UNIVERSIDAD POLITÉCNICA DE VALENCIA

FACULTAD DE INFORMÁTICA.

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN.

SOPORTE OCL EN MOMENT, UNA HERRAMIENTA DE GESTIÓN DE MODELOS

Proyecto final de carrera

Septiembre de 2006. Valencia

Realizado por

Oriente Cantos, Joaquín

Dirigido por

Artur Boronat Moll

Dr. José Á. Carsí Cubel

Índice de contenidos

Índice de figuras

Índice de tablas

Capítulo 1

Presentación del proyecto

1.1 Introducción

La Ingeniería Dirigida por Modelos (Model Driven Engineering, MDE) es una técnica emergente en la Ingeniería del Software basada en el uso sistemático de modelos. Un modelo describe una realidad física, abstracta o hipotética, recogiendo únicamente la información necesaria que permite conseguir unos objetivos específicos: generación de código, integración de aplicaciones, interoperabilidad entre aplicaciones, etc. Trabajar directamente sobre modelos aumenta el nivel de abstracción de estas tareas y permite automatizarlas. No obstante, en muchas ocasiones los modelos no constituyen una especificación lo suficientemente precisa y libre de ambigüedades de un sistema. Además, en muchas ocasiones, y debido a la complejidad actual de los sistemas y sus requerimientos, no es posible expresar ciertas características o reglas de negocio utilizando únicamente modelos gráficos. Se presentó, pues, el reto de encontrar una solución para aumentar la expresividad y desarrollar modelos completos y precisos. Con este propósito, el Object Management Group (OMG) ha especificado el Object Constraint Language, o como se conoce habitualmente, OCL [WarK03]. Se trata de un lenguaje notacional para el análisis y diseño de sistemas software que originalmente se ideó para especificar modelos UML/OCL combinados. En la actualidad OCL se ve involucrado en múltiples proyectos, en los que se pretende integrar el soporte de OCL en herramientas de desarrollo de software dirigido por modelos.

OCL hace más real la utilización de modelos como artefacto fundamental en el desarrollo de sistemas, siguiendo la propuesta de MDE. No obstante, la Ingeniería Dirigida por Modelos plantea un segundo reto: la manipulación y transformación de modelos, en el contexto de un proceso de desarrollo de refinamiento de modelos desde los requerimientos hasta la generación del código ejecutable de la aplicación. Tradicionalmente, tareas como la integración o transformación de modelos se habían solucionado de manera ad-hoc para un específico contexto: bases de datos relacionales, esquemas XML, ontologías, programación orientada a objetos, etc. La Gestión de Modelos es una nueva disciplina, enmarcada dentro de MDE, que trata de dar soluciones abstractas y reutilizables para problemas de esta clase. La disciplina de Gestión de Modelos manipula los artefactos software mediante una serie de operadores genéricos basados en relaciones establecidas entre modelos. Estos operadores tratan a los modelos como ciudadanos de primera clase, incrementando el nivel de abstracción de la solución, siendo ésta independiente del contexto o tecnología utilizados para representarlos.

El éxito de esta emergente disciplina dependerá de la calidad y productividad de las aplicaciones que soporten estos conceptos. Así, en este proyecto final de carrera se amplia la funcionalidad de una herramienta de Gestión de Modelos existente que es MOMENT (MOdel manageMENT).

MOMENT es un framework, el cual está integrado en la plataforma Eclipse y que proporciona una serie de operadores genéricos para manipular modelos a través de un entorno de modelado industrial como es Eclipse Modeling Framework (EMF). El álgebra de los operadores de Gestión de Modelos se ha especificado de manera genérica usando el formalismo de especificación algebraica Maude. Además se proporciona soporte para la trazabilidad de las transformaciones aplicadas a un

conjunto de modelos, así como para la propagación de cambios y composición de operadores.

Por tanto, el framework MOMENT muestra un caso exitoso de la aplicación de una aproximación formal a una herramienta industrial de modelado, en el contexto de la Gestión de Modelos.

Para la definición de transformaciones y relaciones de equivalencia entre modelos, MOMENT ha optado por la propuesta de OMG, el estándar QVT (Query/Views/Transformations), que utiliza OCL como un lenguaje de alto nivel para realizar consultas sobre modelos, formando parte y completando las especificaciones de las transformaciones y relaciones de equivalencia.

1.2 Objetivos

La finalidad de este proyecto final de carrera es dar el soporte necesario para la utilización de OCL en la especificación de transformaciones y relaciones de equivalencia en MOMENT, así como proporcionar una interfaz que permita la validación de restricciones y la evaluación de consultas sobre modelos. Por tanto, este proyecto se puede dividir en dos partes fundamentales, o subproyectos:

1 *Proyecto OCLParser. Soporte para OCL en MOMENT*

El objetivo es dar soporte a las consultas OCL en la definición de transformaciones sobre modelos. Siguiendo la filosofía de la Ingeniería Dirigida por Modelos, un programa QVT que define una transformación o relación de equivalencia no es ejecutado de manera directa, sino que antes de la ejecución se genera un modelo QVT que contiene la información definida en el programa. Por tanto, este modelo contendrá las consultas OCL definidas en el programa QVT. Para poder ejecutar estas consultas es necesario realizar un proceso de traducción desde las expresiones OCL a código Maude para obtener su semántica operacional. OCL proporciona un lenguaje para realizar consultas de gran nivel de abstracción lo cual se traduce en una mayor productividad en la definición de transformaciones y relaciones de equivalencia.

2 *Proyecto OCLEditor. Interfaz para la validación de expresiones OCL sobre modelos*

Se plantea el desarrollo de una interfaz en Eclipse para la validación de invariantes y consultas OCL sobre modelos. Se añaden las siguientes funcionalidades: análisis sintáctico y semántico de las expresiones, editores para visualizar y editar las expresiones, así como la posibilidad de visualizar un esquema del Augmented Abstract Syntax Tree generado en el proceso de traducción, lo cual proporciona una vista estructurada de las expresiones. Mediante esta herramienta se permite la evaluación de una batería de invariantes sobre un determinado modelo lo cual se muestra útil en la realización de métricas sobre modelos.

1.3 Descripción del documento

El siguiente trabajo se organiza en los siguientes puntos:

- El capítulo uno corresponde a esta sección donde se presenta el proyecto y se plantean los objetivos de éste.

- El capítulo dos presenta los fundamentos teóricos sobre los que se asienta MOMENT: las propuestas *Model Driven Engineering, Model Driven Architecture y Meta-Object Facility*, se introduce el lenguaje para la definición de consultas y restricciones sobre modelos *Object Constraint Language*, y finalmente se realiza una aproximación a la disciplina de Gestión de Modelos.

  A continuación se realiza una presentación del framework para la gestión de modelos MOMENT, en cuyo marco se ha desarrollado este proyecto, para pasar a describir las dos tecnologías principales utilizadas: *Eclipse Modeling Framework*, a su vez soportado sobre la plataforma *Eclipse*, y el sistema de reescritura de términos *Maude*. Esto da paso a la explicación de los espacios tecnológicos utilizados y a una descripción de los diferentes componentes del framework. A continuación se presenta el soporte para transformaciones y relaciones de equivalencia, así como la especificación algebraica de OCL 2.0 de MOMENT.

  Para terminar este capítulo se presenta *Kent Modeling Framework* (KMF), que ha servido de base para este proyecto final de carrera, así como otros trabajos relacionados con la integración del soporte OCL en el desarrollo de software dirigido por modelos.

- En el capítulo tres se describe la primera parte desarrollada en este proyecto final de carrera: el componente *OCLParser* que da soporte a la traducción de expresiones OCL a código Maude para completar la especificación de las transformaciones y relaciones de equivalencia en MOMENT.

- En el capítulo cuatro se explica la segunda parte de este proyecto: el componente *OCLEditor* que proporciona una interfaz en Eclipse para desplegar la funcionalidad desarrollada en el componente OCLParser.

- En el capítulo cinco se presentan las conclusiones derivadas de este proyecto, así como los trabajos futuros.

- Finalmente se incluye un apartado con anexos. En el primer anexo se muestra el modelo Royal And Loyal, que servirá para ilustrar ejemplos. En el segundo anexo se muestra la gramática de OCL 2.0 utilizada por KMF. El tercer anexo hace un listado de los métodos de la interfaz *SemanticsVisitor* de KMF, consecuencia de la utilización del patrón de diseño *Visitor*. Por último, en el cuarto anexo se incluye el manual de usuario de la herramienta *OCLEditor*.

Capítulo 2

Fundamentos

2.1 Model Driven Engineering y Model Driven Architecture

2.1.1 Motivación y presentación

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

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

La Ingeniería Dirigida por Modelos (Model Driven Engineering, MDE) tiene como objetivo organizar los niveles de abstracción y las metodologías de desarrollo, todo ello promoviendo el uso de modelos como artefactos principales a ser construidos y mantenidos. Un modelo está constituido por un conjunto de elementos que proporcionan una descripción sintética y abstracta de un sistema, concreto o hipotético. El término MDE fue propuesto por primera vez por Stuart Kent [KentMDE02], en lo que se define como una marco general para la especificación de los modelos y tareas de modelado necesarias para llevar a cabo un proyecto de desarrollo software desde principio a fin. Cualquier especificación puede ser expresada con modelos, y éstos pueden tener cualquier nivel de abstracción y expresar cualquier aspecto de un sistema. El proceso de desarrollo se convierte en un proceso de refinamiento y transformación entre modelos, de manera que el nivel de abstracción cada vez es menor, hasta que en un último paso se genera código para una plataforma específica. Un proceso MDE debe definir claramente la secuencia de modelos a desarrollar en cada nivel y describir cómo derivar a partir de un modelo un modelo de menor nivel de abstracción. El sistema a desarrollar es inicialmente descrito por un modelo que captura los requerimientos, independientemente de los detalles específicos de la plataforma o

de cualquier decisión de implementación. Se trata de un modelo con el mayor nivel de abstracción posible, una descripción del problema a abordar.

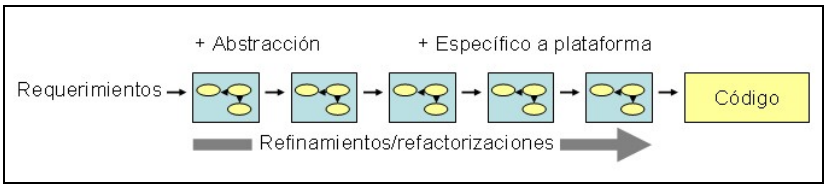

Figura 1. MDE. Esquema de un proceso de desarrollo

La aplicación de las propuestas de MDE a las herramientas CASE solamente pasaba por un escollo: la ausencia de lenguajes de modelado y metodologías de desarrollo estándar que dieran soporte a los sistemas software en todo su ciclo de vida a través de estas herramientas. Además, la existencia de estándares permitiría una mayor interoperabilidad. Para dar respuesta a estos problemas en el contexto de MDE, el Object Management Group (OMG) [OMG] ha lanzado la iniciativa Model Driven Arquitecture (MDA) [MDA], como una aproximación a la especificación e interoperabilidad de sistemas basada en el uso de modelos formales. En MDA, los modelos independientes de la plataforma (platform-independent models, PIMs) son inicialmente expresados en un lenguaje de modelado independiente de la plataforma, como UML. El modelo independiente de la plataforma es traducido a un modelo específico para la plataforma considerada (platform-specific model, PSM), por ejemplo, la plataforma Java, usando reglas formales. Por último, y a partir del modelo específico para la plataforma, se genera el código del sistema en el lenguaje de programación objetivo (Java, C#,...). Además, se propone la automatización de las transformaciones entre modelos y de la generación de código, pudiendo centrar el proceso de desarrollo de software en las tareas de modelado.

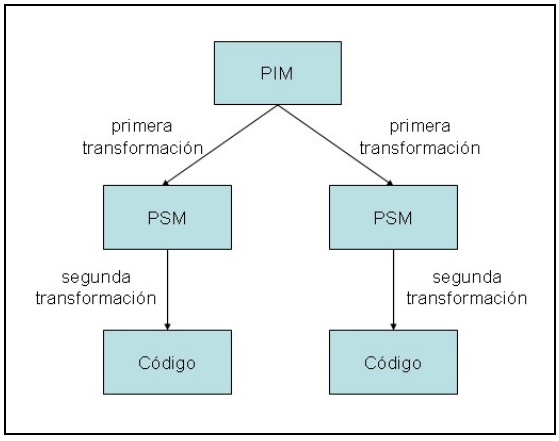

Figura 2. MDA. Relación entre PIM, PSM y código

MDA define un gran número de estándares de OMG:

- UML (*Unified Modelling Language*), que proporciona un vocabulario para describir gran cantidad de sistemas. UML se caracteriza por ser un vocabulario independiente de dominio, si bien tiene sus raíces en el modelado orientado a objetos.

- CWM (*Common Warehouse Metamodel*), un vocabulario específico para el dominio de los sistemas relacionados con la minería o explotación de datos.

- OCL (*Object Constraint Language*), un vocabulario que permite expresar consultas y restricciones sobre modelos. En una sección posterior nos centraremos en este lenguaje, en torno el cual gira el desarrollo de este proyecto.

- QVT (*Query/View/Transformation*), un vocabulario que utiliza OCL para expresar transformaciones y relaciones de equivalencia sobre modelos.

- XMI (*XML Metadata Interchange*), un vocabulario que permite el intercambio de modelos vía XML.

- MOF (*Meta Object Facility*), es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo.

2.1.2 Modelos y metamodelos. MOF

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

MOF (Meta Object Facility) es el metamodelo facilitado por MDA como vocabulario básico o metametamodelo. MOF, concretamente, define un subconjunto de UML para describir conceptos del modelado de clases mediante un repositorio de objetos. Mediante MOF pueden definir nuevos vocabularios, es decir nuevos metamodelos, con las mismas herramientas con que se definen modelos. Por otra parte, cabe preguntarse si existe un vocabulario de modelos superior que se utiliza para definir metamodelos. La respuesta es que sí, a este metamodelo de metamodelos se le denomina metametamodelo. Pero como también es un modelo, ¿se podría seguir extendiendo esta pirámide de forma infinita?

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

- En el nivel M1 se sitúan los modelos, tal y como los hemos introducido aquí, *descripciones abstractas de un sistema*.

- En la capa inmediatamente superior, denominada M2, se sitúan los metamodelos, "vocabularios para definir modelos".

- El nivel M3, que cierra la estructura por arriba, contiene el vocabulario base que permite definir metamodelos. Cabe resaltar que este nivel suele contener un único vocabulario, que caracteriza la aproximación de modelos escogida. Es

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

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

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

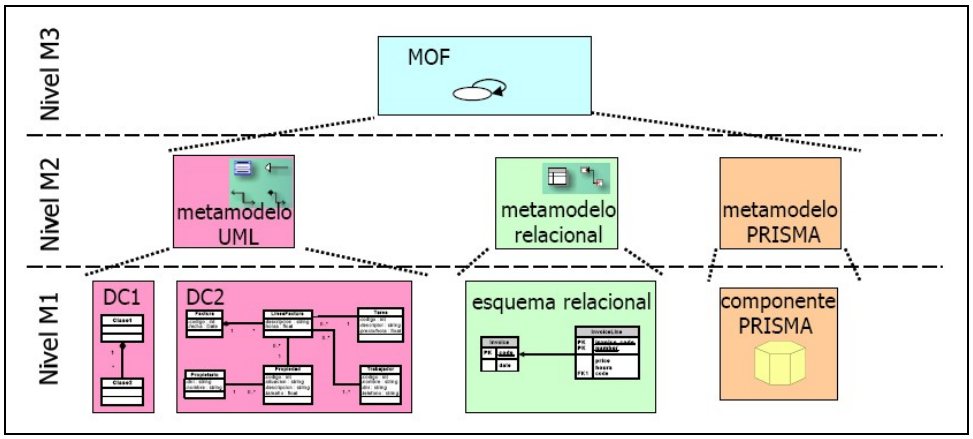

Figura 3. MOF. Arquitectura de niveles

Resulta interesante destacar que esta asignación fija de niveles puede resultar confusa en ocasiones. Quizá es más interesante fijar como idea fundamental la relación entre un modelo y su vocabulario, y darse cuenta de que esta relación ocurre en todos los niveles descritos. Esta relación se denomina *reificación*.

En el contexto del paradigma orientado a objetos (más concretamente, si consideramos el metamodelo de UML) se puede expresar la reificación mediante la «*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.

2.1.3 Nivel de madurez de los modelos

Siguiendo los procesos que plantea MDA, uno de los objetivos a perseguir en el desarrollo de software será producir un modelo del sistema de alto nivel de abstracción. Actualmente muchos desarrolladores utilizan UML u otro lenguaje para modelar los sistemas. Sin embargo, UML, aún siendo un estándar, se utiliza de maneras muy diferentes. Para poner algo de orden y transparencia en el trabajo con modelos se plantean los *niveles de madurez de los modelos* (*Modeling Maturity*

Levels, MMLs) [WarK03]. Estos niveles indican el rol que juegan los modelos en el proceso de desarrollo, y la dirección a tomar para mejorar este proceso. Tradicionalmente ha existido una *gap* semántico entre un sistema y su modelo. El modelo, que es usado como un plano, documentación o como reflejo de un conjunto de ideas, está a más alto nivel que el sistema, que es el objeto real. En cada nivel de madurez esta separación se reduce. Conforme el nivel es mayor, el término programación toma un nuevo significado, hasta alcanzar un extremo en el cual modelar y programar llega a ser lo mismo.

Veamos una descripción de cada uno de los niveles de madurez:

Nivel 0. Sin especificación

En el nivel más bajo, la especificación del software solamente está en la cabeza de los desarrolladores. Este nivel es típico en desarrolladores de software no profesionales. Simplemente se comentan las ideas, no importando que quede constancia por escrito. Esto es aplicable para pequeñas aplicaciones. No se da prácticamente reutilización de código y el mantenimiento es muy costoso.

Nivel 1. Textual

En este nivel la especificación se escribe en uno o más documentos de manera textual. El grado de detalle o formalismo depende totalmente de quien la realiza. Es el nivel más bajo en el desarrollo de software profesional. La especificación es ambigua al utilizar lenguaje natural y es tarea imposible sincronizarla con el código que se va desarrollando.

Nivel 2. Texto con diagramas

En el nivel de madurez 2 la especificación del software se realiza mediante uno o más documentos en lenguaje natural ampliados mediante diagramas de alto nivel. Los diagramas se utilizan para explicar la arquitectura general del sistema y algunos detalles complejos. Aún cuando se mejora la comprensión de la especificación, permanecen todos los problemas del nivel 1.

Nivel 3. Modelos con texto

Un conjunto de modelos, entendidos como diagramas o texto con una semántica bien definida, forma la especificación en este nivel. Adicionalmente, se utiliza el lenguaje natural para explicar la motivación y detalles de los modelos. De esta manera, diagramas o texto formal se convierten en representaciones reales del software. Aún así, la transición entre el modelo y el código es en la mayor parte manual.

Nivel 4. Modelos precisos

Un conjunto de modelos, entendidos como un conjunto coherente de textos y diagramas con una semántica muy concreta y bien definida, especifica el software en este nivel de madurez. También se utiliza lenguaje natural para explicar la motivación y detalles del modelo. Los modelos

tienen la precisión necesaria para tener un relación directa con el código actual, sin embargo, entre ellos se observan diferentes niveles de abstracción. Es el nivel que se persigue mediante un proceso MDA. Se facilita el desarrollo incremental e iterativo mediante una transformación directa entre el modelo y el código.

Nivel 5. Solamente modelos

En este nivel un modelo es una descripción del sistema completa, consistente, detallada y precisa. No se necesita ajustes para obtener el código resultante mediante transformaciones. El código generado es invisible al desarrollador. Este nivel aún no ha podido ser alcanzado, aunque la tecnología está caminando en esta dirección.

2.1.4 OCL. Object Constraint Language

2.1.4.1 Motivación

Para aplicar el proceso MDA se necesitan modelos con un nivel de madurez 4. Este es el primer nivel en el cual un modelo es más que un papel. Si se quiere ser capaz de transformar un modelo desde un PIM, pasando por un PSM, a código esa precisión es necesaria. Y una buena opción, en este caso, es utilizar la combinación de los lenguajes UML y OCL.

Veamos un pequeño ejemplo, en el cual un diagrama UML no tiene la suficiente expresividad para especificar una simple regla de negocio. La siguiente figura muestra un diagrama de clases que modela una compañía aérea.

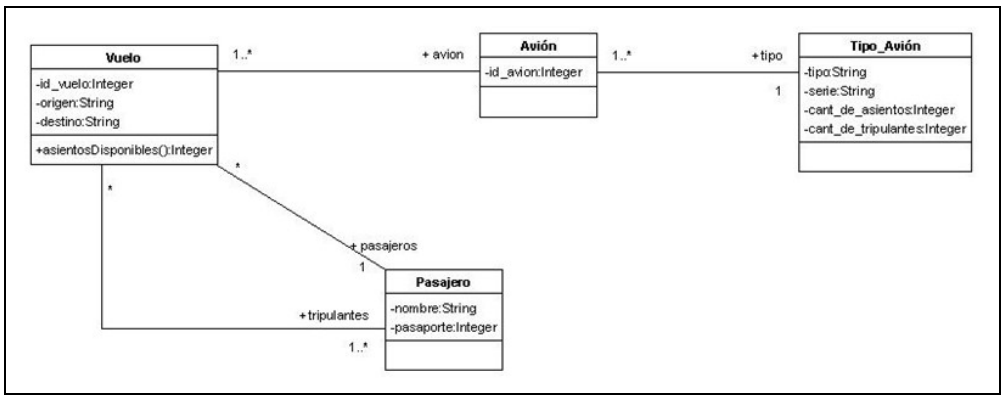

Figura 4. Diagrama de clases. Compañía aérea

En este modelo, un vuelo tiene un conjunto de pasajeros, y le corresponde un avión de un tipo, el cual tendrá un número de asientos determinados.

Mediante este diagrama de clases UML no podemos expresar una regla de negocio tan simple como que "*un vuelo tendrá un número de pasajeros no superior al número de asientos del tipo de avión que tiene asignado*".

Esto se expresa de manera intuitiva mediante una sencilla expresión OCL asociada al diagrama (en la figura se muestra como una etiqueta asociada a la clase *Vuelo).*

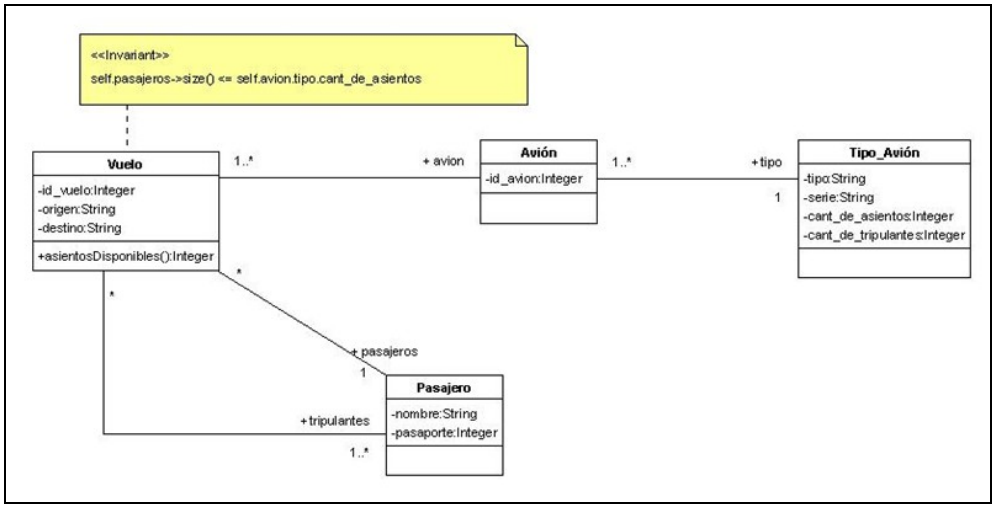

Figura 5. Diagrama de clases con expresión OCL. Compañía aérea

Al tratarse la expresión OCL de un invariante (el cual modela una obligación que debe evaluarse a cierta en todo el ciclo de vida de un objeto) tiene asignado un tipo booleano. La clase *Vuelo* es el contexto de la expresión, y se utiliza la notación de puntos para navegar por las asociaciones del diagrama de clases. En cada navegación se obtiene un elemento o un conjunto dependiendo de las multiplicidades de los extremos de las asociaciones. Mediante el operador de colección *size* se obtiene el número de instancias de una clase, que se han obtenido a través de la navegación.

2.1.4.2 Definición. Uso de modelos UML/OCL combinados

Object Constraint Language (OCL) es un lenguaje notacional para el análisis y diseño de sistemas software. Está definido como lenguaje estándar para dar soporte al *Unified Modeling Language* (UML), el cual es un estándar de OMG para el análisis y diseño orientado a objetos. OCL da soporte a UML para la especificación de restricciones y consultas sobre modelos, permitiendo definir y documentar los modelos UML de forma más precisa.

Un modelo UML, como por ejemplo un diagrama de clases o un diagrama de estados no constituye una especificación lo suficientemente precisa y libre de ambigüedades de un sistema. Las expresiones OCL completan esta especificación, convirtiéndose en información vital para los modelos orientados a objetos y otros objetos para el modelado. Esta información a menudo no puede ser expresada mediante un diagrama.

Cada expresión OCL se refiere a tipos (por ejemplo, clases, interfaces,…) definidos en diagramas UML. De esta manera, una expresión OCL estará siempre ligada a un diagrama UML (no tendría sentido de manera aislada), y en la dirección opuesta, un diagrama UML no estará especificado completamente sin expresiones OCL.

2.1.4.3 Características de OCL

Como hemos visto, UML necesita OCL para construir modelos completos, consistentes y sin ambigüedades. Veamos algunas de las características principales de este lenguaje:

i. Un lenguaje para restricciones y consultas

En UML 1.1, OCL era un lenguaje para expresar obligaciones (*constraints*) sobre elementos de un modelo, definidas como restricciones sobre uno o mas valores de un modelo orientado a objetos o sistema.

En UML 2 se incluye información adicional, como consultas, valores de referencia, condiciones de estado, reglas de negocio, etc. En definitiva, se puede utilizar OCL para expresar cualquier expresión sobre elementos de un diagrama.

Las expresiones OCL pueden ser usadas en cualquier punto de un modelo para indicar un valor. Un valor puede ser un valor simple, como un entero, pero puede ser también la referencia a un objeto, una colección de valores, o una colección de referencias a objetos. Una expresión OCL puede representar, por ejemplo, un valor booleano usado en la condición de un diagrama de estados, o un mensaje en un diagrama de interacción. Una expresión OCL puede referirse a un objeto específico en un diagrama de interacción o de objetos. Por ejemplo, la siguiente expresión define el cuerpo de la operación *asientosDisponibles()* de la clase Vuelo:

```
context Vuelo::asientosDisponibles() : Integer
body: self.avion.tipo.cant_de_asientos - self.pasajeros->size()
```

ii. Fundamentos matemáticos, sin símbolos matemáticos

OCL está basado en la teoría de conjuntos y lógica de predicados, y tiene una semántica matemática formal [Rich01]. Su notación, sin embargo, no utiliza símbolos matemáticos. De esta manera, OCL proporciona el rigor y precisión de un lenguaje formal y la facilidad de uso de un lenguaje natural.

iii. Lenguaje fuertemente tipado

Una característica esencial de OCL es que es un lenguaje tipado. Las expresiones OCL son usadas para modelar y especificar. Debido a que muchos modelos no son ejecutables directamente, muchas expresiones OCL estarán reflejadas aún cuando no existan versiones ejecutables del sistema. Sin embargo, debe ser la posible la comprobación de la corrección de estas expresiones, sin tener que producir una versión ejecutable del modelo. Como lenguaje tipado, las expresiones OCL pueden ser comprobadas durante el modelado, antes de la ejecución. De esta manera, los errores del modelo pueden ser eliminados en etapas tempranas.

iv. Lenguaje declarativo

Otro aspecto esencial de OCL es que se trata de un lenguaje declarativo. En los lenguajes procedurales, como son la mayoría de los lenguajes de programación, las expresiones son descripciones de las acciones que se quieren llevar a cabo. En un lenguaje declarativo, una expresión establece lo que debería hacerse, pero no cómo. Para asegurar esto, las expresiones OCL no tienen efectos laterales. Esto quiere decir que una expresión OCL no puede cambiar el estado del sistema.

El modelador puede tomar decisiones a un alto nivel de abstracción. Por ejemplo, en la definición del cuerpo de la expresión de la operación *asientosDisponibles()* del ejemplo de motivación, se especifica aquello que la operación debe calcular, pero no se establece cómo debe hacerse. El cómo dependerá de la estrategia que se siga en la implementación del sistema.

2.2 Gestión de modelos

2.2.1 Motivación

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

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

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

2.2.2 Operadores de gestión de modelos

En la literatura se discuten los operadores de gestión de modelos 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.

2.2.3 Aproximaciones existentes

Existen herramientas que facilitan la gestión de modelos. Entre ellas destaca RONDO [RONDO], desarrollada entre otros por P. Bernstein. RONDO propone la representación de modelos mediante teoría de grafos y una serie de operadores de alto nivel que permiten manipular modelos y las correspondencias entre ellos. Los modelos se traducen a grafos mediante conversores específicos para cada tipo de modelo que deben tener en cuenta consideraciones sobre las operaciones de manipulación para que se puedan aplicar sobre los modelos resultantes. La plataforma permite definir tanto los metamodelos utilizados para gestionar modelos, como los modelos utilizados para gestionar información, además de establecer correspondencias entre elementos de un mismo nivel de abstracción.

Una vez llegados al ámbito de las herramientas para la gestión de modelos es el momento de presentar MOMENT, un sistema de gestión de modelos que se apoya en el formalismo de las especificaciones algebraicas. A continuación se realiza una descripción de la plataforma, en la cual se engloba este proyecto final de carrera.

2.3 MOMENT. Una herramienta de gestión de modelos

2.3.1 Introducción

La disciplina de Gestión de Modelos trata con modelos mediante una serie de operadores genéricos que pueden aplicarse en tareas, como la generación de código, integración de aplicaciones, interoperabilidad entre aplicaciones, etc. Estos operadores constituyen una solución abstracta y reutilizable para trabajar sobre los modelos como ciudadanos de primer orden, independientemente del contexto o tecnología utilizada para representarlos.

Dada la experiencia previa, del grupo de investigación ISSI, en la aplicación del formalismo de especificaciones algebraicas a la recuperación de sistemas legados [BoPCR04][BoCR05], se ha propuesto el desarrollo de una herramienta que da soporte algebraico a este tipo de operadores genéricos desde un entorno de modelado industrial. Esta herramienta de gestión de modelos denominada MOMENT (*MOdel manageMENT*) [MOMENT], utiliza el entorno Maude, un sistema de reescritura de términos, desde un entorno de modelado industrial, como es Eclipse Modeling Framework (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, MOMENT representa un caso exitoso de la aplicación de métodos formales en problemas reales de la Ingeniería del Software, demostrando que estos métodos pueden aplicarse más allá de hipotéticos ejemplos teóricos.

A continuación se presentan las dos tecnologías principales utilizadas en el desarrollo de este proyecto, el lenguaje Maude y el entorno de modelado EMF, este segundo integrado en la plataforma Eclipse.

2.3.2 Maude

Maude es un lenguaje de programación de alto rendimiento que soporta la especificación y programación lógica tanto ecuacional como de reescritura para una amplia gama de aplicaciones [Maude]. La lógica de reescritura [Me92] permite expresar cambios concurrentes en computaciones altamente no deterministas. Además, proporciona un entorno de trabajo con una semántica general para dar semántica a un amplio abanico de lenguajes y modelos de concurrencia. En particular, soporta muy bien la computación concurrente orientada a objetos. Esto ha sido realzado en el diseño de Maude dando una especial sintaxis para la creación de módulos orientados a objetos. Se proporciona una buena semántica tanto a nivel computacional como en el lógico. En definitiva, se proporciona una *metalógica* en la cual otras lógicas pueden representadas e implementadas de manera natural. Consecuentemente, muchas aplicaciones de Maude están dirigidas a aplicaciones de *metalenguaje*, en la cual Maude es usado para crear entornos ejecutables para diferentes lógicas, probadores de teoremas, lenguajes y modelos de computación. Explotando el hecho de que la reescritura de términos es reflexiva, una característica que hace a Maude extensible y potente es el uso sistemático y eficiente de la reflexión, lo cual permite una *metaprogramación* avanzada. 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.

2.3.2.1 Características de Maude

En definitiva, Maude es un lenguaje de programación que permite modelar sistemas y las acciones entre estos sistemas. A continuación se presenta un listado de las principales características de Maude.

- *Maude es potente*. Puede modelar casi cualquier cosa, desde un conjunto de números racionales o un sistema biológico, hasta el propio lenguaje de programación Maude. Cualquier cosa que se pueda expresar con lenguaje natural se puede expresar en código Maude. Además, Maude soporta de una manera sistemática y eficiente la reflexión lógica. Esto le permite ser un lenguaje extremadamente potente y extensible, permitiéndole soportar un álgebra de operaciones de composición de módulos extensible.

- *Maude es simple*. Su semántica está basada en los fundamentos de la teoría de categorías. Aunque es necesario manejar con soltura sus conceptos y notaciones específicas para tener un conocimiento profundo del

funcionamiento de Maude, al principio no es necesario conocerlos para empezar a programar. Comparado con otros lenguajes, Maude no tiene mucha sintaxis que memorizar, solamente un conjunto de símbolos y palabras clave, además de algunas convenciones generales para poder empezar.

- *Maude es concreto*. 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. La programación se realiza a un nivel muy cercano al propio problema.

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 [Maude2Man]. 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 proporcionará soporte para ella directamente en Core Maude. Es destacable apuntar esto, ya que MOMENT se apoya de manera muy importante en este mecanismo, por lo que actualmente se está trabajando en portar el álgebra escrita en Full Maude a Core Maude con soporte para parametrización (actualmente en versión *Core Maude alpha86a*, que dará origen a Core Maude 2.2) ya que proporciona importantes mejoras en la eficiencia.

2.3.3 Eclipse y Eclipse Modeling Framework

2.3.3.1 Eclipse

Eclipse es un proyecto de desarrollo de software de código abierto, cuyo propósito es proporcionar una plataforma de herramientas altamente integradas [Eclipse]. El trabajo en Eclipse consiste en un proyecto central, el cual incluye un framework genérico para la integración de las herramientas, y un entorno de desarrollo Java construido utilizando el primero. Otros proyectos extienden el framework central para soportar clases específicas de herramientas y entornos de desarrollo. Los proyectos en Eclipse están implementados en Java y se pueden ejecutar en una multitud de sistemas operativos, incluyendo Windows y Linux.

El software producido por Eclipse está disponible bajo la *Common Public License* (CPL), la cual básicamente dice que puedes usar, modificar, y distribuir el software de manera gratuita, o incluirlo como parte de un producto propietario. CPL es una *Open Source Initiative* (OSI), aprobada y reconocida por la *Free Software Foundation* como una licencia de software libre. Cualquier software que contribuya a Eclipse debe hacerlo bajo la licencia CPL.

Eclipse.org es un consorcio de compañías las cuales tienen un acuerdo para proporcionar el soporte esencial al proyecto Eclipse en términos de tiempo, experiencia, tecnología y conocimiento.

2.3.3.2 Los proyectos

El trabajo de desarrollo en Eclipse se divide en tres proyectos principales: el proyecto Eclipse, el proyecto de Herramientas y el proyecto de Tecnología. El proyecto Eclipse contiene los componentes centrales necesarios para desarrollar mediante Eclipse. Sus componentes conforman una única unidad conocida como *Eclipse SDK* (*Software Development Kit*). Los componentes de los otros dos proyectos son usados para propósitos específicos y son generalmente independientes.

i. El proyecto Eclipse

El proyecto Eclipse soporta el desarrollo de una plataforma, o framework, para la implementación de entornos de desarrollo integrados (*Integrated Development Environments*, IDEs). El framework Eclipse está implementado utilizando Java pero es usado para implementar herramientas de desarrollo para otros lenguajes (por ejemplo, C++ y XML, entre otros).

El proyecto Eclipse se divide en tres subproyectos. La Plataforma (*Platform*) es el componente central de Eclipse, y es a menudo considerado como el propio Eclipse. Se utiliza para definir frameworks y los servicios requeridos para soportar la conexión e integración de las herramientas. En segundo lugar, las Herramientas de Desarrollo Java (*Java Development Tools*, JDT) proporcionan un completo entorno de desarrollo Java. Pueden ser utilizadas para desarrollar programas Java para Eclipse o para otras plataformas. Por último, el Entorno de Desarrollo de Plug-ins (*Plug-in Development Environment*, PDE) proporciona vistas y editores

para facilitar la creación de plug-ins para Eclipse. Además proporciona soporte para actividades propias del desarrollo de un plug-in, como el registro de extensiones.

En conjunto, estos tres subproyectos proporcionan todo lo necesario para extender el framework y desarrollar herramientas basadas en Eclipse.

ii. El proyecto de Herramientas

El proyecto de Herramientas define y coordina la integración de diferentes conjuntos o categorías de herramientas basadas en la plataforma Eclipse. El subproyecto de Herramientas de Desarrollo C/C++ (*C/C++ Development Tools*, CDT), por ejemplo, engloba un conjunto de herramientas que definen un IDE C++. Los editores gráficos que utilizan *Graphical Editing Framework* (GEF), y los editores basados en modelos usando EMF representan categorías de herramientas de Eclipse para los cuales se da un soporte común desde los subproyectos de Herramientas.

iii. El proyecto de Tecnología

El proyecto de Tecnología proporciona una oportunidad para los investigadores y educadores para formar parte de la continua evolución de Eclipse.

2.3.3.3 Descripción de la Plataforma Eclipse

La Plataforma Eclipse es un framework para construir entornos de desarrollo integrados (IDEs). Ha sido descrita como una "una IDE para cualquier cosa, y para nada en particular" [EclOv03]. Su cometido es definir una estructura básica para un IDE. Las herramientas específicas extienden el framework, conectándose a él para definir un IDE particular de manera colectiva.

Veamos una descripción de los componentes principales de la Plataforma Eclipse.

i. La arquitectura de plug-ins

Una unidad funcional básica, o componente, en Eclipse es llamado un plug-in. La misma Plataforma Eclipse, y las herramientas que la extienden, están compuestas por plug-ins.

Desde una perspectiva de paquetes, un plug-in incluye todo lo que se necesita para que funcione 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. En este archivo se declaran, entre otras cosas, lo siguiente:

- *Requires*: sus dependencias con otros plug-ins.
- *Exports*: la visibilidad de sus clases públicas con otros plug-ins.
- *Extension points*: declaración de la funcionalidad que se pone a la disposición de otros plug-ins.
- *Extensions*: implementación de puntos de extensión de otros plug-ins

En el arranque, la Plataforma Eclipse (específicamente *Platform Runtime*) descubre todos los plug-ins disponibles y asocial extensiones con sus correspondientes puntos de extensión. Un plug-in, sin embargo, solamente se activa cuando su código necesita ejecutarse, evitando una lenta secuencia de arranque. Cuando se activa, un plug-in es asignado a su propia clase de carga, la cual provee la visibilidad declarada en su archivo de manifiesto.

ii. Recursos del espacio de trabajo (workspace)

Las herramientas integradas en Eclipse trabajan con archivos y carpetas ordinarios, pero utilizan una API de alto nivel basada en recursos (*resources*), proyectos (*projects*), y un espacio de trabajo (*workspace*).

Un recurso es la representación que utiliza Eclipse para los archivos y carpetas, proporcionando funcionalidades adicionales (registro de controladores de cambios en el recurso, marcadores, mantenimiento de historiales,…).

Un proyecto es un tipo especial de recurso que se asocia a una carpeta del usuario en el sistema de ficheros. Las subcarpetas del proyecto son las mismas que las de la carpeta física, pero los proyectos son carpetas de alto nivel en un contenedor virtual de proyectos, llamado espacio de trabajo (*workspace*).

iii. El framework UI

El framework UI de Eclipse consiste en dos conjuntos de herramientas de propósito general, SWT y JFace, y el Escritorio de Eclipse (*workbench UI*).

SWT (*Standard Widget Toolkit*) es un conjunto de librerías gráficas independientes del sistema operativo utilizado.

JFace es un conjunto de herramientas de alto nivel, implementado usando SWT. Proporciona clases para soportar tareas comunes relativas a interfaces de usuario, como manejo de registros de imágenes y fuentes, diálogos, asistentes, monitores de progreso, etc. Una parte importante de JFace son las clases utilizadas como visores para listas, árboles y tablas, que proporcionan un mayor nivel de conexión con los datos que SWT (por ejemplo, mecanismos de población a partir de un modelo de datos y sincronización con éste). Otra parte importante de JFace es su framework para acciones, usado para añadir comandos a menús y barras de herramientas.

Por último, el Escritorio de Eclipse (*workbench*) es ventana principal que el usuario ve cuando arranca Eclipse. Está implementado mediante SWT y JFace. Como interfaz principal de usuario, se considera a menudo que es la propia Plataforma.

iv. Soporte para el trabajo en grupo

La Plataforma Eclipse permite asignar una versión a un proyecto en el *workspace* y gestionar sus versiones mediante un repositorio de trabajo en grupo. Multitud de repositorios de trabajo en grupo pueden coexistir con la Plataforma, no obstante la Plataforma Eclipse incluye soporte para repositorios CVS accedidos mediante protocolos *pserver* o *ssh*.

v. Ayuda

Eclipse Platform Help incluye herramientas que permiten definir y contribuir a la documentación de uno o más libros en línea.

