MARCO PARA LA TRANSFORMACIÓN DE MODELOS BASADO EN GRAMÁTICAS DE ATRIBUTOS

UNIVERSIDAD COMPLUTENSE
MADRID

PROYECTO FIN DE MÁSTER EN SISTEMAS INTELIGENTES

Autor: Juan Pablo Gracia Benítez

Director: José Luis Sierra Rodríguez

Máster en Investigación en Informática

Facultad de Informática

Universidad Complutense de Madrid

Curso académico: 2009/2010

MARCO PARA LA TRANSFORMACIÓN DE MODELOS BASADO EN GRAMÁTICAS DE ATRIBUTOS

PROYECTO FIN DE MÁSTER EN SISTEMAS INTELIGENTES

Autor: Juan Pablo Gracia Benítez

Director: José Luis Sierra Rodríguez

Máster en Investigación en Informática

Facultad de Informática

Universidad Complutense de Madrid

Curso académico: 2009/2010

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "MARCO PARA LA TRANSFORMACIÓN DE MODELOS BASADO EN GRAMÁTICAS DE ATRIBUTOS", realizado durante el curso académico 2009-2010 bajo la dirección de José Luis Sierra en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Madrid, a 21 de Junio de 2010

Juan Pablo Gracia Benítez

AGRADECIMIENTOS

El enfrentarse a un proyecto de fin de Máster, en un tiempo tan limitado y mientras se encuentra cursando el máster y desempeñando una actividad laboral parece una tarea imposible de lograr, sin embargo, la ayuda constante e incondicional de José Luis Sierra desde el inicio, ha hecho posible alcanzar este objetivo, bajo tantas circunstancias adversas, con resultados mejores de los esperados. A él, principal creador y gestor de este proyecto, le debo toda mi gratitud por su total apoyo y por haber compartido conmigo parte de su conocimiento.

Agradezco el apoyo de toda mi familia, que desde la distancia no dejaron de alentarme para cumplir este sueño en tierras lejanas. A Luz, que con su cariño y comprensión, me dio la fuerza y tranquilidad para seguir adelante. A todos los compañeros y amigos que siempre estuvieron pendientes del progreso de mi trabajo, por hacer más tolerable la vida lejos de casa.

-A todos mil gracias-

ABSTRACT

The *Model Driven Software Development* paradigm is currently a hot topic in Software Engineering, which exhibits many different advantages during the construction of software systems. Model transformation is a key aspect of this proposal, since it makes possible the automatic generation of multiple interpretations of the same modeled system depending on the metamodel used. Currently there are different approaches to carry out model transformations, such as: graphical languages, based on graph grammars, direct manipulation based on a suitable API, use of XSLT, relational, declarative and hybrid languages, and proposals based on attribute grammars. The last one is particularly relevant because it is a formal approach and it has shown good performance in similar problems in computer science (e.j., language translation). The goal of this research project is to verify the feasibility of creating and using a syntax-directed transformation specification language based on attribute grammars. In order to achieve this goal, we have developed a framework called *Attribute Grammar Transformer* – AGT, which includes: a metamodel for representing transformations, a specification language called AGTL (*Attribute Grammar Transformer Language*), a translator from AGTL to instances of the aforementioned metamodel, and an evaluation / transformation engine. Also, this project explores the practical applicability of AGT through a case study regarding interactive tutorials.

Keywords: *Model Driven Software Development, Attribute Grammars, Metamodels, Model Transformation, Model Driven Architecture, Language Processors, Attribute Evaluators, Interactive Tutorials*

