygbin"
  RMDir "$INSTDIR"

  StrCpy $INSTDRIVE $INSTDIR 2

  MessageBox MB_ICONQUESTION|MB_YESNO|MB_DEFBUTTON2 "Do you want to delete
$INSTDRIVE\tmp folder? (Be sure you know what are you doing!)" IDNO +2
  RMDir /R "$INSTDRIVE\tmp"

  DeleteRegKey ${PRODUCT_UNINST_ROOT_KEY} "${PRODUCT_UNINST_KEY}"
  DeleteRegKey HKLM "${PRODUCT_DIR_REGKEY}"
  SetAutoClose true
SectionEnd
```

ANEXO IV. FICHEROS DE GRAMÁTICA DE ANTLR PARA DESCOMPONER COMANDOS DE FULL MAUDE.

IV.1. fm.g

```
header {
     package es.upv.dsic.issi.moment.maudedaemon.parser;
}

class FullMaudeCommandsParser extends Parser;

options {
buildAST = true;
classHeaderSuffix=org.norecess.antlr.TestableParser;
exportVocab=FullMaudeVocab;
}
    tokens {
    TERM;
    PROGRAM;
    }
    {
        boolean myFailure;
    boolean myQuietErrors;
    java.util.logging.Logger log =
java.util.logging.Logger.getLogger(FullMaudeCommandsParser.class.getName());

    public boolean successfulParse() {
        return !myFailure;
    }

    public void setQuietErrors(boolean quiet) {
        myQuietErrors=quiet;
    }

    public void atEOF() {
        try{
            match(EOF);
        } catch(Exception e) {
            myFailure=true;
        }
    }
    public void reportError(RecognitionException arg0) {
        myFailure=true;
        if(!myQuietErrors)
            super.reportError(arg0);
    }
}


program : (term|!TODO)* {## = #(#[PROGRAM, "programa"], ##);};

term : (p:PARENT_AB^)(~(PARENT_AB|PARENT_CE)|intermediateTerm)* PARENT_CE
        {#p.setType(TERM);}
        {log.finer("Term found");};

intermediateTerm : PARENT_AB (~(PARENT_AB|PARENT_CE)|intermediateTerm)* PARENT_CE
        {log.finest("Intermediate Term found");};

class FullMaudeCommandsLexer extends Lexer;

options {
          filter = false;
```

```
          charVocabulary = '\3'..'\377';
}
TODO_MENOS_LOS_COMENTARIOS: ("***") => COMENTARIO {$setType(Token.SKIP);}
        //                    | (NL) => NL {$setType(Token.SKIP);}
        //                    | (WS) => WS {$setType(Token.SKIP);}
                            | TODO {$setType(TODO);};

protected COMENTARIO : "***" (~('\n'|'\r'))* { $setType(Token.SKIP);} ;
protected TODO : ~('('|')') (~('('|')'|'*'))* ;

protected WS : ' ' | '\t';
//protected LETRA : 'a'..'z' | 'A'..'Z';
//protected DIGITO : '0'..'9';
protected NL :
  (
    ("\r\n") => "\r\n"
  | '\r'
  | '\n'
  ) {newline(); $setType(Token.SKIP); } ;

PARENT_AB : '(';
PARENT_CE : ')';
```

IV.2. baseTermsJoiner.g

```
header {
     package es.upv.dsic.issi.moment.maudedaemon.parser;

     import java.util.List;
     import java.util.ArrayList;
}

class BaseTermsJoinerTreeParser extends TreeParser;

options {
    buildAST = false;
    importVocab = FullMaudeVocab;
}
{
       private String extractText(AST t) {
                        StringBuffer sb = new StringBuffer();
                       sb.append(t.getText());
                       AST child = t.getFirstChild();
                       while(child!=null) {
                              sb.append(extractText(child));
                              child = child.getNextSibling();
                       }
                  return sb.toString();
       }
}
program returns [List<String> terms = new ArrayList<String>()] {String termino;}
       : #(PROGRAM (termino=baseterm {terms.add(termino);})*) ;

baseterm returns [String s=null;]
    : #(t:TERM (term|TODO)* )
    {
      s = extractText(#t);
    };
term : #(TERM (term|TODO)*);
```

ANEXO V. MODELOS EMF Y SU REPRESENTACIÓN COMO TÉRMINOS DEL ÁLGEBRA DE MOMENT PARA UN EJEMPLO SENCILLO DE TRANSFORMACIÓN DE MODELOS.

V.1. Modelo EMF de Simple Purchase Order (UML).