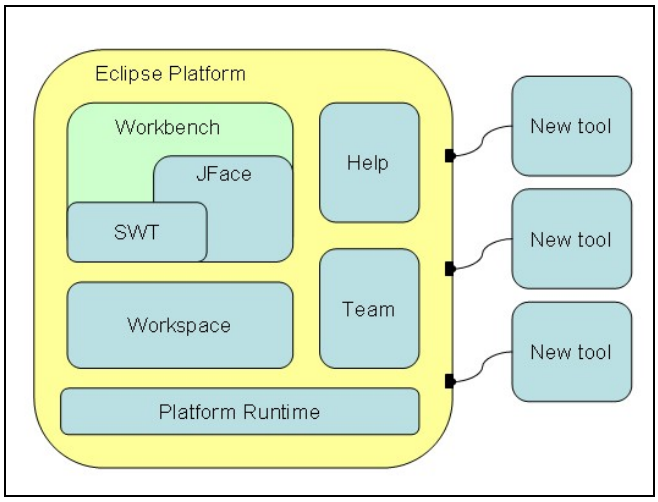

Figura 6. Arquitectura de la Plataforma Eclipse

2.3.3.4 Eclipse Modelling Framework

EMF es básicamente un entorno de modelado y una herramienta de generación de código para la plataforma Eclipse [BuSte03]. Su objetivo es construir herramientas y otras aplicaciones basadas en modelos estructurados. EMF ayuda además a generar código Java a partir de estos modelos de una manera fácil, correcta, personalizable y eficiente. EMF utiliza XMI como una manera canónica de definir y persistir los modelos. Además proporciona un editor gráfico para definir modelos.

Una vez se especifica un modelo EMF, el generador EMF puede crear una serie de clases Java que permiten crear instancias del modelo y manipular sus elementos. Una vez generado el código se pueden editar las clases generadas para añadir nuevos métodos y variables. Además, se proporcionan mecanismos para la notificación de cambios en el modelo, serialización por defecto en XMI y XML Schema, un entorno de trabajo para la validación de modelos, y una API reflexiva

eficiente para manipular objetos EMF de manera genérica. Y lo más importante, EMF proporciona las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

EMF consiste en dos entornos de trabajo fundamentales: el *core* y *EMF.Edit*. El entorno *core* proporciona la generación de código básica y el soporte en tiempo de ejecución para la creación de las clases Java del modelo. *EMF.Edit* extiende y se construye sobre el *core*, añadiendo soporte para la generación de las clases que permiten utilizar los comandos para editar los modelos, además de activar los visores y editores gráficos básicos para un modelo.

2.3.3.5 Ecore

EMF comenzó como una implementación de la especificación MOF de OMG. En realidad EMF se puede considerar una eficiente implementación en Java de un subconjunto del *core* de la API de MOF. Sin embargo, para evitar cualquier confusión, el *metametamodelo* en EMF se llama Ecore. Por tanto, Ecore permite definir los vocabularios locales de dominio o metamodelos, que permiten la creación o modificación de modelos en distintos contextos. Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de metamodelos. Para ello proporciona elementos útiles para describir conceptos y las relaciones entre ellos. En definitiva, Ecore es un subconjunto de MOF, el cual está basado en el diagrama de clases de UML. En la siguiente figura se muestran los elementos principales del metamodelo de Ecore.

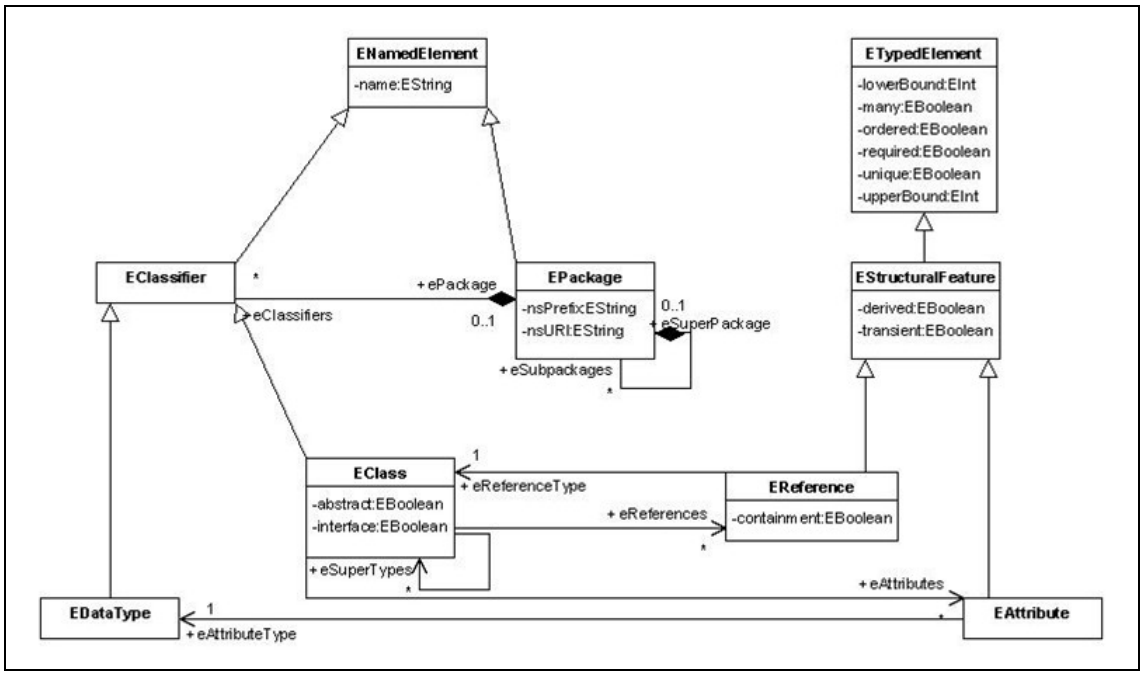

Figura 7. Metamodelo Ecore simplificado

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.

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.

2.3.4 Visión global del framework MOMENT

En MOMENT los operadores de gestión de modelos [Ber03] 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 la siguiente figura utilizando notación UML.

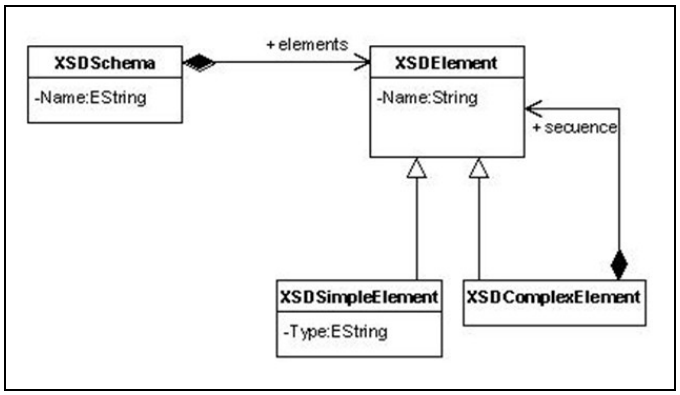

Figura 8. Parte del metamodelo XSD

Utilizando el editor en forma de árbol que proporciona EMF, definimos los esquemas XML *A* y *B* en la siguiente figura. Se aplica el operador *Merge* a ambos, obteniendo el esquema XML integrado *C* y dos modelos de trazas (map_{AC} y map_{BC}) que enlazan los elementos de los modelos de entrada con los elementos del modelo de salida. La invocación del operador es la siguiente:

$$<C, mapAC, mapBC> = Merge(A, B)$$

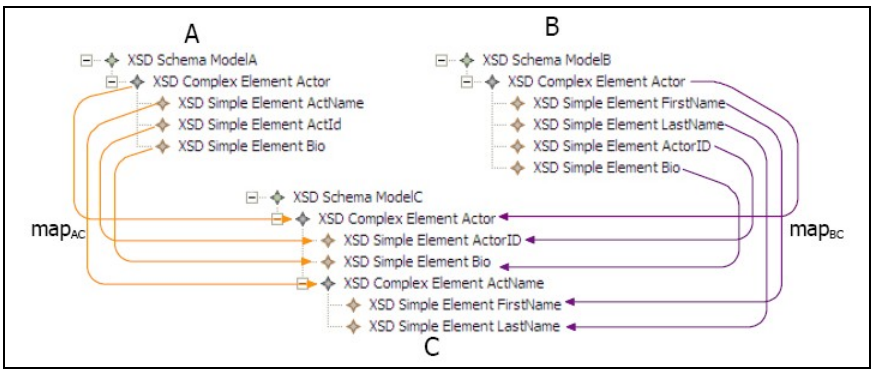

Figura 9. Aplicación del operador Merge

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.

2.3.5 Espacios tecnológicos: EMF y Maude

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 [BeDD05]. El concepto de espacios tecnológicos fue introducido por Kurtev et al. en la discusión sobre el enlace de tecnologías heterogéneas [KurBA02]. Por ejemplo, en este artículo tratamos con los ETs de Eclipse Modeling Framework y de Maude.

Siguiendo un enfoque de Ingeniería de Modelos, para tratar con artefactos software utilizamos la terminología que define el estándar MOF de la iniciativa MDA. Como ya se ha presentado, este estándar 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*). Se define un vocabulario básico en el nivel M3, que puede ser utilizado 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 (por ejemplo, el metamodelo de definición de esquemas XML, XML Schema Definition, XSD), constituyendo un paradigma de modelado. Un metamodelo especifica los elementos que se pueden utilizar para construir un modelo que lo conforme en el nivel M1 (en este caso, un modelo sería un esquema XML específico, que conforma el metamodelo XSD). Por último, un modelo especifica aquellos elementos que pueden utilizar sus instancias, en el nivel M0 (en el ejemplo, un documento XML concreto, instancia del modelo XML considerado).

2.3.5.1 Enlaces entre los espacios tecnológicos de EMF y Maude

Un ET 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 ET EMF se caracteriza por las facilidades que ofrece para representar una buena variedad de artefactos sofware como modelos y por su interoperabilidad con otras herramientas industriales de modelado. El metametamodelo de EMF es Ecore, los metamodelos que lo conforman están descritos en los archivos con extensión *ecore*, y por último, los modelos conformarán estos metamodelos.

El ET Maude se caracteriza por las ventajas que aporta el formalismo de especificaciones algebraicas: abstracción, subtipado, modularización, genericidad mediante parametrización, etc. Este ET está asociado a un paradigma de modelado, siendo el lenguaje Maude el metametamodelo en el nivel M3, y los módulos que proporcionan especificaciones algebraicas los metamodelos, en el nivel M2. Por tanto, una especificación algebraica representa un metamodelo, en el nivel M2, proporcionando la descripción sintáctica de las primitivas necesarias para especificar un artefacto software en el nivel M1. Estas primitivas son denominadas *constructores* en el campo de las especificaciones algebraicas. Los modelos, en el nivel M1, 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 (o puentes) entre ambos ETs. Estas proyecciones permiten representar un modelo como un término algebraico, manipularlo desde Maude, y devolverlo como un modelo EMF.

Por tanto, EMF proporciona una interfaz para la creación y visualización de modelos Ecore desde una herramienta de desarrollo de ámbito industrial, como es Eclipse, y Maude proporciona la semántica operacional necesaria para la manipulación de estos modelos mediante operadores genéricos, independientes del metamodelo utilizado. Por último, EMF permite visualizar el resultado de la ejecución de estas operaciones.

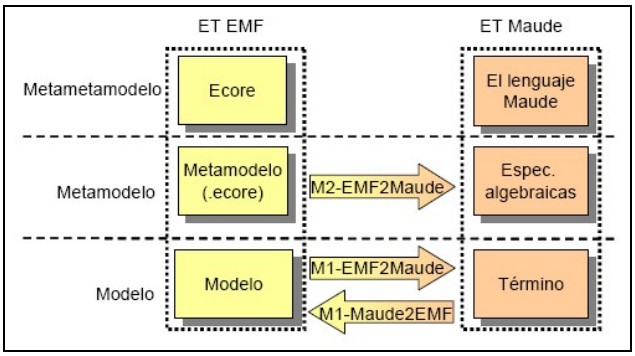

Figura 10. Enlaces entre el ET EMF y el ET Maude

Se han desarrollado en [Ibo05] dos enlaces entre el ET de EMF y el de Maude, uno en el nivel M2 y otro en el nivel M1.

i. Interoperabilidad en el nivel M2

En el nivel M2 se proyecta un metamodelo EMF sobre el ET Maude para obtener una especificación algebraica, dando lugar al enlace unidireccional *M2-EMF2Maude*. De esta manera, un metamodelo se interpreta como un álgebra que proporciona los constructores necesarios para definir modelos y las operaciones necesarias para manipularlos, en el contexto de la Gestión de Modelos.

Los metamodelos se especifican mediante editores visuales de modelado en EMF. Esto permite hacer transparente al usuario el uso del formalismo Maude.

Además, MOMENT trabaja directamente sobre un álgebra de operadores genéricos de manipulación de modelos. Estos operadores pueden ser adaptados a un metamodelo específico haciendo uso de las capacidades de parametrización que ofrece Maude, basándose en el concepto formal de Pushout [EhrigM85] de teoría de categorías.

ii. Interoperabilidad en el nivel M1

En el nivel de modelos existe otro enlace entre el ET de EMF y el de Maude. En este caso el enlace es bidireccional, al estar formado por dos tipos de proyecciones:

- *M1-EMF2Maude*: Este mecanismo proyecta un modelo EMF, definido mediante un metamodelo EMF, sobre el ET de Maude como un término. Para proyectar un modelo sobre el ET de Maude, la herramienta MOMENT consulta el correspondiente metamodelo y obtiene el constructor de la correspondiente especificación algebraica, que es necesario para especificar el término de forma automática.

- *M1-Maude2EMF*: Este mecanismo proporciona la proyección inversa a la anterior, obteniendo un modelo EMF a partir de un término Maude. Es en este paso, cuando la herramienta MOMENT lee un término que representa un modelo, determina las primitivas del metamodelo EMF que debe utilizar para construir el modelo EMF a partir de los símbolos de los constructores

utilizados en el término. Con estas primitivas se construye el modelo dinámicamente y se persiste en formato XMI.

Los enlaces definidos a nivel M2 y M1, entre el ET de EMF y el ET de Maude, permiten la utilización del sistema de reescritura de términos de Maude para la aplicación de operadores algebraicos sobre modelos, definidos de forma gráfica mediante un entorno industrial de modelado, que es EMF sobre la plataforma Eclipse.

Siguiendo con el ejemplo del metamodelo XSD. Supongamos que queremos integrar dos esquemas XML. El proceso ha seguir es el siguiente:

1. Se obtiene la especificación algebraica a partir del metamodelo XSD.

2. Se proyectan los esquemas XML *A* y *B* a términos del álgebra, interpretada a partir de la especificación algebraica del metamodelo.

3. Se aplica el operador *Merge* a ambos términos y se obtiene el término resultante mediante el mecanismo de reducción de Maude. Como resultado de la operación de integración se obtiene de integración se obtiene el término *C*, que representa el esquema XML integrado. Además, el operador *Merge* produce dos modelos de trazabilidad (*mapAC, mapBC*) que relacionan los modelos de entrada *A* y *B* con el modelo de salida *C*, respectivamente.

4. Finalmente, el término *C* es proyectado al ET de EMF como un modelo que conforma el metamodelo Ecore.

2.3.6 Soporte para transformaciones y relaciones de equivalencia

2.3.6.1 Introducción

El paradigma de desarrollo dirigido por modelos es una disciplina de la Ingeniería del Software donde los modelos son considerados como artefactos de primera clase, y donde la transformación genérica de modelos realiza un papel fundamental. Han sido numerosos los lenguajes de transformación surgidos con el propósito de especificar transformaciones genéricas entre modelos, pero la dificultad de aprendizaje y la escasa productividad de las herramientas que soportan estos lenguajes ha limitado su repercusión dentro de la comunidad del software. En un intento de aunar los esfuerzos que se están llevando a cabo en este campo, OMG ha propuesto un lenguaje estándar para la especificación de transformaciones llamado *Query/View/Transformations* (QVT) [QVT], que depende a su vez de otros dos estándares: MOF 2.0 y OCL 2.0. De esta manera se pretende promover el uso estándar de las transformaciones entre modelos, lo que lleva asociado notables mejoras en la productividad, la interoperabilidad y la calidad, permitiendo que el desarrollo de software se lleve a cabo en un nivel de abstracción más elevado y utilizando conceptos más cercanos al dominio del problema.

Para que la visión del desarrollo dirigido por modelos sea una realidad, las herramientas que lo soportan deben ser capaces de proporcionar la automatización de transformaciones de modelos. Éstas, no sólo deben ofrecer transformaciones de modelos de forma *ad-hoc*, sino también proporcionar un lenguaje que permita a los usuarios definir sus propias transformaciones y ejecutarlas posteriormente, con lo que se podrán automatizar tareas como la aplicación de patrones, el refinamiento o la refactorización. En este sentido, MOMENT proporciona soporte para el lenguaje de QVT *Relations*, integrando un motor para la ejecución de transformaciones sobre modelos.

2.3.6.2 QVT y MOMENT

La especificación QVT se define a través de dos dimensiones ortogonales: la dimensión del lenguaje y la dimensión de la interoperabilidad, donde cada una de las cuales tiene una serie de niveles. La dimensión del lenguaje define los diferentes lenguajes de transformación presentes en la especificación QVT. Concretamente son tres: *Relations, Core* y *Operational*, y la principal diferencia entre ellos es su naturaleza declarativa o imperativa. En la dimensión de la interoperabilidad se encuentran aquellas características que permiten a una herramienta que cumple el estándar QVT interoperar con otras herramientas. La intersección de niveles de las dos dimensiones especifica un punto de cumplimiento QVT (*QVT-compliance*).

Para integrar QVT en MOMENT se ha elegido el lenguaje *Relations* para aprovechar su naturaleza declarativa, su especificación de trazabilidad transparente al usuario, así como la sintaxis bastante sencilla, en comparación con otros lenguajes declarativos, que ofrece para definir transformaciones y relaciones de equivalencia entre modelos.

Como se ha comentado anteriormente, siguiendo la disciplina de la Gestión de Modelos se han especificado en MOMENT un conjunto de operadores genéricos para manipular modelos: *Merge, Diff, ModelGen*, etc. En el ámbito de las transformaciones sobre modelos, es el operador *ModelGen*[1] el que proporciona soporte para el lenguaje QVT *Relations*, dando soporte además para la trazabilidad de las transformaciones. *ModelGen* es, además, usado por otros operadores de gestión de modelos del framework, cuando se tiene que realizar una manipulación sobre un modelo.

En el lenguaje Relations las transformaciones quedan descritas a través de la enumeración de los metamodelos participantes, un conjunto de reglas que especifican la relación existente entre términos de los metamodelos, un conjunto de dominios por regla que se ajusta al conjunto de términos para los que se expresa relaciones y un conjunto de patrones que cumplen con la estructura de los términos y a los que se aplican operaciones OCL.

El que una regla sea de transformación o no viene determinado por el dominio destino, que estará marcado como *checkonly* o *enforce*. La marca *checkonly* comprueba si existe una correspondencia válida entre los modelos, mientras que la

[1] El operador *ModelGen* toma un modelo A y lo proyecta en otro metamodelo, obteniendo un modelo B y un *mapping* entre A y B.

marca *enforce* fuerza a que los términos de los respectivos modelos cumplan la relación descrita en la regla.

Respecto a la interoperabilidad en MOMENT, se permite la ejecución de modelos que conforman al metamodelo QVT Relations, la ejecución de cualquier especificación textual de una transformación expresada con el lenguaje Relations, así como la importación y exportación de modelos serializados con formato XMI.

En MOMENT, a través de QVT Relations, es posible la especificación de transformaciones endógenas y exógenas. Las transformaciones endógenas son aquellas transformaciones que comparten el mismo metamodelo, mientras que las transformaciones exógenas se definen entre modelos que conforman a metamodelos diferentes[2] [MensV05].

		Interoperabilidad			
Lenguaje		Sintaxis Ejecutable	XMI Ejecutable	Sintaxis Exportable	XMI Exportable
	Core				
	Relations	**X**	**X**		**X**
	Operational				

Tabla 1. MOMENT. Cumplimiento del estándar QVT

En cuanto a la cardinalidad de modelos participantes en una transformación, el número máximo de modelos origen no está limitado. Por otra parte, el número de modelos destino queda restringido a uno tal y como se especifica en el estándar QVT, aunque es posible alcanzar cardinalidad múltiple en el resultado aplicando composición de transformaciones que tengan diferentes modelos resultantes y que pasen a formar parte del resultado final. La forma con la que actualmente se lleva a cabo la composición de transformaciones es manual y mediante interacción del usuario, pero en un futuro próximo se tiene previsto que este proceso se lleve a cabo de manera automática. Dado que en MOMENT-QVT las transformaciones se proyectan como funciones en Maude, la composición de transformaciones se basa en la composición algebraica de funciones. En definitiva, son posibles las transformaciones 1-1, n-1, 1-n y n-n, bien aplicando una sola transformación o mediante composición de transformaciones.

Por último, hay que considerar que el hecho de que MOMENT se base en estándares como MOF, QVT, OCL y XMI, conlleva implícitamente un alto nivel de interoperabilidad con herramientas similares. Por otro lado, la utilización y aplicación de Maude como método formal subyacente permite el estudio formal de ciertas propiedades de las transformaciones de modelos: como la terminación o el indeterminismo.

[2] Si en una transformación se utilizan varios modelos origen y/o varios modelos destino, entonces pueden estar involucrados varios metamodelos diferentes.

2.3.6.3 Ejecución de QVT Relations en MOMENT

El proceso de transformación que se lleva a cabo en MOMENT-QVT consta de tres fases: análisis, proyección a Maude y ejecución y proyección a EMF (ver siguiente figura). Todas ellas quedan ocultas al usuario, siendo únicamente necesaria la especificación de la transformación por parte del mismo.

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

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

El componente *OCL Parser* se ha desarrollado y se documenta en este proyecto final de carrera.

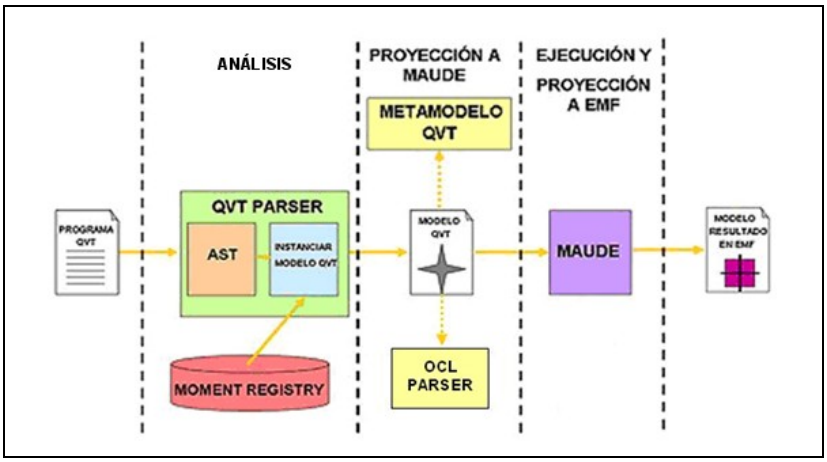

Figura 11. Proceso de ejecución de un programa QVT Relations en MOMENT

i. Fase de análisis

Esta etapa comienza con un análisis léxico y sintáctico del programa QVT Relations. Recorriendo el árbol de sintaxis abstracta (AST) generado por esta etapa, se crea un modelo instancia del metamodelo de QVT Relations que se encuentra integrado en la plataforma MOMENT. Durante este recorrido es necesario consultar los metamodelos respectivos a los modelos participantes en la transformación y obtener de este modo los tipos y las propiedades de sus elementos. Por consiguiente, es prerrequisito imprescindible que estos metamodelos estén registrados en el MOMENT *Registry*.

A continuación se muestra un ejemplo de transformación, el cual ha sido propuesto en el apéndice de la especificación del estándar MOF QVT, donde se especifica la transformación de un diagrama de clases UML en su correspondiente modelo relacional. En la parte izquierda de la siguiente figura, se muestra el segmento de código de la regla de transformación *ClassToTable*. Esta regla establece la correspondencia entre una clase perteneciente al metamodelo origen (UML) y una tabla perteneciente al metamodelo destino (relacional), de forma que a partir del término clase perteneciente a uno de los dominios de la regla se obtienen los términos tabla, columna y clave primaria, que pertenecen al dominio destino de la regla tal como inidica la palabra reservada *enforce*. Tras llevar a cabo la fase de análisis, se muestra la parte del modelo QVT Relations correspondiente a la regla "ClassToTable" en la parte derecha de la figura.

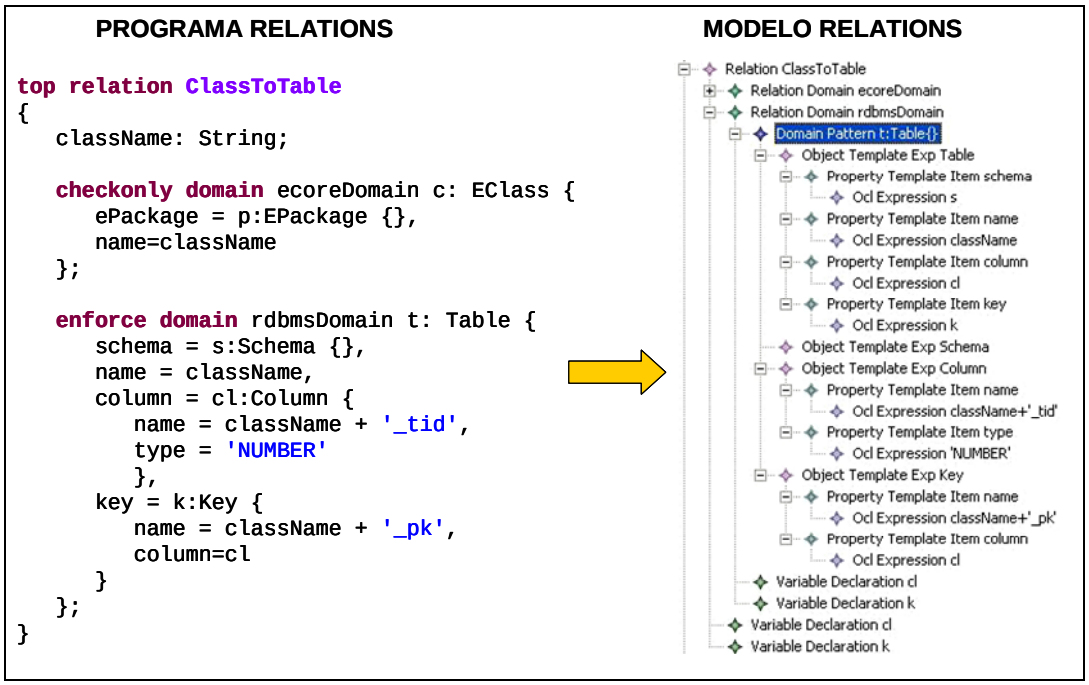

Figura 12. Parte de programa Relations y modelo asociado

ii. Fase de proyección a Maude

Los metamodelos origen y destino son compilados en teorías basadas en lógica ecuacional [BoICRG05]. Los conceptos del metamodelo se proyectan como *sorts* y las propiedades como constructores[3] y operadores de consulta. Así, se crea una especificación algebraica en el espacio tecnológico de Maude para cada metamodelo participante en la transformación. Además, se toman todos los modelos origen (en la capa M1 de MOF) y se obtiene para cada uno de ellos el término algebraico que conforma la especificación algebraica definida para su respectivo metamodelo [BoCR06].

Seguidamente, el modelo QVT Relations resultado de la etapa de análisis se proyecta en otra especificación algebraica. Una parte muy importante de esta

[3] Un constructor en Maude es un operador que permite definir información. En este se tiene un constructor para definir la instancia de una clase.

proyección es la traducción de las expresiones OCL utilizadas en la especificación de la transformación, y que se realiza a través del componente OCL Parser. El desarrollo de este componente ha sido uno de los objetivos principales de este proyecto final de carrera.

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

La siguiente figura muestra el resultado de proyectar a Maude la regla de transformación *ClassToTable* obtenida en la fase anterior. En ella se aprecia la ecuación *TransformElements* que lleva a cabo su ejecución a través de los mecanismos de *pattern-matching* y de recursión. La generación de los términos resultantes *Table, Column y Key* se realiza a través del operador *New*, el cálculo de sus respectivos identificadores se consigue con el operador *AddOID* y la asignación de valores y contenido a cada una de las propiedades de estos términos se obtiene por medio del operador asignación "←" y las navegaciones del modelo origen partiendo de su respectivo dominio en la regla, como es el caso de "*ecoreEClass0 :: name*".

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

Figura 13. Proyección de la transformación a Maude

iii. Ejecución y proyección a EMF

Llegado este punto, se dispone en el espacio tecnológico Maude de toda la información necesaria para lanzar la transformación a ejecución. Desde Maude, se aprovecha el mecanismo de reducción modulo las ecuaciones generadas a partir de un programa QVT Relations, como brevemente se indica en el apartado anterior, y de *pattern-matching*, obteniendo el término resultante que en el ejemplo anterior se corresponde con un esquema relacional. Finalmente, este término es proyectado de vuelta a EMF utilizando los puentes definidos a nivel M1 entre ambos espacios

tecnológicos. Se puede encontrar más detalles del proceso de compilación a código Maude en [BoCR06].

2.3.7 Especificación algebraica de OCL 2.0 en Maude

2.3.7.1 Contexto

En esta sección se describe la especificación algebraica parametrizada de OCL que permite la consulta sobre metamodelos o modelos Ecore en MOMENT [BoOGRC06]. El sistema de reescritura Maude ha sido usado para este propósito. Maude proporciona un lenguaje de especificación algebraica que pertenece a la familia del lenguaje OBJ. Su mecanismo de deducción ecuacional anima la especificación algebraica de OCL sobre una instancia del modelo específica, proporcionando la semántica operacional de las expresiones OCL. En el marco del proyecto MOMENT se ha desarrollado un plug-in que embebe el entorno de trabajo de Maude en la plataforma Eclipse. De esta manera se ha podido utilizar para nuestros propósitos.

2.3.7.2 Aproximación a la especificación algebraica parametrizada de OCL

En Maude, los módulos funcionales describen tipos de datos y operaciones sobre ellos a través de una teoría ecuacional de pertenencia. Matemáticamente, una teoría puede ser descrita como un par $(\Sigma, E \cup A)$, donde Σ es la signatura que especifica la estructura del tipo (*sorts*[4], *subsorts*, clases, y operadores sobrecargados); E es la colección de ecuaciones y relaciones de pertenencia declaradas en el módulo funcional; y A es la colección de atributos ecuacionales (asociatividad, conmutatividad, etc.) que son declarados para los diferentes operadores.

Los tipos de colección de OCL y sus operadores han sido definidos en una especificación algebraica parametrizada, llamada *OCL-SUPPORT{X :: TRIV}*. La siguiente figura muestra los elementos involucrados en el mecanismo de paso de parámetros. *TRIV* es la especificación algebraica del parámetro formal, lo cual es denominado *teoría* en Maude.

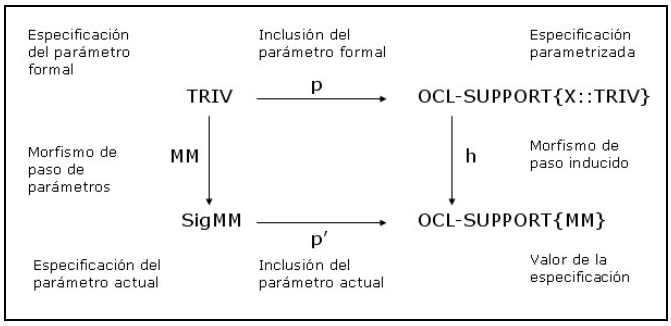

Figura 14. Diagrama de paso de parámetros para el módulo parametrizado *OCL-SUPPORT{X :: TRIV}*

[4] Un *sort* es el nombre que recibe un tipo en Maude

SigMM es la especificación algebraica que se obtiene automáticamente a partir de un metamodelo específico. La especificación *SigMM* constituye el parámetro actual para el módulo *OCL-SUPPORT{X :: TRIV}* y proporciona un constructor para cada tipo que es definido en un metamodelo y una jerarquía de herencia sobre los tipos que aparecen en el metamodelo. La vista *MM* es el morfismo que asocia los elementos del parámetro formal *TRIV* a los elementos del parámetro actual *SigMM*.

Los tipos de colección de OCL y sus operaciones asociadas han sido especificados de manera genérica en el módulo parametrizado *OCL-SUPPORT{X :: TRIV}*, donde el parámetro formal *X* tiene la teoría trivial como tipo. La teoría trivial solo contiene el *sort Elt* (referido a *X$Elt* en la especificación de OCL) que representa el conjunto de elementos que pueden ser contenidos en una colección OCL. Este *sort* representa el tipo *OCLAny* de la especificación de OCL estándar. El módulo *OCL-SUPPORT{X :: TRIV}* importa los tipos básicos de datos y proporciona los constructores que son necesarios para definir colecciones de elementos. También proporciona las operaciones de colección.

En la figura anterior, *p* y *p'* son morfismos de inclusión que indican que la especificación del parámetro formal se incluye en la especificación parametrizada, y que la especificación del parámetro actual es incluida en el valor de la especificación, respectivamente. El morfismo *h* es el morfismo de paso inducido que relaciona los elementos del modulo parametrizado con los elementos del valor de la especificación *OCL-SUPPORT{MM}*, usando el morfismo de paso del parámetro *MM*.

2.3.7.3 Especificación algebraica de los tipos de OCL

Los tipos de datos en OCL se dividen en tipos básicos, tipos de colección, y tipos definidos por el usuario. A continuación se presenta el soporte para las dos primeras clases de tipos.

i. Tipos básicos

En OCL, hay cuatro tipos de datos básicos que tienen una correspondencia directa con tipos de datos básicos de Maude. En la siguiente tabla, se muestra la correspondencia entre el sistema de tipos básicos de OCL 2.0 y Maude, y sus correspondientes primitivas. En la tabla, cuando las operaciones tienen diferentes símbolos en OCL y en Maude, se indica el símbolo en Maude entre paréntesis.

OCL 2.0	Maude	Operadores comunes
Boolean	Bool	or, and, xor, not, = (==), <> (=/=), implies, if-then-else-endif (if-then-else-fi)
Integer	Int	= (==), <> (=/=), <, <=, >, >=, +, -, *, / (quo), mod (rem), abs, max, min
Real	Float	/, round (ceiling), floor
String	String	concat (+), size (length), substring (substr), = (==), <> (=/=)

Tabla 2. Correspondencia entre los tipos de datos de OCL y Maude

ii. Tipos de colección

OCL proporciona cuatro tipos específicos de colección:

- *Set*: Colección que contiene instancias de un tipo OCL válido, en la cual el orden no es relevante y no se permiten elementos repetidos.
- *OrderedSet*: Es un *Set* cuyos elementos están ordenados.
- *Bag*: Es una colección que puede incluir elementos repetidos. Los elementos en este tipo de colección no están ordenados.
- *Sequence*: Es un *Bag* cuyos elementos están ordenados.

Para tener en cuenta las características de orden y unicidad de una colección en OCL, se han introducido dos *shorts* intermedios y sus constructores (mostrados en la siguiente tabla) *Magma{X}* y *OrderedMagma{X}*.

```
1. sort Magma{X} OrderedMagma{X} .
2. subsort X$Elt < Magma{X} OrderedMagma{X} .
3. sorts Collection{X} Set{X} OrderedSet{X} .
4. subsort Collection{X} < X$Elt .
5. subsorts Set{X} OrderedSet{X} < Collection{X} .
6. op _,_ : Magma{X} Magma{X} -> Magma{X} [assoc comm ctor] .
7. op _::_ : Magma{X} Magma{X} -> Magma{X} [assoc ctor] .
8. op Set{_} : Magma{X} -> Set{X} [ctor] .
9. op empty-set : -> Set{X} [ctor] .
10. op OrderedSet{_} : OrderedMagma{X} -> OrderedSet{X} [ctor] .
11. op empty-orderedset : -> OrderedSet{X} [ctor] .
```

Tabla 3. Especificación algebraica de OCL. Especificación de los grupos de elementos

Básicamente, se define el sort *Magma{X}* como el sort del termino que representa un grupo de elementos que no están ordenados (lo cual se indica a través de los atributos de asociatividad y conmutatividad). El constructor para este sort tiene el símbolo *","* y es asociativo y conmutativo. Por ejemplo, trabajando con enteros, *"1, 2, 3"* es un término que representa un *Magma{Int}* válido. Además, se establece que *"1, 2, 3"* y *"3, 2, 1"* representa el mismo grupo de elementos, a través de los atributos de conmutatividad y asociatividad.

El constructor del sort *OrderedMagma{X}* no tiene la propiedad conmutativa, produciendo términos que representan concatenaciones ordenadas de elementos. El constructor para este sort tiene como símbolo *"::"*, permitiendo construir grupos ordenados de elementos usando la sintaxis común para listas en programación funcional. Así, el término *"1 :: 2 :: 3"* representa un *OrderedMagma{Int}* válido, y *"1 :: 2 :: 3"* es diferente a *"3 :: 2 :: 1"* porque el constructor "::" no es conmutativo.

Los términos del sort *Magma{X}* son usados para definir *sets* (línea 8 en la tabla anterior), mientras que los términos del sort *OrderedMagma{X}* son usados en *ordered sets* (línea 10). En la tabla anterior se muestra el código Maude que especifica los tipos *Set* y OrderedSet. En esta especificación, se permite las colecciones de colecciones, al indicarse que una colección puede ser elemento de otra colección (línea 4). El sort *Collection{X}* puede ser considerado como un

concepto abstracto, ya que no existe un constructor específico para él. Cada colección tiene un constructor constante que define una colección vacía (líneas 9 y 11). Los tipos *Bag* y *Sequence* también han sido especificados, de manera similar a los tipos *Set* y *OrderedSet*, respectivamente. En la especificación, la propiedad de unicidad de las colecciones *Set* y *OrderedSet* son comprobadas en las operaciones que unen dos colecciones: *union, intersection,* e *including* para *Set, y union, append, prepend, insertAt* e *including* para *OrderedSet.*

Se ha definido una vista para cada tipo básico de Maude, para utilizar colecciones de tipos de datos básicos. Por ejemplo, para utilizar colecciones de enteros, se ha definido la siguiente vista:

```
view Int from TRIV to INT is sort Elt to Int . endv
```

Esta vista es utilizada para instanciar el módulo *OCL-SUPPORT{X}* como *OCL-SUPPORT{Int}*. De esta manera, el siguiente ejemplo es una colección de enteros válida:

```
OrderedSet{ Set{1, 2, 3} :: Bag{1, 2, 3, 3} :: Sequence{3 :: 3 :: 2 :: 1}}
```

iii. Operadores de iteración o iteradores

Se pueden distinguir dos clases de operaciones sobre colecciones en OCL 2.0: operaciones regulares y operaciones de iteración o iteradores. Las operaciones regulares proporcionan una funcionalidad común sobre colecciones. Las operaciones de iteración o iteradores permiten iterar sobre los elementos de una colección mientras se realiza una acción específica. Este apartado se centra en el segundo tipo de operaciones.

Cada operación de iteración contiene una expresión OCL como parámetro, denominada cuerpo (*body*) de la operación. Por ejemplo, en la siguiente expresión OCL se permite obtener los números pares de un *set* de enteros:

```
Set{1,2,3,4,5,6} -> select(i | i.mod(2) <> 0)
```

En esta expresión, *select* es la operación iteradora y la expresión *(i | i.mod(2) <> 0)* es el cuerpo. Las operaciones iteradoras y las expresiones "*cuerpo*" de la operación (*body expression*) son consideradas en la especificación algebraica de manera separada. Esta separación es necesaria para simular funciones de orden superior en Maude, considerando las funciones del cuerpo como términos que pueden ser pasados como argumentos a las operaciones de iteración.

Utilizando el ejemplo de la selección de números pares de un *set* de enteros, primero hay que estudiar cómo especificar el cuerpo de la operación *i | i.mod(2) <> 0.* Los cuerpos de las operaciones pueden ser evaluados a diferentes tipos dependiendo de la clase de operador que está siendo utilizado. Por ejemplo, el cuerpo de una operación *select* se evalúa a un valor booleano. Dependiendo del tipo

de retorno del cuerpo, un símbolo es asociado a éste indicando el nombre del cuerpo de la operación. Por ejemplo, se obtiene:

```
op isOdd : -> BoolBody{Int} [ctor] .
```

El cuerpo de la operación es construido usando la siguiente operación:

```
op _::_`(_;_`) : Magma{X} BoolBody{X} ParameterList Collection{X} -> Bool .
```

donde el primer argumento es un término que representa un magma de elementos, el segundo argumento es el correspondiente símbolo del cuerpo, el tercer argumento es una lista variable de parámetros que puede ser vacía, y el cuarto argumento es la colección inicial completa a la cual pertenece el primer argumento. Para definir la función del cuerpo, los axiomas deben ser proporcionados por el usuario en notación Maude. Por ejemplo, se define la siguiente ecuación:

```
var intN : Int . var intCol : Collection{Int} . var PL : ParameterList .
eq intN :: isOdd ( PL ; intCol ) = ((intN rem 2) =/= 0) .
```

Una vez el cuerpo de la operación ha sido definido, se proporciona una especificación algebraica de la semántica operacional de la operación *select* para *sets*. Las diferentes operaciones de colección han sido definidas como símbolos de función (términos que son mostrados en la siguiente tabla), dependiendo del tipo de retorno de cada operación. Por ejemplo, la operación *select*, la cual retorna una colección de elementos, se define de la siguiente manera:

```
op select : -> Fun{X} [ctor] .
```

	Tipo de retorno	**Símbolos de los operadores de colección**					**Símbolos de los iteradores**
		Collection	**Set**	**OrderedSet**	**Bag**	**Sequence**	**Collection**
Fun{X}	Collection	union, flatten, including, excluding, iterate	--, inter-section	--, insertAt, append, prepend	intersection	insertAt, append, prepend	select, reject, any, sortedBy, collect, collectNested, iterate
EltFun{X}	Element			first, last, at		first, last at	
BoolFun{X}	Valor booleano	includes, includesAll, excludes, excludesAll, isEmpty, notEmpty					one, forAll, forAll2[5], exists, isUnique
IntFun{X}	Valor entero	count, size, sum, product		indexOf		indexOf	

Tabla 4. Especificación algebraica de OCL. Operaciones colección especificadas

[5] La operación *forAll2* se ha incluido para proporcionar soporte cuando se utilizan dos variables iteradoras en la operación *forAll*

La semántica operacional de las operaciones de iteración se ha definido independientemente de las operaciones del cuerpo. Este hecho permite la reutilización de la especificación algebraica de las operaciones de iteración simulando funciones de orden superior. Tres axiomas constituyen la especificación algebraica del operador *select* para *sets* (como se muestra, mediante notación Maude, en la siguiente tabla).

```
eq Set{ N , M } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } -> including ( ( Set{ M
} -> select ( BB ; PL ; Col )) ) -> flatten
else Set{ M } -> select ( BB ; PL ; Col ) fi .
eq Set{ N } -> select ( BB ; PL ; Col ) = if (N :: BB ( PL ; Col )) then Set{ N } else empty-set fi .
eq empty-set -> select ( BB ; PL ; Col ) = empty-set .
```

Tabla 5. Especificación algebraica de OCL. Especificación de la operación *select* para *sets*

Aparecen tres argumentos en el operador *select*: *BB* es una variable que contiene la expresión booleana del cuerpo, *PL* es lista de parámetros para el operador del cuerpo, y *Col* es el *set* original. El primer axioma considera el caso recursivo, donde hay más de un elemento en el *set*. Si la función del cuerpo se valida a *true*, el elemento es añadido al *set* resultante. Finalmente, la recursión sobre el resto de elementos continúa. El segundo axioma considera la recursión en el caso de que solo haya un elemento en el *set*, siendo un caso baso de la recursión. El tercer axioma considera el caso de que el *set* este vacío.