RESUMEN

El paradigma de desarrollo de software dirigido por modelos está tenido un importante auge debido a las múltiples ventajas que ofrece. La transformación de modelos es parte fundamental de dicha propuesta, ya que ofrece la forma de obtener automáticamente diferentes interpretaciones del sistema modelado, dependiendo del metamodelo que lo defina. Actualmente existen diferentes enfoques para realizar dichas transformaciones, tales como lenguajes gráficos, basados en gramáticas de grafos, manipulación directa vía API, basados en XSLT, propuestas de lenguajes relacionales, declarativos e híbridos, y basadas en gramática de atributos. Este último enfoque es de especial interés, por su formalidad y buenos resultados comprobados en problemáticas de computación similares, como la traducción de lenguajes. La propuesta de este proyecto de investigación pretende verificar la factibilidad de crear y utilizar un lenguaje de especificación de transformación de modelos dirigido por la sintaxis, basándose en el formalismo de las gramáticas de atributos. Para comprobar e investigar las cualidades de dicho enfoque se ha desarrollado un marco de transformaciones llamado *Attribute Grammar Transformer* – AGT, el cual consta de un metamodelo para la representación de transformaciones, un lenguaje de especificación denominado AGTL (*Attribute Grammar Transformer Language)*, un traductor de AGTL a instancias del citado metamodelo, y un motor de evaluación de atributos / transformación. En este proyecto se explora, además, la aplicabilidad práctica de AGT mediante su uso en un caso de estudio relativo a tutoriales interactivos.

Palabras Clave: *Desarrollo de Software Dirigido por Modelos, Gramáticas de Atributos, Metamodelos, Transformación de Modelos, Arquitectura Dirigida por Modelos, Procesadores de Lenguaje, Evaluadores de Atributos, Tutoriales Interactivos*

ÍNDICE

INTRODUCCIÓN 1

1 ESTADO DE LA CUESTIÓN 5

1.1 DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS 5
1.1.1 Metamodelado 7
1.1.2 Un ejemplo: Model Driven Architecture (MDA) 8
1.2 REVISIÓN DE LENGUAJES DE TRANSFORMACIÓN DE MODELOS...13
1.2.1 Compendio lenguajes de transformación de modelos...13
1.2.2 QVT (Query/View/Transformation) 16
1.2.3 ATL (Atlas Transformation Language) 22
1.2.4 MT model transformation language 27
1.2.5 MOFScript 30
1.2.6 UMLX 33
1.2.7 Comparativa y discusión...37
1.3 ENFOQUES GRAMATICALES A LA TRANSFORMACIÓN DE MODELOS 38
1.3.1 Conceptos básicos...38
1.3.2 Transformación de modelos utilizando gramáticas de grafos.................................45
1.3.3 Transformación de modelos utilizando gramáticas de atributos 46
1.4 A MODO DE CONCLUSIÓN 48

2 EL MARCO DE TRANSFORMACIÓN AGT 51

2.1 INTRODUCCIÓN..51
2.2 UN EJEMPLO DE TRANSFORMACIÓN EN AGT 52
2.3 EL LENGUAJE DE ESPECIFICACIÓN AGTL...55
2.3.1 Sintaxis de AGTL...56
2.3.2 Semántica de AGTL 60
2.4 EL METAMODELO AGT 65
2.5 LA CLASE SEMÁNTICA 68
2.6 EL TRADUCTOR...69
2.6.1 El analizador léxico 69
2.6.2 El analizador sintáctico – analizador semántico – traductor...................................71
2.7 EL TRANSFORMADOR...74
2.7.1 La fase de análisis 74
2.7.2 La fase de evaluación..79
2.8 A MODO DE CONCLUSIÓN 80

3 CASO DE ESTUDIO 83

3.1 INTRODUCCIÓN..83
3.2 <E-TUTOR> 83
3.3 UN METAMODELO DE ALTO NIVEL PARA LA REPRESENTACIÓN DE TUTORIALES INTERACTIVOS.......................86
3.4 TRANSFORMACIÓN AGT 87
3.4.1 Estilo de Transformación 87
3.4.2 Descripción de la transformación 88