```
<?xml version="1.0" encoding="UTF-8"?>
<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="purchaseOrder"
    nsURI="http://es.upv.dsic.issi/moment/purchaseOrder.ecore" nsPrefix="purchaseOrder">
  <eClassifiers xsi:type="ecore:EClass" name="Item">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="productName" unique="false"
        lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="quantity" unique="false"
        lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="USPrice" unique="false"
        lowerBound="1" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="PurchaseOrder">
    <eStructuralFeatures xsi:type="ecore:EReference" name="items" upperBound="-1"
        eType="#//Item" containment="true" resolveProxies="false"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="shipTo" unique="false"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="billTo" unique="false"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>
```

V.2. Término para Simple Purchase Order (UML).

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

Set {
```

```
(ecore-EPackage 'platform:/resource/PropagateChanges/Models/uml.ecore#/ "purchaseOrder"
"http://es.upv.dsic.issi/moment/purchaseOrder.ecore" "purchaseOrder" empty-orderedset
OrderedSet {'#// } OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder } empty-orderedset
empty-orderedset ),
(ecore-EClass 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item "Item" "" "0"
"0" false false empty-orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#/ } empty-orderedset empty-
orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice } empty-orderedset
empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice } empty-orderedset
empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice } empty-orderedset
empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice } ),
(ecore-EAttribute
'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName "productName"
true false 1 1 false true true false false "" "0" false false false empty-orderedset
OrderedSet {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } ),
(ecore-EAttribute 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity
"quantity" true false 1 1 false true true false false "" "0" false false false empty-
orderedset OrderedSet {'http://www.eclipse.org/emf/2002/Ecore#//EInt } OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EInt } ),
(ecore-EAttribute 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice
"USPrice" true false 1 1 false true true false false "" "0" false false false empty-
orderedset OrderedSet {'http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal } OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal } ),
(ecore-EClass 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder
"PurchaseOrder" "" "0" "0" false false empty-orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#/ } empty-orderedset empty-
orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items } empty-
orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo } empty-
orderedset empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo  ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo } ),
(ecore-EReference
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/items "items" true
true 0 -1 true false true false false "" "0" false false true false false empty-
orderedset OrderedSet {'platform:/resource/PropagateChanges/Models/uml.ecore#//Item }
OrderedSet {'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder }
empty-orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item } ),
(ecore-EAttribute
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo "shipTo"
true false 0 1 false false true false false "" "0" false false false empty-orderedset
OrderedSet {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } ),
(ecore-EAttribute
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo "billTo"
true false 0 1 false false true false false "" "0" false false false empty-orderedset
OrderedSet {'http://www.eclipse.org/emf/2002/Ecore#//EString } OrderedSet
{'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder } OrderedSet
{'http://www.eclipse.org/emf/2002/Ecore#//EString } )
}
```