Para invocar al iterador a la manera de OCL, se utiliza la siguiente operación:

```
op _->_`(_;_;_`) : Collection{X} Fun{X} BoolBody{X} ParameterList Collection{X} ->
Collection{X} .
```

donde el primer argumento es la colección a ser iterada, el segundo argumento es un símbolo de iteración, el tercer argumento es el cuerpo de la operación, el cuarto argumento es una lista de argumentos para el cuerpo de la operación, y el quinto argumento es útil cuando la colección debe ser navegada en el cuerpo de la operación.

Para invocar el iterador *select* sobre un *set* de enteros, con el cuerpo *isOdd* antes especificado, hay que utilizar la siguiente expresión:

```
Set{1, 2, 3, 4, 5, 6} -> select(isOdd ; empty-params ; empty-set) .
```

2.3.7.4 Ejemplo: especificación de una consulta OCL en Maude

Para este ejemplo, presentamos un simple modelo UML de una compañía de autocares. En este diseño, un autocar (*coach*) tiene un número específico de asientos y puede ser usado para viajes regulares (*regular trips*) o privados (*private trips*). En los viajes regulares, los tickets son comprados individualmente. En los viajes privados se alquila un autocar entero.

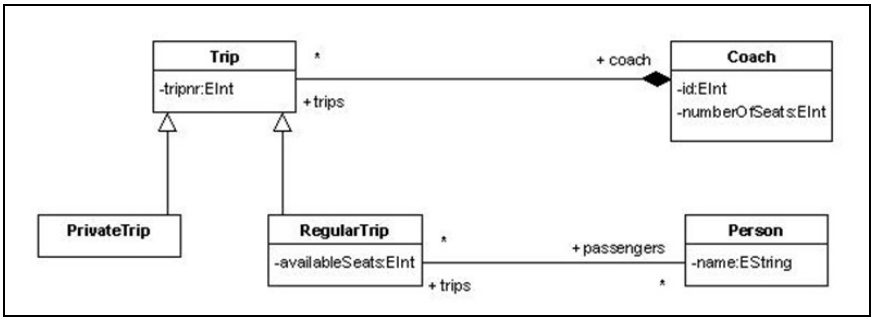

Figura 15. Modelo Coach Company

Hay que tener en cuenta que la especificación de OCL en Maude puede ser utilizada para consultar cualquier artefacto definido en el entorno conceptual MOF: metamodelos, modelos, o instancias de modelos.

Las consultas OCL permiten una mayor precisión en la especificación de un modelo al añadir a éste invariantes[6], o restricciones que deberán cumplirse en todo momento para una instancia del modelo. Por ejemplo, se puede indicar que el temido *"overbooking"* no está permitido en un viaje regular, lo cual queda expresado con el siguiente invariante:

```
context Coach:
inv: self.trips -> select( t:Trip | t.oclIsType(RegularTrip))
-> forAll(r:Trip | r.oclAsType(RegularTrip).passengers -> size() <= r.coach.numberOfSeats)
```

La consulta utilizada en este invariante puede expresarse definiendo una expresión "*cuerpo*" para el iterador *forAll* como una operación. Esta operación comprueba que todos los viajes regulares tienen un número de pasajeros menor al número de pasajeros que se han establecido en el atributo *numberOfSeats* de la correspondiente instancia *Coach*. El código Maude que equivale al cuerpo del operador *forAll* en el ejemplo es el que sigue:

```
var self : trip-Trip . var tripModel : Set{trip} .
op notOverbooked : -> BoolBody{trip} [ctor].
ceq self :: notOverbooked ( PL ; tripModel ) =
((((self :: oclAsType ( ? "RegularTrip" ; tripModel )) :: passengers ( tripModel))
-> size) <= (self :: coach ( tripModel ) :: numberOfSeats))
if self :: trip-Trip .
eq self :: notOverbooked ( PL ; tripModel ) = false [owise].
```

[6] Se puede considerar que un invariante se construye a partir de una consulta OCL que retorna un valor booleano.

donde *self* es una variable del tipo *Trip,* y *tripModel* es un *set* que representa la instancia a ser consultada. La consulta que es utilizada en el cuerpo del invariante puede ser codificada como sigue:

red self :: trips (tripModel) -> select (oclIsTypeOf ; "RegularTrip" ; tripModel) -> forAll (notOverbooked ; empty-params ; tripModel) .

donde *self* es una variable de tipo *Coach,* y *tripModel* es una variable que contiene la instancia del modelo a ser comprobada. Las operaciones *select* y *forAll* proporcionan la expresión "*cuerpo*" de un invariante en código Maude, utilizando el operador *oclIsTypeOf* y el cuerpo *notOverbooked* antes especificado.

2.3.8 Componentes del framework MOMENT

A continuación se presenta el diagrama de los diferentes componentes que participan en MOMENT, con las dependencias existentes entre ellos. A continuación se hace una breve descripción de cada uno de ellos.

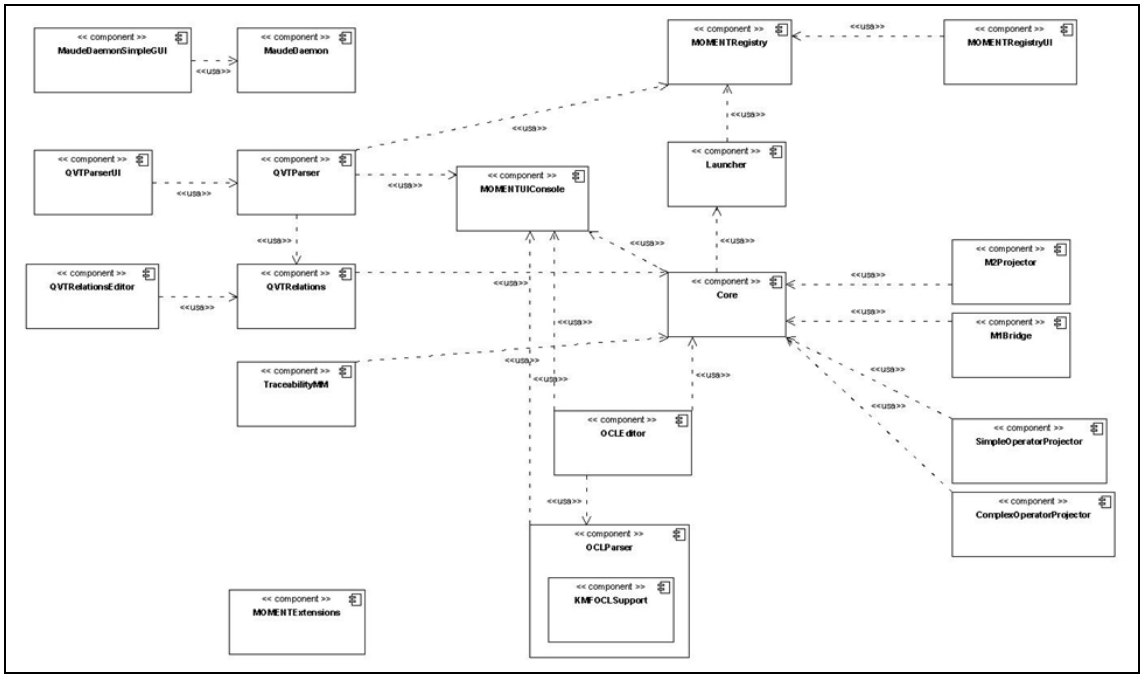

Figura 16. Framework MOMENT. Diagrama de componentes

Kernel

- *Core*: Contiene una serie de módulos parametrizados en Maude que especifica los diferentes operadores para modelos (*Merge*, *Cross*, *Diff,…*), el soporte para el lenguaje OCL 2.0, así como el soporte para transformaciones. Se denomina también *"kernel"*, ya que se trata de la especificación del motor de reducción en Maude de la herramienta. Además, incluye la API principal de MOMENT para acceder a su funcionalidad general.

El resto de componentes representan plug-ins que utilizan la tecnología EMF.

- *Launcher*: Proporciona la interfaz de usuario principal de MOMENT.

Soporte para transformaciones

- *QVTParser*: Construye un modelo QVT a partir de un programa que especifica una transformación, utilizando el lenguaje QVT Relations y OCL.
- *QVTParserUI*: Interfaz de usuario asociada al componente *QVTParser*.
- *QVTRelations*: Genera la especificación de la transformación en Maude a partir de un modelo QVT. Para ello necesita traducir las expresiones OCL contenidas en el modelo a código Maude. Este último se realiza mediante llamadas al módulo *OCLParser*.
- *QVTRelationsEditor*: Un editor textual con coloreado de sintaxis para programas escritos en el lenguaje QVT Relations.

Soporte para OCL 2.0

- *KMFOCLSupport*: Soporte para OCL 2.0 proporcionado por KMF.
- ***OCLParser***: Soporte para la traducción de expresiones OCL a código Maude. Encapsula el KMFOCLSupport. *Conforma la primera parte de este proyecto final de carrera*.
- ***OCLEditor***: Interfaz de usuario que despliega toda la funcionalidad de OCLParser. *Conforma la segunda parte de este proyecto final de carrera*.

Proyectores entre los espacios tecnológicos EMF y Maude (Puentes)

- *M1Bridge*: Da soporte a la interoperabilidad en el nivel M1.
- *M2Projector*: Proporciona soporte a la interoperabilidad en el nivel M2.

Proyectores para operaciones

- *SimpleOperatorProjector*: Proyecta a código Maude las operaciones simples.
- *ComplexOperatorProjector*: Proyecta a código Maude las operaciones complejas, como composición de operaciones simples.

Soporte para la interacción Maude-Eclipse

- *MaudeDaemon*: Da soporte para lanzar comandos en Maude y controlar procesos en Maude desde Eclipse. No se representan sus dependencias por simplicidad del diagrama.
- *MaudeDaemonSimpleGUI*: Consola de Maude integrada en el escritorio de Eclipse.

Registro de metadatos

- *MOMENTRegistry*: Permite registrar metadatos relativos al framework, como metamodelos utilizados, operaciones…

- *MOMENTRegistryUI*: Interfaz de usuario para interaccionar con el componente *MOMENTRegistry*.

Trazabilidad

- *TraceabilityMM*: Proporciona soporte para la trazabilidad entre transformaciones.

Utilidades

- *MOMENTConsoleUI*: Consola de MOMENT integrada en el escritorio de Eclipse.

- *MOMENTExtensions*: Utilidades auxiliares, como una consola para el registro de metamodelos en el registro de EMF.

2.4 KENT MODELING FRAMEWORK

2.4.1 ¿Qué es KMF?

Kent Modeling Framework [KMF] es un proyecto desarrollado por la Universidad de Kent (situada en Canterbury, en el Reino Unido). Su propósito es proporcionar un conjunto de herramientas para dar soporte al desarrollo de software dirigido por modelos. El corazón de KMF es *KMFStudio*, una herramienta que genera herramientas de edición de modelos a partir de la especificación de lenguajes de modelado expresados como metamodelos. *KMFStudio* integra *OCLCommon* y *OCL4KMF*, dos librerías Java que permiten la evaluación dinámica de restricciones OCL. De esta manera, las herramientas generadas utilizan estas librerías para dar soporte a la construcción de modelos coherentes respecto un conjunto de invariantes.

Se incluye un lenguaje propio de metamodelado, KMF (la versión 2.0 está basada en el metamodelo de UML 1.4). KMF utiliza un archivo UML 1.4 XMI para construir la implementación de un modelo. Una vez construido el archivo XMI se genera el código Java que implementa el modelo [AkeLP03]. Además, se proporciona una API reflexiva para construir modelos dinámicamente y acceder a sus propiedades, de manera análoga a EMF.

Este proyecto presenta similitudes claras con EMF y su lenguaje de metamodelado Ecore. EMF y KMF son entornos de trabajo basados en Java que permiten construir herramientas basadas en modelos estructurados, sirviéndose de mecanismos de generación de código Java que parten de modelos persistidos en formato XMI. Sobre la comprobación de la consistencia de los modelos mediante restricciones OCL, el proyecto EMF-OCL persigue el mismo cometido dentro del proyecto EMF.

2.4.2 Implementación de OCL 2.0 en KMF

La Universidad de Kent ha desarrollado una implementación del estándar OCL 2.0 en el marco del *Kent Modeling Framework*. La motivación de este trabajo es dar soporte para la comprobación de restricciones OCL sobre poblaciones de modelos [AkeLP03].

2.4.2.1 Librerías para KMF y EMF

Como KMF y EMF son entornos de trabajo similares, para proporcionar soporte para el análisis y ejecución de expresiones OCL en el contexto de un modelo EMF, se ha desarrollado un puente que permite utilizar KMF sobre modelos EMF. Así pues, se proporciona un conjunto de librerías Java que sirven para dar soporte a KMF y las herramientas que genera, así como a otras herramientas que utilicen tecnología EMF. Además se proporciona integración con la plataforma Eclipse a través de un plug-in.

El paquete de distribución contiene las siguientes librerías [AkeP03]:

- OCLCommon. Una librería diseñada para soportar funciones de OCL comunes, independientemente del entorno de trabajo (KMF o EMF).
- OCL4KMF. Una extensión de *OCLCommon* para KMF.
- OCL4EMF. Una extensión de *OCLCommon* para EMF.
- OCL4EMFPlugin. Un plug-in para Eclipse.

El paquete permite los siguientes casos de uso:

- Análisis sintáctico de las expresiones OCL.
- Análisis semántico de las expresiones OCL.
- Evaluación de las restricciones OCL.
- Generación de código Java para las restricciones OCL.

En general, los escenarios de uso siguen el siguiente patrón:

- Inicialización del modelo.
- Inicialización de la instancia del modelo, si es necesario (evaluación).
- Instanciación de un apropiado procesador de OCL (para KMF o EMF).

2.4.2.2 Estructura de la implementación

La estructura de su implementación puede separarse en dos partes, según la clásica división en dos fases de un traductor, que muestra un orden a seguir para dos tareas fundamentales: análisis y síntesis. En la siguiente figura se muestra un esquema del proceso.

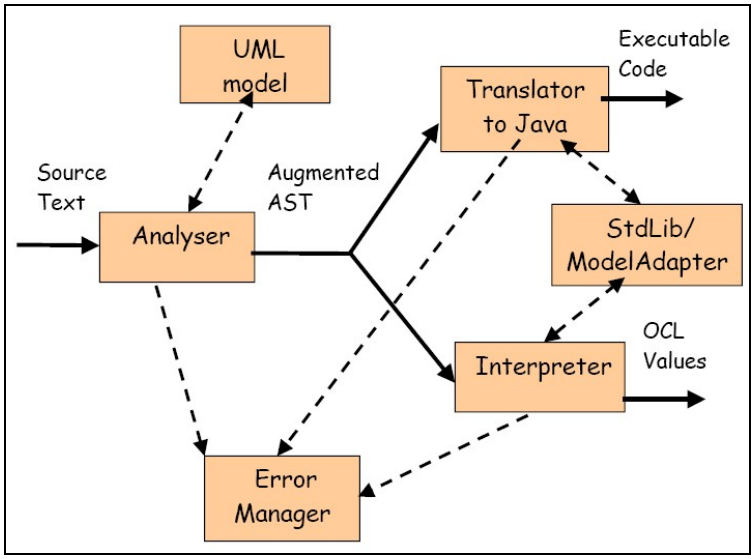

Figura 17. Estructura de la implementación de OCL para KMF

En una primera fase (análisis), se proporciona soporte para la comprobación léxica, sintáctica y semántica de las expresiones OCL. Las entradas son la expresión OCL como texto y el modelo al cual se refiere. Esto último se realiza indicando el archivo XMI donde se especifica dicho modelo, aceptando archivos que

siguen la especificación tanto del metamodelo KMF como Ecore. Indicar el modelo es necesario para el proceso de comprobación de la coherencia semántica de la expresión.

Como resultado de la fase de análisis se tiene un *Augmented Abstract Syntax Tree* (*Augmented AST*) que proporciona una visión estructurada de la expresión OCL y facilita la generación de código a partir de ella. Se trata de una especificación de la semántica del texto introducido. El AST resultado es instancia de un modelo semántico de OCL (que simplifica el modelo sintáctico).

En la segunda fase (síntesis), se permite la traducción de la expresión OCL a código Java ejecutable o su interpretación, en ambos casos utilizando reflexión. Ambos procesos se implementan como *visitors* (siguiendo el patrón de diseño *Visitor*, reseñado en la sección 3.2.7) sobre el modelo semántico.

Para soportar la evaluación de expresiones OCL, se proporciona una librería, llamada *Standard Library*, que contiene los conceptos OCL que dependen dinámicamente de la semántica de la expresión (*OCL values*): cadenas de texto, números, elementos del modelo, enumeraciones…

Por ejemplo, para la expresión OCL

```
context library::Library inv: self.books→asSequence()→first().author
```

se genera o se interpreta el siguiente código Java utilizando reflexión:

```
try {
// Llama propiedad 'books'
OclSet t17 = StdLibAdapterImpl.INSTANCE.Set(self.getBooks());
// Llama operación 'asSequence'
OclSequence t16 = (OclSequence)t17.asSequence();
// Llama operación 'first'
library.Book t15 = (library.Book)t16.first();
// Llama propiedad 'author'
library.Author t14 = (library.Author)t15.getAuthor();
// Devuelve resultado
if (t14 != null) return t14;
else return StdLibAdapterImpl.INSTANCE.Undefined();
} catch (Exception e) {
return StdLibAdapterImpl.INSTANCE.Undefined();
}
```

Por último, la gestión de los mensajes de error de los diferentes módulos se realiza de manera centralizada desde el módulo *Error Manager*.

2.4.3 ¿Qué aporta KMF a MOMENT?

Dentro del marco de la herramienta MOMENT, la implementación de OCL 2.0 de KMF proporciona soporte para el análisis (léxico, sintáctico y semántico) de expresiones OCL y a su vez, la generación del AST facilita la traducción de estas expresiones a código Maude, el cual da su semántica ejecutable. Por tanto nos servimos del front-end de esta implementación, que proporciona la fase de análisis de las expresiones y da como resultado el AST, para posteriormente crear un back-end propio de generación de código.

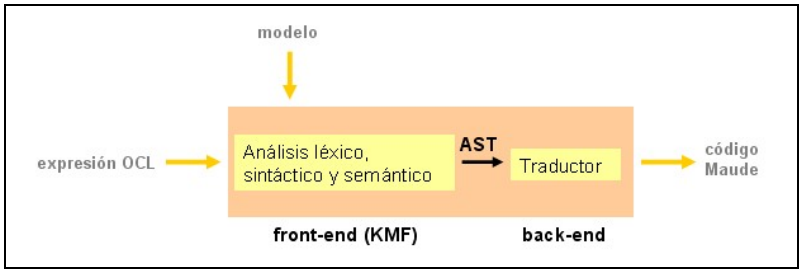

Figura 18. KMF en MOMENT (estructura del traductor)

2.4.3.1 Estructura del front-end

La estructura del front-end es la típica en un traductor y comprende tres acciones fundamentales:

1. Análisis léxico: el analizador léxico toma la entrada textual carácter a carácter y retorna tokens, o unidades léxicas.

2. Análisis sintáctico: el analizador sintáctico reconoce si la entrada pertenece al lenguaje generado por una gramática. Para ello toma como entrada los tokens de salida del analizador léxico y construye el árbol de derivación o *parse*. Se denomina *Abstract Syntax Tree* (AST) a una representación comprimida del árbol de derivación, donde los operadores aparecen como nodos y los operandos de dicho operador como hijos del nodo que representa el operador. Hablaremos indistintamente de árbol de derivación, parse y AST para mayor simplicidad.

3. Análisis semántico: el analizador semántico conecta las definiciones de las variables a sus usos, comprueba que cada expresión tiene un tipo correcto, y transforma el AST en una representación simplificada, adecuada para la generación de código o su interpretación. Esta versión simplificada del árbol se denomina *Augmented Abstract Syntax Tree*. Este análisis se realiza a partir de las acciones semánticas definidas para los símbolos de la gramática del lenguaje (hablamos de una gramática de atributos, o atribuida) y se hace en paralelo con el análisis sintáctico.

El lenguaje OCL entra dentro del grupo de lenguajes que pueden ser generados a partir de una gramática LALR(1) (como la gran mayoría de lenguajes de programación), subconjunto de las gramáticas incontextuales. Esto último permite la construcción del analizador sintáctico a partir de los numerosos

generadores automáticos de *parsers* LALR(1) existentes. Llamamos *parser* al analizador sintáctico que produce finalmente un árbol de derivación. Mediante acciones semánticas introducidas en el proceso de análisis sintáctico se puede realizar el proceso de análisis semántico y generación de código. En el caso de KMF se ha utilizado el generador de analizadores léxicos Flex [Flex] y el generador de *parsers* para Java CUP [CUP], a los cuales hay que precisar, respectivamente, los símbolos terminales que reconocerá el analizador léxico y la gramática del lenguaje para la que se quiere generar un *parser*. De esta manera se da lugar a una implementación completa a partir de especificaciones que pueden ser muy sencillas.

En nuestro caso, para realizar el análisis semántico se necesita información del modelo al cual se refiere la expresión OCL.

Todo mensaje de error es procesado por un módulo gestor de errores.

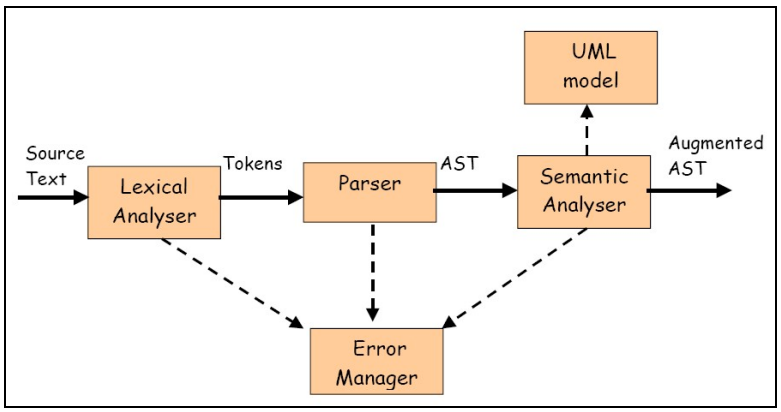

Figura 19. KMF. Estructura del front-end

2.4.4 ¿Cómo analizar una expresión OCL y obtener su AST?

Vamos a ver un ejemplo sencillo donde se utilizarán los métodos necesarios de las librerías de KMF para EMF (básicamente de *OCLCommon* y *OCL4EMF*).

Supongamos que queremos analizar la corrección de una expresión OCL y obtener su AST asociado. El AST se podrá obtener siempre que la expresión sea léxica, sintáctica y semánticamente correcta. Supongamos que así es. Supongamos que la expresión OCL completa la especificación de un modelo Ecore, cuya descripción se encuentra físicamente en un archivo XMI. Por último, supongamos que tenemos acceso a las clases Java que definen al modelo en EMF.

```
//Nos guardamos en una variable el paquete raíz del modelo
//Para este ejemplo utilizamos el clásico modelo Royal and Loyal [R&L]
RoyalPackage pkg = RoyalPackage.eINSTANCE;

//El log de mensajes de error queremos redireccionarlo a la salida de error
ILog log = new OutputStreamLog(System.err);
```

```
//Inicializamos el procesador EMF de OCL, indicándole el log a utilizar
OclProcessor processor = new EmfOclProcessorImpl(log);

//Cargamos la información del modelo en el procesador
processor.addModel(pkg);

//La expresión a analizar es la siguiente, del modelo Royal and Loyal [R&L]
String ocl_expression = "context LoyaltyAccount inv: self.points > 100";

//Analizamos sintácticamente la expresión. En caso de error
// se mostraría en el log de errores
PackageDeclarationAS pkgAS = processor.parse(ocl_expression);
// Si el análisis es correcto,
// retorna la raíz del AST, resultado del análisis sintáctico

//Analizamos semánticamente la expresión. En caso de error
// se mostraría en el log de errores
List l = processor.analyse(ocl_expression);

//Un análisis correcto retorna una lista de ContextDeclaration,
//uno por cada declaración de contexto que incluya la expresión.
//Se trata de una clase incluida en el modelo semántico de OCL de KMF
//y que básicamente representa la declaración de un contexto del
//cual depende una o más expresiones OCL.
// Tomamos el primer elemento de la lista
ContextDeclaration contextDeclaration = (ContextDeclaration) l.get(0);

//Establecemos un visitador de depuración que imprime la estructura
//del AST de manera textual
OclDebugVisitorImpl visitor = new OclDebugVisitorImpl();

//Por último, recorremos el AST
//siguiendo la estrategia del patrón de diseño Visitor
System.out.println(contextDeclaration.accept(visitor,"").toString());

//Se imprime el resultado por la salida estándar. Como ya hemos dicho,
//se trata de una representación textual del AST vinculado a la expresión:

OperationCall {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(LoyaltyAccount)
      }
      Property points:Integer
   }
   Operation >()
   Integer(100)
}
Type = Boolean
```

Supongamos que la expresión fuera sintácticamente incorrecta. El mensaje de error que se almacenaría en el log sería el siguiente:

Expresión OCL

```
context LoyaltyAccount inv: self.points >< 100
(no existe el operador ><)
```

Mensaje de error

```
Error: []1:42 Syntax error near '<'
Error: []1:42 Couldn't repair and continue parse near '<'
```

El mensaje de error indica el punto donde se ha detenido el análisis.

Supongamos que la expresión fuera semánticamente incorrecta. El mensaje de error que se almacenaría en el log sería el siguiente:

Expresión OCL

```
context LoyaltyAccount inv: self.colors > 100
(no existe la propiedad colors para el contexto LoyaltyAccount)
```

Mensaje de error

```
Error: []Unknown property 'colors' on 'OclModelElementType(LoyaltyAccount)'
Error: []Problem analysing expression : DotSelectionExpAS {7}
```

De igual manera se muestra un mensaje indicando el motivo del error.

2.4.5 Modelo sintáctico de OCL 2.0

Las instancias de este modelo sintáctico [AkeLP03] serán los diferentes AST's generados a partir del análisis sintáctico de expresiones OCL. Esta estructura se ve simplificada en el proceso de análisis semántico para facilitar la generación de código o interpretación a partir de él.

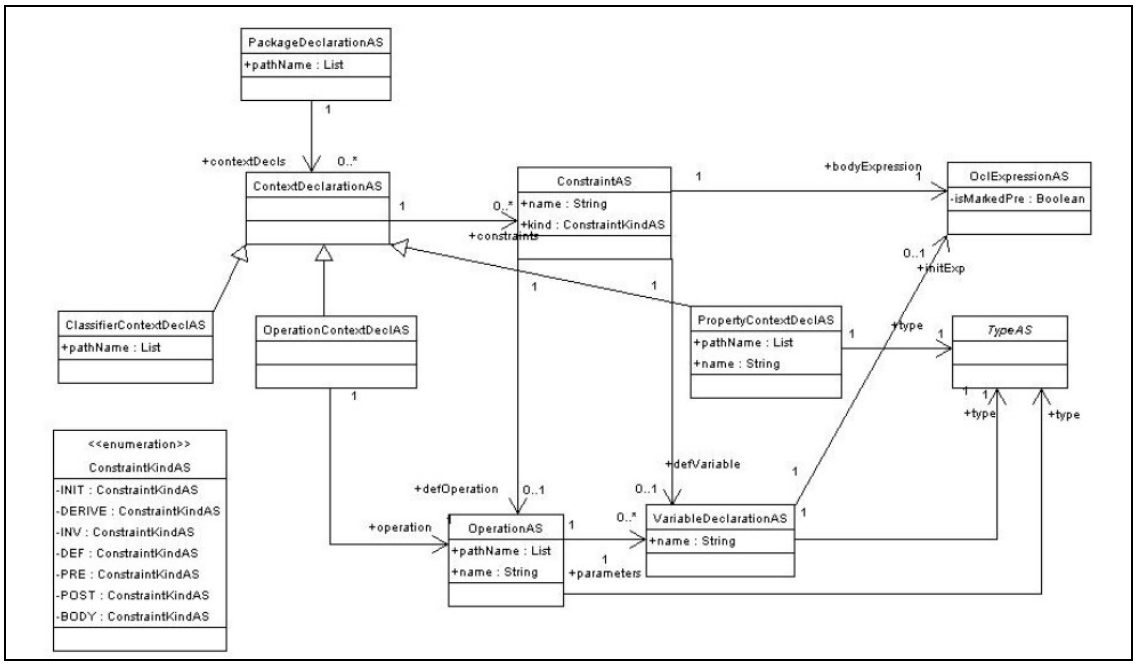

Figura 20. Modelo sintáctico de OCL. Contextos sintácticos

La raíz del modelo es la declaración de un paquete (*PackageDeclarationAS*) que se vincula con un conjunto de declaraciones de contexto (*ContextDeclarationAS*). Las declaraciones de contexto pueden ser de tres tipos:

- *ClassifierContextDeclAS*: asociadas a una clase del modelo. Es el caso, por ejemplo, de la definición de un invariante.

```
context LoyaltyProgram
inv: self.Membership.account->isUnique( number )
```

- *OperationContextDeclAS*: llevan asociadas una operación con sus parámetros y su tipo de retorno, o tipo de la operación. Es el caso de la definición de la semántica de un método de una clase.

```
context CustomerCard::getTransactions(from : Date, until: Date )
                        : Set(Transaction)
body: transactions->select( date.isAfter( from ) and
                    date.isBefore( until ) )
```

- *PropertyContextDeclAS*: para expresar propiedades. Llevan asociadas un tipo.

En los tres casos, las declaraciones de contexto llevan asociadas un conjunto de restricciones OCL de cierta clase (*ConstraintAS* de clase *ConstraintKindAS*), que define si se trata de un invariante (*INV*), la definición de un dato derivado (*DERIVE*), una precondición (*PRE*)…

Cada restricción OCL tiene asociado un cuerpo (*OclExpressionAS*) que es la expresión OCL en sí.

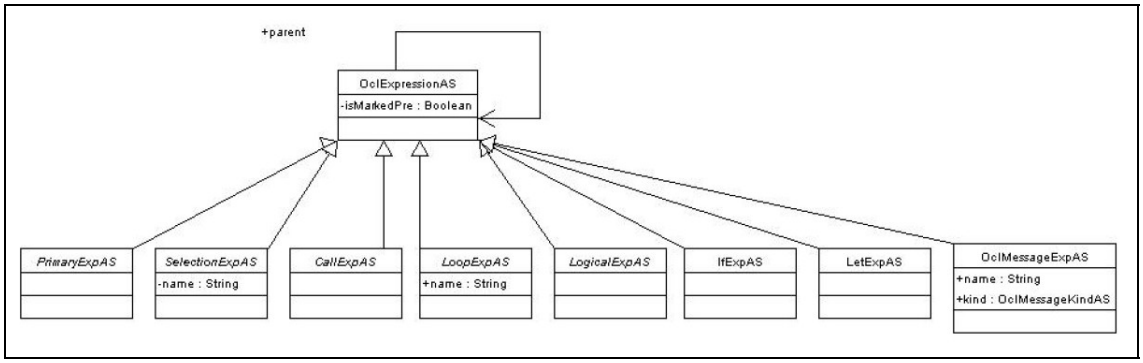

Figura 21. Modelo sintáctico de OCL. Expresiones

Las expresiones OCL se dividen en diferentes grupos: primarias, de selección, de llamada, iteradoras, lógicas, If, Let, y mensajes. Mediante la asociación *parent* se establece un orden de anidamiento entre ellas. Pasamos a ver los modelos específicos de cada uno.

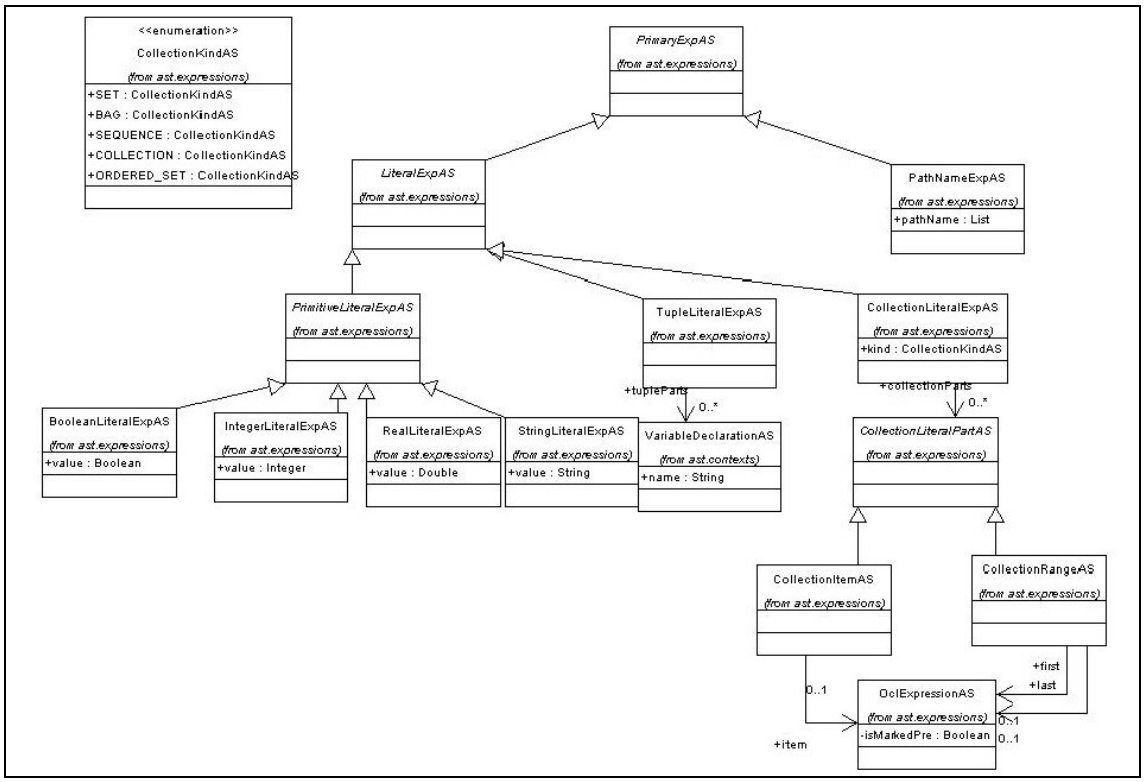

Figura 22. Modelo sintáctico de OCL. Expresiones primarias

Las expresiones primarias (*PrimaryExpAS*) se dividen en literales (*LiteralExpAS*) y navegaciones (*PathNameExpAS*). Las navegaciones son simplemente una lista de nombres. Los literales incluyen a los llamados primitivos o básicos (*PrimitiveLiteralExpAS*), que incluyen a los valores booleanos (*BooleanLiteralExpAS*), enteros (*IntegerLiteralExpAS*), reales (Real*LiteralExpAS*), y cadenas de caracteres (*StringLiteralExpAS*). Otro tipo de literales son las tuplas (*TupleLiteralExpAS*), que incluye un conjunto de declaraciones de variables (*VariableDeclarationAS*). Por último tenemos las colecciones (*CollectionLiteralExpAS*), las cuales están formadas por un conjunto de partes (*CollectionLiteralPartAS*), que a su vez cada una puede ser un ítem individual (*CollectionItemAS*) o expresar un rango (*CollectionRangeAS*). Las colecciones pueden ser de diferentes clases expresadas en *CollectionKindAS*, dependiendo de si se tratan de colecciones ordenadas o de si se pueden o no repetir sus elementos. Hablamos de Set (no ordenada y no repetición), OrderedSet (ordenada y no repetición), Bag (no ordenada y repetición) y Sequence (ordenada y repetición).

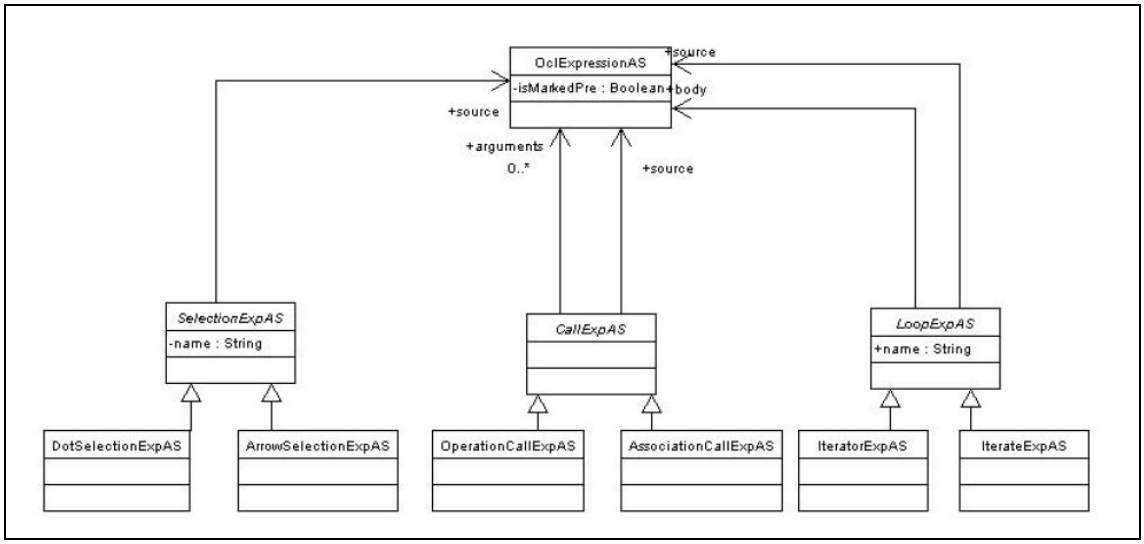

Figura 23. Modelo sintáctico de OCL. Expresiones de selección, llamada e iteración

Las expresiones de selección (*SelectionExpAs*) se dividen en "tipo punto" y "tipo flecha", utilizadas para navegaciones y operaciones de colección respectivamente. Poseen una expresión origen (*source*), a la cual se refieren.

Las expresiones de llamada (*CallExpAS*) se dividen en operaciones (*OperationCallExpAS*) y asociaciones (*AssociationCallExpAS*). En ambos casos hay una expresión a la cual se hace referencia (*source*) y un conjunto de argumentos (*arguments*), que puede ser vacío.

Por ejemplo, en la expresión:

Set{1,2,3,4} -> union(Set{5,6,7})

Set{1,2,3,4} sería la expresión a la que se hace referencia, *union* la operación a realizar y *Set{5,6,7}* sería el único argumento.

Las expresiones de iteración (*LoopExpAS*) diferencian entre el iterador genérico *iterate* (*IterateExpAS*) y el resto de más alto nivel (*IteratorExpAS*). En

ambos casos se tiene una expresión a la que se hace referencia (*source*), generalmente una colección de elementos, y por otra parte el cuerpo de la expresión iteradora (*body*), que será otra expresión OCL.

Por ejemplo, en la expresión:

self.partners -> select(p: ProgramPartner | p.numberOfCustomers > 0)

self.partners, representa la colección sobre la que se aplica la operación iteradora, *select* es la operación iteradora a aplicar y *p.numberOfCustomers > 0* es el cuerpo de la expresión.

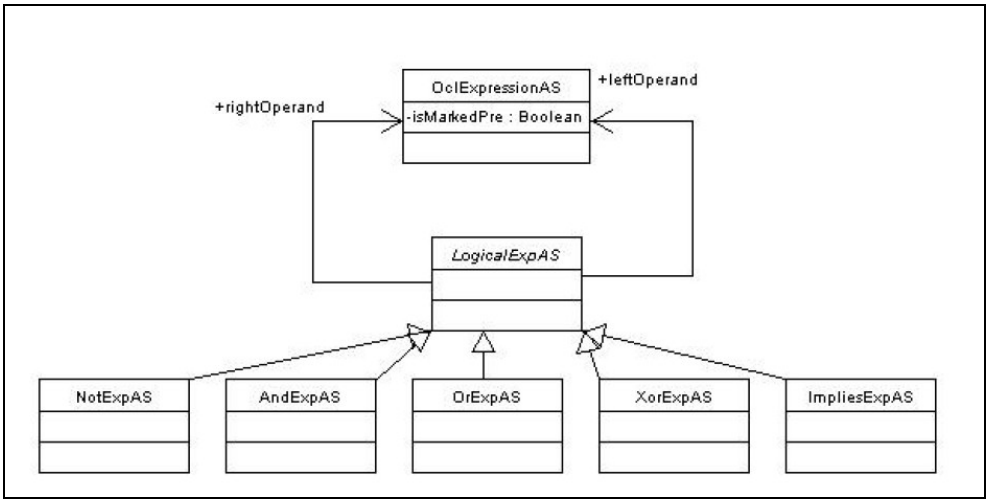

Figura 24. Modelo sintáctico de OCL. Expresiones lógicas

Las expresiones lógicas (*LogicalExpAS*) se componen de dos operandos (*leftOperand*, *rightOperand*) y se clasifican en expresiones de negación (*NotExpAS*), conjunción (*AndExpAS*), disyunción (*OrExpAS*), disyunción exclusiva (*XorExpAS*) e implicación lógica (*ImpliesExpAS*).