3.4.3 La Clase semántica *92*
3.4.4 Aplicación de la transformación *94*
3.5 A MODO DE CONCLUSIÓN 95

CONCLUSIONES Y TRABAJO FUTURO 97

REFERENCIAS 101

APÉNDICE A. CLASE SEMÁNTICA PARA EL CASO DE ESTUDIO 105

INTRODUCCIÓN

El paradigma de desarrollo de software dirigido por modelos, el cual parte del uso de modelos como base para un proceso de producción automática de *software*, ha adquirido considerable relevancia en los últimos años, debido a las ventajas que supone su uso. Entre dichas ventajas destacan el incremento de la velocidad de desarrollo y calidad del software producido, la mejora en la mantenibilidad, el incremento en la reutilización de componentes de software, la mejora en el manejo de la complejidad de dicho software y, por consiguiente, la mejora de la productividad por parte de los desarrolladores. Sin embargo, muchas de las implementaciones de los conceptos pertenecientes a dicho paradigma se encuentran en etapa de desarrollo, refinamiento y optimización mediante la utilización de métodos formales, comprobación teórica y generación de herramientas prácticas, buscando lograr su mejor aprovechamiento. Este proyecto de investigación se encamina precisamente a buscar posibles mejoras y alternativas en un aspecto fundamental dentro del desarrollo de software dirigido por modelos: la transformación automática de modelos.

Realizando un estudio previo dentro de las diferentes vertientes para realizar transformaciones de modelos, se pueden encontrar varios enfoques comunes tales como: (i) lenguajes gráficos, algunos basados en la utilización de gramáticas de grafos, (ii) manipulación directa vía API, (iii) basados en XSLT, (iv) propuestas de lenguajes relacionales, declarativos e híbridos, y basadas en gramática de atributos (v). Este proyecto se ha centrado en la última de estas alternativas, orientándose a comprobar la factibilidad de crear y utilizar un lenguaje que permita expresar la transformación directa de modelos utilizando gramáticas de atributos. Las gramáticas de atributos constituyen un lenguaje declarativo de especificación de muy alto nivel, específicamente orientado a tareas de procesamiento de lenguajes dirigido por la sintaxis, y promueven una clara separación entre sintaxis y semántica, a diferencia de otros lenguajes típicos en el dominio de la transformación de modelos, que tienden a ignorar completamente tal separación, lo que puede agregar complejidad innecesaria al proceso y ocasionar dificultades al especificador. Sin embargo, a pesar de la exitosa utilización de las gramáticas de atributos en aplicaciones computacionales relativas al procesamiento de lenguajes formales, éstas han sido relativamente poco aplicadas en el desarrollo de software dirigido por modelos, ya que la única propuesta encontrada al respecto es la de May Dehayni y Louis Féraud [1], propuesta que, además, se basa en expresar los modelos en texto, y en realizar la transformación entre dichos formatos textuales. En contraposición con dicha propuesta, en este proyecto investigaremos la posibilidad de aplicar los mismos principios

directamente sobre las redes de objetos que representan los modelos de entrada a las transformaciones, evitando, de esta forma, la necesidad de implicar sintaxis textuales intermedias.

Para contrastar la factibilidad del uso de gramáticas de atributos en la transformación directa de modelos, en este proyecto se ha desarrollado el marco de transformación AGT (*Attribute Grammar Transformer*). AGT consta de un metamodelo que permite representar las transformaciones basadas en gramáticas de atributos y de un motor para realizar dichas transformaciones. Además define un lenguaje textual para especificarlas llamado AGTL (*Attribute Grammar Transformation Language*), y provee para su procesamiento un traductor que traduce las especificaciones AGTL en instancias del citado metamodelo de representación de transformaciones. AGT pretende ser la base de futuras investigaciones en cuanto a eficiencia del enfoque y usabilidad del lenguaje, además de contemplar su posible integración con otras herramientas, ya sean pertenecientes al paradigma, o bien que puedan sacar provecho de sus mecanismos de transformación.

Este documento se divide en cuatro capítulos:

- En el primer capítulo se realiza una revisión del estado de la cuestión, la cual fundamenta esta propuesta. En ésta se revisan los principales conceptos del desarrollo de software dirigido por modelos, haciendo especial énfasis en una de las propuestas más populares: *Model Driven Architecture (MDA)*. Así mismo, se hace una revisión general de los principales lenguajes de transformación, centrándose en cinco lenguajes en particular: *QVT, ATL, MOFScript, UMLX* y MT *model transformation language*. Estos lenguajes representan diferentes enfoques para realizar transformación de modelos, por lo que se recalcan sus principales características, ventajas y desventajas. Para finalizar el capítulo, se presentan los enfoques gramaticales utilizados para llevar a cabo la transformación de modelos, y algunas propuestas que han hecho uso de estos formalismos.