V.3. Término para Simple Purchase Order (RDBMS) devuelto por ModelGen.

```
***$ rdbms - "http:///es.upv.dsic.issi/moment/rdbms_mm.ecore"

Set{(rdbms-Schema
    'platform:/resource/PropagateChanges/Models/uml.ecore#/ "purchaseOrder" "0"
    OrderedSet{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item ::
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}),(
    rdbms-Key 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_pk
    "Item_pk" "0" OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item} OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_tid}),(
    rdbms-Key
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_pk
    "PurchaseOrder_pk" "0" OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}

OrderedSet{'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_tid}),(
rdbms-ForeignKey 'PurchaseOrder_items_Item
    "PurchaseOrder_items_Item" "0" OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_pk}
    OrderedSet{'PurchaseOrder_items_Item_tid} OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}),(
    rdbms-Table 'platform:/resource/PropagateChanges/Models/uml.ecore#//Item
    "Item" "0" OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_pk}
    empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice ::
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName ::
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity ::
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_tid}
    OrderedSet{'platform:/resource/PropagateChanges/Models/uml.ecore#/}),(
    rdbms-Table
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder
    "PurchaseOrder" "0" OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_pk}
    OrderedSet{'PurchaseOrder_items_Item} OrderedSet{
    'PurchaseOrder_items_Item_tid ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo ::
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo ::
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_tid}
    OrderedSet{'platform:/resource/PropagateChanges/Models/uml.ecore#/}),(
    rdbms-Column 'PurchaseOrder_items_Item_tid "PurchaseOrder_items_Item_tid"
    "0" "NUMBER" OrderedSet{'PurchaseOrder_items_Item} empty-orderedset
    OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}),(
    rdbms-Column
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/USPrice
    "USPrice" "0" "http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal"
    empty-orderedset empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item}),(
    rdbms-Column
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/productName
    "productName" "0" "http://www.eclipse.org/emf/2002/Ecore#//EString"
    empty-orderedset empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item}),(
    rdbms-Column
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item/quantity
    "quantity" "0" "http://www.eclipse.org/emf/2002/Ecore#//EInt"
    empty-orderedset empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item}),(
    rdbms-Column
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_tid "Item_tid"
    "0" "NUMBER" empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//Item_pk}
    OrderedSet{'platform:/resource/PropagateChanges/Models/uml.ecore#//Item}),(
    rdbms-Column
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/billTo "billTo"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EString"
    empty-orderedset empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}),(
    rdbms-Column
'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder/shipTo "shipTo"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EString"
    empty-orderedset empty-orderedset OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder}),(
    rdbms-Column
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_tid
    "PurchaseOrder_tid" "0" "NUMBER" empty-orderedset OrderedSet{
```

```
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder_pk}
    OrderedSet{
    'platform:/resource/PropagateChanges/Models/uml.ecore#//PurchaseOrder})}
```

V.4. Modelo EMF para Simple Purchase Order (RDBMS).

```
<?xml version="1.0" encoding="ASCII"?>
<rdbms:Schema xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:rdbms="http:///es.upv.dsic.issi/moment/rdbms_mm.ecore" name="purchaseOrder"
kind="0">
  <tables name="Item" kind="0">
    <key name="Item_pk" kind="0" column="//@tables.0/@column.3"/>
    <column name="USPrice" kind="0"
type="http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal"/>
    <column name="productName" kind="0"
type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="quantity" kind="0"
type="http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <column name="Item_tid" kind="0" type="NUMBER" key="//@tables.0/@key.0"/>
  </tables>
  <tables name="PurchaseOrder" kind="0">
    <key name="PurchaseOrder_pk" kind="0" column="//@tables.1/@column.3"/>
    <foreignKey name="PurchaseOrder_items_Item" kind="0" refersTo="//@tables.0/@key.0"
column="//@tables.1/@column.0"/>
    <column name="PurchaseOrder_items_Item_tid" kind="0" type="NUMBER"
foreignKey="//@tables.1/@foreignKey.0"/>
    <column name="billTo" kind="0"
type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="shipTo" kind="0"
type="http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <column name="PurchaseOrder_tid" kind="0" type="NUMBER" key="//@tables.1/@key.0"/>
  </tables>
</rdbms:Schema>
```