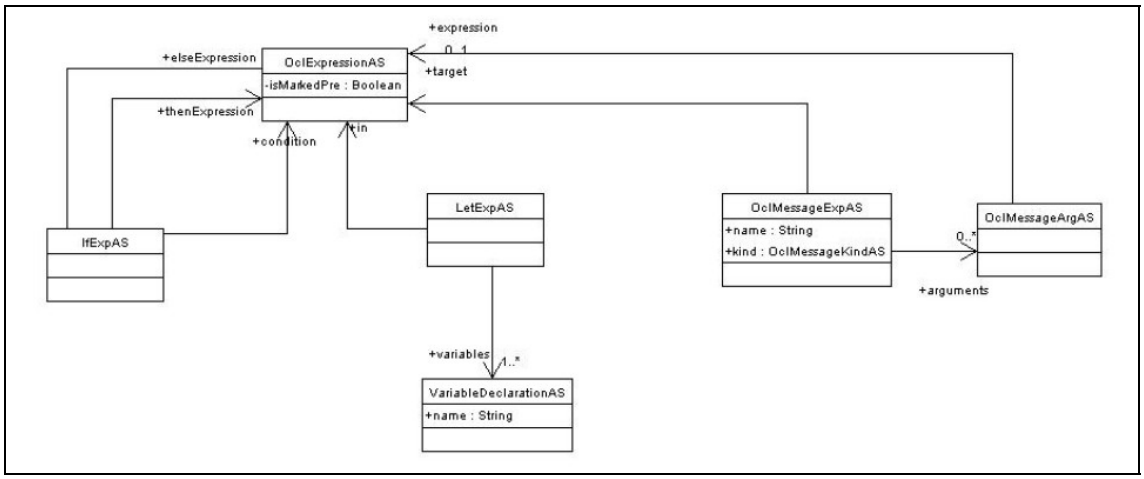

Figura 25. Modelo sintáctico de OCL. Expresiones If, Let y OclMessage

Las expresiones condicionales (*IfExpAS*) siguen la sintaxis:

if (expresión_booleana) then expresión$_1$ else expresión$_2$ endif expresión_booleana corresponde a condition en el modelo, expresión$_1$ a thenExpression, y expresión$_2$ a elseExpression.

Las expresiones Let (*LetExpAS*) se relacionan con un conjunto de declaraciones de variables (*VariableDeclarationAS*) y con la expresión OCL a la que hacen referencia (*in*).

Por último, se especifica la utilización del tipo OclMessage mediante el uso de operadores.

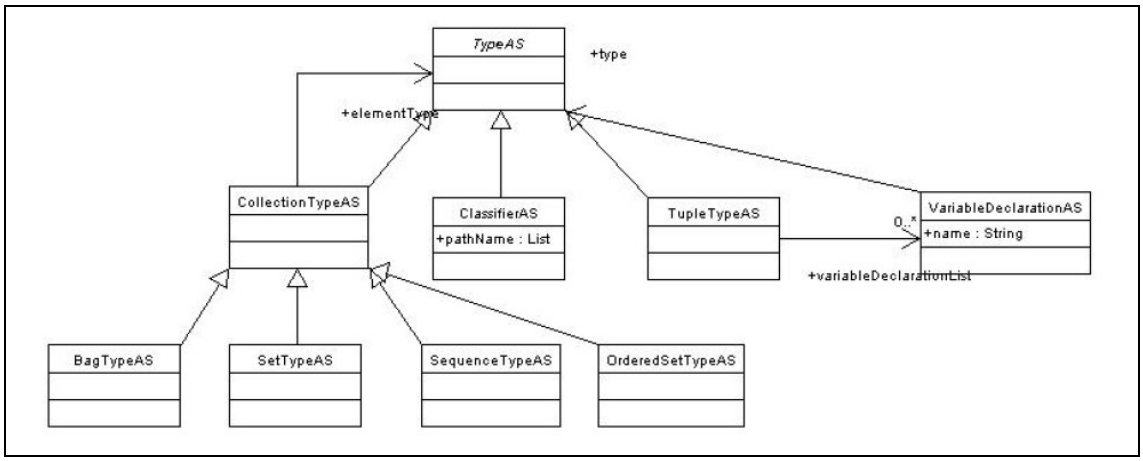

Figura 26. Modelo sintáctico de OCL. Tipos

Esta última parte del modelo expresa los diferentes tipos utilizados en OCL (*TypeAS*), a parte de los tipos básicos. Hablamos de los tipos de colección (*CollectionTypeAS*), los clasificadores o clases del modelo (por ejemplo, UML) al cual se refiere la expresión OCL (*ClassifierAS*), y por último al tipo tupla (*TupleTypeAS*) formado por un conjunto de declaraciones de variable (*VariableDeclarationAS*).

2.4.6 Limitaciones

La implementación de OCL proporcionada por KMF tiene ciertas limitaciones, conceptos del estándar OCL 2.0 no soportados, que se listan a continuación [AkeLP03]:

- Operador *hasSent* ('^') utilizado en postcondiciones, que indica que una comunicación a tenido lugar.
- Operador de mensaje '^^'.
- Operadores relativos a mensajes: isSignalSent(), isOperationCall(), hasReturned() y result().
- Tipos OclState y OclMessage

- Otros tipos de contexto que no sean inv:
- @pre (que hace referencia al valor de una propiedad al comienzo de una operación)

De estas limitaciones, la más importante, en nuestro caso, es la de no poder utilizar otros tipos de contexto que no sean invariantes. No obstante, se ha demostrado que no es tal problema, ya que el proceso de análisis de la expresión no comprueba que se deba retornar siempre un tipo booleano. De esta manera hemos podido utilizar esta herramienta tanto en invariantes como en consultas a modelos, que retornan tipos diferentes al booleano.

Otras limitaciones detectadas con el uso son las siguientes:

- Respecto el nombre de las variables:

  El nombre de una variable debe responder a la expresión regular:

```
[A-Za-Z0-9_][A-Za-Z0-9_]*
```

- Respecto al tipado de variables utilizadas en funciones iteradoras:

  Siempre hay que declarar su tipo explícitamente. Por ejemplo:

```
self.trips->collect(t:Trip | t.id)
```

- Respecto a posibles resultados anómalos en la utilización concatenada de operadores. Uso del parentizado:

  En expresiones donde se concatenan diferentes operadores se han detectado resultados anómalos. En ocasiones concretas KMF devuelve una excepción de puntero nulo y no retorna el AST para el proceso de traducción. En estas situaciones la solución pasa por anidar los operadores mediante paréntesis de la siguiente forma:

```
((((Expresión -> op1) ->op2) ->op3)...)
```

2.5 Trabajos relacionados

A continuación se describen otros proyectos que dan soporte al lenguaje OCL 2.0. En todos ellos se ha implementado una especificación del estándar (generalmente, se da soporte a una parte), y el objetivo fundamental en todos ellos es la aplicación de OCL en la construcción de modelos más expresivos y consistentes. También se da, en general, soporte a la consulta sobre instancias de modelos y en algunos casos se dotan de mecanismos de generación de código.

2.5.1 Dresden OCL Toolkit

Este proyecto se ha realizado en el marco de la Universidad Técnica de Dresden (*Technische Universität Dresden*, en Alemania). La versión 2.0 del *Dresden OCL Toolkit* [Dresden] para OCL 2.0 consta de los siguientes elementos:

Metamodelo OCL común

Todo el conjunto de herramientas está basado en un metamodelo de OCL derivado de los metamodelos de MOF14 y UML15.

Repositorio de metadatos (MDR)

Todos los modelos y metamodelos (MOF14, UML15 y el metamodelo común de OCL) se almacenan en un respositorio de metadatos. Se utiliza una implementación basada en *NetBeans*.

API de los modelos y metamodelos basada en Java

Utilizando el repositorio de metadatos y los metamodelos almacenados, se generan interfaces Java para acceder a los modelos en el repositorio.

Librería OCL estándar

Si ha adaptado la probada "*OCL-Basisbibliothek*" de Frank Finger para crear la nueva arquitectura basada en un metamodelo.

OCL 2.0 Parser

El parser utilizado en este proyecto está optimizado para la gramática L-atribuida de OCL 2.0. Se utiliza un versión mejorada del popular generador de parsers SableCC [SableCC] en Java para crear un analizador léxico y sintáctico (en definitiva, un parser[7]) y una plantilla abstracta para un evaluador de atributos. La clase que implementa el evaluador de atributos se deriva de la plantilla por herencia.

[7] Denominamos *parser* al analizador léxico y sintáctico que produce, como resultado del análisis, un árbol de derivación (también denominado Abstract Syntax Tree, AST). *Parsear* es la acción de aplicar un proceso de análisis léxico y sintáctico sobre un programa fuente, que da como resultado un AST.

El parser se divide en dos fases: un primer paso crea un arbol sintáctico concreto (*Concrete Syntax Tree*, CST), y en un segundo paso el evaluador de atributos realiza una transformación desde el arbol sintáctico concreto a uno abstracto (*Abstract Syntax Tree, AST*).

Generador de código y evaluador de restricciones

El entorno de trabajo para OCL 2.0 es una aplicación de demostración que permite cargar modelos basados en la especificación MOF14, creando restricciones sobre estos modelos, generando código para ellos y evaluando las restricciones en el contexto del modelo específico (nivel MOF M1).

Interfaz de usuario del OCL 2.0 Parser

Se trata de una aplicación de demostración que permite visualizar la funcionalidad de OCL 2.0 Parser. Permite cargar o editar textualmente las restricciones OCL, cargar modelos que siguen la especificación UML15 a través de la importación de archivos XMI, y parsear las restricciones en el contexto del modelo cargado. Después del proceso de parseado, las restricciones pueden adjuntarse a los elementos del modelo considerado, pudiendo exportar los modelos mejorados mediante XMI.

2.5.2 ITP/OCL

La herramienta ITP/OCL [ClavelE06b] es una herramienta experimental desarrollada por Marina Egea y Manuel Clavel en la Universidad Complutense de Madrid. ITP/OCL aprovecha las capacidades reflexivas del sistema Maude. Esta herramienta está basada en el mecanismo de reescritura de Maude, para soportar la validación automática de diagramas de clases UML mediante restricciones OCL. Desde un punto de vista conceptual, ITP/OCL está basada en la semántica ecuacional para los diagramas UML/OCL combinados, desarrollada en [Egea05] y [ClavelE06a], la cual expresa que:

- Los diagramas de clases y objetos están especificados como teorias ecuacionales de pertenencia.
- Los invariantes se representan como términos booleanos sobre extensiones de estas teorias.
- La validación de los diagramas de objetos con respecto a los invariantes se reduce a comprobar cuando el correspondiente término booleano se reescribe a verdadero o a falso.

Desde un punto de vista de implementación, la herramienta ITP/OCL está escrita completamente en Maude, haciendo extensivo el uso de la reflexión para implementar la interfaz de usuario, gracias a lo cual, el usuario no tiene porqué saber de la utilización de una semántica ecuacional. Solamente tendrá que tener nociones sobre la construcción de diagramas UML e invariantes OCL.

La herramienta ITP/OCL puede:

- Crear diagramas UML de clases y objetos.
- Parsear invariantes OCL sobre diagramas de clases UML.
- Comprobar invariantes OCL sobre diagramas de objetos UML.
- Evaluar consultas OCL sobre diagramas de objetos UML.

2.5.3 OCTOPUS

Este proyecto ha sido desarrollado por Jos Warmer y Anneke Kleppe, antiguos colaboradores de OMG. Participaron en la creación de la primera versión standard de UML y han contribuido a la especificación del estándar OCL como integrantes de Klasse Objecten [KlasseO], una compañía dedicada al desarrollo de software dirigido por modelos. Octopus [Octopus] permite comprobar estáticamente expresions OCL. Comprueba la sintaxis, los tipos de las expresiones y el uso correcto de los elementos del modelo, como los roles de las asociaciones y los atributos. Además permite transformar un modelo UML, incluyendo las expresiones OCL, a código Java. Esta herramienta se integra mediante un conjunto de plug-ins sobre Eclipse, y está implementado en Java.

Octopus conforma totalmente la versión de OCL 2.0. Todos los nuevos constructores, como las reglas de derivación y la especificación de valores iniciales, se soportan completamente.

Permite generar prototipos de aplicación en tres capas desde los modelos UML/OCL combinados. La capa lógica incluye objetos Java la comprobación de invariantes y multiplicidades sobre el modelo. La capa de persistencia consiste en un lector y escritor XML dedicado para un modelo UML/OCL. Almacena y permite recuperar toda la información relativa al prototipo de aplicación. La capa de interfaz consiste en una implementación de un plug-in en Eclipse para la Eclipse Rich Client Platform (RCP). Desde una vista se muestran todas las instancias en el sistema, se permite crear y examinar las instancias del modelo UML/OCL, y se da la posibilidad de comprobar los invariantes sobre las instancias de un modelo.

2.5.4 EMFT-OCL

El proyecto EMFT (Eclipse Modeling Framework Technology, [EMFT]) fue iniciado, en el seno del proyecto Eclipse, para desarrollar y promover las nuevas tecnologías que extienden o complementan EMF.

Sus tres componentes principales son: *EMFT-OCL*, *EMFT-Validation y EMFT-Query*.

EMFT-OCL es un componente de este proyecto que da soporte a la integración de OCL en EMF. Permite definir APIs para la sintaxis de las expresiones OCL y da soporte para la definición de consultas y restricciones sobre modelos Ecore. Utiliza, además, una interfaz basada en el patrón de diseño *Visitor* para recorrer el AST asociado a las expresiones OCL. Emplea un parser de OCL generado por el *LALR Parser Generator*, un proyecto SourceForge [SourceF], bajo la licencia EPL (*Eclipse Public License*, cuya versión 1.0 está descrita en [EPL]).

El componente OCL de EMFT proporciona la siguiente funcionalidad:

- Una API para el modelo sintáctico abstracto de OCL, utilizando el patrón de diseño *Visitor* para procesar ASTs.
- Una API para parsear y evaluar consultas y restricciones OCL sobre los elementos de un modelo Ecore.
- Una API para la integración del componente OCL en el *framework EMFT-Query*.
- Las expresiones OCL son usadas para implementar consultas (elementos *Query*) y restricciones.
- Y además, el componente OCL soporta desarrollos *stand-alone*.

Un modelo Ecore, con EMFT-OCL, puede incluir anotaciones para proporcionar la especificación OCL de invariantes, propiedades derivadas y operaciones. Al generar las clases Java asociadas al modelo, se generan métodos que implementan estas especificaciones.

Capítulo 3

OCLParser. Soporte OCL en MOMENT

3.1 Análisis de requisitos

3.1.1 Contexto

En la Ingeniería Dirigida por Modelos, los artefactos software son representados por modelos que pueden ser manipulados. Estas manipulaciones pueden ser realizadas por un conjunto de transformaciones y consultas. El estándar Query/Views/Transformations, utilizando el lenguaje estándar OCL es el lenguaje adecuado para estos propósitos.

En el contexto de la herramienta de gestión de modelos MOMENT se ha dado soporte al lenguaje para la especificación de transformaciones y relaciones de equivalencia QVT Relations. Debido a que un programa escrito en este lenguaje incluye consultas OCL para completar su especificación, y que finalmente hay que traducirlo a código Maude para darle semántica operacional a la transformación, se hacía necesario la creación de un módulo de soporte para el lenguaje de definición de restricciones y consultas OCL 2.0. En este marco han nacido los desarrollos que han permitido este proyecto final de carrera.

3.1.2 Requisitos

En un comienzo se detectaron los siguientes requisitos generales derivados de la necesidad de un soporte para el lenguaje OCL 2.0 en el *framework* MOMENT:

- *Soporte para la traducción de código OCL 2.0 a código Maude*. Se necesitaba un módulo que diera soporte a la traducción de las expresiones OCL de un programa QVT a código Maude, permitiendo la integración de estas expresiones en el código Maude finalmente generado para ejecutar una transformación.

- *Plataforma Eclipse y lenguaje de programación Java.* Debía estar soportado sobre la plataforma tecnológica de Eclipse, y los módulos debían estar desarrollados en el lenguaje de programación Java para integrarse con el resto de los módulos desarrollados para el *framework* MOMENT.

- *Arquitectura de plug-ins*. MOMENT utiliza una arquitectura de plug-ins para integrar todos sus componentes, de esta manera sigue la filosofía de interacción e integración de componentes de Eclipse. De esta manera, el soporte a OCL debería estar conformado por un plug-in o conjunto de ellos.

- *Una interfaz sencilla y funcional.* El soporte OCL debía tener una interfaz clara para facilitar la tarea de automatización de la traducción de la especificación de transformaciones y relaciones de equivalencia desde QVT Relations a Maude.

- *Necesidad de partir de una base sólida.* No solo había que dar soporte a la traducción de expresiones OCL a código Maude. Estas expresiones debían

ser previamente correctas, sintáctica y semánticamente, respecto el estándar OCL 2.0.

3.1.3 Ampliación de requisitos

Una vez comenzó el desarrollo del plug-in que ha dado lugar al proyecto *OCLParser* se perfilaron las funcionalidades que debía proporcionar la interfaz del soporte OCL de MOMENT. Parte de estas funcionalidades se han integrado derivadas de la necesidad de métodos de consulta para la interfaz de evaluación de expresiones OCL sobre modelos que es el proyecto *OCLEditor*, la segunda parte que conforma este proyecto final de carrera. Veamos el listado final de la lista de funcionalidades que se han pedido a *OCLParser*:

- *Traducción de expresiones OCL a código Maude*. Tanto para la integración de este código en el código Maude generado para una transformación o relación de equivalencia, como para permitir la ejecución y evaluación de consultas e invariantes definidos sobre modelos o metamodelos. Este proceso de traducción debe tener implícito un proceso de análisis sintáctico y semántico de las expresiones a traducir. No se debe generar código si la expresión no es correcta según es estándar OCL 2.0.

  Esta es la única funcionalidad utilizada desde el soporte de transformaciones de MOMENT. Los siguientes puntos definen funcionalidades que han sido ser integradas en el contexto del proyecto OCLEditor.

- *Análisis sintáctico y semántico de expresiones* OCL. Además se debe gestionar el retorno de posibles errores.

- *Obtención de un esquema textual del Augmented AST*, resultado del proceso de análisis semántico de una expresión.

A partir de estos requisitos se ha diseñado e implementado una solución que ha cubierto todas estas características y funcionalidades, dando lugar al proyecto denominado OCLParser.

3.2 Diseño de la solución

El desarrollo del componente ***OCLParser*** ha constituido una parte fundamental de este proyecto final de carrera. A partir de este punto se va a realizar una descripción del diseño y la funcionalidad aportada por este módulo, así como su integración con el resto de componentes de MOMENT.

3.2.1 Componentes del Soporte para OCL en MOMENT

El Soporte para OCL en MOMENT está formado por los siguientes componentes:

- *OCLParser*: que permite el análisis y traducción de las expresiones OCL utilizadas en el contexto de una transformación QVT. Encapsula el soporte de KMF para OCL 2.0.

- *OCLEditor*: Interfaz que despliega toda la funcionalidad de OCLParser. Se describe en el capítulo 4 de este documento.

- *La especificación algebraica de OCL 2.0 en Maude incluida en el kernel de MOMENT*: la misión de OCLParser es traducir las expresiones OCL a axiomas y expresiones en Maude que conformen esta especificación algebraica.

No obstante, en el contexto del proyecto MOMENT, se utiliza también la denominación *"Soporte para OCL"* como sinónimo del módulo *OCLParser*.

3.2.2 Relación con el Soporte para transformaciones

Una transformación sobre modelos se especifica, según el estándar QVT, en un programa mediante el lenguaje QVT Relations, el cual utiliza OCL como lenguaje de consulta. Un traductor (el componente de MOMENT, *QVTParser*) transforma el programa en un modelo que conforma el metamodelo de QVT. Este modelo incluye expresiones OCL que deben ser traducidas e integradas en el código Maude final, que proporcionará la ejecutabilidad de la transformación. El proceso de generación de código Maude a partir de un modelo QVT implica un recorrido de este modelo (realizado mediante el componente de MOMENT, *QVTRelations*), que a su vez llevará aparejado un secuencia de llamadas al módulo *OCLParser* para obtener, bajo demanda, las expresiones OCL traducidas a código Maude. Por tanto, OCLParser da soporte a la traducción de las expresiones OCL utilizadas en un programa QVT a código Maude.

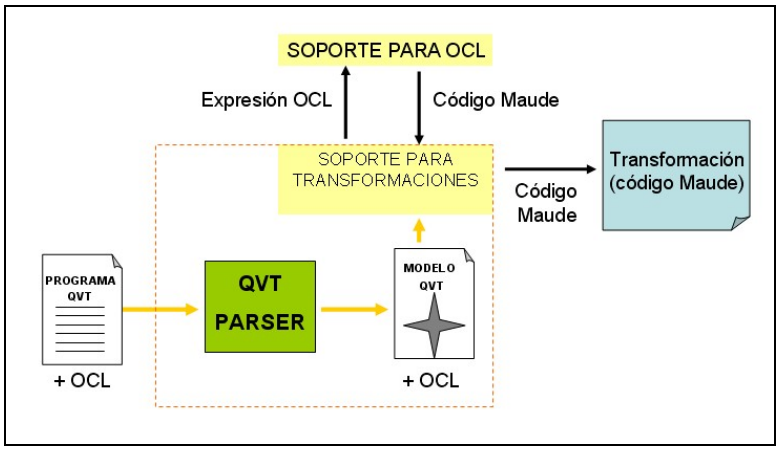

Figura 27. OCLParser. Integración con el Soporte para transformaciones de MOMENT

3.2.3 OCLParser y KMF

KMF dota de un mecanismo de comprobación léxica, sintáctica y semántica de expresiones OCL según el estándar OCL 2.0. Para analizar la semántica de las expresiones se necesita considerar el modelo sobre el que están definidas. Como resultado del análisis semántico de una expresión OCL se obtiene un Augmented AST, que representa la estructura semántica de la expresión de manera jerarquizada. Este árbol de derivación anotado puede utilizarse como "código intermedio", entrada de un proceso de traducción, en este caso a código Maude. El código Maude generado para la expresión conformará a la especificación algebraica de OCL 2.0 en MOMENT. Por tanto, esta estructura intermedia es recorrida para realizar la operación de generación de código Maude. Para ello se ha utilizado el patrón de diseño *Visitor*, que se describe en la sección 3.2.7. Utilizando el mismo patrón se pueden realizar otras operaciones sobre la estructura, como la impresión de un esquema textual de la misma.

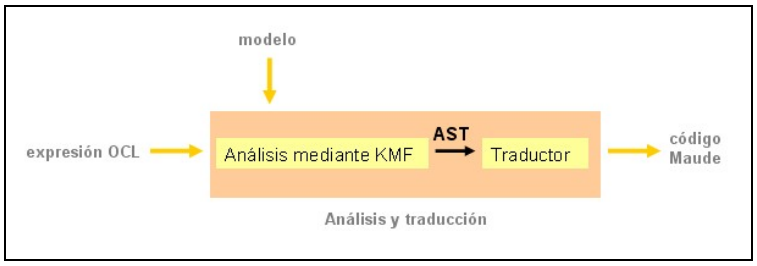

Figura 28. OCLParser. Análisis y traducción de una expresión OCL

3.2.4 Funcionalidad considerada

El proyecto OCLParser ha dado soporte a todas las funcionalidades expresadas como requerimientos en el análisis de requisitos (sección 3.1). Por tanto, no se limita a traducir expresiones OCL a código Maude, sino que además se proporciona soporte para el análisis sintáctico y semántico de las expresiones OCL, y para la visualización de un esquema textual del arbol de derivación asociado a una expresión OCL, resultado de su proceso de análisis. Además, toda expresión traducida para formar parte de la especificación de una transformación en Maude

pasa previamente por un proceso de análisis sintáctico y semántico. Por tanto, solo se pueden definir transformaciones con expresiones OCL sintácticamente y semánticamente correctas, según el estándar OCL 2.0.

3.2.5 Arquitectura y dependencias

El proyecto *OCLParser* se compone de un único plug-in que toma el nombre *es.upv.dsic.issi.moment.oclparser*.

Además, la única dependencia con otro plug-in de la plataforma MOMENT es la dependencia existente con el plug-in que proporciona una consola para visualizar resultados en este framework. Estamos hablando de la consola de MOMENT que corresponde con el plug-in *es.upv.dsic.issi.moment.ui.console*.

En la siguiente figura se puede ver un diagrama con las dependencias existentes entre los diferentes paquetes del plug-in *es.upv.dsic.issi.moment.oclparser*.

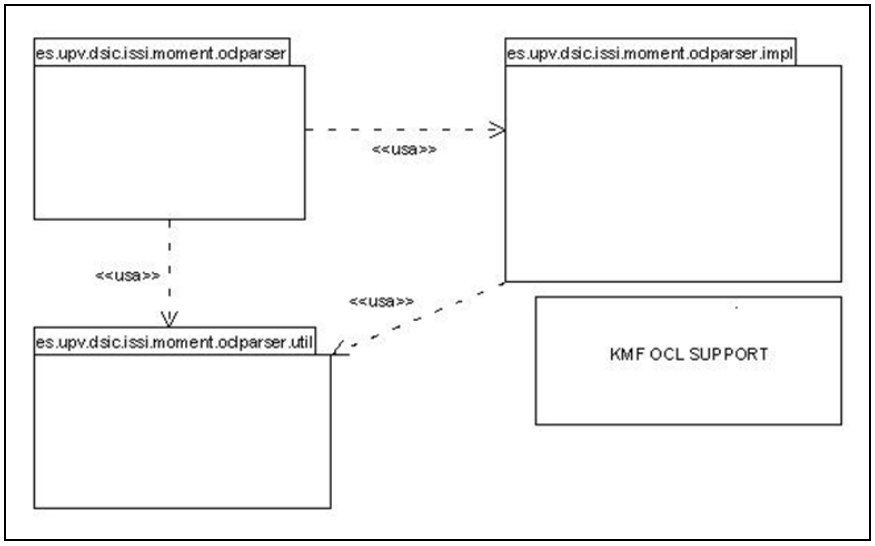

Figura 29. OCLParser. Dependencias entra paquetes y soporte de KMF

El paquete *es.upv.dsic.issi.moment.oclparser* implementa la interfaz del plug-in. El paquete *es.upv.dsic.issi.moment.oclparser.impl* implementa la lógica del proceso de generación de código y proporciona acceso a las funcionalidades de KMF (análisis sintáctico y semántico de expresiones OCL, generación de un esquema textual del Augmented AST y gestión del procesador de OCL para EMF, así como la gestión de los logs de resultados y errores). Finalmente, el paquete *es.upv.dsic.issi.moment.oclparser.util* proporciona una estructura auxiliar para retornar resultados a través de la interfaz de manera estructurada.

Se ha optado por integrar la funcionalidad de la plataforma KMF en el soporte para OCL en MOMENT. Más concretamente, se han integrado las siguientes librerías de KMF dentro del plug-in *es.upv.dsic.issi.moment.oclparser*:

- *CUPRuntime*. Esta librería proporciona el soporte en ejecución del generador de *parsers* para Java CUP.
- *KMF_util*. Una librería con funciones auxiliares y utilidades de KMF.
- *KMFpatterns*. Una librería que da soporte a patrones de diseño.
- *OCLCommon*. Una librería diseñada para soportar funciones de OCL comunes, independientemente del entorno de trabajo (KMF o EMF).
- *OCL4EMF*. Una extensión de *OCLCommon* para EMF.

En el diagrama de dependencias, este conjunto de librerias se agrupan en el denominado *KMF OCL SUPPORT*.

3.2.6 Ejemplo. Proceso de generación de código Maude para transformación. Integración de OCLParser en MOMENT

En este apartado se va a plantear un ejemplo del proceso seguido desde la especificación de una transformación en el lenguaje *QVT Relations* hasta la obtención del código Maude que le da semántica operacional, para mostrar la interacción de estos procesos con el módulo OCLParser.

Se plantea, en la siguiente figura, una de las reglas que especifica la correspondencia entre el metamodelo UML y el metamodelo RDBMS, cuyos alias *ecoreDomain* y *rdbmsDomain* respectivamente. En concreto se trata de la regla que especifica la correspondencia entre una clase y una tabla.

```
top relation ClassToTable
{
   className: String;

   checkonly domain ecoreDomain c: EClass {
      ePackage = p:EPackage {},
      name=className
   };

   enforce domain rdbmsDomain t: Table {
      schema = s:Schema {},
      name = className,
      column = cl:Column {
          name = className + '_tid',
          type = 'NUMBER'
          },
      key = k:Key {
          name = className + '_pk',
          column=cl
      }
   };
}
```

Propiedad "name" anidada en "key"

Figura 30. Regla *ClassToTable*. Código *QVT Relations*

La regla tiene un nombre, *ClassToTable*. Una regla debe tener al menos un dominio origen y un dominio destino. Los dominios se corresponden con al menos uno de los elementos que conforman el metamodelo origen y destino, respectivamente. En el ejemplo, el dominio origen se corresponde con *EClass* y el dominio destino con *Table*. Cada dominio esta formado por una expresión patrón que está formada por un conjunto de propiedades. Para el ejemplo, el dominio origen contiene las propiedaes *ePackage* y *name*, y el dominio destino las propiedades *schema*, *name*, *column y key*.

Cada propiedad puede tener asociado un valor simple (que no hace referencia a otro objeto). Es el caso de la propiedad *name* anidada en la propiedad *key*, que representa el nombre de la columna que va a contener la clave primaria de la tabla. Una propiedad también puede tener asociado un valor estructurado, esto es, una referencia a otro de los objetos definidos en el metamodelo al que pertenece el dominio que estamos tratando. Es el caso de la propiedad *key* que hace referencia al objeto de la clase *Key* del metamodelo RDBMS.

En el caso de que la propiedad tenga asociado un valor simple, dicho valor equivale a una consulta OCL, a evaluar sobre el metamodelo origen o destino. Aunque la sintaxis utilizada en *QVT Relations* para estos valores no corresponde con el estándar OCL, hablaremos de expresiones OCL, ya que existe una correspondencia directa entre estas expresiones en *QVT Relations* y sus equivalentes en OCL.

El programa QVT el cual incluye expresiones OCL para completar su especificación, es traducido a un modelo QVT mediante el módulo *QVT Parser*. El modelo resultado sigue integrando las expresiones OCL, como se puede ver en la siguiente figura:

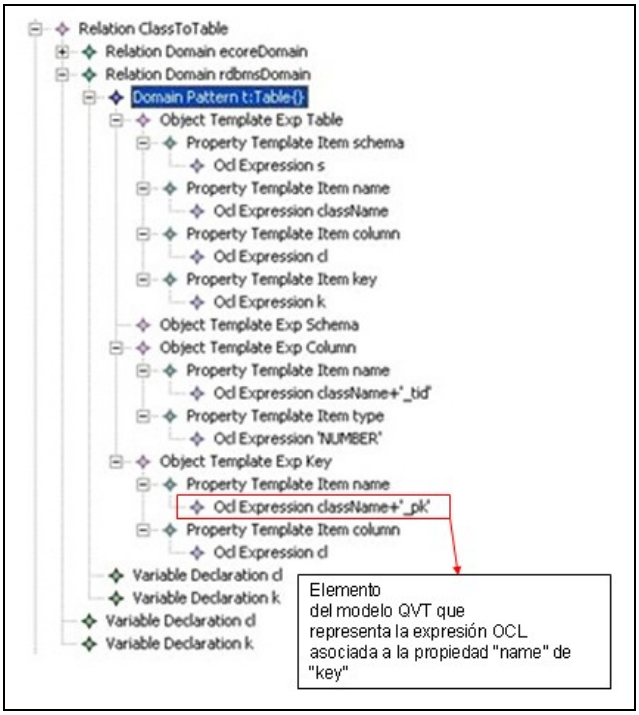

Figura 31. Regla ClassToTable. *Modelo QVT*

Cada propiedad que tenga asociada un valor simple representará finalmente una llamada al componente *OCLParser*, cuando se recorra el modelo QVT que especifica el programa que incluye esta regla, para obtener la especificación en código Maude de la transformación.

En concreto, para el ejemplo de la propiedad *name* anidada en la propiedad *key* se realizará una petición de traducción a código Maude de la siguiente consulta OCL, equivalente a la expresión asociada a la propiedad:

```
let ecoreEClass0:Eclass in c.name.concat('_pk')      (Contexto OCL de la expresión: EClass)
```

Por tanto, el nombre de la clave primaria estará formado por la concatenación del nombre de la clase, del dominio origen, y la cadena *'_pk'*.

El componente OCLParser devolverá la siguiente expresión en código Maude, equivalente a la original en lenguaje OCL:

```
(ecoreEClass0 :: name) + "_pk"
```

Esta expresión conforma la especficación algebraica de OCL 2.0 en Maude de MOMENT. Finalmente es integrada con el código generado para la transformación, como se puede ver en la siguiente figura:

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

Código Maude, derivado de la expresión OCL, integrado en la especificación de la transformación

Figura 32. Regla *ClassToTable*. Código Maude

Hay que tener en cuenta que la expresión OCL a traducir debe ser coherente con el metamodelo sobre el que está definida, en este caso con el metamodelo de UML. En caso contrario no se realiza la traducción. Esto es debido a que OCLParser realiza (con el soporte de KMF) un proceso previo de análisis de la expresión, léxico, sintáctico y semántico (contra el metamodelo considerado) antes de comenzar el proceso de generación de código Maude.

En la siguiente figura se muestra un esquema con todos los pasos descritos en esta sección:

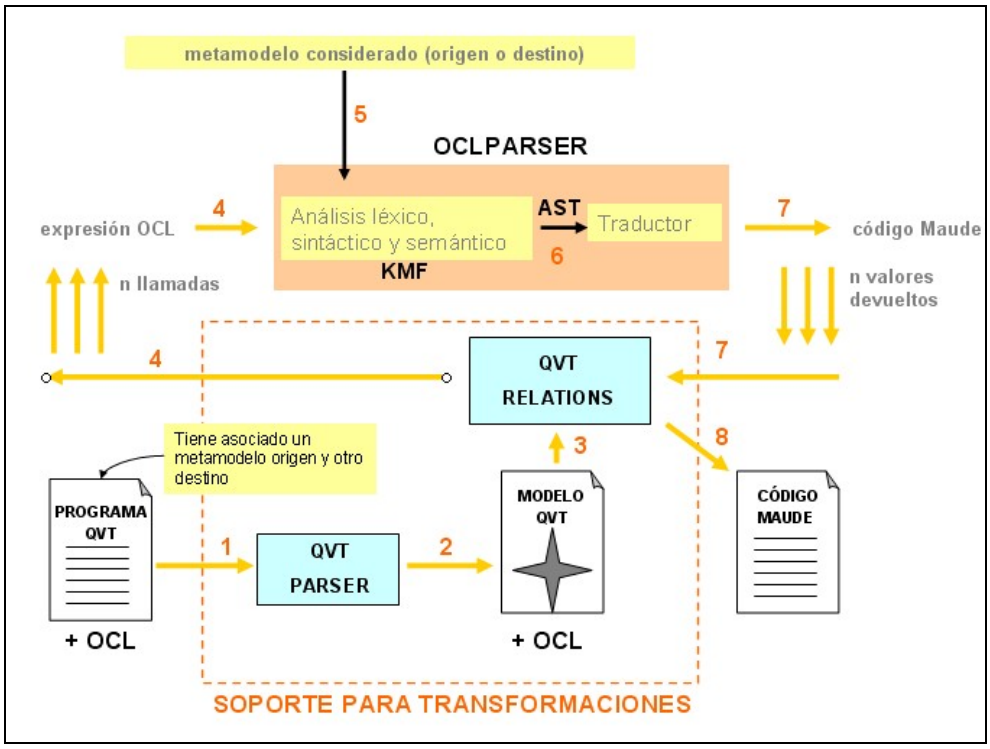

Figura 33. Proceso de generación de código Maude para una transformación

Los números de cada paso corresponden a la siguiente correlación de procesos y acciones:

1. El programa QVT es entrada del componente *QVTParser*.
2. El componente *QVTParser* genera el modelo QVT asociado al programa QVT. Tanto el programa como el modelo tienen asociadas expresiones OCL.
3. El componente *QVTRelations* recorre el modelo QVT en el proceso de generación de código Maude.
4. Para cada expresión OCL que representa una propiedad básica, el componente *QVTRelations* procesa la expresión incluida en el modelo para darle la sintaxis estándar OCL, y la utiliza como entrada para el componente *OCLParser*.
5. La expresión OCL es evaluada sobre el metamodelo origen o destino. Para realizar el proceso de análisis de la expresión, es nececesario cargar la información del metamodelo considerado en el procesador de OCL para EMF de KMF.
6. La expresión es analizada léxica, sintácticamente, y semánticamente (sobre el metamodelo considerado) mediante KMF, produciendo un *Augmented AST*, que da una visión estructurada de la semántica de la expresión.
7. El *Augmented AST* es recorrido y se genera el código Maude que da la semántica operacional a la expresión OCL. Este código Maude es retornado al módulo *QVTRelations*.
8. El código Maude de las expresiones OCL es integrado en el código generado para la transformación.

3.2.7 Patrón de diseño: Visitor

3.2.7.1 Motivación

El proceso de análisis semántico tiene como resultado un *Augmented AST*, instancia del modelo semántico de OCL utilizado por KMF. Partiendo del árbol de derivación, se realizan tareas diferentes sobre sus elementos, como generación de código Java, la impresión de un esquema textual del árbol de derivación, o la generación de código Maude. Se plantea el problema de la integración de un conjunto de operaciones diferentes sobre una estructura de objetos.

En una primera aproximación a la solución, se podría pensar en distribuir las operaciones a través de las clases que representan los elementos. Esta solución es difícil de mantener, al mezclar en las clases de los elementos código correspondiente a diferentes operaciones. Además, añadir una nueva operación supone recompilar todas las clases. En la siguiente figura se presenta un ejemplo de lo que supone esta propuesta. En este caso el lenguaje fuente del compilador es un lenguaje programación, de manera que la estructura del AST contendrá nodos que representen referencias a variables (*VariableRefNode*) o asignaciones de valores a variables (*AssignamentNode*) Cada clase de nodo implementa un conjunto de operaciones, como la comprobación de tipos, generación de código o una impresión de resultados.

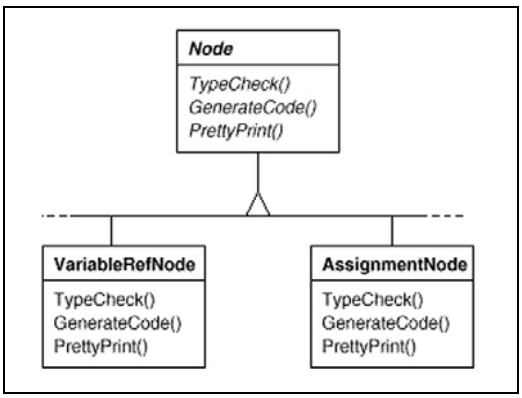

Figura 34. Visitor. Operaciones integradas en las clases de los elementos

El patrón de diseño *Visitor* permite definir operaciones sobre una estructura de objetos sin cambiar las clases de los elementos sobre los que operan [Gam95].

3.2.7.2 Estructura

Mediante esta solución, se empaquetan las operaciones de cada clase en un objeto separado, llamado *visitor*.

Se definen dos jerarquías de clases: una para los elementos sobre los que se opera (la jerarquía *Node*) y otra para los *visitors* que definen las operaciones en los elementos (la jerarquía *NodeVisitor*).

En la jerarquía de clases *visitor* existe una clase abstracta padre (*NodeVisitor*). Crear una nueva operación supone añadir una nueva subclase a esta

jerarquía. Si la gramática que el compilador acepta no cambia (esto es, no hay que añadir una nueva subclase a la jerarquía *Node*), se puede añadir una nueva funcionalidad simplemente añadiendo nuevas subclases *NodeVisitor*. De esta manera, este patrón de diseño encapsula las operaciones de cada fase de la compilación en un *visitor* asociado con dicha fase.

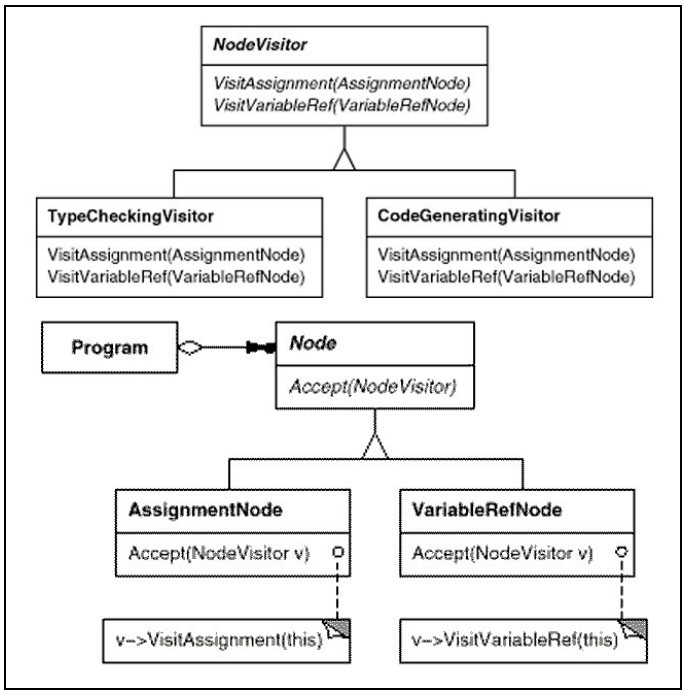

Figura 35. Visitor. Ejemplo

3.2.7.3 Funcionamiento

Veamos el procedimiento para ejecutar una operación sobre un elemento del AST.