- En el segundo capítulo se presenta el marco de transformación AGT. En este capítulo se pueden encontrar detalles de su implementación, de su arquitectura, de la sintaxis y semántica de su lenguaje de especificación AGTL, y, en especial, acerca de cómo se realizan las transformaciones, todo esto apoyado en un ejemplo práctico.

- El tercer capítulo realiza el desarrollo de un caso de estudio real y de mayor complejidad, que consiste en realizar la transformación de un modelo de especificación de alto nivel de tutores interactivos a un modelo ejecutable dentro del *framework* <e-Tutor>, un entorno experimental para la reproducción de este tipo de tutores desarrollado en el grupo de investigación en Ingeniería del Software aplicada al

e-Learning de la UCM. El principal objetivo de este capítulo es contrastar la factibilidad práctica de la propuesta planteada en el proyecto de máster.

- El último capítulo presenta las conclusiones obtenidas con el desarrollo de este proyecto, y esboza algunas posibles líneas de trabajo futuro.

1 ESTADO DE LA CUESTIÓN

1.1 Desarrollo de software dirigido por modelos

La aplicación de modelos al desarrollo de software es una propuesta que, aunque tiene ya una larga tradición en Ingeniería del Software (véase, por ejemplo,[2][3]), ha experimentado un aumento espectacular de popularidad desde la creación del lenguaje unificado de modelado (UML – *Unified Modeling Language*) [4]. Sin embargo, éste fue utilizado inicialmente como herramienta de documentación, al permitir expresar únicamente de forma intencional, no formal, la relación entre el modelo y la implementación. La principal desventaja que se deriva de este hecho se debe a que los sistemas de software no son estáticos, sino que son propensos a cambios, particularmente intensos durante las primeras fases del ciclo de vida. Como la documentación necesita estar meticulosamente sincronizada con la implementación para evitar inconsistencias, el mantenimiento de dichos modelos resulta en una serie de tareas complicadas que termina aumentando costos y esfuerzos, disminuyendo claramente la utilidad real de este tipo de propuestas para los desarrolladores de software.

Como contrapartida al carácter intencional de lenguajes tipo UML, el *desarrollo de software dirigido por modelos* (MDSD – *Model Driven Software Development*) conocido también de manera menos precisa como *desarrollo dirigido por modelos* (MDD – *Model Driven Development*) introduce un enfoque diferente: los modelos no constituyen únicamente la documentación, sino que adquieren un estatus análogo al del código, de tal manera que su implementación puede ser automatizada.

Este uso de los modelos como base a un proceso de producción automática de *software*, puede ejemplificarse mediante una analogía con procesos similares en otras ingenierías, por ejemplo, como en la ingeniería mecánica. Efectivamente, en ingeniería mecánica es posible automatizar la creación automática de piezas físicas a partir de modelos computarizados, producidos generalmente mediante el uso de herramientas de diseño asistido por ordenador (CAD – *Computer Aided Design*). Una vez aprobado el diseño, es posible activar una cadena de producción robotizada, totalmente automática, que produce como resultado la pieza terminada [5]. Este ejemplo demuestra que el dominio es esencial tanto para los modelos como para los procesos de producción automática. Claramente el programa de automatización generalmente no puede abarcar todos los procesos (en el ejemplo, no será posible construir completamente una casa prefabricada o un coche mediante procesos totalmente automatizados, sino únicamente tipos de piezas y componentes muy bien definidos y

delimitados). De esta forma, MDSD busca encontrar una abstracción del dominio específico y hacerlo accesible a través de modelado formal, lo cual proporciona el potencial necesario para automatizar la producción de software, de tal manera que se incremente la productividad. Así mismo, MDSD hace posible también que los modelos puedan ser entendidos por expertos del dominio y a su vez servir como medio de comunicación entre los participantes del proyecto de desarrollo de software, en el tiempo y en el espacio. Estas ventajas son esenciales en el contexto de los proyectos software, debido a las características dinámicas de los mismos. Más concretamente, los principales objetivos de MDSD son:

- Incrementar la velocidad de desarrollo. Esto se puede lograr a través de la citada automatización, ya que se puede generar código ejecutable a partir de modelos formales usando una o varias *transformaciones*. Efectivamente, para aplicar satisfactoriamente los enfoques de MDSD es necesaria la existencia de lenguajes específicos de dominio que permitan formular dichos modelos [6][7]. Para tal fin, es frecuente utilizar lenguajes gráficos, aunque el uso de estos no es obligatorio, ni tampoco adecuado en todos los escenarios posibles. De hecho, el uso de modelos textuales es también una opción perfectamente factible [8][9]. Aparte de disponer de lenguajes de modelado adecuados, es también necesario disponer de lenguajes que puedan expresar la transformación de modelos a código, además de compiladores, generadores y transformadores basados en estos lenguajes, que permitan realizar las transformaciones para generar código ejecutable en las plataformas disponibles.

- Incrementar la calidad del software producido. El uso de transformaciones automatizadas y lenguajes de modelado formalmente definidos permite incrementar la calidad del software. En concreto, una vez definida una arquitectura de software apropiada sobre la que basar los modelos, es posible realizar fácilmente mejoras recurrentes en su implementación sin afectar a los artefactos finales, ya que estos pueden generarse automáticamente a partir de dichos modelos.

- Mejorar la mantenibilidad. Efectivamente, aquellos aspectos que no se pueden localizar fácilmente en un único módulo (*cross-cutting* [10]) pueden situarse y cambiarse ahora en un único lugar, por ejemplo, en las reglas de transformación. De forma análoga, los errores en el código generado pueden ser corregidos sin más que cambiar dichas reglas. Con ello, se evita la redundancia y se permite administrar adecuadamente los cambios tecnológicos.

- Incrementar el grado de reutilización. Una vez que se ha definido la arquitectura, el lenguaje de modelado y las transformaciones, estos pueden ser utilizados dentro de una línea de producción de software para producir múltiples sistemas. Esto fomenta una alta reusabilidad, así como una representación explícita del conocimiento en

forma de modelos de alto nivel, hecho que enlaza directamente con las propuestas generativas clásicas en el campo de la reutilización del *software* [2][3].

- Mejorar el manejo de la complejidad. Los lenguajes de modelado permiten "programar" o configurar software en un nivel más alto de abstracción.

- Mejorar la productividad. Los enfoques dirigidos por modelos ofrecen entornos altamente productivos en campos de la tecnología, ingeniería y administración por medio del uso de procesos construidos en bloques, de acuerdo con las mejores prácticas de desarrollo, lo que redunda también en el refuerzo de los objetivos anteriormente mencionados.

Existen varias implementaciones y enfoques de MDSD, entre los que destacan *Model Driven Architecture* (MDA) [11], *Agile Model-Driven Development*[12] , *Model Integrated Computing* [13], *Domain-Oriented Programming*[14] y *Microsoft's Software Factories*[15]. Dado el predomino y estatus de estándar de MDA, este enfoque será analizado con más profundidad posteriormente. Antes se introducirá, no obstante, el concepto de *metamodelado*, dado su carácter fundamental en relación con MDSD.

1.1.1 Metamodelado

La construcción de *metamodelos*, el *metamodelado*, es uno de los aspectos más importantes del desarrollo de software dirigido por modelos. Efectivamente, un metamodelo describe la posible estructura de modelos de la misma clase. Este aspecto es necesario para:

- Construcción de los lenguajes de modelado específicos de dominio (DSLs – *Domain Specific Languages*). Los metamodelos describen la *sintaxis abstracta* de cada lenguaje [6] [7]. Mientras que la sintaxis concreta de un lenguaje específica si un analizador puede aceptar dicho lenguaje, una sintaxis abstracta sólo específica cómo debe lucir la estructura del lenguaje. Se puede decir que una sintaxis concreta es la realización o representación de una sintaxis abstracta. Un ejemplo de esto es XML (*e*X*tensible* M*arkup* L*anguage*) [16]. Los documentos XML se escriben utilizando una sintaxis concreta de XML; por su parte, los analizadores sintácticos de XML pueden procesar estos documentos instanciando representaciones en memoria, tales como los árboles DOM (*Document Object Model*) [17]. DOM en sí es una sintaxis abstracta de XML, que caracteriza los componentes básicos de XML (p.ej., elementos, contenidos textuales, atributos, etc.), así como las relaciones que existen entre dichos elementos, pero no establece la forma concreta en la que se han de escribir dichos documentos. De forma análoga, los metamodelos definen las construcciones de los lenguajes de modelado y sus relaciones, así como las restricciones y reglas de modelado, pero no la sintaxis concreta de dichos lenguajes.

- Validación de los modelos. Los modelos son validados mediante las reglas definidas en el metamodelo.
- Transformaciones de Modelo a Modelo. Las transformaciones están definidas como reglas de mapeo entre metamodelos.
- Generación de Código. Las plantillas de generación se refieren al metamodelo del DSL.
- Integración de herramientas. Basándose en el metamodelo, las herramientas de modelado pueden adaptarse a cada dominio particular.

Metamodelos y modelos tienen una relación *clase-instancia*. Cada modelo es una instancia de un metamodelo. Para definir un metamodelo, es necesario un lenguaje de *metamodelado* que a su vez es descrito por un *meta-metamodelo*. En teoría, esta abstracción en cascada puede ser infinita, pero en la práctica se utilizan aproximaciones a la misma, que constan de un número predeterminado de niveles, como se discutirá en el apartado dedicado a MOF (*Meta-Object Facility*), que es parte de la propuesta MDA.

1.1.2 Un ejemplo: *Model Driven Architecture* (MDA)

El *Object Management Group* (OMG) ha definido una elaboración particular de MDSD llamada *Model-Driven Architecture* (MDA), la cual proporciona una aproximación a la filosofía general de MDSD, así como permite disponer de herramientas que permitan (véase la Figura 1)[18]:

- Especificar sistemas independientes de las plataformas que los soportan, modelando los mismos mediante modelos independientes de plataforma (PIMs, *Platform Independent Models*).
- Escoger la(s) plataforma(s) más adecuada(s) para cada tipo de sistema y especificar las mismas mediante *modelos específicos de plataformas* (PSMs, *Platform Specific Models*).
- Transformar las especificaciones de los sistemas (los PIMs) a las especificaciones de las plataformas (los PSMs).

De esta forma, los tres principales objetivos de MDA son portabilidad, interoperabilidad y reutilización a través de la citada separación arquitectónica.

En principio, MDA es una aproximación de MDSD, pero también se diferencia del enfoque general en otros detalles, en parte debido a las diferentes motivaciones de cada uno. En particular, MDA tiende a ser más restrictivo, centrándose en un lenguaje de modelado basado en UML, mientras que MDSD no posee esta limitación.