V.5. Término para Simple Purchase Order (RDBMS) con los identificadores actualizados.

```
*** Generated by Moment
***$ rdbms - "http:///es.upv.dsic.issi/moment/rdbms_mm.ecore"

Set {
(rdbms-Schema 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#/ "purchaseOrder"
"0" OrderedSet { 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } ),
(rdbms-Table 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 "Item"
"0" OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@key.0 } empty-
orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.0  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.1  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.2  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.3 }
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#/ } ),
(rdbms-Key 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@key.0
"Item_pk" "0" OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 } OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.3 } ),
```

```
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.0 "USPrice"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EBigDecimal" empty-orderedset empty-
orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.1
"productName" "0" "http://www.eclipse.org/emf/2002/Ecore#//EString" empty-orderedset
empty-orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.2 "quantity"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EInt" empty-orderedset empty-orderedset
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@column.3 "Item_tid"
"0" "NUMBER" empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@key.0 }
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0 } ),
(rdbms-Table 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1
"PurchaseOrder" "0" OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@key.0 }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@foreignKey.0 }
OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.0  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.1  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.2  ::
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.3 }
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#/ } ),
(rdbms-Key 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@key.0
"PurchaseOrder_pk" "0" OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.3 } ),
(rdbms-ForeignKey
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@foreignKey.0
"PurchaseOrder_items_Item" "0" OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.0/@key.0 } OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.0 }
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.0
"PurchaseOrder_items_Item_tid" "0" "NUMBER" OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@foreignKey.0 }
empty-orderedset OrderedSet
{'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.1 "billTo"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EString" empty-orderedset empty-orderedset
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.2 "shipTo"
"0" "http://www.eclipse.org/emf/2002/Ecore#//EString" empty-orderedset empty-orderedset
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } ),
(rdbms-Column
'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@column.3
"PurchaseOrder_tid" "0" "NUMBER" empty-orderedset OrderedSet
{ 'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1/@key.0 }
OrderedSet {'platform:/resource/PropagateChanges/Models/rdbms.rdbms#//@tables.1 } )
}
```

ANEXO VI. CREACIÓN DE UN PROYECTO EMF.

A continuación ilustraremos el procedimiento para generar un proyecto EMF. Mostraremos, por ejemplo, el proceso de generación del proyecto para el metamodelo de trazabilidad básico. En primer lugar, iniciaremos el asistente de creación de nuevo proyecto de Eclipse, y seleccionaremos «*EMF Project*».

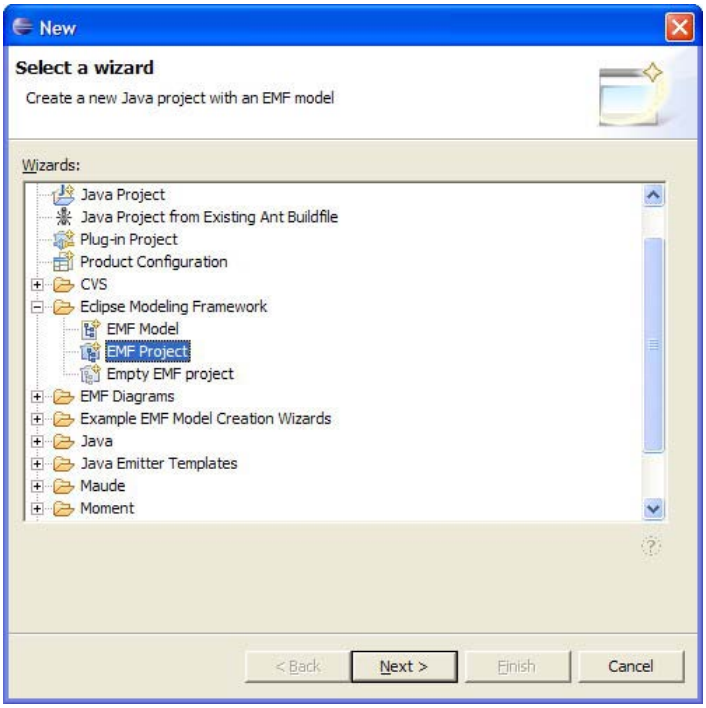

Figura 80: Diálogo de creación de nuevo proyecto EMF.

El siguiente paso del asistente pregunta por el nombre que tendrá el proyecto. Por ejemplo, para la creación del metamodelo de trazabilidad básico, indicaremos «*es.upv.dsic.issi.moment.traceability.basicmetamodel*».