Cada elemento del AST posee un método para "aceptar" (*accept*) la utilización de un *visitor*. De hecho, el mismo *visitor* es el argumento del método. Al ejecutarse se realiza una llamada a un método del *visitor*, específico para el tipo de elemento del AST. Como argumento se le pasa el mismo elemento, de manera que su información estará disponible al realizar la operación. Por último, hay que considerar que este proceso se suele anidar, lo cual se consigue mediante la obtención de otro elemento de la estructura partir del inicial y la aceptación del mismo *visitor*, todo esto dentro del método del *visitor*. Mediante esto se consigue recorrer la estructura de objetos realizando una operación específica, lo cual se muestra útil en tareas como la generación de código.

Veamos un pequeño ejemplo de anidamiento de llamadas. Consideremos un modelo con dos clases, *NodeA* y *NodeB*. La clase *NodeA* tiene una asociación con la clase *NodeB* de multiplicidad uno. Consideremos un *visitor*, definido para el modelo, llamado *GenericVisitor*. Un nodo de la clase *NodeA* es *"visitado"* (mediante *GenericVisitor*), y a partir de esta llamada se accede a otro nodo de la jerarquía de objetos de clase *NodeB*, el cual acepta el mismo "*visitador*".

```
NodeA n := …;
// Un nodo acepta el visitor
n.accept(new GenericVisitor());

…

class NodeA
{
        accept(GenericVisitor v)
        {
                //
                v.visitNodeA(this);
        }
}

…

class GenericVisitor
{
        // Al "visitar" un nodo…
        visitNodeA(NodeA n1)
        {
                //…accedemos a otro nodo de la jerarquía…
                NodeB n2 = n1.getB();

                //...que acepta el mismo visitor…
                n2.accept(this);
        }

        //…anidando una llamada a otro método específico para su clase
        visitNodeB(NodeB n2)
        {
                …
        }
}
```

3.2.7.4 Aplicación al proyecto. Interfaz Java implementada: *SemanticsVisitor*

En el contexto de OCLParser el objetivo es, dada una expresión OCL, recorrer el Augmented AST, resultante del análisis semántico, para generar el código Maude que exprese la semántica operacional de la expresión. Por tanto, se ha añadido una nueva operación de generación de código a esta estructura de objetos.

KMF define dos interfaces Java mediante las clases *SyntaxVisitor* y *SemanticsVisitor*. Al implementar sus métodos se permite la definición de una nueva operación sobre los elementos de un AST (resultante del análisis sintáctico), o de un Augmented AST (resultante del análisis semántico) respectivamente. En este caso se ha implementado la interfaz *SemanticsVisitor*.

En el anexo 3 de este documento se muestra el código Java de la interfaz *SemanticsVisitor* de KMF con los métodos que se han implementado. Cada método

corresponde con las acciones a realizar cuando se "*visita*" un elemento de una clase determinada del modelo de OCL 2.0 en el transcurso de la operación que se quiere implementar utilizando la interfaz.

3.2.8 Flujos de información

El anidamiento de llamadas, siguiendo la filosofía del patrón de diseño *Visitor* permite un recorrido por una estructura de objetos, en este caso un Augmented AST, realizando una operación sobre sus elementos. Si tenemos en cuenta tanto las llamadas a los métodos, con el paso de argumentos, como el retorno de valores al finalizar los métodos, podemos hablar de dos flujos de información sobre la estructura de objetos, uno descendente y otro ascendente respectivamente.

Mediante el flujo descendente se proporciona la información inicial del punto de análisis a considerar. Un método de la clase *Visitor* espera recibir el elemento sobre el cual va a realizar la operación. Esto es lo que especifica el patrón de diseño. No obstante, se ha añadido más información para el análisis mediante un segundo argumento. Se trata del nombre de la clase del nodo "*padre*" del elemento (nodo del árbol, en definitiva) que se está considerando. Como la información no se reutiliza en cada una de las llamadas, se puede hablar de un flujo descendente discontinuo, o puntual.

Por otra parte, el flujo ascendente supone el retorno de los resultados parciales que formarán, finalmente en la raíz del árbol, el resultado final. En este caso la generación del código Maude asociado a la expresión OCL. El tipo de información a retornar es variada (código generado, listas de variables utilizadas, listas de navegaciones parciales…) y en un futuro deberá ser posible su ampliación, si fuera necesario. Debido a esto se ha optado por un objeto específico, cuya clase se ha denominado en la implementación *StructVisitor*, para almacenar y gestionar los resultados. En este caso, estos objetos son utilizados como valor de retorno en las sucesivas llamadas, acumulando o integrando cada vez más información. Es por lo que se puede hablar de un flujo ascendente continuo o acumulativo.

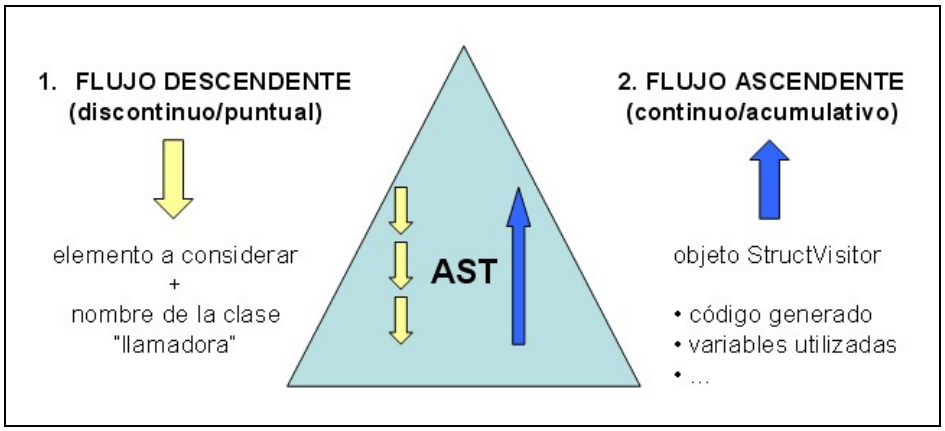

Figura 36. OCLParser. Flujos de información

3.2.9 Repositorios de información auxiliares para el análisis y generación de código

En el proceso de análisis y generación de código a partir de una expresión OCL existe cierta información referida a la sintaxis y semántica de OCL y Maude que no varía. Al hablar de Maude hay que incluir la especificación de OCL para Maude que se ha desarrollado en el proyecto MOMENT.

De esta manera, se deben mantener repositorios de información (en forma, por ejemplo, de listas de elementos o tablas *Hash*) que den soporte al análisis y generación de código. Estas estructuras de datos se deben crear para cada análisis y deben ser accesibles en todo punto de éste.

A continuación se presenta una descripción de estas estructuras auxiliares.

3.2.9.1 Clasificación de los patrones sintácticos de las operaciones de tipos básicos y de colección

Se planteó la necesidad de clasificar y relacionar los patrones sintácticos de las operaciones de tipos básicos (*Integer, Real, String, Boolean*) en OCL y Maude, así como de las operaciones de colección en OCL y en la especificación algebraica de OCL de MOMENT en Maude.

Así, se han creado diferentes listas de nombres de operaciones, según la sintaxis que adoptarán finalmente en la traducción al lenguaje Maude. Estas estructuras son inicializadas al comienzo del proceso de traducción y nunca son modificadas. De esta manera se convierten en estructuras de información auxiliar para el análisis de las expresiones y generación de código.

En el caso de las operaciones de tipos básicos, la sintaxis se conserva salvo en algunas excepciones. En las operaciones sobre colecciones, la especificación algebraica de OCL en MOMENT ha respetado la sintaxis de OCL (salvo en algún detalle), con lo que la traducción es prácticamente directa.

Veamos una descripción de los diferentes grupos de operaciones.

v_collection_ops

Lista que contiene el nombre de todas las operaciones de colección, las cuales siguen el patrón:

```
colección->op_colección(lista_de_argumentos)

{lista_de_argumentos: separados por comas,  puede estar vacía}
```

Este patrón es el mismo en OCL y en la especificación algebraica de de OCL en Maude, salvo cuando la lista de argumentos es vacía, ya que en el segundo caso no se utilizan paréntesis (por ejemplo: *colección->size()* se traduce en *colección->size*).

v_collection_ops_wb

Lista que contiene las operaciones de colección con una lista de argumentos no vacía). Estas operaciones siguen el mismo patrón en OCL y en la especificación algebraica de OCL en Maude:

```
colección->op_colección(arg1, arg2, …)
```

En este grupo se encuentran las operaciones: *includes, excludes, includesAll, excludesAll, union, intersection, symmetricDifference, including, excluding, at, indexOf, insertAt, subSequence, subOrderedSet, append, prepend* y *count*.

v_basic_ops

Lista que contiene las operaciones de tipos básicos (*Integer, Real, Boolean, String*) que siguen el patrón en Maude:

```
elem_a op elem_b
```

Es el caso de las operaciones:

- Booleanas*: and, or, xor,* == (= en OCL)*,* =/= (<> en OCL), *implies*.
- Enteras y reales: >=, <=, >, <, +, -, *, /, *quo* (*div* en OCL).
- Para cadenas de caracteres: + (*concat* en OCL).

v_basic_ops_wb

Lista que contiene las operaciones de tipos básicos (*Integer, Real, Boolean, String*) que siguen el patrón en Maude:

```
op(arg1, arg2, …)
```

Es el caso de las operaciones, de tipo entero y real: *max, min y modExp* (*mod* en OCL).

v_basic_ops_wb2

Lista que contiene las operaciones de tipos básicos (*Integer, Real, Boolean, String*) que siguen el patrón en Maude:

```
op(elem)
```

Es el caso de la operación para cadenas de caracteres de OCL *size* que se traduce a *length* en Maude. Es también el caso de las operaciones, para

enteros y reales, *abs, floor* y *round*. Esta última redondea un número real al entero más cercano. Al no existir una operación equivalente en Maude se ha optado por traducirla por *ceiling* que redondea al mayor entero más cercano. También se considera la operación booleana *not*, aún no incluyendo paréntesis.

3.2.9.2 Otras clasificaciones de conceptos de OCL útiles para la traducción

Se han utilizado listas de elementos para almacenar otros conjuntos de valores útiles para la traducción, los cuales nunca varían (se establecen al comienzo del proceso), al tratarse de clasificaciones propias de la semántica de OCL.

Vector v_collection_kinds

Almacena el nombre de los cuatro tipos de colecciones en OCL: *Set* (colección no ordenada, sin elementos repetidos), *OrderedSet* (colección ordenada, sin elementos repetidos), *Bag* (colección no ordenada, con elementos repetidos), *Sequence* (colección ordenada, con elementos repetidos).

Vector v_basic_types

Almacena el nombre de los cuatro tipos básicos de OCL: *Integer*, *Real*, *Boolean* y *String*.

Vector v_iterator_types

Almacena el nombre de los iteradores de OCL: *collect, collectNested, select, reject, forAll, exists, isUnique, one, any, sortedBy, iterate.*

3.2.9.3 Almacenamiento de la información de las variables OCL en tiempo de ejecución

Se planteó el problema de almacenar y gestionar la información referida a las variables declaradas y utilizadas en las expresiones OCL.

La solución planteada es la siguiente: la información de las variables se almacena en dos estructuras de datos (finalmente implementadas como dos tablas *Hash*): una para las variables definidas en expresiones *let* y otra para las variables definidas en operaciones de iteración (*variables iteradoras*).

```
Map <String,Vector> _let_vars = new HashMap <String,Vector>();
Map <String,Vector> _body_vars = new HashMap <String,Vector>();
```

La clave de búsqueda en estas estructuras es el nombre de la variable, y se almacena para cada una de ellas su tipo de datos y la expresión mediante la cual se ha inicializado (en el caso, por ejemplo, de variables definidas mediante *let*).

Esta información es almacenada en tiempo de ejecución, conforme avanza el análisis para la generación de código, y está accesible en todo momento del proceso.

3.2.10 Generación de identificadores únicos

La especificación de una tranformación escrita en QVT Relations puede estar formada por decenas de expresiones OCL que hay que traducir en sucesivas llamadas al plug-in de *OCLParser*. El nombre de las variables, así como el nombre de las diferentes operaciones generadas en Maude deben ser únicos al generar código para una transformación.

La solución adoptada ha sido añadir a estos nombres un número como sufijo. El último número utilizado se mantiene en sendos atributos de la clase OclParser del paquete *es.upv.dsic.issi.moment.oclparser.impl*:

- *function_number*: para el nombre de operaciones en Maude.
- *variable_number*: para el nombre de variables en Maude.

Estos atributos son accedidos a través de métodos que en cada invocación incrementan su valor. De esta manera se asegura la propiedad de unicidad.

3.2.11 Estrategia para el paso de variables en el anidamiento de operaciones de iteración

3.2.11.1 Motivación

Se propone la siguiente expresión correspondiente a un invariante definido sobre el modelo Royal And Loyal [R&L]:

```
context Customer inv:
self.cards->forAll( c:CustomerCard | self.title.concat(self.name) = c.printedName)
```

Como se puede observar, la variable *self* (remarcada en gris) se utiliza dentro del cuerpo de la operación iteradora *forAll*.

El código Maude que da la semántica operacional al cuerpo de la expresión iteradora y al cuerpo del invariante son las siguientes dos operaciones:

```
// Cuerpo del invariante
op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
((self::0 :: cards ( royalModel )) -> (forAll).BoolFun{royal} (inv::body0 ; ? self::0 ; royalModel ))
if self::0 :: royal-Customer .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .

// Cuerpo de la operación iteradora
op inv::body0 : -> BoolBody{royal} [ctor].
ceq c::1 :: inv::body0 ( ? self::0 ; royalModel ) =
(((self::0 :: title) + (self::0 :: name)) == (c::1 :: printedName))
if c::1 :: royal-CustomerCard .
eq c::1 :: inv::body0 ( PL ; royalModel ) = false [owise].
```

En gris se remarca el paso del parámetro *self* entre las dos operaciones. Este paso de parámetros es la consecuencia de la utilización de una variable en el cuerpo de una operación de iteración anidada.

3.2.11.2 Estrategia de resolución

Cada operación (que representa el cuerpo de una operación de iteración) debe recibir como parámetros las variables que se utilizan su cuerpo, así como las variables que se utilizan en operaciones anidadas.

Para ello la estrategia seguida es la siguiente:

1. El método visitador de cada operación de iteración recibe una lista de variables utilizadas por las operaciones de iteración que tiene anidadas en su cuerpo.

2. Al generar el código para la operación iteradora, se incluyen como parámetros propios las variables recibidas en la lista menos sus propias variables iteradoras.

3. Por último, se incluye sus variables iteradoras en la lista de variables, que es retornada junto con el resto de la información generada en el proceso de análisis.

3.3 Implementación. Descripción de paquetes y clases

A continuación se presenta una descripción de los paquetes y clases del proyecto OCLParser.

3.3.1 es.upv.dsic.issi.moment.ocl.parser

En este paquete se definen los métodos que forman la interfaz del plug-in.

3.3.1.1 Clases

i. OclParserPlugin

Esta clase es generada automáticamente mediante el asistente para crear un nuevo plug-in para Eclipse. Extiende de la clase *org.eclipse.ui.plugin.AbstractUIPlugin*. En esta clase se define los métodos que forman la interfaz del plug-in. A continuación se destacan los atributos más significativos de la clase y se realiza una descripción de estos métodos.

i.i Atributos

- *private static OclParser ocl_ parser*

  La instancia compartida de la clase OclParser del paquete *es.upv.dsic.issi.moment.ocl.parser.impl*, que es donde se redirigen las llamadas a la interfaz.

- *private static OclParserPlugin plugin*

  La instancia compartida del plug-in (representa el propio plug-in).

- *"Alias" para identificar los tipos de expresiones OCL soportados*

  Se han definido variables estáticas asociadas a una representación numérica para especificar los tipos de expresiones OCL que soporta OCLParser.

```
public static final int INVARIANT = 0;
public static final int QUERY = 1;
public static final int QVT_QUERY = 2;
public static final int NONE = 3;
```

Se permite la traducción de invariantes (*INVARIANT*), consultas (*QUERY*) o expresiones OCL utilizadas en la especificación de una transformación o relación de equivalencia sobre modelos en QVT (*QVT_QUERY*).

i.ii Métodos por defecto de Eclipse

- *OclParserPlugin()*

  Este es el método constructor, inicializa el campo "*instancia compartida*".

- *static OclParserPlugin getDefault()*

  Devuelve la instancia compartida.

- *void start(BundleContext context)*

  Este método es llamado al iniciar el plug-in.

- *void stop(BundleContext context)*

  Este método es llamado al detenerse el plug-in.

i.iii Métodos de la interfaz

- *EPackage getEPackage()*

  Este método devuelve el objeto EPackage del último modelo Ecore considerado.

- *OclParserResult parseToMaude(String ocl_expression, String ocl_context, String model_name, EPackage pkg)*

  Este método llama al método *parseToMaude* de la clase *OclParser* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* con l*os mismos argumentos, retornando un objeto del tipo OclParserResult*.

  Mediante este método se obtiene la traducción de una expresión OCL, utilizada en la especificación de una transformación o relación de equivalencia sobre modelos en QVT, a código Maude.

- *OclParserResult parseToMaude(String ocl_expression, String ocl_context, String model_name, Integer option, EPackage pkg)*

  Este método llama al método *parseToMaude* de la clase *OclParser* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* con los mismos argumentos, retornando un objeto del tipo *OclParserResult*.

  Mediante este método se obtiene la traducción de una expresión OCL a código Maude.

  Mediante el atributo *option* se permite la traducción de invariantes, consultas o expresiones OCL utilizadas en la especificación de una transformación o relación de equivalencia sobre modelos en QVT, a código Maude. Se pueden utilizar los "alias", ya descritos, definidos como atributos en esta clase.

- *OclParserResult semanticAnalysis(String ocl_expression, String ocl_context, EPackage pkg)*

  Este método llama al método *semanticAnalisys* de la clase *OclParser* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* con los mismos argumentos, retornando un objeto del tipo *OclParserResult*.

  Mediante este método se obtiene el resultado del análisis léxico, sintáctico y semántico de una expresión OCL, proporcionado por KMF.

- *OclParserResult showAST(String ocl_expression, String ocl_context, EPackage pkg)*

  Este método llama al método *showAST* de la clase *OclParser* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* con los mismos argumentos, retornando un objeto del tipo *OclParserResult*.

  Mediante este método se obtiene una representación textual del Augmented AST, resultado del proceso de análisis de una expresión OCL en KMF.

- *OclParserResult syntaxAnalysis(String ocl_expression, String ocl_context, EPackage pkg)*

  Este método llama al método *syntaxAnalisys* de la clase *OclParser* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* con los mismos argumentos, retornando un objeto del tipo *OclParserResult*.

  Mediante este método se obtiene el resultado del análisis léxico y sintáctico de una expresión OCL, proporcionado por KMF.

3.3.2 es.upv.dsic.issi.moment.ocl.parser.impl

Este paquete es el central del proyecto OCLParser. En él se define un *visitor* para la generación de código Maude (clase *OclVisitor*), una estructura de datos para la gestión de la información recopilada durante el proceso (clase *StructVisitor*) y una clase que recibe y gestiona las llamadas desde los métodos de la interfaz (clase *OclParser*).

3.3.2.1 Clases

i. OclParser

Esta clase es la encargada de recibir las llamadas redirigidas desde los métodos de la interfaz, preparar todo lo necesario para realizar tareas, como la generación de código o el análisis sintáctico y semántico, lanzar dichas tareas y por último devolver los resultados.

i.i Atributos

A continuación se muestra una descripción de los atributos más significativos.

- *static int function_number*

  Número de cuerpos de expresiones en Maude considerados hasta el momento. Se utiliza para obtener identificadores únicos, aún considerando diferentes análisis, al incrementarse su valor en cada acceso a través de métodos específicos.

- *static EPackage model_package*

  EPackage del último modelo Ecore considerado.

- *private String oclELogStr*

  Mediante este atributo se establece la ruta de un archivo de *log* para las excepciones lanzadas desde KMF.

- *private String oclLogStr*

  Mediante este atributo se establece la ruta de un archivo de *log* para los mensajes lanzados desde KMF.

- *OclProcessor processor*

  Instancia del procesador OCL de KMF.

- *static int variable_number*

  Número de variable en Maude consideradas hasta el momento. Se utiliza para obtener identificadores de variables únicos, aún considerando diferentes análisis, al incrementarse su valor en cada acceso a través de métodos específicos.

i.ii Métodos

- *OclParser()*

  Este es el método constructor.

- *OclParserResult parseToMaude(String ocl_expression, String ocl_context, String model_name, Integer option, EPackage pkg)*

  Mediante este método se obtiene la traducción de una expresión OCL a código Maude, utilizando para ello un objeto de la clase OclParserResult.

  Veamos sus atributos:

- *String ocl_expression*: expresión OCL a considerar. No hay que incluir información sobre el contexto, ni hay que especificar en este atributo si se trata de un invariante o una consulta. Una entrada válida para un invariante sobre el modelo Royal and Loyal [R&L] sería:

```
self.age >= 50
```

- *String ocl_context*: contexto OCL a considerar. Siguiendo el ejemplo anterior, un contexto válido sería "*Customer*".

- *String model_name*: nombre del modelo Ecore a considerar en la traducción a código Maude. Si este campo se deja a *null* se tomará el nombre del EPackage del último argumento.

- *Integer option*: codificación numérica del tipo de expresión OCL a considerar. Se pueden utilizar los "alias", ya descritos, definidos como atributos en la clase OclParserPlugin del paquete *es.upv.dsic.issi.moment.ocl.parser*.

- *EPackage pkg*: objeto EPackage del modelo Ecore a considerar en el análisis de la expresión OCL.

- *OclParserResult semanticAnalysis(String ocl_expression, String ocl_context, EPackage pkg)*

  Mediante este método se obtiene el resultado del análisis léxico, sintáctico y semántico de una expresión OCL, proporcionado por KMF, utilizando para ello un objeto de la clase OclParserResult.

  Los argumentos tienen el mismo significado que aquellos con igual nombre del método ParseToMaude.

- *void setErrorMessage()*

  Mediante este método se establece un posible mensaje de error en el análisis para poder ser retornado.

- *void setPackage(EPackage pkg)*

  Este método crea una nueva instancia del procesador OCL para EMF de KMF, para que considere el modelo Ecore definido por el EPackage pasado como argumento.

- *OclParserResult showAST(String ocl_expression, String ocl_context, EPackage pkg)*

  Mediante este método se obtiene una representación textual del Augmented AST, resultado del proceso de análisis de una expresión OCL en KMF, utilizando para ello un objeto de la clase OclParserResult.

  Los argumentos tienen el mismo significado que aquellos con igual nombre del método ParseToMaude.

- *OclParserResult syntaxAnalysis(String ocl_expression, String ocl_context, EPackage pkg)*

  Mediante este método se obtiene el resultado del análisis léxico y sintáctico de una expresión OCL, proporcionado por KMF, utilizando para ello un objeto de la clase OclParserResult.

  Los argumentos tienen el mismo significado que aquellos con igual nombre del método ParseToMaude.

- *void printExcError()*

  Mediante este método se imprime por la consola de MOMENT, como salida de depuración, posibles excepciones lanzadas desde KMF.

- *void redirectOutput()*

  Mediante este método se redirige la salida estándar al archivo de *log* de excepciones lanzadas por KMF.

- *void redirectOutputToOriginal()*

  Mediante este método se reestablece la salida estándar.

ii. StructVisitor

Esta clase proporciona soporte para el almacenamiento y gestión de los resultados parciales del proceso de generación de código, en lo que se ha denominado flujo de información ascendente (ver sección 3.2.8).

ii.i Atributos

- *private String context*

  En este atributo se almacena el nombre del identificador interno de OCLParser que identifica la variable OCL que es el punto de partida de una expresión.

  Por ejemplo, en el punto del proceso de generación de código en el cual se considera el siguiente iterador (para el modelo Royal and Loyal [R&L]):

  ```
  self.partners -> select(p: ProgramPartner | p.numberOfCustomers > 0)
  ```

  Se almacenará en el atributo "*context*" el nombre del identificador interno de la variable *self*. Hay que considerar que, en este punto del análisis, se considera la expresión iteradora como un todo, cuyo punto de partida es esta variable.

- *private String init_expression_var*

  Este atributo almacena el código Maude que da la semántica a una posible expresión OCL utilizada para inicializar la variable cuyo nombre se almacena en el atributo *var*.

- *Vector list_vars*

  Este atributo almacena, en un punto del proceso de generación de código, el conjunto de variables iteradoras que se han detectado. No se almacena su nombre original, sino una variante codificada que proporciona la propiedad de unicidad sobre el resto de variables utilizadas en una transformación. La misión de esta lista de variables es dar soporte al mecanismo de paso de variables en operaciones de iteración anidadas.

- *Vector maude_code*

Este vector almacena el código Maude generado, de manera que la primera posición del vector almacena el código que representa la expresión OCL propiamente dicha, y el resto de posiciones contienen los axiomas que completan su semántica operacional.

Veamos un ejemplo del contenido de este vector al final del proceso de generación de código para la expresión OCL antes referida.

Expresión OCL (para el modelo Royal And Loyal [R&L])

```
context LoyaltyProgram inv : self.partners -> select(p: ProgramPartner |
p.numberOfCustomers > 0) -> size() > 0
```

Vector *maude_code*

```
[Posición 0]
 ((((self::0 :: partners ( royalModel )) -> select ( inv::body0 ; empty-params ; royalModel )) -> size) > 0)

[Posición 1]
op inv::body0 : -> BoolBody{royal} [ctor].
ceq p::1 :: inv::body0 ( PL ; royalModel ) =
((p::1 :: numberOfCustomers) > 0)
if p::1 :: royal-ProgramPartner .
eq p::1 :: inv::body0 ( PL ; royalModel ) = false [owise].
```

- *private String name*

Este atributo está referido al nombre de la clase del último elemento del cual se ha recogido información en el Augmented AST, mediante una instancia de StructVisitor. Este atributo realiza una función similar al flujo de información descendente pero en sentido opuesto (ver sección 3.2.8).

- *Vector navigation*

Este vector almacena los términos de una navegación en OCL.

Por ejemplo, para la navegación OCL *self.partners*, del ejemplo anterior, que devuelve una colección de instancias de la clase *ProgramPartner,* siendo *self* una variable del tipo *LoyaltyProgram*, el vector *navigation* contendrá lo siguiente cuando se procese la información referida a esta subexpresión OCL:

Vector *navigation*

```
[Posición 0]
self

[Posición 1]
partners
```

- *private String var*

  Este atributo almacena el nombre de una variable OCL en el punto del análisis en el que es declarada. No se almacena su nombre original, sino una variante codificada que proporciona la propiedad de unicidad sobre el resto de variables utilizadas en una transformación.

- *private String var_type*

  Este atributo almacena el nombre del tipo de la variable OCL cuyo nombre se ha almacenado en el atributo *var*.

ii.ii Métodos

- *StructVisitor()*

  Este es el método constructor.

- *void addListVars(String str)*

  Añade el nombre de una variable al vector atributo *list_vars* (no se almacenan valores duplicados).

- *void addToNavigation(String str)*

  Método para almacenar un término de una navegación OCL en la última posición del vector atributo *navigation*.

- *void copyListVars(Vector v)*

  Este método fusiona el contenido del vector atributo *list_vars* y el contenido del vector pasado como parámetro. Añade el contenido de las posiciones del vector *v* a las últimas posiciones libres del vector *list_vars*. Equivale por tanto a una llamada al método *addListVars* por cada posición del vector *v*. Mediante este método se recopilan el nombre de las variables iteradoras utilizadas. Esto se realiza para la gestión de estas variables en operaciones de iteración anidadas.

- *void copyOtherCode(Vector v)*

  Este método fusiona el contenido del vector atributo *maude_code* y el contenido del vector pasado como parámetro.

Simplemente añade el contenido de las posiciones del vector *v* a las últimas posiciones libres del vector *maude_code*. Equivale por tanto a una llamada al método *setOtherMaudeCode* por cada posición del vector *v*. Mediante este método se fusionan los resultados parciales del proceso de generación de código, en lo que se ha denominado flujo de información ascendente (sección 3.2.8).

- *void deleteVar(String str)*

  Este método elimina el nombre de una variable iteradora del vector atributo *list_vars* (en el caso de que esté almacenado). Esto se realiza en el proceso de gestión de variables iteradoras en operaciones de iteración anidadas.

- *String getContext()*

  Método para obtener el valor del atributo *context*.

- *String getInitExpressionVar()*

  Método para obtener el valor del atributo *init_expression_var*.

- *String getMaudeCode()*

  Método para obtener la primera posición del vector atributo *maude_code* (se obtiene cadena vacía en el caso de que el vector esté vacío).

- *String getName()*

  Método para obtener el valor del atributo *name*.

- *String getVarName()*

  Método para obtener el valor del atributo *var*.

- *String getVarType()*

  Método para obtener el valor del atributo *var_type*.

- *void refreshListVars()*

  Este método elimina el nombre de las variables iteradoras del vector atributo *list_vars* en el caso de que no se haga referencia a

ellas en el código generado hasta el momento (en el vector atributo *maude_code*). Se trata de una tarea de refresco para no generar variables que nunca vayan a ser referenciadas. Esto puede ocurrir ya que internamente se generan variables auxiliares que en la generación de código pueden ser obviadas para conseguir mayor eficiencia (o simplemente, una mayor legibilidad para facilitar tareas de depuración).

- *void setContext(String str)*

  Método para establecer el valor del atributo *context*.

- *void setInitExpressionVar(String str)*

  Método para establecer el valor del atributo *init_expression_var*.

- *void setMaudeCode(String str)*

  Método para almacenar una cadena con código generado en la primera posición del vector atributo *maude_code*. Se reemplaza el contenido de la primera posición en el caso de que no esté vacía. Este método se utiliza para almacenar el código Maude generado para la expresión OCL, el cual puede utilizar otros axiomas para completar su semántica operacional.

- *void setName(String str)*

  Método para establecer el valor del atributo *name*.

- *void setOtherMaudeCode(String str)*

  Método para almacenar una cadena con código Maude generado en la última posición libre del vector atributo *maude_code*. Este método se utiliza para almacenar el código Maude de los axiomas que completan la semántica operacional de la expresión OCL.

- *void setVarName(String str)*

  Método para establecer el valor del atributo *var*.

- *void setVarType(String str)*

  Método para establecer el valor del atributo *var_type*.

iii. OclVisitor

Esta clase es la principal del proyecto *OCLParser*, ya que es la que implementa la interfaz de KMF *SemanticsVisitor* (ver anexo 3) definiendo la operación de generación de código Maude sobre el Augmented AST. Esta estructura de datos es el resultado del proceso de análisis semántico de una expresión OCL en KMF, y es instancia del modelo semántico de OCL 2.0. Como ya se ha comentado, el modelo semántico es una simplificación del modelo sintáctico de OCL 2.0 (ver sección 2.4.5), ya que está diseñado para facilitar la generación de código o la interpretación a partir de él. Para definir una operación sobre una estructura de datos de manera separada a esta estructura, y además permitir el anidamiento de operaciones sobre una estructura en árbol se recurre al patrón de diseño *Visitor* (este patrón se ha explicado en la sección 3.2.7).

iii.i Ejemplo

Para ilustrar el significado de los atributos y métodos de esta clase se va a utilizar un ejemplo: un invariante OCL referido al modelo Royal And Loyal [R&L].

```
context Customer inv:
self.cards->forAll( c:CustomerCard |  self.title.concat(self.name)

                                                =
                                        c.printedName

                )
```

El significado del invariante es el siguiente: "Todas las tarjetas de un cliente *tendrán como nombre impreso la concatenación del título y el nombre del cliente*".

iii.ii Atributos

- *private String _body_name*

Este atributo referencia al prefijo utilizado para el nombre de los axiomas del código Maude generado.

```
// Se remarca el prefijo de un axioma
// (se ha obviado una parte de  la ecuación del  cuerpo de la operación)

op inv::body0 : -> BoolBody{royal} [ctor].
ceq c::1 :: inv::body0 ( PL ; royalModel ) =

…

if c::1 :: royal-CustomerCard .
eq c::1 :: inv::body0 ( PL ; royalModel ) = false [owise].
```

- *private Map <String,Vector> _body_vars*

  Esta tabla Hash almacena la información referida a las variables declaradas en operaciones de iteración (ver sección 3.2.9.3).

- *private String _context*

  Este atributo almacena el nombre del contexto de la consulta o invariante considerado.

```
// Por ejemplo, para el invariante:

Context Customer inv:
self.cards->forAll( c:CustomerCard |  self.title.concat(self.name) =    c.printedName )

// se almacena el nombre "Customer"
```

- *private String _instance_name*

  Este atributo hace referencia al nombre de la operación que retorna la instancia del modelo (o metamodelo) considerado.

```
// Operación que retorna la instancia
op royal-instance : -> Set{royal} .
eq royal-instance =
Set{
        …
    } .

// Se remarca la utilización de esta variable en la  expresión
// de reducción para la comprobación del invariante ejemplo

red royal-instance -> select ( oclIsTypeOf ; ? "Customer" ; royal-instance ) -> forAll(
inv1 ; empty-params ; royal-instance ) .
```

- *private String _inv_name*

  Este atributo hace referencia al prefijo utilizado para el nombre de la operación que representa el cuerpo de un invariante.

```
//  Se remarca el prefijo de la operación
// (se ha obviado una parte de la ecuación del  cuerpo)

op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
…
if self::0 :: royal-Customer .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .
```

- *private Map <String,Vector> _let_vars*

  Esta tabla Hash almacena la información referida a las variables declaradas en una expresión *let* (ver sección 3.2.9.3).

- *private String _main_function_name*

  Este atributo hace referencia al nombre utilizado para el nombre de la operación que representa el cuerpo de un invariante.

```
//  Se remarca el nombre de la operación
// (se ha obviado una parte de la ecuación del  cuerpo)

op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
…
if self::0 :: royal-Customer .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .
```

- *private String _model_name*

  Este atributo almacena el nombre del modelo (o metamodelo) que se está considerando.

```
// Se remarca el nombre del modelo

op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
…
if self::0 :: royal-Customer .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .
```

- *private String _model_str*

  Este atributo hace referencia a una variable de la especificación en Maude. Esta variable es de tipo *set* de términos referidos a instancias de clases del modelo (o metamodelo).

```
// Para el ejemplo del modelo Royal And Loyal [R&L],
// se hace referencia a la siguiente variable en Maude

var royalModel : Set{royal} .

// Se remarca la utilización de esta variable en la  operación
// que representa el cuerpo del invariante

op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
((self::0 :: cards ( royalModel )) -> (forAll).BoolFun{royal} (inv::body0 ; ? self::0 ;
royalModel ))
if self::0 :: royal-Customer .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .
```

- *private int _option*

  Este atributo almacena el código de opción introducido a través de la interfaz. Mediante esta opción se permite generar código para un invariante, una consulta, o una consulta para una transformación QVT. En la clase *OclParserPlugin* del paquete *es.upv.dsic.issi.moment.ocl.parser.impl* estan definidos los diferentes códigos.

- *private Vector v_basic_ops*

  Lista que contiene el nombre de todas las operaciones de tipos básicos que siguen el patrón en Maude:

  elem_a op elem_b

  (ver sección 3.2.9.1)

- *private Vector v_basic_ops_wb*

  Lista que contiene el nombre de todas las operaciones de tipos básicos que siguen el patrón en Maude:

  op(arg1, arg2, …)

  (ver sección 3.2.9.1)

- *private Vector v_basic_ops_wb2*

  Lista que contiene el nombre de todas las operaciones de tipos básicos que siguen el patrón en Maude:

  op(elem)

  (ver sección 3.2.9.1)

- *private Vector v_basic_types*

  Almacena el nombre de los cuatro tipos básicos de OCL: *Integer*, *Real*, *Boolean* y *String*.

- *private Vector v_collection_kinds*

  Almacena el nombre de los cuatro tipos de colecciones en OCL: *Set* (colección no ordenada, sin elementos repetidos), *OrderedSet* (colección ordenada, sin elementos repetidos), *Bag* (colección no ordenada, con elementos repetidos), *Sequence* (colección ordenada, con elementos repetidos).

- *private Vector v_collection_ops*

  Lista que contiene el nombre de todas las operaciones de colección que siguen el patrón en OCL y Maude:

  colección->op_colección(lista_de_argumentos)

  {lista_de_argumentos: separados por comas, puede estar vacía}

  (ver sección 3.2.9.1)

- *private Vector v_collection_ops_wb*

  Lista que contiene el nombre de todas las operaciones de colección que siguen el patrón en OCL y Maude:

  colección->op_colección(arg_1, arg_2, …)

  (ver sección 3.2.9.1)

- *private Vector v_iterator_types*

  Almacena el nombre de los iteradores de OCL: *collect, collectNested, select, reject, forAll, exists, isUnique, one, any, sortedBy, iterate.*

- *private Vector v_model_classes*

  Este vector almacena el nombre de las clases del modelo (o metamodelo) considerado.

- *private String _tuple_prefix*

  Este atributo hace referencia al prefijo utilizado para el nombre de la clase que se genera dinámicamente cuando se quiere representar y hacer referencia a una tupla OCL. Por defecto toma el valor "*MOMENTOCLTUPLE*".

iii.iii Métodos. Implementación de la interfaz SemanticsVisitor

A continuación se muestra una descripción de los métodos que se han implementado de la interfaz *SemanticsVisitor* (ver anexo 3). Los métodos de la interfaz que no están descritos retornan un valor nulo en la versión actual de *OCLParser*.

Cada método está referido a una clase distinta del modelo semántico de OCL, y se invocará cuando se deba generar código Maude para un objeto (nodo de un Augmented AST) de dicha clase.

Por último, señalar que todos los métodos tienen dos argumentos. El primero es el propio objeto, correspondiente al Augmented AST, para el cual se está realizando la operación de generación de código. El segundo argumento es el nombre de la clase del objeto del Augmented AST en cuyo proceso de generación de código se ha provocado la llamada al método. Dicho de otra manera, se trata del nombre de la clase del nodo padre en el arbol de derivación. Este flujo de información descendente en el arbol de derivación se ha definido en la sección 3.2.8. El único método en el que el segundo argumento no tiene el mismo contenido, ni significado, es en el método referido a la clase *ClassifierContextDecl*. Esto se explica en el siguiente punto.

- *Contextos. Object visit(ClassifierContextDecl host, Object data)*

  Es el primer método de la clase *OclVisitor* que se invoca en el proceso de generación de código Maude, cuando un objeto de la clase *ClassifierContextDecl* acepta el visitador. Esta clase representa una declaración de contexto y puede verse como la raíz del árbol de derivación que se va a recorrer en el proceso de generación de código.

  El proceso de análisis semántico de KMF retorna una lista de objetos de tipo *ClassifierContextDecl*, respondiendo a un conjunto de declaraciones de contexto a considerar. En el proyecto *OCLParser* se ha considerado que esta lista siempre tendrá un único elemento, considerándose las declaraciones de contexto por separado.

  Por tanto, el objeto de tipo *ClassifierContextDecl* retornado por el proceso de análisis semántico de KMF aceptará el visitor de clase *OclVisitor* en el inicio del proceso de generación de código. Es entonces cuando se invocará a este método, el cual incluye los siguientes argumentos: el propio objeto de clase *ClassifierContextDecl* sobre el que se va a ejecutar la operación, y como segundo argumento un vector con la siguiente información:

  - [Posición 0] → Contexto de la expresión OCL (valor del atributo *_context*)
  - [Posición 1] → Codificación numérica definida en la clase del tipo de traducción a realizar (valor del atributo *_option*): invariante, consulta, o consulta para tranformación QVT.[8]
  - [Posición 2] → Valor del atributo *_model_str*.

  El método anida una llamada en la cual el objeto de la clase *OclExpression* acepta el visitor de la clase *OclVisitor*. En el contexto del proyecto *OCLParser* se ha restringido el número de restricciones OCL (clase *Constraint*) que se relacionan con una declaración de contexto (clase *ClassifierContextDecl*) a una. Asimismo, se ha limitado el número de expresiones OCL (clase *OclExpression*) que forman el cuerpo de una restricción a una.

  Se realizan otra serie de funciones:

  - Se inicializan las estructuras de datos auxiliares (esta información será constante en todo el proceso).
  - Si la expresión OCL que se esta analizando es un invariante se comprueba que su cuerpo es de tipo booleano.

[8] Estos códigos se definen en la clase *OclParserPlugin del* paquete *es.upv.dsic.issi.moment.ocl.parser.impl*

- En el caso de tratarse de la generación de código para un invariante, se construye la operación en Maude que representa su cuerpo.

- Se eliminan las declaraciones de variables generadas que finalmente no son referenciadas en el código Maude.

- Se contruye el comando que permitirá la reducción y obtención del resultado de un invariante o consulta.

- Se prepara el retorno del resultado del proceso en un objeto de la clase *OclParserResult*.

- *Variables. Object visit(VariableDeclaration host, Object data)*

Este método se invoca cuando un objeto de clase *VariableDeclaration* acepta el visitador. Esta clase simboliza la declaración de una variable (en una operación de iteración, en una expresión *let*, en la definición de una tupla…).