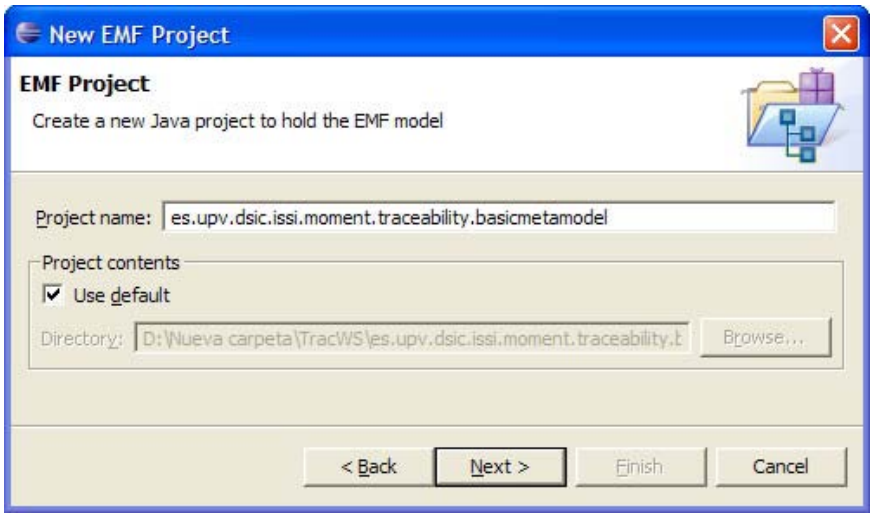

Figura 81: Diálogo para indicar el nombre del nuevo proyecto EMF.

En este punto del asistente se nos solicitará a partir de qué tipo de modelo se desea crear un proyecto EMF. Inicialmente se pueden seleccionar un modelo Ecore, un esquema XML y un modelo de Racional Rose. Si se ha instalado el soporte para UML2, también aparecerá la opción correspondiente.

En caso de que se desee crear un proyecto para un modelo a partir de Java anotado, es necesario crear primero el modelo Ecore mediante el asistente "Nuevo modelo EMF", y seleccionar el archivo *.ecore* generado en este punto del asistente.

En este caso, nosotros seleccionamos la opción de un modelo Ecore.

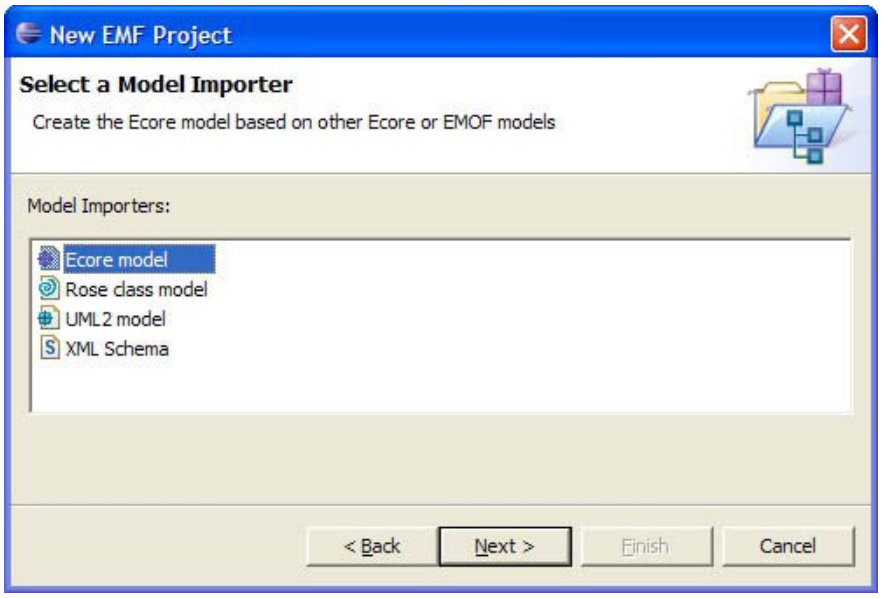

Figura 82: Diálogo de selección de modelo origen.

Una vez seleccionado el tipo de fuente para generar el proyecto EMF, deberemos indicar el nombre del fichero ecore a partir del cual generaremos el proyecto. Igualmente, se indicará en este paso del asistente el nombre que tendrá el fichero del modelo generador.

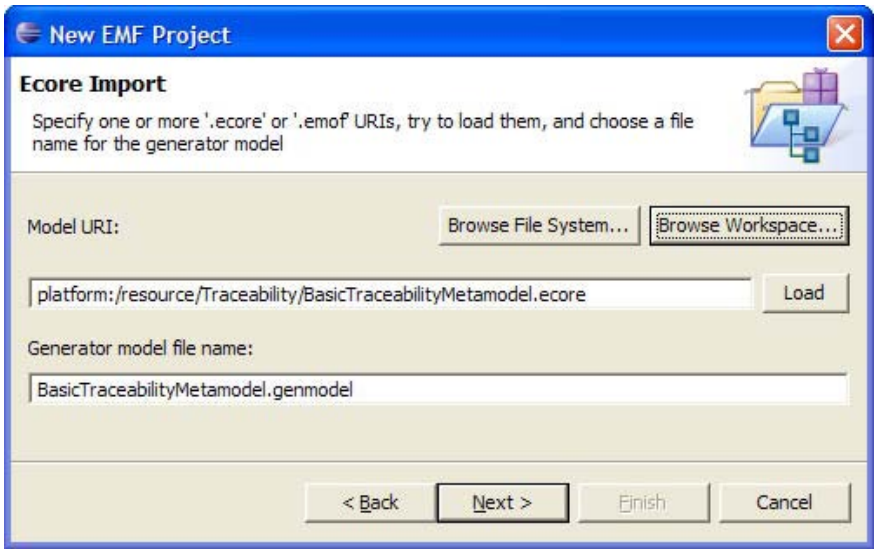

Figura 83: Élección del modelo Ecore y el modelo generador.

Por último, seleccionaremos del modelo ecore seleccionado sobre qué paquetes desearemos generar código, y el nombre para sus ficheros de modelo. En caso de que el modelo *core* tenga referencias a otros modelos, se indicarán en este punto. Una vez realizados todos estos pasos, puede pulsarse el botón *Finish*, terminando el asistente, y mostrándose en pantalla el modelo generador.

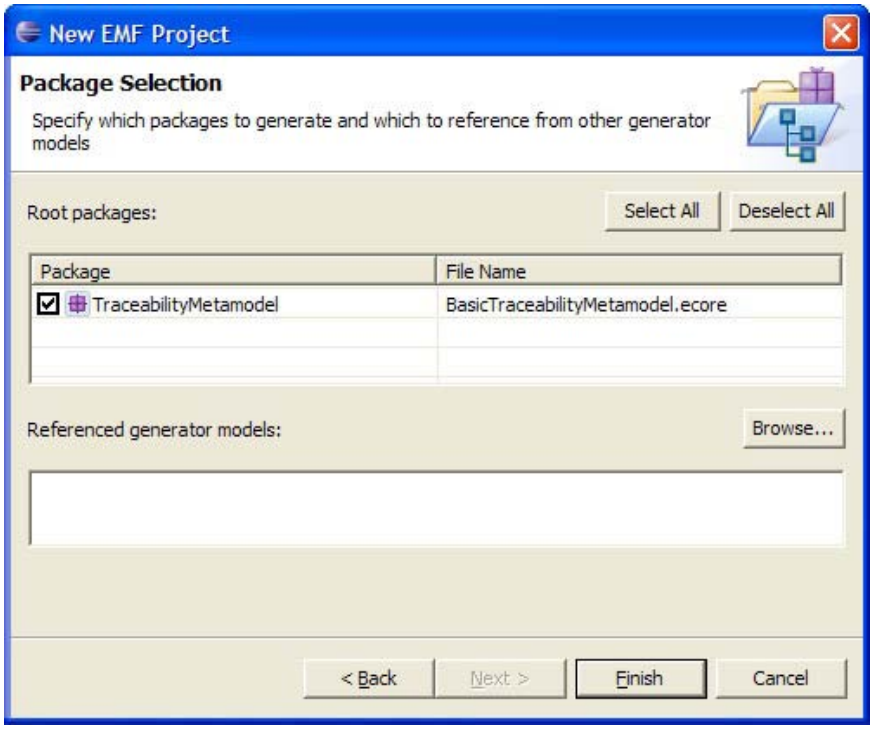

Figura 84: Selección de elementos a generar y sus referencias.