En este punto se realiza una única tarea básicamente: retornar el nombre (original) y tipo de la variable, y el código Maude que da la semántica operacional de una posible expresión OCL para su inicialización (por tanto, se recoge el objeto de la clase *OclExpression* con el que está relacionado y se visita).

Veamos una representación textual del Augmented AST generado para el invariante del ejemplo para situar la llamada a este método en la jerarquía de llamadas anidadas.

```
// Se remarcan los nodos de la clase VariableDeclaration

Iterator forAll {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
   Variable c:OclModelElementType(CustomerCard)
   null
   OperationCall {
      OperationCall {
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         Operation concat()
```

```
        PropertyCall {
            VariableExp {
                Variable self:OclModelElementType(Customer)
            }
            Property name:String
        }
    }
    Operation =()
    PropertyCall {
        VariableExp {
            Variable c:OclModelElementType(CustomerCard)
        }
        Property printedName:String
    }
  }
}
Type = Boolean
```

Hay que notar que la jerarquía de llamadas a los métodos de la interfaz *SemanticsVisitor* tiene una relación directa con la jerarquía establecida por la estructura del Augmented AST (que a su vez se especifica mediante el modelo semántico de OCL 2.0 definido por KMF).

- *Variables. Object visit(VariableExp host, Object data)*

Este método se invoca cuando un objeto de clase *VariableExp* acepta el visitador. Esta clase simboliza la utilización de una variable en una expresión OCL.

Este método anida la aceptación del visitador por parte del objeto de la clase *VariableDeclaration* con el que se relaciona. Una vez obtenidos los datos de la declaración de la variable (nombre, tipo y código Maude de una posible inicialización) se registran en la tabla auxiliar *_body_vars* mediante una llamada al método *register_body_variable*.

Veamos una representación textual del Augmented AST generado para el invariante del ejemplo.

```
// Se remarcan los nodos de la clase VariableExp

Iterator forAll {
  PropertyCall {
    VariableExp {
      Variable self:OclModelElementType(Customer)
    }
    Property cards:OrderedSet(OclModelElementType(CustomerCard))
  }
  Variable c:OclModelElementType(CustomerCard)
  null
```

```
  OperationCall {
     OperationCall {
        PropertyCall {
           VariableExp {
              Variable self:OclModelElementType(Customer)
           }
           Property title:String
        }
        Operation concat()
        PropertyCall {
           VariableExp {
              Variable self:OclModelElementType(Customer)
           }
           Property name:String
        }
     }
     Operation =()
     PropertyCall {
        VariableExp {
           Variable c:OclModelElementType(CustomerCard)
        }
        Property printedName:String
     }
  }
}
Type = Boolean
```

- *Propiedades. Object visit(Property host, Object data)*

Este método se invoca cuando un objeto de clase *Property* acepta el visitador. Esta clase simboliza el acceso a una propiedad concreta, como por ejemplo el acceso al atributo de una clase o el acceso a una clase a través de una asociación en una navegación. En el segundo caso se añade como argumento, en el código Maude generado, la variable cuyo nombre se almacena en el atributo *_model_str*. Se trata de una variable de tipo *set* de términos, los cuales representan instancias de clases, del modelo (o metamodelo) considerado. La propiedad *owner* de la navegación *self.owner* del ejemplo generada el código *owner(royalModel)*, siendo *royalModel* el nombre de la variable que se almacena en *_model_str*.

Se anida una llamada al método visitador de la clase *Classifier* que retorna el tipo de la propiedad (tipo básico, de colección, clase del modelo o metamodelo, o tupla).

Veamos una representación textual del Augmented AST generado para el invariante del ejemplo.

```
// Se remarcan los nodos de la clase Property

Iterator forAll {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
   Variable c:OclModelElementType(CustomerCard)
   null
   OperationCall {
      OperationCall {
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         Operation concat()
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property name:String
         }
      }
      Operation =()
      PropertyCall {
         VariableExp {
            Variable c:OclModelElementType(CustomerCard)
         }
         Property printedName:String
      }
   }
}
Type = Boolean
```

- *Propiedades. Object visit(PropertyCallExp host, Object data)*

Este método se invoca cuando un objeto de clase *PropertyCallExp* acepta el visitador. Esta clase simboliza el acceso a una propiedad desde una variable o desde una navegación, que atraviesa más de una asociación, con cardinalidad a 1 en todos los extremos de las asociaciones navegadas.

Anida una llamada al método visitador de la clase OclExpression (más concretamente de *VariableExp* o *PropertyCallExp*). Este primer término se denomina origen (*source*) de la expresión.

En segundo lugar, también incluye una llamada al método visitador de la clase *Property*. Este segundo término es la propiedad accedida.

En la estructura de retorno se deja constancia de una posible navegación.

Veamos una representación textual del Augmented AST generado para el invariante del ejemplo.

```
// Se remarcan los nodos de la clase PropertyCallExp

Iterator forAll {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
   Variable c:OclModelElementType(CustomerCard)
   null
   OperationCall {
      OperationCall {
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         Operation concat()
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property name:String
         }
      }
      Operation =()
      PropertyCall {
         VariableExp {
            Variable c:OclModelElementType(CustomerCard)
         }
         Property printedName:String
      }
   }
}
Type = Boolean
```

- *Operaciones básicas y de colección. Object visit(Operation host, Object data)*

Este método se invoca cuando un objeto de clase *Operation* acepta el visitador. Esta clase simboliza un operador concreto (de tipo básico o de colección).

La función principal de este método es retornar el símbolo del operador. En caso de que el símbolo en OCL y en Maude sea diferente se retorna el símbolo para Maude.

```
// Se remarcan los nodos de la clase Operation para el ejemplo

Iterator forAll {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
   Variable c:OclModelElementType(CustomerCard)
   null
   OperationCall {
      OperationCall {
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         Operation concat()
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property name:String
         }
      }
      Operation =()
      PropertyCall {
         VariableExp {
            Variable c:OclModelElementType(CustomerCard)
         }
         Property printedName:String
      }
   }
}
Type = Boolean
```

- *Operaciones básicas y de colección. Object visit(OperationCallExp host, Object data)*

Este método se invoca cuando un objeto de clase *OperationCallExp* acepta el visitador. Esta clase simboliza la llamada a una operación (de tipo básico o de colección).

Este método anida las siguientes llamadas a métodos visitadores:

- Una llamada al método de la clase *Operation*: para retornar el símbolo de la operación en Maude.
- Una llamada al método de la clase *OclExpression*: para retornar el código Maude e información asociada del origen (*source*) de la operación, que se corresponde con el único operando en las operaciones monarias y con el primer operando en las operaciones binarias y de más operandos.
- Cero o más llamadas al método de la clase *OclExpression*: para retornar el código Maude y la información asociada con el resto de operandos, en el caso de que los hubiese.

Las acciones que se realizan en el cuerpo del método se pueden resumir en los siguientes puntos:

1. Obtención del símbolo del operador en Maude.
2. Detección del patrón de traducción utilizado por la operación.
3. Obtención y procesado de la información asociada a los diferentes operandos.
4. Aplicación del patrón de traducción y fusión de la información en una estructura *StructVisitor*.
5. Retorno de la información.

El paso 3 supone la repetición de, prácticamente, las mismas acciones para cada uno de los operandos. Debido a esto, y para una mayor claridad del código, el paso 3 se ha implementado mediante una función auxiliar llamada *analyze_OperationCallExp*.

```
// Se remarcan los nodos de la clase OperationCallExp para el ejemplo

Iterator forAll {
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
   Variable c:OclModelElementType(CustomerCard)
   null
   OperationCall {  // ← OPERACIÓN A (=)

      // OP-A Primer operando
      OperationCall { // ← OPERACIÓN B (concat)

         // OP-B Primer operando
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         // OP-B Fin primer operando

         Operation concat()  // OP-B Operando

         // OP-B Segundo operando
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property name:String
         }
        // OP-B Fin segundo operando
      }
      // OP-A Fin primer operando

      Operation =()  // OP-A Operando

      // OP-A  Segundo operando
      PropertyCall {
         VariableExp {
            Variable c:OclModelElementType(CustomerCard)
         }
         Property printedName:String
      }
      // OP-A  Fin segundo operando
   }
}
Type = Boolean
```

- *Operaciones de iteración. Object visit(IteratorExp host, Object data)*

Este método se invoca cuando un objeto de clase *IteratorExp* acepta el visitador. Esta clase simboliza la llamada a una operación iteradora.

Este método anida las siguientes llamadas a métodos visitadores:

- Una llamada al método de la clase *OclExpression*: para retornar la información referente al origen (*source*) de la operación. Se trata de la colección sobre la que se aplica la operación iteradora.

  En el ejemplo, el origen de la operación iteradora *forAll* es la expresión OCL *self.cards*

- Otra llamada al método de la clase *OclExpression*: para retornar la información referente al cuerpo (*body*) de la operación iteradora.

  En el ejemplo, el cuerpo de la operación iteradora *forAll* es la expresión OCL *self.title.concat(self.name) = c.printedName*.

- Y por último, una o dos llamadas al método de la clase *VariableDeclaration*: para obtener la información referente a la variable iteradora, o de las dos, en el caso específico de una operación de iteración *forAll* con dos variables iteradoras.

  En el ejemplo, la operación de iteración *forAll* solo tiene una variable iteradora ("*c*").

Para el procesado de la información obtenida del origen, cuerpo, y variable o variables iteradoras de la operación iteradora se utiliza el método auxiliar *analyze_IteratorExp*.

```
// Se remarca el nodo de la clase IteratorExp para el ejemplo

Iterator forAll {

  // Origen de la expresión
   PropertyCall {
      VariableExp {
         Variable self:OclModelElementType(Customer)
      }
      Property cards:OrderedSet(OclModelElementType(CustomerCard))
   }
  // Fin del origen de la expresión

   // Variables iteradoras
   Variable c:OclModelElementType(CustomerCard)
   null
   // Fin variables iteradoras

   // Cuerpo de la expresión
   OperationCall {
      OperationCall {
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property title:String
         }
         Operation concat()
         PropertyCall {
            VariableExp {
               Variable self:OclModelElementType(Customer)
            }
            Property name:String
         }
      }
      Operation =()
      PropertyCall {
         VariableExp {
            Variable c:OclModelElementType(CustomerCard)
         }
         Property printedName:String
      }
   }
   // Fin del cuerpo de la expresión
}
Type = Boolean
```

- *Operaciones de iteración. Object visit(IterateExp host, Object data)*

Este método se invoca cuando un objeto de clase *IterateExp* acepta el visitador. Esta clase simboliza la llamada a la operación iteradora *iterate*.

La sintaxis de la operación iteradora *iterate* es la siguiente:

```
colección_origen->iterate(elemento:Tipo1; acumulador:Tipo2 = <expresión>|
                          <cuerpo con el elemento y el acumulador>)
```

Un ejemplo de uso de *iterate* es la siguiente consulta (del modelo Royal And Loyal [R&L]):

```
LoyaltyProgram::allInstances().partners
->iterate(p: ProgramPartner; resultSet: Set(ProgramPartner) = Set{} |
if (p.numberOfCustomers > 0) then resultSet.including(p) else resultSet endif)
```

Mediante esta consulta se obtiene un set con las instancias de la clase ProgramPartner que tienen un valor en el atributo numberOfCustomers mayor que cero.

Este método anida las siguientes llamadas a métodos visitadores:

- Una llamada al método de la clase *OclExpression*: para retornar la información referente al origen (*source*) de la operación. Se trata de la colección sobre la que se aplica la operación iteradora.

  En el ejemplo, el origen de la operación iteradora *iterate* es la expresión OCL *LoyaltyProgram::allInstances().partners*

- Una llamada al método de la clase *VariableDeclaration*: para obtener la información referente a la variable *elemento*.

  En el ejemplo, la variable *elemento* es "*p*".

- Una llamada al método de la clase *VariableDeclaration*: para obtener la información referente a la variable *acumulador*.

  En el ejemplo, la variable *acumulador* es "*resultSet*".

- Otra llamada al método de la clase *OclExpression*: para retornar la información referente al cuerpo (*body*) de la operación iteradora.

> En el ejemplo, el cuerpo de la operación iteradora *iterate* es la expresión OCL *if (p.numberOfCustomers > 0) then resultSet.including(p) else resultSet endif*.

Veamos una representación textual del Augmented AST generado para la consulta del ejemplo.

```
// Se remarca el nodo de la clase IterateExp para el ejemplo

Iterate {

  // Origen de la expresión
  Iterator collect {
    OperationCall {
      null
      Operation allInstances()
    }
    Variable _tempIt3:OclModelElementType(LoyaltyProgram)
    null
    PropertyCall {
      VariableExp {
        Variable _tempIt3:OclModelElementType(LoyaltyProgram)
      }
      Property partners:OrderedSet(OclModelElementType(ProgramPartner))
    }
  }
  // Fin del origen de la expresión

   // Variable elemento
  Variable p:OclModelElementType(ProgramPartner)

   // Variable acumulador
  Variable resultSet:Set(OclModelElementType(ProgramPartner)) =
    Set{
    }

  // Cuerpo de la expresión
  IfExp {
    OperationCall {
      PropertyCall {
        VariableExp {
          Variable p:OclModelElementType(ProgramPartner)
        }
        Property numberOfCustomers:Integer
      }
      Operation >()
      Integer(0)
    }
```

```
      OperationCall {
         VariableExp {
            Variable resultSet:Set(OclModelElementType(ProgramPartner))
         }
         Operation including()
         VariableExp {
            Variable p:OclModelElementType(ProgramPartner)
         }
      }
      VariableExp {
         Variable resultSet:Set(OclModelElementType(ProgramPartner))
      }
   }
   // Fin del cuerpo de la expresión

}
Type = Set(OclModelElementType(ProgramPartner))
```

- *Expresiones Let. Object visit(LetExp host, Object data)*

Este método se invoca cuando un objeto de clase *LetExp* acepta el visitador. Esta clase simboliza una expresión *let*, utilizada para declarar variables que van a ser utilizadas en el cuerpo de un invariante o consulta.

Un ejemplo de uso de let es la siguiente consulta:

```
let m:String = 'Hola' in m.concat(' mundo.')
```

Este método anida una llamada al método visitador de la clase VariableDeclaration. Si se declara más de una variable en el *let* se anidarán las llamadas al método de *LetExp*. Además, se anida una llamada al método visitador de *OclExpression* para recoger la información de la expresión OCL definida en el cuerpo (o "*in*") de la expresión *let*.

La información relativa a la variable declarada (nombre, tipo y código Maude de una posible expresión de inicialización) se registra en la tabla auxiliar *_let_vars* mediante el método *register_let_variable*.

Veamos una representación textual del Augmented AST generado para la consulta del ejemplo.

```
// Se remarca el nodo de la clase LetExp para el ejemplo

LetExp {

   // Declaración de variable
   Variable m:String =
      String(Hola)

   // "in" de la expresión
   OperationCall {
      VariableExp {
         Variable m:String
      }
      Operation concat()
      String( mundo.)
   }
}
Type = String
```

- *Expresiones condicionales. Object visit(IfExp host, Object data)*

Este método se invoca cuando un objeto de clase *IfExp* acepta el visitador. Esta clase simboliza una expresión condicional *if-then-else*, la cual tiene la semántica habitual.

La sintaxis de la expresión es la siguiente:

```
if <expresión OCL booleana>
then < expression OCL>
else < expression OCL>
endif
```

Se anidan tres llamadas al método visitador de la clase *OCLExpression*: una para la expresión booleana, otra para la expresión *then* y otra para la expresión *else*.

En el punto del método visitador para IterateExp aparecía un ejemplo con una expresión condicional. A continuación se muestra parte del Augmented AST remarcando las partes de la expresión condicional.

```
// Expresión condicional
//if (p.numberOfCustomers > 0) then resultSet.including(p) else resultSet endif

// Se remarca el nodo de clase IfExp

IfExp {

   // Condición if
   OperationCall {
      PropertyCall {
         VariableExp {
            Variable p:OclModelElementType(ProgramPartner)
         }
         Property numberOfCustomers:Integer
      }
      Operation >()
      Integer(0)
   }

   // Expresión then
   OperationCall {
      VariableExp {
         Variable resultSet:Set(OclModelElementType(ProgramPartner))
      }
      Operation including()
      VariableExp {
         Variable p:OclModelElementType(ProgramPartner)
      }
   }

   // Expresión else
   VariableExp {
      Variable resultSet:Set(OclModelElementType(ProgramPartner))
   }
}
```

- *Colecciones. Object visit(CollectionKind host, Object data)*

  Este método se invoca cuando un objeto de clase *CollectionKind* acepta el visitador. Esta clase simboliza los tipos de colección (*set, orderedSet, bag* o *sequence*). El método retorna el símbolo del tipo de colección.

- *Colecciones. Object visit(CollectionItem host, Object data)*

  Este método se invoca cuando un objeto de clase *CollectionItem* acepta el visitador. Esta clase simboliza un elemento de una colección de literales.

El método anida una llamada al método visitador de *OclExpression* para obtener la información del elemento.

- *Colecciones. Object visit(CollectionLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase *CollectionLiteralExp* acepta el visitador. Esta clase simboliza una expresión colección de literales.

Se realizan las siguientes llamadas a métodos visitadores:

- Una al método de la clase *CollectionKind*: para saber el tipo de colección (*set, orderedSet, bag* o *sequence*).
- Cero o más llamadas a la clase *CollectionItem*: para obtener la información de los elementos de la colección.

- *Literales. Object visit(BooleanLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase *BooleanLiteralExp* acepta el visitador. Esta clase simboliza un literal booleano.

El método retorna el símbolo del literal.

- *Literales. Object visit(IntegerLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase *IntegerLiteralExp* acepta el visitador. Esta clase simboliza un literal numérico entero.

El método retorna el símbolo del literal.

- *Literales. Object visit(RealLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase *RealLiteralExp* acepta el visitador. Esta clase simboliza un literal numérico real.

- *Literales. Object visit(StringLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase *StringLiteralExp* acepta el visitador. Esta clase simboliza un literal de tipo cadena de carácteres.

El método retorna el símbolo del literal.

- *Tuplas: Object visit(TupleLiteralExp host, Object data)*

Este método se invoca cuando un objeto de clase Tuple*LiteralExp* acepta el visitador. Esta clase simboliza una expresión tupla de literales.

Veamos un ejemplo (una consulta del modelo Royal And Loyal [R&L]):

```
LoyaltyProgram::allInstances().partners->collect(p:ProgramPartner |
    Tuple {
                partnerName: String = p.name,
                numServices: Integer = p.deliveredServices->size()
            }
  )
```

Esta consulta obtiene un conjunto de tuplas con la información, para un programa, del nombre de los socios y su número de servicios.

En este método se realizan las siguientes acciones:

1. Crear dinámicamente una clase en el modelo o metamodelo considerado (mediante la API reflexiva de EMF) para representar el tipo tupla, con la estructura de campos necesaria. El nombre de la clase que representa la tupla toma un valor único para un conjunto de llamadas sucesivas a OCLParser. De esta manera se imposibilita la aparición de conflictos en el transcurso de una transformación (por ejemplo). Este nombre se obtiene mediante una llamada al método auxiliar *get_tuple_class_name()*.

2. Realizar las llamadas que sean necesarias al método de la clase *VariableDeclaration*, tantas como campos tenga la tupla (cada campo esta nominado con una variable).

3. Procesar la información y generar el código Maude que da la semántica operacional a la expresión tupla.

A continuación se muestra el código Maude generado para la tupla del ejemplo:

```
(New("MOMENTOCLTUPLEroyal4", MM((empty-set).Set{royal}))).royalNode
 :: partnerName <-- p::3 :: name
 :: numServices <-- ((p::3 :: deliveredServices ( royalModel )) -> size)

// La función New permite crear la nueva clase (de nombre
// "MOMENTOCLTUPLEroyal4") en la especificación algebraica de la instancia del
// modelo en Maude dinámicamente.
```

- *Métodos para expresar los tipos de OCL*

Se han codificado los siguientes métodos:

Tipos básicos

Object visit(BooleanType host, Object data)
Object visit(IntegerType host, Object data)
Object visit(RealType host, Object data)
Object visit(StringType host, Object data)

Tipos de colección

Object visit(CollectionType host, Object data)
Object visit(SetType host, Object data)
Object visit(OrderedSetType host, Object data)
Object visit(BagType host, Object data)
Object visit(SequenceType host, Object data)

Otros tipos

Object visit(DataType host, Object data)
Object visit(OclAnyType host, Object data)
Object visit(OclMessageType host, Object data)
Object visit(OclModelElementType host, Object data)
Object visit(PrimitiveType host, Object data)
Object visit(TupleType host, Object data)
Object visit(VoidType host, Object data)

Estos métodos se invocan cuando un objeto de la clase del primer argumento acepta el visitador. Estas clases simbolizan los tipos de OCL.

La única acción que realizan estos métodos es devolver el símbolo de su tipo correspondiente.

iii.iv Métodos auxiliares

A continuación se describen los métodos auxiliares, necesarios para modularizar las acciones más frecuentes en el proceso de generación de código.

- *private void analyze_IteratorExp (StructVisitor return_struct, StructVisitor return_struct_source, StructVisitor return_struct_body, String body_type, String previous_class, Vector v_iterator_vars)*

  Este método analiza la información asociada al origen, cuerpo y variable iteradora (pudiendo tratarse de dos variables en el caso del iterador *forAll*) de una llamada al método visitador de la clase *IteratorCallExp*.

  Recibe como argumentos:

  - Un objeto *StructVisitor* para retornar el resultado del proceso de generación de código.
  - El objeto *StructVisitor* de la expresión OCL origen.
  - El objeto *StructVisitor* de la expresión OCL cuerpo.
  - El valor del nombre del tipo de la expresión OCL cuerpo.
  - El nombre de la clase del nodo padre del objeto de la clase IteratorExp, cuyo método se está ejecutando.
  - Un vector con el nombre de las variables iteradoras.

- *private String analyze_OperationCallExp(StructVisitor source_struct, String op, String type, Boolean is_arg)*

  Este método analiza la información asociada a un operando obtenida desde una llamada al método visitador de la clase *OperationCallExp*.

  Recibe como argumentos:

  - El objeto *StructVisitor*, resultado de la visita del operando.
  - El nombre de la operación para Maude.
  - El nombre del tipo de la expresión OCL que representa el operando.
  - Un valor booleano que indica si se trata del primer operando (*false*) o de cualquier otro (*true*).

  Retorna el código Maude que da la semántica operacional a la expresión OCL que representa al operando.

- *private String build_main_body(String str, String self_var, String type_exp)*

  Este método retorna la operación que representa el cuerpo de una expresión OCL de un invariante. Recibe como argumentos el código generado para la expresión OCL cuerpo (que formará parte de una ecuación de la operación), el nombre de la variable que inicia la expresión OCL y el nombre del tipo de la expresión OCL.

- *private String build_main_call(String self_var)*

  Este método retorna el comando que permite la reducción y comprobación del valor de un invariante para un modelo o metamodelo. Como argumento se le pasa el nombre de la variable que inicia la expresión OCL, cuerpo del invariante.

  Retorna el código del comando generado.

- *private Boolean contains_metamodel_type(String type)*

  Este método devuelve true si la cadena que se le pasa como argumento corresponde al nombre de una clase del modelo o metamodelo considerado. Devuelve false en caso contrario.

- *private String get_body_function_name()*

  Este método retorna la versión codificada del nombre de un axioma. Se conserva la propiedad de unicidad del nombre retornado en llamadas sucesivas.

  El nombre codificado se construye con el patrón:

  ```
  _body_name + get_function_number()
  ```

- *private int get_function_number()*

  Este método retorna un número para construir la codificación del nombre de una operación en Maude. El número se incrementa en cada llamada sucesiva a la función para asegurar la propiedad de unicidad en la codificación (el valor se mantiene en el atributo function_number de la clase OclParser del paquete es.upv.dsic.issi.moment.ocl.parser.impl).

- *private int get_let_nav_size(String code)*

  Este método retorna, para el nombre codificado de una variable declarada en un let, el número de asociaciones atravesadas, en el caso de que la expresión de inicialización sea una navegación.

- *private String get_main_function_name()*

  Este método retorna la versión codificada del nombre de la operación que representa el cuerpo de un invariante. Se conserva la propiedad de unicidad del nombre retornado en llamadas sucesivas.

  El nombre codificado se construye con el patrón:

  ```
  _inv_name + get_function_number()
  ```

- *private void init(Vector ocl_struct)*

  Este método es invocado en el inicio del proceso de generación de código y básicamente realiza las siguientes dos funciones:

- Inicializar las estructuras de datos auxiliares (ver sección 3.2.9).
- Establecer el valor de los atributos de la clase (su valor no variará durante el proceso de generación de código).

Como argumento recibe un vector con la misma estructura que el que recibe el método referido a la clase *ClassifierContextDecl*.

No retorna ningún valor.

- *private String get_original_var_name(String var_name)*

  Este método retorna el valor original de una variable OCL dada su versión codificada.

- *private String get_tuple_class_name()*

  Este método retorna la versión codificada del nombre de la clase generada dinámicamente en el modelo o metamodelo para representar una tupla. Se conserva la propiedad de unicidad del nombre retornado en llamadas sucesivas.

  El nombre codificado se construye con el patrón:

  ```
  _tuple_prefix + _model_name + get_variable_number()
  ```

- *private int get_variable_number()*

  Este método retorna un número para construir la codificación del nombre de una variable en Maude. El número se incrementa en cada llamada sucesiva a la función para asegurar la propiedad de unicidad en la codificación (el valor se mantiene en el atributo variable_number de la clase OclParser del paquete es.upv.dsic.issi.moment.ocl.parser.impl).

- *private String get_var_name(String var_name)*

  Este método retorna la versión codificada del nombre de una variable para conservar la propiedad de unicidad en llamadas sucesivas al método.

El nombre codificado se construye con el patrón:

var_name + "::" + <valor retornado por get_variable_number()>

- *private String parse_type_variable(String var_type)*

  Este método procesa el valor del símbolo de un tipo obtenido en el proceso de análisis para adecuarlo a la generación de código. Devuelve el símbolo del tipo procesado.

- *private String print_body_vars()*

  Este método retorna el código generado para la declaración de las variables insertadas en las tablas *_body_vars* y *_let_vars*.

- *private void refresh_body_vars(String code)*

  Este método recibe como argumento el código Maude generado para una expresión OCL y elimina de la tabla _body_vars aquellas variables que no se utilizan.

  No retorna ningún valor.

- *private String register_body_variable(String var_name, String var_type, String var_init_exp)*

  Este método registra la información relativa a una variable utilizada en el cuerpo de una expresión OCL en la tabla *_body_vars* (se almacena el nombre, tipo, y código Maude relativo a una posible expresión OCL de inicialización). El nombre original de la variable se codifica de manera que posea la propiedad de unicidad para un conjunto de llamadas sucesivas a *OCLParser*.

  Retorna el valor del nombre de la variable codificado.

- *private void register_let_variable(String var_name, String var_type, String var_init_exp, int length_nav)*

  Este método registra la información relativa a una variable declarada en una expresión *let* en la tabla *_let_vars* (se almacena el nombre, tipo, el código Maude relativo a una posible expresión OCL de inicialización, y el número de asociaciones atravesadas en el caso

de tratarse el tercer atributo de una navegación). El nombre original de la variable se codifica de manera que posea la propiedad de unicidad para un conjunto de llamadas sucesivas a *OCLParser*.

Retorna el valor del nombre de la variable codificado.

3.3.3 es.upv.dsic.issi.moment.ocl.parser.util

Este paquete almacena las clases auxiliares del proyecto OCLParser, en este caso una sola clase, *OclParserResult*, que se presenta a continuación.

3.3.3.1 Clases

i. OclParserResult

Esta clase tiene una misión muy específica: especificar un objeto para devolver el resultado del análisis y generación de código de una expresión OCL de manera estructurada. Los objetos de esta clase se utilizan para devolver el resultado de las siguientes operaciones a traves de la interfaz de OCLParser (referidas a métodos de la clase *OclParserPlugin* del paquete *es.upv.dsic.issi.moment.ocl.parser*):

- Análisis sintáctico (método *syntaxAnalysis*)
- análisis semántico (método *semanticAnalysis*)
- generación de código (método *parseToMaude*)
- y generación de un esquema textual de la estructura del *Augmented AST*, asociado a una expresión OCL (método *showAST*).

ii.i Ejemplo

Para ilustrar el significado de los atributos de esta clase se va a utilizar un ejemplo. En concreto, un invariante OCL referido al modelo Royal And Loyal [R&L].

```
Context LoyaltyProgram inv:
self.partners.deliveredServices->forAll(s:Service |  (s.pointsEarned = 0 and s.pointsBurned = 0)

                                                                                    implies

                                                                (self.Membership.account->isEmpty())
                                                )
```

El significado del invariante es el siguiente: "*Si todos los servicios de todos los socios de un programa no tienen asociado ningun tipo de puntos las cuentas de puntos de sus clientes estarán vacías*".

ii.ii Atributos

- *private String ast*

  Este atributo almacena una representación textual del *Augmented AST* derivado del análisis semántico de la expresión OCL considerada.

- *private String error_message*

  Atributo que almacena un posible error en la operación solicitada. En cado de que no se haya dado ningún error almacenará la cadena vacía.

- *private Boolean has_errors*

  Este atributo indica si se ha dado algún error en la operación solicitada. En caso de que tome valor de verdad, el atributo *error_message* almacenará un mensaje descriptivo del error.

- *private Object ocl_object*

  Este atributo almacena un objeto de una subclase de *OCLExpression* del metamodelo de *OCLEditor*. Se trata de una referencia auxiliar para el tratamiento y visualización de información en la interfaz del proyecto *OCLEditor*.

- *private String parsed_axioms_result*

  Este atributo almacena el código Maude que da la semántica operacional a los axiomas que son necesarios para completar la semántica operacional de la expresión OCL.

```
// Para el ejemplo:

op inv::body0 : -> BoolBody{royal} [ctor].
ceq s::2 :: inv::body0 ( ? self::0 ; royalModel ) =
((((s::2 :: pointsEarned) == 0) and ((s::2 :: pointsBurned) == 0))
implies ((self::0 :: membership ( royalModel ) :: account ( royalModel )) -> isEmpty))
if s::2 :: royal-Service .
eq s::2 :: inv::body0 ( PL ; royalModel ) = false [owise].

// La operación iteradora forAll retorna un valor booleano.
```

- *private String parsed_expression_result*

  Este atributo almacena el código Maude que da la semántica operacional a la expresión OCL que forma el cuerpo de un invariante o consulta OCL[9].

```
// Para el ejemplo:

op inv1 : -> BoolBody{royal} [ctor] .
ceq self::0 :: inv1 ( PL ; royalModel ) =
((self::0 :: partners ( royalModel ) :: deliveredServices ( royalModel )) ->
(forAll).BoolFun{royal} (inv::body0 ; ? self::0 ; royalModel ))
if self::0 :: royal-LoyaltyProgram .
eq self::0 :: inv1 ( PL ; royalModel ) = false [owise] .

// Este es un ejemplo de paso de variables. La variable self::0 se utiliza en el cuerpo del
// iterador forAll y por ello se pasa como parámetro.

// Al tratarse de un invariante la operación devuelve un valor booleano.
```

- *private String parsed_main_execution*

  Este atributo almacena el comando que es necesario utilizar en el entorno Maude para ejecutar la reducción del invariante o consulta OCL considerada.

```
// Para el ejemplo:

red royal-instance
-> select ( oclIsTypeOf ; ? "LoyaltyProgram" ; royal-instance )
 -> forAll( inv1 ; empty-params ; royal-instance ) .

// Estrategia de ejecución para este invariante:

// A partir de la instancia del modelo, se seleccionan las instancias de la clase
// LoyaltyProgram (mediante el operador oclIsTypeOf).
// Para todas ellas se deberá cumplir el invariante cuya semántica
// se ha codificado y almacenado en los atributos parsed_expression_result,
// parsed_axioms_result y parsed_variables_result.
```

- *private String parsed_variables_result*

  En este atributo se almacena la declaración de las variables utilizadas en la expresión OCL.

[9] Un invariante o una consulta OCL esta formado, básicamente, por un contexto, una lista de declaraciones de variables y una expresión OCL que forma su cuerpo.

```
// Para el ejemplo se generan las siguientes variables:

var s::2 : royal-Service .  // variable iteradora
var self::0 : royalNode .   // variable self ( tipo: instancia de clase del modelo)
```

ii.iii Relación entre los atributos de OclParserResult y los métodos de la interfaz de OCLParser

En la siguiente tabla se muestra que atributos tomarán valor en todos los casos (marcados con una cruz) y cuales tomarán valor dependiendo de cada caso (marcados con la causa). Un atributo sin valor almacena la cadena vacía.

Los cuatro métodos de la interfaz, como ya se ha comentado, están codificados en la clase *OclParserPlugin* del paquete *es.upv.dsic.issi.moment.ocl.parser*.

	parseToMaude	**syntaxAnalysis**	**semanticAnalysis**	**showAST**
parsed_variables_result	(sólo si se necesitan)			
parsed_expression_result	**X**			
parsed_axioms_result	(sólo si se necesitan)			
parsed_main_execution	**X**			
error_message	(sólo si se ha dado un error)	(sólo si se ha dado un error)	(sólo si se ha dado un error)	(sólo si se ha dado un error)
has_errors	true (error_message con valor) / false (en caso contrario)	true (error_message con valor) / false (en caso contrario)	true (error_message con valor) / false (en caso contrario)	true (error_message con valor) / false (en caso contrario)
ast				**X**

Tabla 6. Relación entre los atributos de *OclParserResult* y los métodos de la interfaz de *OCLParser*

Hay que notar que una expresión OCL es sintácticamente o semánticamente correcta cuando el atributo *has_errors* tiene valor *false* en el objeto de la clase *OclParserResult* retornado por los métodos *syntaxAnalisys* o *semanticAnalisys*, respectivamente.

ii.iv Métodos

- *OclParserResult()*

Este es el método constructor.

- *String getParsedAxioms()*

  Método para obtener el valor del atributo *parsed_axioms_result*.

- *String getErrorMessage()*

  Método para obtener el valor del atributo *error_message*.

- *String getOCLObject()*

  Método para obtener el valor del atributo *ocl_object*.

- *String getParsedExpression()*

  Método para obtener el valor del atributo *parsed_expression_result*.

- *String getParsedMainExecution()*

  Método para obtener el valor del atributo *parsed_main_execution*.

- *String getParsedVariables()*

  Método para obtener el valor del atributo *parsed_variables_result*.

- *void setParsedAxioms(String str)*

  Método para establecer el valor del atributo *parsed_axioms_result*.

- *void setErrorMessage(String str)*

  Método para establecer el valor del atributo *error_message*. Además, inicializa el atributo *has_errors* con el valor *true*.

- *void setOCLObject(Object o)*

  Método para establecer el valor del atributo *ocl_object*.

- *void setParsedExpression(String str)*

  Método para establecer el valor del atributo *parsed_expression_result*.

- *void setParsedMainExecution(String str)*

  Método para establecer el valor del atributo *parsed_main_execution*.

- *void setParsedVariables(String str)*

  Método para establecer el valor del atributo *parsed_variables_result*.

Capítulo 4

OCLEditor. Interfaz para la validación de expresiones OCL sobre modelos

4.1 Análisis de requisitos

Se planteó la posibilidad de aprovechar el esfuerzo invertido en el Soporte OCL de MOMENT para crear además una herramienta dirigida al análisis y comprobación de expresiones OCL definidas sobre modelos o metamodelos. La herramienta se llamaría *OCLEditor*.

Se plantearon las siguientes funcionalidades a implementar:

- Análisis léxico, sintáctico y semántico de invariantes y consultas OCL.
- Validación de un conjunto de invariantes sobre un modelo que conforma un metamodelo, o sobre una instancia específica de un modelo.
- Ejecución de consultas sobre un modelo que conforma un metamodelo, o sobre instancias específicas de un modelo.
- Visualización de un esquema del *Augmented AST* producido en el proceso de traducción.
- Visualización del código Maude generado para la ejecución de invariantes y consultas, el cual proporciona la semántica operacional a estas expresiones, para tareas de depuración.
- Visualización del código Maude generado para consultas OCL utilizadas en la especificación de transformaciones y relaciones de equivalencia sobre modelos en QVT, para tareas de depuración.
- Persistencia de las expresiones OCL utilizadas por el usuario, para su almacenado, reutilización…

Desde un principio se pensó en esta herramienta como adecuada para la realización de métricas (validación de baterías de invariantes y consultas definidas sobre modelos para su posterior tratamiento estadístico).

La *usabilidad*, entendida como facilidad de utilización desde una interfaz sencilla y clara, debía ser un factor importante. De esta manera, y siguiendo la filosofía de MOMENT, la herramienta debía integrarse en forma de un conjunto plug-ins en la plataforma Eclipse, y mostrarse de la manera más intuitiva posible.

4.2 Diseño de la solución

4.2.1 Visión

La herramienta se ha desarrollado integrada en la plataforma Eclipse. Para persistir las expresiones OCL, y siguiendo las propuestas de MDE que se basan en el uso sistemático de modelos en el desarrollo de software, se ha creado un modelo Ecore de los elementos e información a persistir. Aprovechando la persistencia de la información en instancias de un modelo Ecore, se ha reutilizado el código del visor en forma de árbol (*tree viewer*) que proporciona Eclipse para visualizar instancias de este tipo de modelos y crear un editor propio, que implementa la funcionalidad de la herramienta. De esta manera la información se muestra de una

manera jerárquica e intuitiva para usuarios de la plataforma Eclipse y la tecnología EMF.

Por un lado, OCLEditor se comporta de manera análoga al *Sample Ecore Model Editor*, de EMF que permite editar modelos Ecore o sus instancias, mediante un visor en forma de árbol. Los archivos que edita OCLEditor son las instancias del modelo definido para persistir la información de las expresiones OCL. Estos archivos (en formato XMI) deben tener la extensión *mocl* para poder ser abiertos por el editor. Así, la herramienta incorpora un visor en forma de árbol (*tree viewer*) y una vista de propiedades (*properties view*), de aquellos elementos seleccionados en el visor. Esto permite la visualización y edición de los elementos de una instancia.

Por otro lado, la herramienta incorpora opciones específicas (para analizar, ejecutar expresiones…), que pueden ser accedidas bien con un menú de botón derecho encima de cada elemento de la instancia, o bien mediante una barra de herramientas. Además, se dispone de una consola (*console*) para mostrar mensajes y resultados.

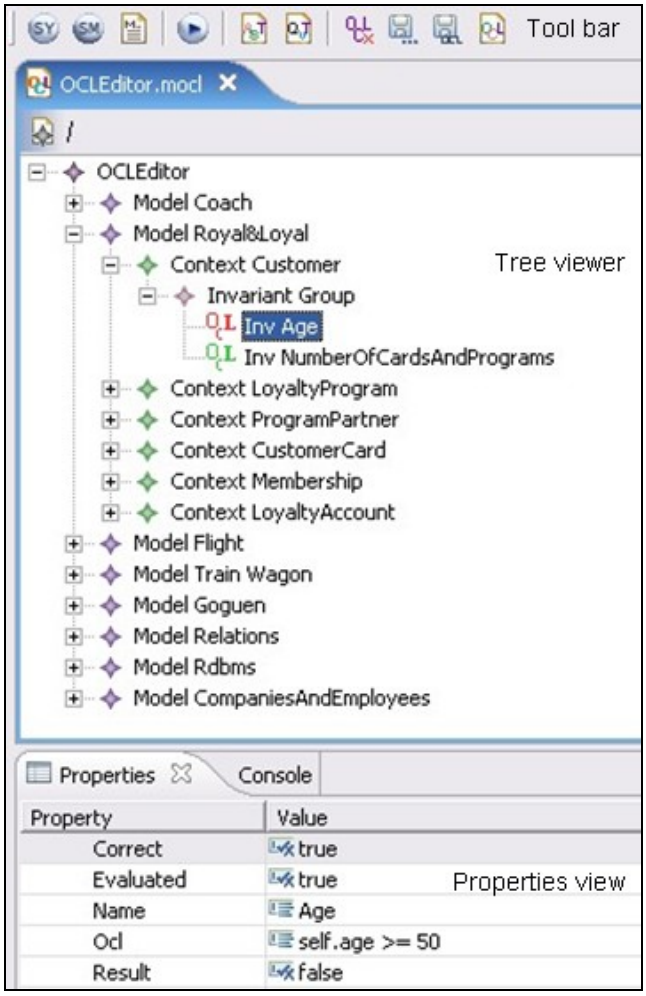

Figura 37. OCLEditor. Partes de la interfaz

4.2.2 Modelo de los elementos a persistir

El modelo que refleja los diferentes elementos e información a persistir es el siguiente. A continuación se muestra, en detalle, el significado de cada clase.

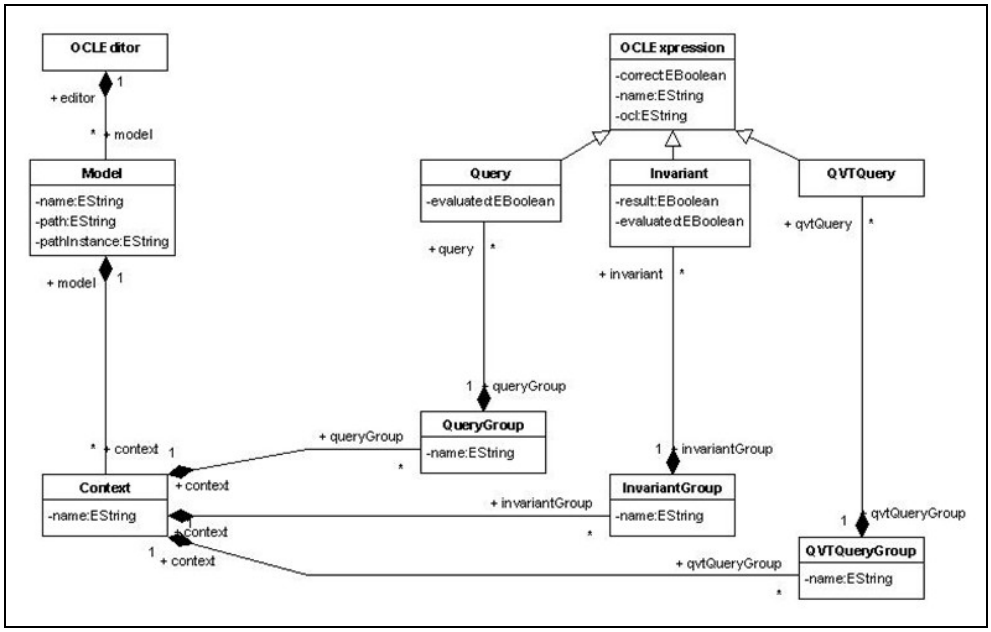

Figura 38. OCLEditor. Modelo de los elementos a persistir

OCLEditor — Esta clase es la raíz del modelo y representa la propia herramienta.

Model — La herramienta puede persistir información sobre expresiones OCL definidas sobre diferentes modelos, que serán evaluadas sobre instancias de estos[10]. De esta manera, la primera agrupación de las expresiones OCL se hace siguiendo el criterio del par modelo/instancia sobre el que van a hacer referencia las expresiones.

Sus atributos son:

- *name* (EString): nombre del modelo.
- *path* (EString): ruta relativa al *workspace* de Eclipse en uso donde se almacena el archivo XMI que describe el modelo.
- *pathInstance* (EString): ruta relativa al *workspace* de Eclipse en uso donde se almacena el archivo XMI que describe la instancia del modelo indicado en el atributo *path*.

[10] Por tanto, la herramienta puede persistir información sobre expresiones OCL definidas sobre modelos que también son metamodelos, que serán evaluadas sobre los modelos que los conforman (que son por tanto sus intancias).

Context	La siguiente agrupación es respecto al contexto utilizado en la expresión OCL. Se almacena en el atributo *name* su nombre.
OCLExpression	Esta clase representa una expresión OCL individual. Sus atributos son: ▪ *correct* (EBoolean): indica si la expresión es sintácticamente y semánticamente correcta. Este indicador también se utiliza para indicar si ha habido algún problema en el proceso de traducción. El usuario no lo debe modificar directamente, sino a través de la herramienta. Su valor por defecto es *true*. ▪ *name* (EString): nombre de la expresión OCL. ▪ *ocl* (EString): expresión OCL a considerar.
Invariant	Esta clase representa un invariante OCL. Hereda de *OCLExpression* y añade los siguientes atributos: ▪ *result* (EBoolean): indica el resultado, cierto o falso, de la evaluación del invariante. Su valor por defecto es *false*. ▪ *evaluated* (EBoolean): indica si el invariante ha sido evaluado, a cierto o falso, con anterioridad. Su valor por defecto es *false*. El usuario no debe modificar directamente estos dos atributos, sino que lo hará a través de la herramienta.
Query	Esta clase representa una consulta OCL. Hereda de *OCLExpression* y añade el atributo evaluated, con el mismo significado que en el caso de la clase Invariant.
QVTQuery	Esta clase representa una consulta OCL utilizada para la especificación de una transformación o relación de equivalencia sobre modelos en QVT. Hereda de *OCLExpression* y no añade atributos.
InvariantGroup	Esta clase representa un grupo de invariantes. Mediante el atributo *name* se le puede asignar un nombre.
QueryGroup	Esta clase representa un grupo de consultas. Mediante el atributo *name* se le puede asignar un nombre.
QVTQueryGroup	Esta clase representa un grupo de consultas utilizadas para la especificación de transformaciones o relaciones de equivalencia sobre modelos en QVT. Mediante el atributo *name* se le puede asignar un nombre.

Hay que observar que una instancia de este modelo puede tener una representación en forma de árbol, donde la raíz es una instancia de la clase *OCLEditor*. Esta representación es la elegida para visualizar las instancias en la herramienta, mediante un *tree viewer*. De esta manera, el árbol es la instancia, con todos sus elementos, un nodo del árbol es una instancia de una clase determinada, y la relación padre-hijo entre nodos en el árbol viene determinada por la relación de agregación entre clases.

4.2.3 Manual de usuario

Se ha desarrollado un manual para la utilización de la herramienta OCLEditor. Este manual de usuario se ha adjuntado en el anexo 4 de este documento.

4.2.4 Arquitectura de plug-ins

A continuación se muestra, mediante un diagrama de componentes, los diferentes plug-ins desarrollados y sus dependencias.

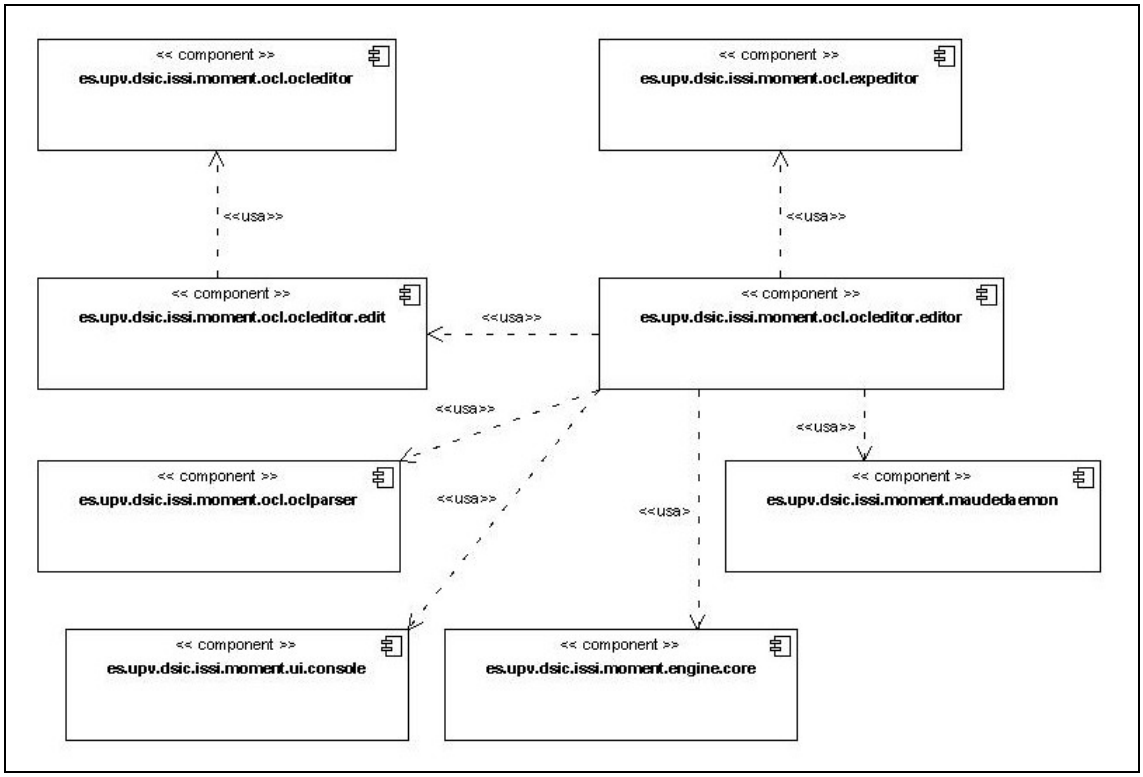

Figura 39. OCLEditor. Arquitectura de plug-ins

Veamos una breve descripción de los plug-ins, que se ampliará en la sección de implementación.

EMF proporciona por defecto una arquitectura en tres plug-ins para la construcción de un editor para instancias de un modelo Ecore:

- *es.upv.dsic.issi.moment.ocl.ocleditor*: proporciona las clases Java (interfaces, implementaciones y utilidades) que especifican el modelo de OCLEditor.

- *es.upv.dsic.issi.moment.ocl.ocleditor.edit*: da soporte para editar instancias del modelo mediante la API reflexiva de EMF.

- *es.upv.dsic.issi.moment.ocl.ocleditor.editor*: da soporte para editar instancias del modelo a través de un editor gráfico que proporciona una visión jerarquizada en forma de árbol. Se ha ampliado su funcionalidad para los propósitos del proyecto de OCLEditor.

Además, se ha desarrollado un editor de archivos de texto con coloreado de sintaxis para OCL. Corresponde al plug-in,

- *es.upv.dsic.issi.moment.ocl.ocleditor.expeditor*

Por último, el plug-in tiene una serie de dependencias con otros plug-ins de del framework MOMENT. Se listan a continuación:

- *es.upv.dsic.issi.moment.ocl.parser*: el proyecto *OCLParser* proporciona soporte a OCLEditor para la traducción de las expresiones OCL a código Maude, análisis sintáctico y semántico de las expresiones y obtención de un esquema textual del Augmented AST resultado del proceso de análisis semántico de una expresión OCL.

- *es.upv.dsic.issi.moment.ui.console*: consola de MOMENT para visualizar resultados.

- *es.upv.dsic.issi.moment.engine.core*: plug-in principal y API principal de MOMENT. Contiene los puentes de conexión de los espacios tecnológicos de EMF y Maude, permite la carga de módulos en Maude, etc.

- *es.upv.dsic.issi.moment.maudedaemon*: integra 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.

4.3 Implementación de OCLEditor. Descripción de paquetes y clases

A continuación se realiza una descripción de los tres paquetes que forman el plug-in *es.upv.dsic.issi.moment.ocl.ocleditor.editor*, las clases que contienen, y sus métodos más interesantes.

4.3.1 es.upv.dsic.issi.moment.ocl.ocleditor.editor

4.3.1.1 Clases

i. OclEditorPlugin

Esta es la clase central del plug-in de *OCLEditor*. Extiende de la clase *org.eclipse.emf.common.EMFPlugin*, y tiene como clase anidada a la clase *Implementation*. Su estructura es básicamente la generada por defecto para un editor EMF.

i.i Métodos por defecto de EMF

- *OclEditorPlugin()*

Este es el método constructor por defecto. Llama al constructor de la clase padre, creando la instancia del plug-in EMF.

- *static OclEditorPlugin.Implementation getPlugin()*

Devuelve la instancia del plug-in de Eclipse.

- *org.eclipse.emf.common.util.ResourceLocator getPluginResourceLocator()*

Devuelve, al igual que el método anterior, la instancia compartida del plug-in.

i.ii Clase anidada. OclEditorPlugin.Implementation

Esta es la implementación del plug-in de Eclipse. Hereda de la clase *org.eclipse.emf.common.EMFPlugin.EclipsePlugin*.

- OclEditorPlugin.Implementation()

Método constructor que crea la instancia compartida del plug-in.

4.3.2 es.upv.dsic.issi.moment.ocl.oclEditor.presentation

Ese paquete incluye los métodos necesarios para la visualización y presentación de contenidos, así como para la gestión de eventos de OCLEditor.

4.3.2.1 Clases

i. OclEditor

Esta es la clase que comprende el editor del modelo de persistencia de OCLEditor. 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, esta clase ha sido modificada para soportar la funcionalidad asociada a OCLEditor.

i.i Métodos por defecto de EMF

- *OclEditor()*

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.

- *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 menú contextual para el visor *viewer*, y añade un listener, registrando este menú.

- *void createModel()*

  Este es el menú que se llama para cargar los recursos (*resources*) en su dominio de edición. Esto es, en este método se carga la instancia del modelo de OCLEditor.

- *void createPages()*

  Este es el método llamado por el framework para crear el visor en forma de árbol que forma el editor. En primer lugar, se realiza una llamada a *createModel()*, para cargar la instancia del modelo OCLEditor, y después se crea el panel para el visor. Posteriormente se realizan las invocaciones a métodos necesarias para configurar las acciones que responden a los eventos, configurar la barra de herramientas y crear el menú contextual. Por último se selecciona un proveedor de etiquetas para el visor de la clase *OclEditorLabelProvider* (contenida en este mismo paquete).

- 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 la instancia del modelo OCLEditor 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 a la instancia del modelo OCLEditor 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* de la instancia del modelo OCLEditor.

- *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 de la instancia del modelo OCLEditor.

- *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 *OclEditorPropertySource*.

- *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 requiere 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*». Este método generado

por defecto por EMF no se utiliza en el contexto de OCLEditor, ya que solamente hay una vista.

- *protected boolean handleDirtyConflict()*

Este método muestra un diálogo que pregunta si los cambios no salvados deben guardarse.

- *protected void hideTabs()*

El método *hideTabs()* oculta las pestañas inferiores del editor multipágina si éste tiene únicamente una. Es el caso de OCLEditor.

- *void init(org.eclipse.ui.IEditorSite site, org.eclipse.ui.IEditorInput editorInput)*

Este método se llama al iniciar el editor. Inicializa el editor, estableciendo su contenido, etc.

- *boolean isDirty()*

Este método comprueba si es existen cambios pendientes de guardar a disco en la instancia del modelo OCLEditor que está siendo editado en un momento determinado.

- *boolean isSaveAsAllowed()*

Este método devuelve siempre cierto, ya que actualmente no se soporta esta característica.

- *void menuAboutToShow(org.eclipse.jface.action.IMenuManager menuManager)*

Este método es el encargado de completar los menús contextuales con las acciones propias de OCLEditor.

- *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, 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.

i.ii Métodos propios de OCLEditor

- *protected void createActions()*

Este método crea acciones asociadas a las diferentes funcionalidades de OCLEditor. Se les asocia un texto y una imagen descriptiva, así como un método a ejecutar cuando sean activadas.

- *protected void createToolbar()*

Este método añade a la barra de herramientas botones con las acciones propias de OCLEditor.

- *private void executeOCLExpression()*

Este método permite la ejecución de un invariante, una consulta, o un conjunto de invariantes. Se ejecutará aquella expresión OCL individual seleccionada en el visor o, en el caso de que se haya seleccionado en el visor un contenedor de expresiones (un metamodelo, o un grupo de invariantes), se ejecutarán aquellas expresiones que dependan de él.

El resultado de la ejecución se visualiza en la consola de MOMENT.

Se realizan, básicamente, las siguientes tareas:

1. Recopilación de información relativa a la expresión o expresiones a ejecutar. Para ello se consulta la instancia del modelo OCLEditor cargada en el visor.

2. Comprobación de que la información es correcta y suficiente para la ejecución.

3. Carga en el registro de metamodelos de EMF del modelo (o metamodelo) sobre el que se van evaluar las expresiones.

4. Traducción de la expresión o expresiones a código Maude. Para esta tarea se invoca el método *parseToMaude* de la interfaz de OCLParser.

5. Si se ha devuelto un error para una expresión en el punto 4, se cambian los flags de estado de la expresión en la instancia del modelo para mostrar un código visual de error.

6. Se lanza un trabajo concurrente (mediante una instancia de la clase *OclEditorLauncherJob* del paquete *es.upv.dsic.issi.moment.ocl.ocleditor.util*) que ejecutará la expresión o expresiones, visualizando mediante la consola de MOMENT el resultado.

- *private void newOCLEditorFile()*

Este método permite crear una nueva instancia del modelo, con parte de la información de la instancia original. Se copian los elementos dependientes de los elementos de la clase "*Model*" seleccionados. Se utiliza un asistente para que el usuario elija el nombre y carpeta contenedora del nuevo archivo de extensión "*mocl*" a crear.

- *private void openExpEditor()*

Este método permite visualizar, mediante un editor textual con coloreado de sintaxis de OCL (cuya implementación se encuentra en el plug-in *es.upv.dsic.issi.moment.ocl.expeditor*), un invariante, una consulta, una consulta para QVT, un grupo de invariantes, un grupo de consultas, un grupo de consultas para QVT, o las expresiones asociadas a un modelo (o metamodelo). La expresión o contenedor de expresiones a considerar debe estar seleccionado. Se utiliza un asistente para que el usuario elija el nombre y carpeta contenedora del nuevo archivo de extensión "*eocl*" a crear.

- *private void parseToMaude()*

Este método permite visualizar por consola el código Maude generado mediante OCLParser para un invariante o consulta seleccionados. En caso de error se muestra por consola un mensaje descriptivo y se modifican los flags del elemento considerado en la instancia del modelo OCLEditor cargada para mostrar un código visual de error.

- *private void parseToMaudeTrans()*

Este método permite visualizar por consola el código Maude generado mediante OCLParser para una consulta para QVT. En caso de error se muestra por consola un mensaje descriptivo y se modifican los flags del

elemento considerado en la instancia del modelo OCLEditor cargada para mostrar un código visual de error.

- *private void semanticAnalysis()*

Este método permite visualizar por consola el resultado del análisis semántico de un invariante, consulta o consulta para QVT. En caso de error se muestra por consola un mensaje descriptivo y se modifican los flags del elemento considerado en la instancia del modelo OCLEditor cargada para mostrar un código visual de error.

- *private void semanticAnalysisForDialog(Object obj)*

Da soporte al análisis semántico automático de una expresión OCL (cuya instancia se pasa como argumento) en el cierre del asistente de introducción de una expresión OCL en la vista de Propiedades (campo Ocl).

- *private void syntaxAnalysis()*

Análogo al método *semanticAnalysis* utilizando un análisis sintáctico.

- *private void showAST()*

Este método permite visualizar por consola un esquema textual del Augmented AST, resultado del proceso de análisis semántico, de un invariante, consulta o consulta para QVT. En caso de error se muestra por consola un mensaje descriptivo y se modifican los flags del elemento considerado en la instancia del modelo OCLEditor cargada para mostrar un código visual de error.

- *private void resetOCLTags()*

Este método permite inicializar los flags de una expresión o conjunto de expresiones a su valor por defecto (*Correct* a *true*, para invariantes, consultas y consultas para QVT, *Evaluated* a *false*, para invariantes y consultas,y *Result* a false, para invariantes).

- *private void viewExp()*

Este método permite visualizar por la consola de MOMENT una expresión OCL o conjunto de ellas, seleccionando una expresión individual o un contenedor de expresiones en el visor.

i.iii Correspondencia entre los botones de la barra de herramientas de la interfaz, acciones y métodos asociados

Los nombres de acciones corresponden a nombres de atributos pertenecientes a la clase *org.eclipse.jface.action.Action*. Veamos el listado de botones de la interfaz y sus correspondencias:

Botones	Acciones	Métodos
Syntax analysis	syntaxAction	syntaxAnalysis
Semantic analysis	semanticAction	semanticAnalysis
ParseToMaude	maudeAction	parseToMaude
Execute invariants/group/OCL	executeMetamodelAction/ executeGroupAction/ executeAction	executeOCLExpression
Show AST	astAction	showAST
Show code for transformation	transAction	parseToMaudeTrans
Reset OCL tags	resetOCLTagsAction	resetOCLTags
Save OCLEditor file	newOCLEditorFileAction	newOCLEditorFile
Save OCL expressions	openExpEditorAction	openExpEditor
View OCL expressions	viewExpAction	viewExp

Tabla 7. OCLEditor. Correspondencia entre botones, acciones y métodos asociados

ii. OclEditorActionBarContributor

Esta clase especifica la contribución que se hace a los menús de Eclipse por parte de OCLEditor. Hereda de la clase *org.eclipse.emf.edit.ui.action.EditingDomainActionBarContributor* e implementa la clase *org.eclipse.jface.viewers.ISelectionChangedListener*.

ii.i Métodos

- *EditingDomainActionBarContributor()*

  Este método es el constructor por defecto.

- *protected void addGlobalActions(org.eclipse.jface.action.IMenuManager menuManager)*

Este método inserta las acciones globales antes del separador «*additions-end*».

- *void contributeToMenu(org.eclipse.jface.action.IMenuManager menuManager)*

Por medio de este método se permite añadir un menú a la barra de menús.

- *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.CreateSiblingAction* 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.

- *protected void populateManager(org.eclipse.jface.action.IContributionManager manager, java.util.Collection actions, String contributionID)*

  Este método puebla 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.

iii. OclEditorDialogExp

Esta clase hereda de *org.eclipse.swt.widgets.Dialog*, especificando la ventana de diálogo que asiste la introducción de una expresión OCL en el campo Ocl de la vista de propiedades del editor.

iii.i Métodos

- *OclEditorDialogExp(Shell parent, String oclExpression)*

  Este método es el constructor, que recibe como argumentos el *shell* gráfico padre y una cadena de texto con la expresión OCL inicial que mostrará el cuadro de texto del asistente.

- *String getResult()*

Este método devuelve el contenido actual del cuadro de texto que contiene el cuadro de diálogo.

- *void open()*

Este método construye los elementos gráficos del cuadro de diálogo y y finalmente permite su visualización.

iv. OclEditorLabelProvider

Esta clase dota de un proveedor de etiquetas personalizado para el visor del editor. De esta manera se pueden elegir las imágenes y etiquetas que se muestran para cada elemento del modelo.

Extende de la clase *org.eclipse.jface.viewers.LavelProvider*.

iv.i Métodos

- *void dispose()*

Mediante este método se limpia la caché de imágenes.

- *org.eclipse.swt.graphics.Image getImage(Object element)*

Este método permite elegir un descriptor de imagen para cada clase del modelo del editor. Un descriptor hace referencia a un archivo gráfico (usualmente de tipo GIF) que se utilizará como icono para una clase de elementos en el visor. Dependiendo de la clase del objeto pasado como argumento, se cargará la imagen referida a su descriptor en una caché, retornándose finalmente la imagen. Esta clase permite, además, visualizar para una expresión OCL un icono u otro, dependiendo de la combinación de flags *Correct*, *Evaluated* y *Result* del objeto de clase "*OCLExpression*" que representa la expresión.

- *String getText(Object element)*

Este método permite elegir una etiqueta para cada clase del modelo del editor. Dependiendo de la clase del objeto pasado como argumento, se devolverá una etiqueta determinada.

- *protected java.lang.RuntimeException unknownElement(Object element)*

Lanza una excepción si se procesa un objeto de una clase no perteneciente al modelo del editor.

v. OclEditorModelWizard

Esta clase se encarga de la gestión del asistente para la creación de una instancia del modelo de OCLEditor. 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 *org.eclipse.ui.INewWizard*.

v.i Métodos

- *void addPages()*

En este método se realiza la creación e inclusión en el asistente de las diversas ventanas de diálogo que lo forman.

- *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(…)*.

v.i Clases anidadas

- *OcleditorModelWizardNewFileCreationPage*

  Especifica la primera página del asistente de selección del nombre del archivo de extensión *"mocl"* y de la carpeta contenedora. Extiende de la clase *org.eclipse.ui.dialogs.WizardNewFileCreationPage*.

- *OcleditorModelWizardInitialObjectCreationPage*

  Extiende de la clase org.eclipse.jface.wizard.WizardPage y especifica la página del asistente donde se selecciona la clase del objeto a crear. Solamente se da opción de crear un objeto de clase "*OCLEditor*" del modelo de OCLEditor, que será la raíz del árbol a visualizar.

vi. OclEditorPropertySource

Esta clase permite personalizar la edición de los campos de la vista de propiedades.

vi.i Métodos

- *OclEditorPropertySource(org.eclipse.emf.common.notify.AdapterFactory adapterFactory, org.eclipse.ui.IEditorSite site)*

  Este es el método constructor. Llama al constructor del padre con el objeto de la clase *AdapterFactory* proporcionado.

- *protected IPropertySource createPropertySource(Object object, org.eclipse.emf.edit.provider.ItemPropertySource itemPropertySource)*

  Este método permite crear un descriptor para especificar un editor para un tipo de propiedad en la vista de propiedades de OCLEditor. Dependiendo del tipo de propiedad se puede especificar un editor. En este caso, a un conjunto de campos se han asociado editores de clase org.eclipse.jface.viewers.DialogCellEditor. Dependiendo del tipo de propiedad se realizan unas acciones al acceder al editor. Las acciones se especifican sobreescribiendo el método *openDialogBox(...)* de DialogCellEditor.

  Se han especificado tres tipos de acciones, dependiendo del tipo de propiedad y de la clase del elemento sobre el que se observa:

- *Propiedades "Path" y "PathInstance" de la clase "Model"*: Se abre un diálogo de clase org.eclipse.ui.dialogs.ElementTreeSelectionDialog para seleccionar un archivo que especifique el modelo (en el caso de la propiedad "*Path*") o instancia (en el caso de la propiedad "*PathInstance*") a considerar.

- *Propiedad "Ocl" de la clase "OCLExpression"*: Se abre una ventana de diálogo de clase *OclEditorDialogExp*, contenida en este mismo paquete, para introducir una expresión OCL. Cuando se cierra el diálogo se llama al método *semanticAnalysisForDialog*(…) de la clase OclEditor, para analizar automáticamente la semántica de la expresión.

- *Propiedad "Name" de la clase "Context":* Esta última, como excepción, utiliza un editor de clase *org.eclipse.jface.viewers.ComboBoxCellEditor*, para utilizar una lista con los nombres de la clase del modelo sobre el que se consideran las expresiones OCL, como soporte para elegir un contexto OCL. Se ha considerado la posibilidad de un modelo Ecore estructurado con objetos de clase *EPackage* anidados.

- *private Vector mergeVectors(Vector v1, Vector v2)*

Este método fusiona el contenido de dos vectores y devuelve el resultado.

- *private Vector getESubPackagesListRecursive(org.eclipse.emf.ecore.EPackage pkg)*

Mediante este método se devuelve una lista con los objetos de tipo *EPackage* anidados, a cualquier nivel, en un objeto de la misma clase.

4.3.3 es.upv.dsic.issi.moment.ocl.ocleditor.util

Ese paquete incluye las clases que permiten la ejecución de expresiones OCL, así como métodos auxiliares de ámbito general.

4.3.3.1 Clases

i. OclEditorLauncherJob

Esta es la clase especifica un trabajo concurrente (extiende de la clase *org.eclipse.core.runtime.jobs.Job*) para la ejecución de un invariante, un conjunto de invariantes o una consulta OCL. La ejecución del trabajo se lanza en paralelo con el hilo que atiende los eventos de la interfaz.

El único método de la interfaz que instancia este trabajo de ejecución es *executeOCLExpression()* de la clase *OclEditor* (contenida en el paquete *es.upv.dsic.issi.moment.ocl.ocleditor.presentation*).

i.i Métodos

- *OclEditorLauncherJob(String name, org.eclipse.core.resources.IFile model_file, es.upv.dsic.issi.moment.engine.core.model.Model mdl, es.upv.dsic.issi.moment.ocl.parser.util.OclParserResult ocl_parser_result, Boolean createContext, String messageHeader)*

  Constructor de la clase. Permite la ejecución de un invariante o consulta OCL individual. Llama al constructor de la clase padre e inicializa atributos.

  Tiene los siguientes argumentos:

  - *String name*: nombre del trabajo. Es pasado en la llamada al constructor a la clase padre.
  - org.eclipse.core.resources.*IFile model_file*: recurso del *workspace* de Eclipse de tipo archivo que hace referencia al archivo de la instancia del modelo sobre la que se evaluan las expresiones OCL. Esta referencia se especifica en la propiedad *pathInstance* del elemento de la clase "*Model*" contenedor de la expresión OCL a evaluar.
  - es.upv.dsic.issi.moment.engine.core.model.*Model mdl*: representación de la instancia referida mediante el atributo anterior. Se trata de una instancia de la clase *"Model"* del modelo del componente *es.upv.dsic.issi.moment.engine.core*, plug-in y API principal de MOMENT.
  - es.upv.dsic.issi.moment.ocl.parser.util.*OclParserResult ocl_parser_result*: estructura que contiene el resultado del proceso de generación de código Maude, obtenida mediante una llamada al método parseToMaude(..) de la interfaz de OCLParser.

- *Boolean createContext*: flag que indica si se ha cargado en una ejecución anterior los módulos en Maude con la especificación algebraica de OCL 2.0. Si ya se ha realizado esta tarea no será necesario realizarla nuevamente.

- *String messageHeader*: Cabecera para los mensajes de resultado de la ejecución.

- *OclEditorLauncherJob(String name, org.eclipse.core.resources.IFile model_file, es.upv.dsic.issi.moment.engine,core.model.Model mdl, Vector oclExpVector, Boolean createContext)*

Constructor de la clase. Permite la ejecución de un conjunto de invariantes. Llama al constructor de la clase padre e inicializa atributos.

En vez vez de un objeto de la clase es.upv.dsic.issi.moment.ocl.parser.util.*OclParserResult*, se pasa como argumento una lista (*oclExpVector*) con un objeto de la clase *OclParserResult* por cada invariante a evaluar.

- *final IStatus run(IProgressMonitor monitor)*

Este método especifica las acciones a realizar cuando se programe mediante el método *schedule*() la ejecución de una tarea concurrente de tipo *OclEditorLauncherJob*.

Como argumento se tiene un monitor de progreso de la tarea de la clase *org.eclipse.core.runtime.IProgressMonitor*. Este monitor se puede visualizar mediante la vista básica *Progress* de Eclipse (Menú *Window>Show View>Other…>Basic>Progress*).

Las acciones básicas de este método son:

1. Se crea un punto de interacción con Maude para permitir la carga de módulos. Esto se realiza mediante la creación de una instancia de la clase *es.upv.dsic.issi.moment.engine. core.model.MaudeContext*. Si no se trata de la primera ejecución de la sesión de trabajo se realiza una actualización del contexto de ejecución.

2. En el caso de no haberse realizado ninguna ejecución anterior, se cargan en Maude los módulos que dan soporte a la especificación algebraica de OCL 2.0 desarrollada en el marco del proyecto MOMENT.

3. Se carga la signatura y la vista[11] del modelo sobre el que se ha definido la expresión o el conjunto de expresiones OCL a evaluar.

4. Se carga la especificación algebraica que representa al modelo. Es lo que se denomina *"módulo SOUP*".

5. Se crea un módulo denominado "*SOUP-OCL"* que incluirá el el término algebraico que representa la instancia del modelo. Este módulo se crea mediante un objeto de la clase *es.upv.dsic.issi.moment.engine.core.model.MaudeModule.*

6. Tambien se introducen en este módulo "*SOUP-OCL"* las declaraciones de variables y axiomas, resultado de la traducción de las expresiones OCL. Este paso se realiza una vez por cada expresión OCL a ejecutar en el mismo trabajo concurrente.

7. Se carga el módulo "SOUP-OCL" en Maude, a través del contexto de ejecución.

8. Se lanza la ejecución de la reducción asociada a cada expresión OCL considerada. Esto se realiza mediante una instancia de la clase *es.upv.dsic.issi.moment.maudedaemon. maude.IMaudeJob*.

9. Se visualiza el resultado de la ejecución para cada expresión OCL a través de la consola de MOMENT. Si se ha producido un error en la ejecución de alguna de las expresiones OCL, se muestra un mensaje textual de error.

10. Se actualizan los flags Correct, Evaluated y Result para cada instancia de la clase OCLExpression del modelo OCLEditor considerada en la ejecución. En este punto, si se ha producido un error, se muestra un código visual de error asociado al elemento involucrado.

11. Se retorna un objeto *org.eclipse.core.runtime.IStatus* que informa de si la operación a finalizado correctamente.

i. OclEditorUtil

Esta clase especifica un conjunto de métodos auxiliares.

[11] La signatura especifica un sort para cada clase, que representa su colección de instancias respectiva, y un constructor para cada una. La vista se utiliza para utilizar la signatura para un modelo como parámetro actual en el módulo *OCL-SUPPORT{X::TRIV}*.

i.i Métodos

- *OclEditorUtil()*

  Constructor de la clase. No tiene asociada ninguna acción.

- *String getContextName(Object o)*

  Devuelve el nombre del objeto de la clase *"Context"* del modelo de OCLEditor asociado a un objeto de la clase *"Invariant"*, *"Query"* o *"QVTQuery"*.

- *es.upv.dsic.issi.moment.ocl.ocleditor.Model getModel(Object o)*

  Devuelve un objeto de la clase *"Model"* del modelo de OCLEditor asociado a un objeto de la clase *"Invariant"*, *"Query"* o *"QVTQuery"*.

- *Vector getOCLExpressions(Object o)*

  Devuelve una lista con los objetos de la clase "*Invariant*" del modelo de OCLEditor asociados a un objeto de la clase *"OCLEditor"*, *"Model"* o *"InvariantGroup"*.

4.4 Implementación de OCLExpEditor. Descripción de paquetes y clases

OCLExpEditor es una editor textual con coloreado de sintaxis para OCL 2.0. Su implementación se ha realizado en el plug-in *es.upv.dsic.issi.moment.ocl.expeditor*. Su código se ha generado automáticamente en Eclipse y se asocia a archivos de texto con extensión *"eocl"*. Se ha indicado el léxico propio de OCL para ser detectado por el mecanismo de coloreado.

4.4.1 es.upv.dsic.issi.moment.ocl.expeditor

4.4.1.1 Clases

i. ExpEditorPlugin

Esta es la clase de inicialización del plug-in *OCLExpEditor*.

i.i Métodos

- *ExpeditorPlugin ()*

  Este es el método constructor, inicializa el campo "*instancia compartida*".

- *static ExpeditorPlugin getDefault()*

  Devuelve la instancia compartida.

- *void start(BundleContext context)*

  Este método es llamado al iniciar el plug-in.

- *void stop(BundleContext context)*

  Este método es llamado al detenerse el plug-in.

4.4.2 es.upv.dsic.issi.moment.expeditor.editor

Este paquete incluye la lógica de OCLExpEditor.

4.4.2.1 Clases

i. ExpEditor

Esta es la clase representa el editor textual. Hereda de la clase *org.eclipse.ui.editors.text.TextEditor*.

i.i Métodos

- *ExpEditor ()*

  Este es el método constructor. Llama al método constructor padre y establece la configuración de visualización.

- *protected void createActions()*

  Crea las acciones asociadas al editor, llamando mismo método de la clase padre.

- *protected void editorContextMenuAboutToShow(org.eclipse.jface.action.IMenuManager menu)*

  Establece el manejador de contribuciones a la barra de menús, llamando al mismo método de la clase padre.

ii. ExpEditorContributor

Esta clase maneja la instalación y desinstalación de las acciones globales para editores con múltiples páginas. Es responsable de la redirección de las acciones globales al editor activo. Hereda de la clase *org.eclipse.ui.part. EditorActionBarContributor*.

ii.i Métodos

- *ExpEditorContributor ()*

  Este es el método constructor. Crea el contribuidor multipágina.

- *protected IAction getAction(org.eclipse.ui.texteditor.ITextEditor editor, String actionID)*

  Devuelve la acción registrada con el editor de texto pasado como parámetro.

- *protected void setActiveEditor(org.eclipse.ui.IEditorPart part)*

  Método declarado en *AbstractMultiPageEditorActionBarContributor*.

iii. ExpRuleScanner

Esta clase especifica los parámetros del detector léxico. Detalla los términos a detectar y los colores a utilizar para resaltar los términos.

iii.i Atributos

- *String[] keyWords*

  Vector que contiene el nombre de las palabras clave a detectar.

```
// Se detectan las siguientes palabras clave del estándar OCL

public static String[] keyWords = { "inv" , "query", "context", "Set", "Bag", "Sequence", "Collection", "Tuple", "TupleType", "OrderedSet", "if", "then", "else", "endif", "let", " in ", "iterate", "implies", "and", "or", "xor", "not", "true", "false", "mod", "div", "real", "integer", "string", "collect", "collectNested", "select", "reject", "forAll", "exists", "one", "isUnique", "any", "sortedBy", "allInstances","asBag", "asSet", "asOrderedSet", "asSequence", "size", "sum"};
```

- org.eclipse.swt.graphics.Color KEY_WORDS_COLOR

  Establece el color a utilizar para las palabras claves.

- org.eclipse.swt.graphics.Color STRING_COLOR

  Establece el color a utilizar para las cadenas de texto, representadas mediante un texto entre comillas simples o dobles.

iii.i Métodos

- *ExpRuleScanner()*

  Este es el método constructor. Se realizan las siguientes tareas:

  - Creación del detector de palabras (instancia de la clase *org.eclipse.jface.text.rules. IWordDetector*).
  - Creación de una regla de detección de palabras (instancia de la clase *org.eclipse.jface.text.rules.WordRule*).

- Se añaden las palabras claves a la regla de detección.
- Se instalan esta regla de detección, junto a reglas para detectar comentarios y cadenas de texto, y una regla para detectar espacios en blanco.

iii. ExpSourceViewerConfig

Especifica la configuración de la visualización del editor de texto. Hereda de *org.eclipse.jface.text.source.SourceViewerConfiguration*

iii.i Métodos

- *ExpSourceViewerConfig ()*

  Este es el método constructor. No realiza ninguna tarea.

- *ExpRuleScanner getTagScanner()*

  Retorna el *ExpRuleScanner* actual, creándolo en el caso de que no se haya definido.

- *org.eclipse.jface.presentation.IPresentationReconciler getPresentationReconciler(org.eclipse.jface.text.source.ISourceViewer sourceViewer)*

  Retorna el reconciliador de presentación para el editor para el caso que haya que reparar errores.

- *(org.eclipse.jface.text.contentassist.IContentAssistant getContentAssistant(org.eclipse.jface.text.source.ISourceViewer sourceViewer)*

  Retorna el asistente de contenidos del editor, que da soporte para completar interactivamente contenidos.

iv. WhitespaceDetector

Especifica el detector de espacios en blanco para el editor de texto. Implementa la clase *org.eclipse.jface.text.rules.IWhitespaceDetector*

iv.i Métodos

- *boolean isWhitespace(char c)*

  Comprueba si el carácter pasado como parámetro es un espacio en blanco, devolviendo *true* en caso afirmativo y *false* en contrario.

Capítulo 5

Conclusiones y trabajos futuros

5.1 Conclusiones

Desde que OMG propuso el lenguaje OCL, se ha tratado de integrar en gran cantidad de herramientas para el desarrollo de software dirigido por modelos. OCL está teniendo una gran aceptación entre los seguidores de las propuestas de MDE, y por extensión de MDA y OMG, convirtiéndose en un lenguaje estándar de facto para la definición de restricciones y consultas sobre modelos. Esto se ha visto refrendado por la utilización de este lenguaje, desde la disciplina de la Gestión de Modelos, en el estándar QVT, el cual permite la especificación de transformaciones y relaciones de equivalencia sobre modelos.

En este trabajo se ha dado soporte para la utilización del lenguaje de consulta y definición de restricciones sobre modelos OCL 2.0 en MOMENT, una herramienta para la gestión de modelos. MOMENT ha conseguido demostrar la falsedad de la poca productividad de los métodos formales, integrando un sistema para la especificación algebraica, como es Maude, en una plataforma industrial, como es Eclipse con la tecnología de EMF. En el ámbito de la definición de transformaciones sobre modelos, MOMENT es una de las primeras herramientas que ofrece soporte para el lenguaje QVT Relations. OCL permite aumentar la productividad en la especificación de los programas escritos en QVT Relations, al proporcionar un lenguaje de alto nivel de abstracción para la consulta de los metamodelos origen y destino de una transformación.

Se ha integrado la funcionalidad de la plataforma KMF, lo cual ha permitido reaprovechar el esfuerzo invertido por parte de la Universidad de Kent en una implementación del estándar OCL 2.0. KMF ha proporcionado el soporte para el análisis léxico, sintáctico y semántico de las expresiones OCL, lo cual permite partir de la base de expresiones OCL correctamente especificadas. Se ha extendido esta plataforma, con el desarrollo de un sistema de traducción de expresiones OCL a código Maude, utilizando el patrón de diseño *Visitor*. Este componente se ha integrado con el resto de plug-ins que forman el framework MOMENT (más concretamente, con el Soporte para transformaciones de esta herramienta), permitiendo la traducción bajo demanda de expresiones OCL en el proceso de generación de código Maude para una transformación a partir de un modelo QVT.

Este sistema de traducción, denominado OCLParser, ha formado la primera parte de este proyecto final de carrera. Sus características se pueden resumir en los siguientes puntos, siendo las tres primeras una consecuencia directa de la utilización del patrón de diseño *Visitor*:

- *Sin interferencias con las estructuras a analizar*. El patrón de diseño *Visitor* permite separar la lógica de la operación de generación de código del árbol de derivación que representa la estructura semántica de la expresión OCL.

- *Diseño altamente modular*. La interfaz *SemanticsVisitor* implementada separa las acciones a ejecutar en la operación de generación de código según la clase del elemento a analizar.

- *Diseño extensible*. Implementar una nueva operación sobre la estructura del árbol de derivación, resultado del análisis semántico

de la expresión OCL, únicamente supone especificar una nueva clase que implemente la interfaz Java *SemanticsVisitor*.

- *Estructuras internas preparadas para satisfacer futuros requerimientos*. Las estructuras de datos que dan soporte al proceso de generación de código son totalmente extensibles.

- *Interfaz clara y sencilla*. Se ha minimizado el número de métodos de la interfaz, proporcionando un conjunto de métodos intuitivos y fáciles de manejar, tanto en la llamada como en la recogida de resultados.

- *Retorno estructurado de resultados*. El resultado de los métodos invocados a través de la interfaz es un objeto estructurado, lo cual permite devolver los datos de manera ordenada y dotar de extensibilidad en cuanto a aquello que se va a retornar.

El proyecto OCLEditor es la segunda parte de este proyecto final de carrera, en el cual se ha extendido la funcionalidad de un visor de modelos Ecore para interaccionar con el módulo de traducción de expresiones OCL a código Maude desarrollado. Esta interfaz tiene como objetivo principal evaluar invariantes y consultas sobre instancias de modelos Ecore. Sus características se pueden resumir en los siguientes puntos:

- *Interfaz clara y familiar*, para los usuarios de la plataforma Eclipse y la tecnología EMF.

- *Modelo de persistencia*. Las diferentes expresiones OCL definidas sobre los modelos se almacenan en archivos XMI, que definen instancias de un modelo Ecore de persistencia de la herramienta OCLEditor.

- *Acceso gráfico a la funcionalidad de OCLParser*. Esto ha facilitado en primera instancia las tareas de depuración del traductor, lo cual ha acelerado su desarrollo y extensión con nuevas funcionalidades.

5.2 Trabajos futuros

Durante el desarrollo del sistema de traducción OCLParser y la interfaz OCLEditor para Eclipse se han ido integrando nuevas funcionalidades, derivadas del plan de desarrollo preestablecido, o por la detección de nuevos requisitos. A continuación, se detallan los nuevos requisitos que, a fecha de septiembre de 2006, se han detectado, y que darán lugar a nuevos desarrollos y refactorizaciones sobre los componentes ya existentes.

5.2.1 Ampliación del Soporte para transformaciones en MOMENT

Se ha planteado la posibilidad de definir invariantes sobre los modelos a los cuales se aplican transformaciones en el framework MOMENT. De esta manera, una operación de transformación, o bien, un conjunto de ellas, se puede ver como una transacción que debe conducir a los modelos integrantes de la operación de un estado consistente a otro de igual manera consistente. Por tanto, siguiendo la misma filosofía que la utilizada en los sistemas de gestión de bases de datos, después de cada transformación se deberá comprobar la consistencia de los modelos, evaluando los invariantes previamente definidos.

Por otra parte, el componente OCLParser se encuentra continuamente en proceso de revisión, en ocasiones debidas a modificaciones sobre la especificación algebraica de OCL 2.0 en Maude de MOMENT, lenguaje objeto del traductor desarrollado. El objetivo de estas revisiones no es otro que conseguir mayores cotas de eficiencia, reduciendo los tiempos necesarios para la evaluación de invariantes y consultas sobre las instancias de los modelos considerados.

5.2.2 Ampliación de OCLEditor

Las consultas a las que se da soporte en la actualidad, a través de OCLEditor, se realizan sobre el conjunto de instancias de una clase del modelo considerado. Se ha planteado la posibilidad de definir mediante OCLEditor una consulta dirigida a una instancia específica de una clase del modelo. Se propone dar soporte a la evaluación de la regla de derivación asociada a un atributo derivado de una clase de modelo.

Para ello se plantea la siguiente estrategia:

```
Clase_del_elemento::AllInstances()->select(selección de la instancia específica)
[Consulta sobre la instancia específica]
```

Se debe resolver la selección de una instancia concreta de una clase a través de la interfaz de OCLEditor. Se ha propuesto la utilización de identificadores basados en la URI del modelo.

A parte de este punto, a través del uso de la herramienta para la realización de métricas, se espera la detección de nuevas necesidades que deban tener reflejo en el diseño e implementación de nuevas funcionalidades.

BIBLIOGRAFÍA

[AkeLP03] Akehurst, D., Linington, P., Patrascoiu, O., «OCL 2.0: Implementing the Standard». Technical Report No. 12-03. University of Kent. Canterbury. 2003.

[AkeP03] Akehurst, D., Patrascoiu, O., «OCL for EMF: User guide». University of Kent. Canterbury. June 2003.

[BeDD05] Bézivin, J., Devedzic, V., Djuric, D., Favreau, J.M., Gasevic, D., Jouault, F., «An M3-Neutral infrastructure for bridging model engineering and ontology engineering». INTEROP-ESA. Switzerland. 2005.

[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.

[BoPCR04] 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.

[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.

[BoICRG05] Boronat, A., Iborra, J., Carsí, J. Á., Ramos, I., Gómez A., «Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos». Desarrollo de Software Dirigido por Modelos (DSDM, junto a JISBD). Granada. Septiembre 2005.

[BoCR06] Boronat, A., Carsí, J.Á., Ramos, I., «Algebraic Specification of a Model Transformation Engine». LNCS. Fundamental Approaches to Software Engineering (FASE). ETAPS. Vienna. Austria. Marzo 2006.

[BoOGRC06] Boronat, A., Oriente, J., Gómez A., Ramos, I., Carsí, J. Á., «An Algebraic Specification of Generic OCL Queries within the Eclipse Modeling Framework». European Conference on Model Driven Architecture, Foundations and Applications (ECMDA-FA). Julio 2006.

[BuSte03] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J., «Eclipse Modeling Framework: A Developer's Guide». Addison-Wesley. 2003.

[ClavelE06a] Clavel, M., Egea, M., «Equational Specification of UML+OCL Static Class Diagrams». Universidad Complutense de Madrid. 2006.

[ClavelE06b] Clavel, M., Egea, M., «ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams». Universidad Complutense de Madrid. 2006.

[CUP] Hudson, S.E., Graphics Visualization and Usability Center, Georgia Institute of Technology. Java(tm) Based Constructor of Useful Parsers. http://www2.cs.tum.edu/projects/cup

[Dresden] Dresden OCL Toolkit Website. http://dresden-ocl.sourceforge.net. Septiembre 2006.

[Eclipse] Eclipse Website. http://www.eclipse.org. Septiembre 2006.

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

[Egea05] Egea, M., «ITP/OCL: a Theorem Prover-Based Tool for UML+OCL Class Diagrams». Universidad Complutense de Madrid. 2005.

[EhrigM85] Ehrig, H., Mahr, B.: «Fundamentals of Algebraic Specification 1». Springer-Verlag Berlin Heidelberg New York Tokio (1985). ISBN: 3-540-13718-1.

[EMFT] EMFT Website. http://www.eclipse.org/emft/projects. Septiembre 2006.

[EPL] Eclipse Public License, EPL. V.1.0. http://www.eclipse.org/legal/epl-v10.html. Septiembre 2006.

[Flex] Flex. A Fast scanner generator. Online manual. http://dinosaur.compilertools.net/flex/index.html

[Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., «Design Patterns. Elements of Reusable Object-Oriented Software». Addison-Wesley. 1995.

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

[KentMDE02] Stuart Kent. Model Driven Engineering. In *IFM 2002*, volume 2335 of *LNCS*. Springer-Verlag, 2002.

[KlasseO] Klasse Objecten Website. http://www.klasse.nl. Septiembre 2006.

[KMF] Kent Modelling Framework. University of Kent http://www.cs.kent.ac.uk/projects/kmf

[KurBA02] Kurtev, I., Bézivin, J., Aksit, «M.: Technological Spaces: An Initial Appraisal». Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.

[Maude] The Maude System Website. http://maude.cs.uiuc.edu. Septiembre 2006.

[Maude2Man] 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

[MensV05] Mens, T., Gorp, P.V., «A Taxonomy of Model Transformation». Proc. Int'1 Workshop on Graph and Model Transformation (GraMoT). 2005.

[Me92] Meseguer, J., «Conditional rewriting logic as a unified model of concurrency». *Theoretical Computer Science*, 96(1):73-155, 1992.

[MDA] Object Management Group, MDA Specification, http://www.omg.org/mda/spec.htm, Septiembre 2006.

[MOMENT] MOMENT Website. «MOMENT: A framework for MOdel manageMENT». http://moment.dsic.upv.es. Septiembre 2006.

[Octopus] Octopus: OCL Tool for Precise Uml Specifications. Website. http://www.klasse.nl/octopus. Septiembre 2006.

[OMG] Object Management Group, http://www.omg.org, Septiembre 2006.

[QVT] MOF QVT Standard Specification. http://www.omg.org/docs/ptc/05-11-01.pdf

[R&L] Modelo UML Royal and Loyal. Utilizado en [WarK03] (Figura C-1).

[Rich01] Richters, M., «A Precise Approach to Validating UML Models and OCL Constraints», Logos Verlag Berlin, 2001.

[RONDO] Melnik, S. «Rondo: A Programming Platform for Model Management», Junio 2003. http://www-db.stanford.edu/~melnik/mm/rondo/

[SableCC] Java parser generator. SableCC Website. http://sablecc.org. Septiembre 2006.

[SourceF] SourceForge Website. http://sourceforge.net. Septiembre 2006.

[WarK03] Warmer, J., Kleppe, A., «The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition». Addison-Wesley. 2003.

Anexo 1. Modelo Royal & Loyal

Utilizado para ilustrar ejemplos en [WarK03] (Figura C-1).

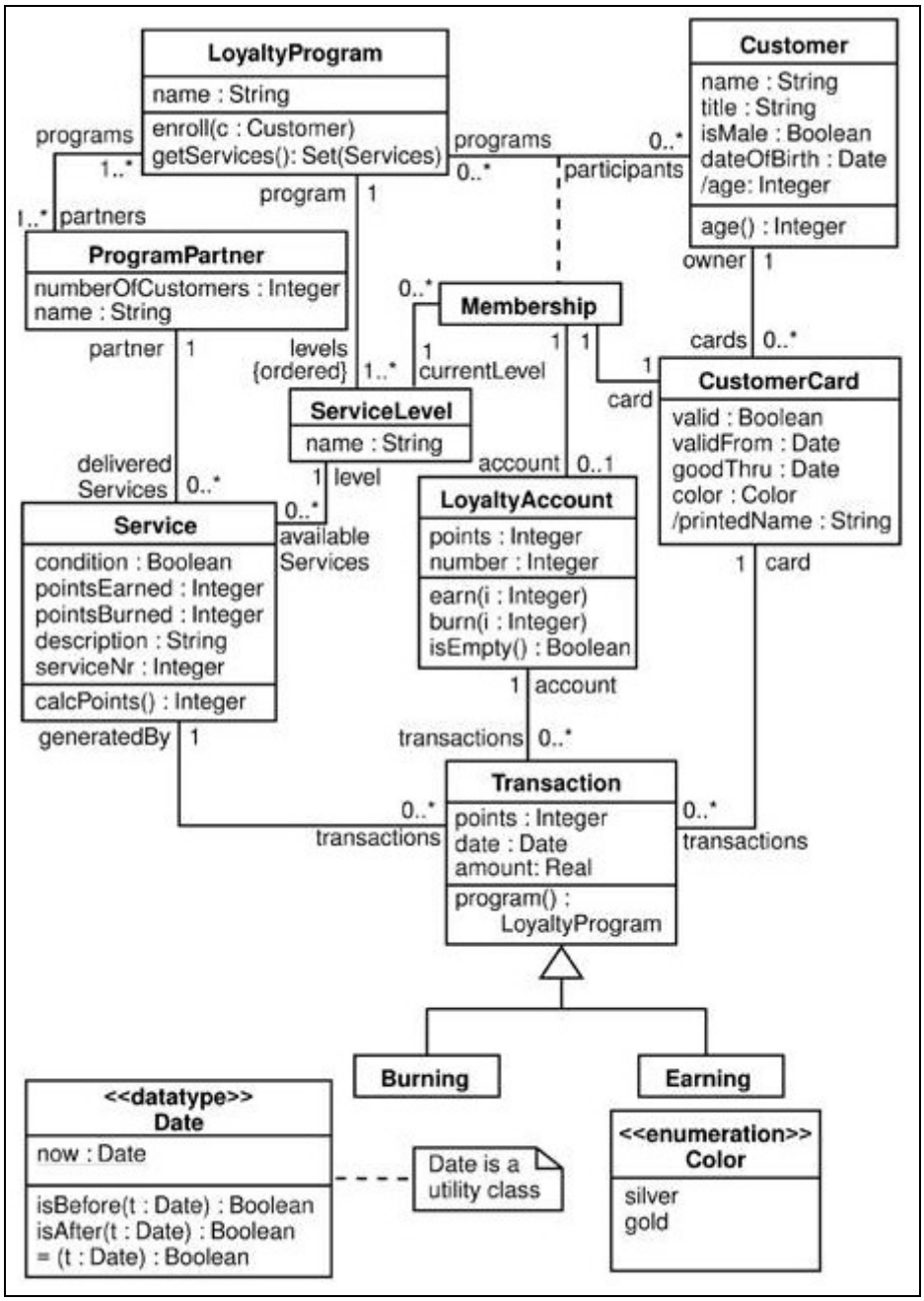

Figura 40. Modelo Royal And Loyal

Anexo 2. KMF. Gramática de OCL 2.0

A continuación se muestra la gramática de OCL [AkeLP03] utilizada para la construcción del analizador sintáctico de KMF, mediante el generador automático de parsers CUP [CUP].

```
packageDeclaration → 'package' pathName contextDeclList 'endpackage'.
packageDeclaration → contextDeclList .
contextDeclList → contextDeclaration* .
contextDeclaration → propertyContextDecl .
contextDeclaration → classifierContextDecl .
contextDeclaration → operationContextDecl .
propertycontext'Decl → 'context' pathName simpleName ':' type initOrDerValue+ .
initOrDerValue → 'init' ':' oclExpression | 'derive' ':' oclExpression
classifiercontext'Decl → 'context' pathName invOrDef+ .
invOrDef → 'inv' [simpleName] ':' oclExpression .
invOrDef → 'def' [simpleName] ':' defExpression .
defExpression → simpleName ':' type '=' oclExpression .
defExpression → operation '=' oclExpression .
operation'context'Decl → 'context' operation prePostOrBodyDecl+ .
prePostOrBodyDecl → 'pre' [simpleName] ':' oclExpression .
prePostOrBodyDecl → 'post' [simpleName] ':' oclExpression .
prePostOrBodyDecl → 'body' [simpleName] ':' oclExpression .
operation → pathName '(' [variableDeclarationList] ')' [':' type]
variableDeclarationList → variableDeclarationList (',' variableDeclaration)* .
variableDeclaration → simpleName [':' type] ['=' oclExpression]
type → pathName | collectionType | tupleType .
collectionType → collectionKind '(' type ')' .
tupleType → 'TupleType' '(' variableDeclarationList ')' .
oclExpression →
literalExp |
'(' oclExpression ')' |
pathName isMarkedPre |
oclExpression DOT simpleName isMarkedPre |
oclExpression '->' simpleName |
oclExpression '(' ')' |
oclExpression '(' oclExpression ')' |
oclExpression '(' oclExpression ',' argumentList ')' |
oclExpression '(' variableDeclaration '|' oclExpression ')' |
oclExpression '(' oclExpression ',' variableDeclaration '|' oclExpression ')' |
oclExpression '(' oclExpression ':' type ',' variableDeclaration '|' oclExpression ')' |
oclExpression '[' argumentList ']' isMarkedPre |
oclExpression '->' 'iterate' '(' variableDeclaration [';' variableDeclaration] '|' oclExpression ')' |
'not' oclExpression |
'-' oclExpression |
oclExpression '*' oclExpression |
oclExpression '/' oclExpression |
oclExpression 'div' oclExpression |
oclExpression 'mod' oclExpression |
oclExpression '+' oclExpression |
oclExpression -' oclExpression |
'if' oclExpression 'then' oclExpression 'else' oclExpression 'endif' |
oclExpression '<' oclExpression |
oclExpression '>' oclExpression |
oclExpression '<' oclExpression |
oclExpression '<' oclExpression |
```

```
oclExpression '=' oclExpression |
oclExpression '<>' oclExpression |
oclExpression 'and' oclExpression |
oclExpression 'or' oclExpression |
oclExpression 'xor' oclExpression |
oclExpression 'implies' oclExpression |
'let' variableDeclarationList 'in' oclExpression |
oclExpression '^^' simpleName '(' [oclMessageArgumentList] ')' |
oclExpression '^' simpleName '(' [oclMessageArgumentList] ')' .
argumentList → oclExpression (',' oclExpression)* .
oclMessageArgumentList → oclMessageArgument (',' oclMessageArgument)* .
oclMessageArgument → oclExpression | '?' [':' type] .
isMarkedPre → ['@' 'pre'] .
literalExp → collectionLiteralExp .
literalExp → tupleLiteralExp .
literalExp → primitiveLiteralExp .
collectionLiteralExp → collectionKind '{' collectionLiteralParts '}' .
collectionLiteralExp → collectionKind '{' '}' .
collectionKind → 'Set' | 'Bag' | 'Sequence' | 'Collection' | 'OrderedSet'.
collectionLiteralParts → collectionLiteralPart (',' collectionLiteralPart)* .
collectionLiteralPart → oclExpression | collectionRange .
collectionRange → oclExpression '..' oclExpression .
tupleLiteralExp → 'Tuple' '{' variableDeclarationList '}' .
primitiveLiteralExp → integer .
primitiveLiteralExp → real .
primitiveLiteralExp → string .
primitiveLiteralExp → 'true' .
primitiveLiteralExp → 'false' .
pathName → simpleName .
pathName → pathName '::' simpleName .
```

Anexo 3. KMF. Interfaz *SemanticsVisitor*

La siguiente interfaz Java ha sido implementada en el contexto del proyecto OCLParser. De esta manera se proporciona soporte a la operación de generación de código Maude sobre el Augmented AST, resultado del proceso de análisis semántico de una expresión OCL. Para dotar de una nueva operación a esta estructura, se utiliza el patrón de diseño *Visitor*.

Cada método está referido a una clase distinta del modelo semántico de OCL, y se invocará cuando se deba generar código Maude para un objeto (nodo de un Augmented AST) de dicha clase.

En este caso se ha optado por sobrecargar el método *visit* con diferentes perfiles, para uniformizar la interfaz.

```
public interface SemanticsVisitor {

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.TypeExp' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.TypeExp host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.StringType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.StringType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.VoidType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.VoidType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.BagType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.BagType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.OrderedSetType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.OrderedSetType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.BooleanType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.BooleanType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.OclAnyType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.OclAnyType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.SetType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.SetType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.OclMessageType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.OclMessageType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.RealType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.RealType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.CollectionType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.CollectionType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.TupleType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.TupleType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.IntegerType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.IntegerType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.types.SequenceType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.types.SequenceType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionKind' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionKind host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CallExp' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CallExp host, Object data);
```

```
/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.LetExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.LetExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.BooleanLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.BooleanLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.OclExpression' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.OclExpression host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.VariableDeclaration' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.VariableDeclaration host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.LoopExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.LoopExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.IntegerLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.IntegerLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionRange' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionRange host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.OclMessageExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.OclMessageExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.VariableExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.VariableExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionLiteralPart' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionLiteralPart host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.RealLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.RealLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.NumericalLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.NumericalLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.IterateExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.IterateExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.ModelPropertyCallExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.ModelPropertyCallExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.PropertyCallExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.PropertyCallExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.OclMessageArg' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.OclMessageArg host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.UnspecifiedValueExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.UnspecifiedValueExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.OperationCallExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.OperationCallExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.EnumLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.EnumLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.LiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.LiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.IfExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.IfExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.StringLiteralExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.StringLiteralExp host, Object data);

/** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.IteratorExp' */
public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.IteratorExp host, Object data);
```

```
        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.TupleLiteralExp' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.TupleLiteralExp host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionItem' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.expressions.CollectionItem host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.ConstraintKind' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.ConstraintKind host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.ContextDeclaration' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.ContextDeclaration host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.Constraint' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.Constraint host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.OperationContextDecl' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.OperationContextDecl host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.PropertyContextDecl' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.PropertyContextDecl host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.contexts.ClassifierContextDecl' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.model.contexts.ClassifierContextDecl host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.SendAction' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.SendAction host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.ModelElement' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.ModelElement host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.EnumLiteral' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.EnumLiteral host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.CallAction' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.CallAction host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Signal' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Signal host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.OclModelElementType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.OclModelElementType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.DataType' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.DataType host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Namespace' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Namespace host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Environment' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Environment host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Classifier' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Classifier host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Enumeration_' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Enumeration_ host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Property' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Property host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.NamedElement' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.NamedElement host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Operation' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Operation host, Object data);

        /** Visit class 'uk.ac.kent.cs.ocl20.semantics.model.bridge.Primitive' */
        public Object visit(uk.ac.kent.cs.ocl20.semantics.bridge.Primitive host, Object data);
}
```

Anexo 4. OCLEditor. Manual de usuario

En esta sección se van a detallar los pasos a seguir para utilizar esta herramienta. Se suponen instalados y configurados los plug-ins necesarios en una distribución de Eclipse versión 3.1.0.

1. Creación de una instancia del modelo OCLEditor

Partamos de cero. Lo primer es crear un archivo con extensión *mocl* para almacenar nuestras expresiones OCL. Para ello nos serviremos de un asistente.

Accedemos al asistente mediante la opción *"File > New > Other"*, eligiendo en la categoría *"Moment"* la opción "*New OCLEditor model file*".

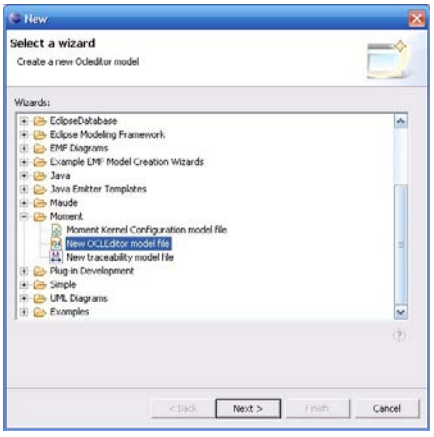

Figura 41. OCLEditor. Asistente para la creación de una instancia del modelo (i)

En la siguiente ventana del asistente se nos pide una carpeta del *workspace* en uso y el nombre del archivo, con extensión *mocl*, que se va a crear.

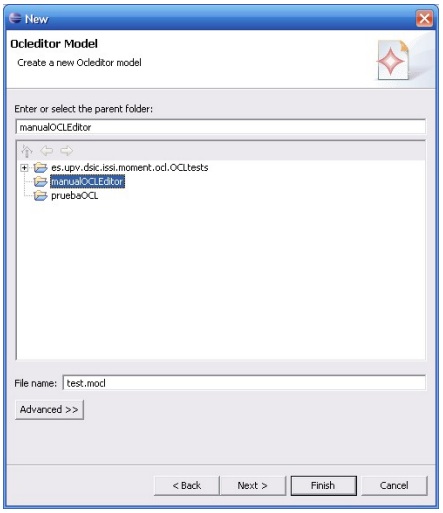

Figura 42. OCLEditor. Asistente para la creación de una instancia del modelo (ii)

En la última ventana del asistente se nos informa de que se va a crear, por defecto, una instancia de la clase *OCLEditor* (que será la raíz del árbol). Además, se da la opción de modificar el tipo de codificación XML. Estas opciones no hay que cambiarlas.

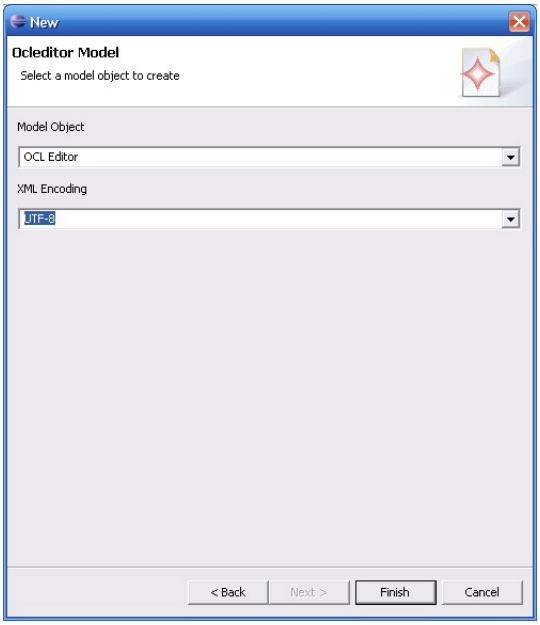

Figura 43. OCLEditor. Asistente para la creación de una instancia del modelo (iii)

Al apretar el botón '*Finish*' se crea el archivo en la carpeta seleccionada. Además, se abre el archivo con el editor, mostrando en un visor en forma de árbol la instancia de la clase OCLEditor que se ha creado por defecto.

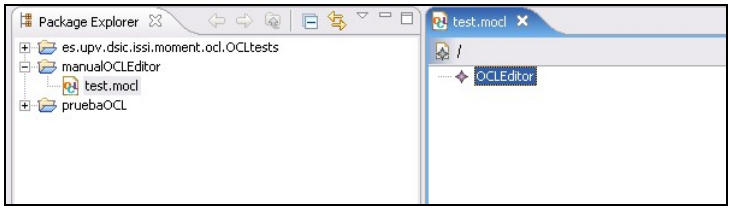

Figura 44. OCLEditor. Archivo *mocl* y raíz del árbol

2. Población de la instancia

Para poblar la instancia del modelo a partir de la raíz nos valemos de la representación jerárquica en forma de árbol que muestra el *tree viewer*, y de los mecanismos que por defecto incluye EMF en este tipo de visores.

Para ello tan fácil como colocarse en cualquier punto del árbol y desplegar el menú contextual de botón derecho del ratón. En el caso de que el nodo del árbol seleccionado sea "contenedor" de otros elementos aparecerá la opción "*New Child*", la cual permitirá crear hijos de ese nodo (de una clase determinada). Por ejemplo, podemos crear nodos de la clase *Model*, como hijos de la raíz *OCLEditor*. A su vez esos nodos *Model* podrán tener como hijos nodos de la clase *Context*…, siguiendo las relaciones de agregación reflejadas en el modelo de persistencia de OCLEditor (ver sección 4.2.2).

En la siguiente figura se muestra una instancia totalmente poblada:

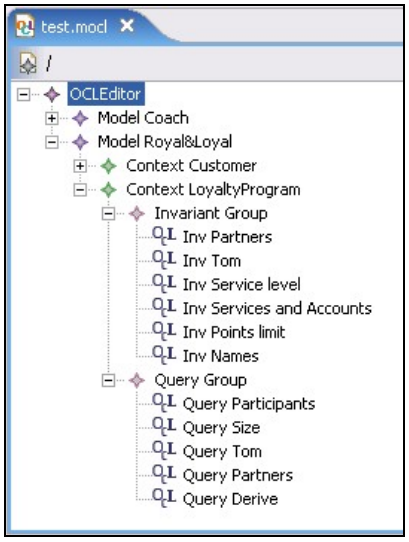

Figura 45. OCLEditor. Instancia poblada

3. Establecimiento de las propiedades de los elementos de la instancia

Una vez poblado nuestro árbol es momento de indicar las propiedades específicas de cada uno de sus elementos.

Veamos que atributos hay que especificar en la vista de propiedades dependiendo de la clase de cada nodo.

3.1 OCLEditor

Es el elemento raíz y no tiene atributos que especificar.

3.2 Model

Se pueden especificar expresiones OCL sobre diferentes metamodelos, evaluadas sobre modelos que los conforman, o sobre diferentes modelos y evaluarlas sobre sus instancias. En la clase "*Model*" se especifica un par concreto, metamodelo/modelo o modelo/instancia, a considerar.

A partir de ahora hablaremos de modelos e instancias, teniendo en cuenta que se debería proceder de manera análoga en el caso de metamodelos y modelos.

Se debe especificar un nombre para el par modelo/instancia (propiedad "*Name*") y la ruta relativa en el *workspace* en uso del archivo XMI que especifica tanto el modelo como la instancia (mediante las propiedades "*Path*" y "*Path Instance*", respectivamente). Para especificar estas rutas se dispone de la ayuda de

un asistente, al cual se accede a través del campo de estas propiedades, en la vista de propiedades.

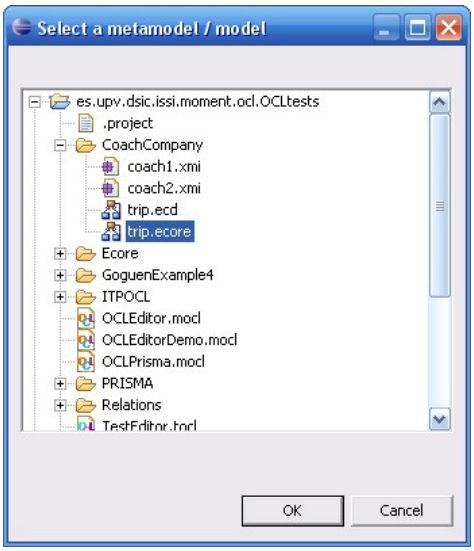

Figura 46. OCLEditor. Asistente para la selección de metamodelo o modelo

3.3 Context

Se pueden definir expresiones OCL sobre diferentes contextos, siendo éste concepto el propio de OCL.

Solamente hay que especificar su nombre mediante la propiedad "*Name*" de este elemento. Por ejemplo, se podría considerar el contexto "*Customer*" del modelo *Royal And Loyal* [R&L].

3.4 Grupos

Un contexto puede "contener" un conjunto de grupos de invariantes ("*Invariant Group*"), consultas ("*Query Group*") o consultas utilizadas para la especificación de transformaciones o relaciones de equivalencia sobre modelos mediante QVT ("*QVT Query Group*"). Estos grupos se utilizan para organizar las expresiones OCL y solamente hay que especificar su nombre mediante la propiedad "*Name*".

3.5 Invariant

En el caso de un invariante hay que especificar las siguientes propiedades:

- *Name*: un nombre identificativo.
- *Ocl*: la expresión OCL en modo textual. Para ello se dispone de la ayuda de un asistente al acceder al campo de esta propiedad en la vista de propiedades.

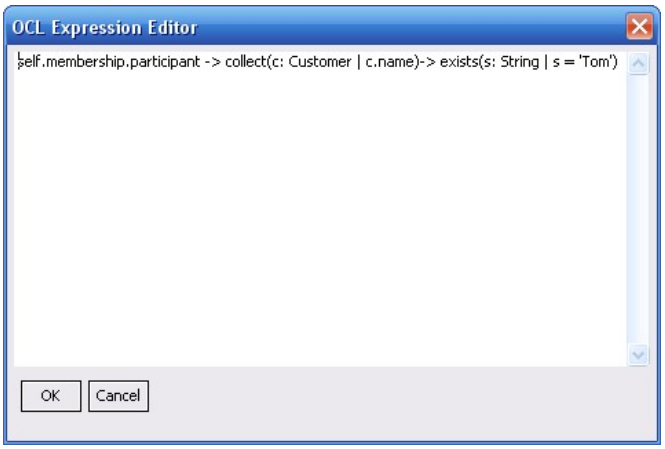

Figura 47. OCLEditor. Asistente para la edición de expresiones OCL

El resto de propiedades son gestionadas internamente por la herramienta y el usuario no deberá modificarlas.

Hay que tener en cuenta que no hay que especificar el contexto, ni indicar que se trata de un invariante, al especificar mediante la propiedad "*Ocl*" la expresión OCL. Esto se debe a que esta información ya se introduce en otros puntos.

Por ejemplo, para una invariante referido al modelo Royal And Loyal [R&L], de contexto "*LoyaltyProgram*" sería correcto especificar en la propiedad "*Ocl*" la siguiente expresión:

self.partners.size() > 0

3.6 Query

Sirve todo lo dicho para "*Invariant*". Solamente hay que indicar que este tipo de expresiones realizan una consulta sobre todas las instancias de una clase determinada. Por ello siempre deben comenzar con la notación:

nombre_de_contexto::allInstances()

Para facilitar esto, el asistente para la edición de expresiones OCL añade por defecto esta línea al modificar la propiedad "*Ocl*", cuando ésta aún no ha sido modificada y se encuentra vacía.

Por ejemplo, para una consulta referida al modelo Royal and Loyal [R&L], de contexto "*LoyaltyProgram*" sería correcto especificar en la propiedad "*Ocl*" la siguiente expresión:

LoyaltyProgram::allInstances()->collect(p:LoyaltyProgram| p.partners->size())

3.7 QVT Query

Sirve todo lo dicho para "*Invariant*". Se ha dado la posibilidad de introducir este tipo consultas en OCLEditor para tareas de depuración del Soporte de OCL en MOMENT. Se trata de consultas que pueden ser utilizadas para la especificación de transformaciones y relaciones de equivalencia sobre modelos en QVT.

4. Acciones de la barra de herramientas

OCLEditor incorpora una barra de herramientas (*tool bar*) a través de la cual se puede acceder a diferentes acciones. Todas las acciones no están disponibles para todas las clases de elementos, y una acción puede estar disponible para diferentes clases de elementos.

A continuación se hace una descripción de cada una de estas acciones.

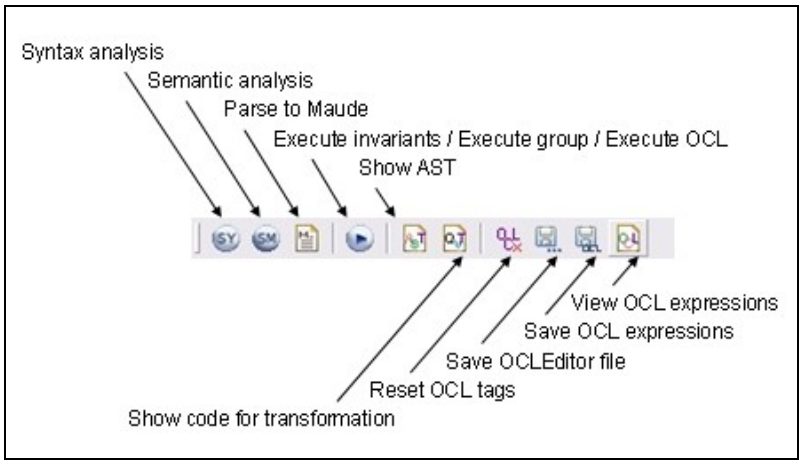

Figura 48. OCLEditor. Acciones de la barra de herramientas

4.1 Syntax analysis

Mediante está acción se realiza un análisis sintáctico de las expresiones OCL. Si la expresión es sintácticamente correcta aparecerá un mensaje de confirmación en la consola. En el caso de que no lo sea aparecerá un mensaje de error igualmente por consola.

En el caso de que la expresión OCL no sea sintácticamente correcta, el icono que representa el elemento en el visor en forma de árbol tomará color amarillo.

4.2 Semantic analysis

Mediante está acción se realiza un análisis semántico de las expresiones OCL. Los mensajes se muestran por consola como en el caso anterior.

En el caso de que la expresión OCL no sea semánticamente correcta, el icono que representa el elemento en el visor en forma de árbol tomará color amarillo.

Por consistencia de las expresiones almacenadas, siempre que se edita una expresión OCL y se almacena se realiza un análisis semántico de la expresión de manera implícita.

4.3 Parse to Maude

Esta acción permite la visualización del código Maude resultado del proceso de traducción de una expresión OCL, el cual le da su semántica operativa (permitiendo su ejecución). Se trata de una operación disponible para tareas de depuración.

4.4 Execute invariants / Execute group / Execute OCL

Esta acción permite la evaluación de todos los invariantes asociados a un elemento de la clase "*Model*" (acción "*Execute invariants*"), la evaluación de un grupo de invariantes (acción "*Execute group*" asociada a elementos de la clase "*Invariant Group*"), o la evaluación de una expresión OCL concreta (acción "*Execute OCL*" asociada a los elementos de las clases "*Invariant*", "*Query*" y "*QVT Query*").

El resultado de la evaluación se muestra por consola. En el caso de que se evalúe el valor de un invariante, el icono que representa el invariante en el visor en forma de árbol tomará color verde en el caso de que se evalúe a cierto y tomará color rojo en el caso de que se evalúe a falso. La información del resultado de la evaluación también aparece por consola.

En cualquier caso, si tras ejecutarse y evaluarse una expresión OCL se detecta un error sintáctico, semántico o un error interno de la herramienta, el icono que representa la expresión OCL en el visor en forma de árbol tomará color amarillo, informándose del error por consola.

4.5 Show AST

Se da la opción de visualizar un esquema textual del Augmented AST resultado de la traducción de una expresión OCL, proporcionando una visión estructurada de ésta.

Por ejemplo, para la siguiente expresión OCL, especificada sobre el modelo Royal And Loyal [R&L],

context LoyaltyProgram inv : self.partners.size() > 0

se obtiene la siguiente representación textual del Augmented AST (instancia del modelo semántico de OCL 2.0 de KMF):

```
OperationCall {
   OperationCall {
      PropertyCall {
         VariableExp {
            Variable self:OclModelElementType(LoyaltyProgram)
         }
         Property partners:OrderedSet(OclModelElementType(ProgramPartner))
      }
      Operation size()
   }
   Operation >()
   Integer(0)
}
Type = Boolean
```

4.6 Show code for transformation

Se trata de una acción análoga a la opción "Parse to Maude" específica para elementos de la clase "*QVT Query*".

4.7 Reset OCL Tags

Esta acción inicializa, a las opciones por defecto, las etiquetas booleanas referidas a las expresiones OCL.

4.8 Save OCLEditor file

Permite almacenar un conjunto de expresiones OCL del archivo actual en otro archivo. Para ello se utiliza un asistente similar al utilizado en la creación de un nuevo archivo "*mocl"*, seleccionando una carpeta del *workspace* actual y un nombre de archivo "*mocl"* de destino. Para acceder a esta acción se debe seleccionar un conjunto de elementos de la clase "*Model*", guardándose todos los elementos dependientes de ellos.

4.9 Save OCL Expressions

Permite almacenar un conjunto de expresiones OCL del archivo actual en un archivo de texto de extensión "*eocl"*. Se utiliza un asistente en el cual se selecciona una carpeta del *workspace* actual y un nombre de archivo "*eocl"* de destino. Este tipo de archivos pueden ser editados mediante un editor específico con coloreado de sintaxis para OCL que incorpora OCLEditor.

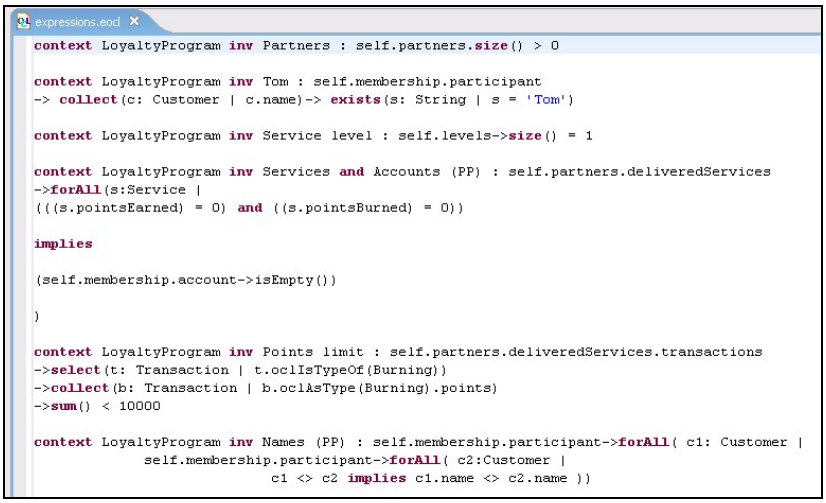

Figura 49. OCLEditor. Editor con coloreado de sintaxis para OCL

4.10 View OCL Expressions

Mediante esta acción se puede visualizar un conjunto de expresiones OCL por la consola.

5. Menú contextual

No todas las acciones descritas están disponibles para todas las clases de elementos. Esto implica que, en el caso de que un elemento no disponga de cierta acción y se presione el botón de esta acción en la barra de herramientas, no se observará ninguna respuesta.

Esto se muestra de manera más clara en el caso del menú contextual de los elementos, al cual se puede acceder mediante el botón derecho del ratón. Para cada clase de elemento solo aparecerán aquellas acciones que estén disponibles.

En la siguiente tabla se muestran las diferentes clases de elementos y las acciones que están disponibles en cada caso. El orden de las acciones, de arriba abajo, es el mismo que el seguido en la barra de herramientas, de izquierda a derecha.

	OCLEditor	Model	Context	Invariant Group	Query Group	QVT Query Group	Invariant	Query	QVT Query Group
Syntax							X	X	X
Semantic							X	X	X
Maude							X	X	
Execute		X		X			X	X	
AST							X	X	X
QVT									X
Tags	X	X		X			X	X	X
Save file		X							
Save exp		X		X	X	X	X	X	X
View exp		X		X	X	X	X	X	X

Tabla 8. OCLEditor. Elementos del modelo y acciones disponibles

6. Código de colores para las expresiones OCL

Los iconos que representan las expresiones OCL, en la representación jerárquica en forma de árbol de la instancia del modelo de OCLEditor, toman diferentes colores dependiendo de la combinación del valor de una serie de atributos booleanos.

En la siguiente tabla se muestran los diferentes colores que pueden tomar, la combinación de valores de los atributos que lo provocan y su significado.

	correct	evaluated	result	Significado
Azul	**true**	**false**	true/false	Estado por defecto. No evaluada y correcta.
Amarillo	**false**	true/false	true/false	Error sintáctico, semántico (o interno de la herramienta).
Verde	**true**	**true**	true	Invariante evaluado a *true*.
Rojo	**true**	**true**	false	Invariante evaluado a *false*.

Tabla 9. OCLEditor. Combinación de colores para las expresiones OCL e interpretación

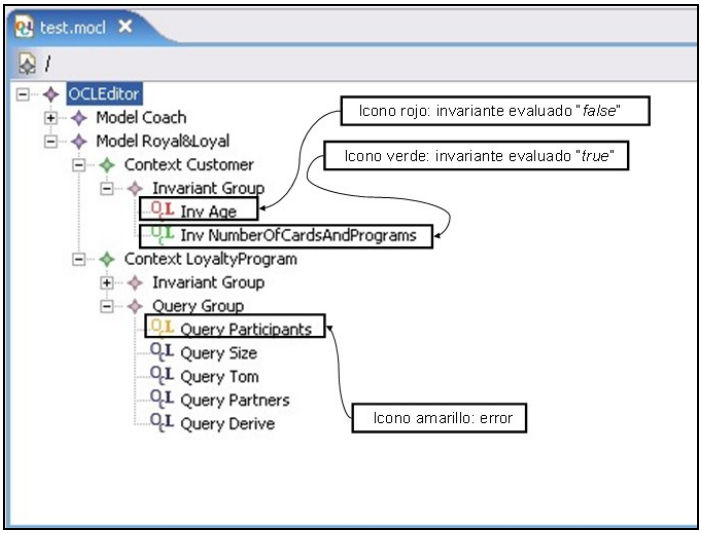

Figura 50. OCLEditor. Código de colores para las expresiones OCL

7. Preferencias de la consola de MOMENT

Desde la ventana de preferencias de la consola de MOMENT, a la cual se puede acceder desde el menú "*Window>Preferences…*", seleccionando la categoría "*MOMENT>Console*". Desde aquí se puede activar o desactivar la salida de información, de depuración, de *warning* y de error de la consola de MOMENT.

Además, se puede activar y desactivar la salida de mensajes por defecto de OCLEditor (*OCLEditor default messages*), así como la de otros mensajes adicionales (*OCLEditor other messages*).