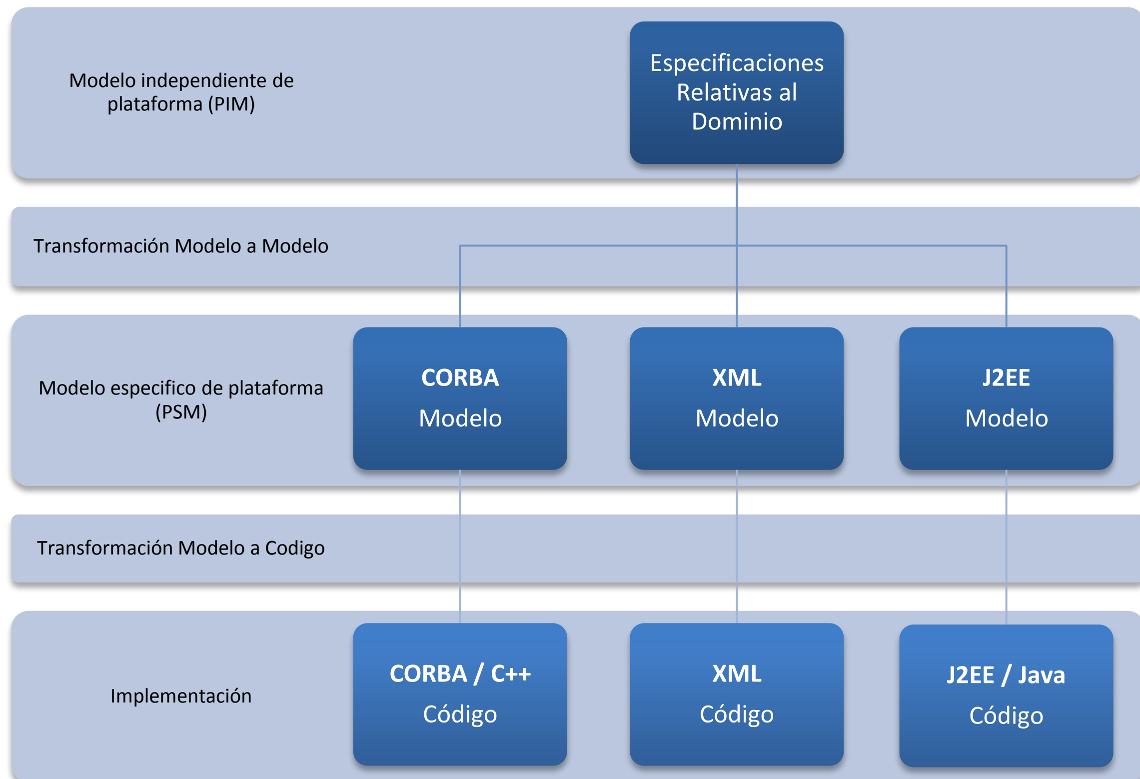

Figura 1. Ejemplo de elementos básicos de MDA.

1.1.2.1 Meta-Object Facility (MOF)

Meta-Object Facility (MOF)[19] es un estándar del OMG para su propuesta MDA. MOF se basa en la *InfrastructrureLibrary* de UML [19], un metalenguaje mínimo en términos del cuál es posible definir la sintaxis abstracta de UML (así como la de otros lenguajes de modelado). De esta forma, MOF se configura como un lenguaje de meta-modelado versátil. De hecho, actualmente UML es un lenguaje MOF [19][20]. Por tanto, se encuentran alineados arquitectónicamente UML, MOF y XMI (*XML Metadata Interchange*) de tal manera que se soporta completamente el modelo de intercambio y se permitir la adaptación de UML a través de perfiles y creación de nuevos lenguajes basados en el mismo metalenguaje núcleo utilizado para definir UML. De esta forma, el metamodelo UML ha sido estructurado con los principios de diseño de modularidad, manejo de capas, división en partes, extensibilidad y reutilización. MOF induce una jerarquía de modelos de cuatro capas que satisface los principios anteriormente expuestos, y en la que puede encuadrarse consistentemente UML (véase la Figura 2):

- En la capa superior, nivel M3, se sitúan los meta-metamodelos. Esta capa incluye el modelo de MOF como un caso concreto de meta-metamodelo.
- La capa siguiente, nivel M2, aloja a los metamodelos (entre ellos, el modelo de UML y CWM - *Common Warehouse Metamodel* [21]). De esta forma, los metamodelos serán las instancias de los meta-metamodelos. Efectivamente, cada elemento de un metamodelo será una instancia de un elemento en el correspondiente meta-metamodelo. Así, por ejemplo, el modelo de UML (un metamodelo) es una instancia del modelo de MOF (un meta-metamodelo).
- La capa de nivel M1 aloja modelos de dominios semánticos concretos (sistemas software, procesos de negocios, requisitos, etc.). Cada modelo en esta capa será una instancia de un metamodelo.
- En la parte más baja de la jerarquía se encuentra la capa de nivel M0, la cual contiene las instancias en tiempo de ejecución de los elementos de los modelos M1.

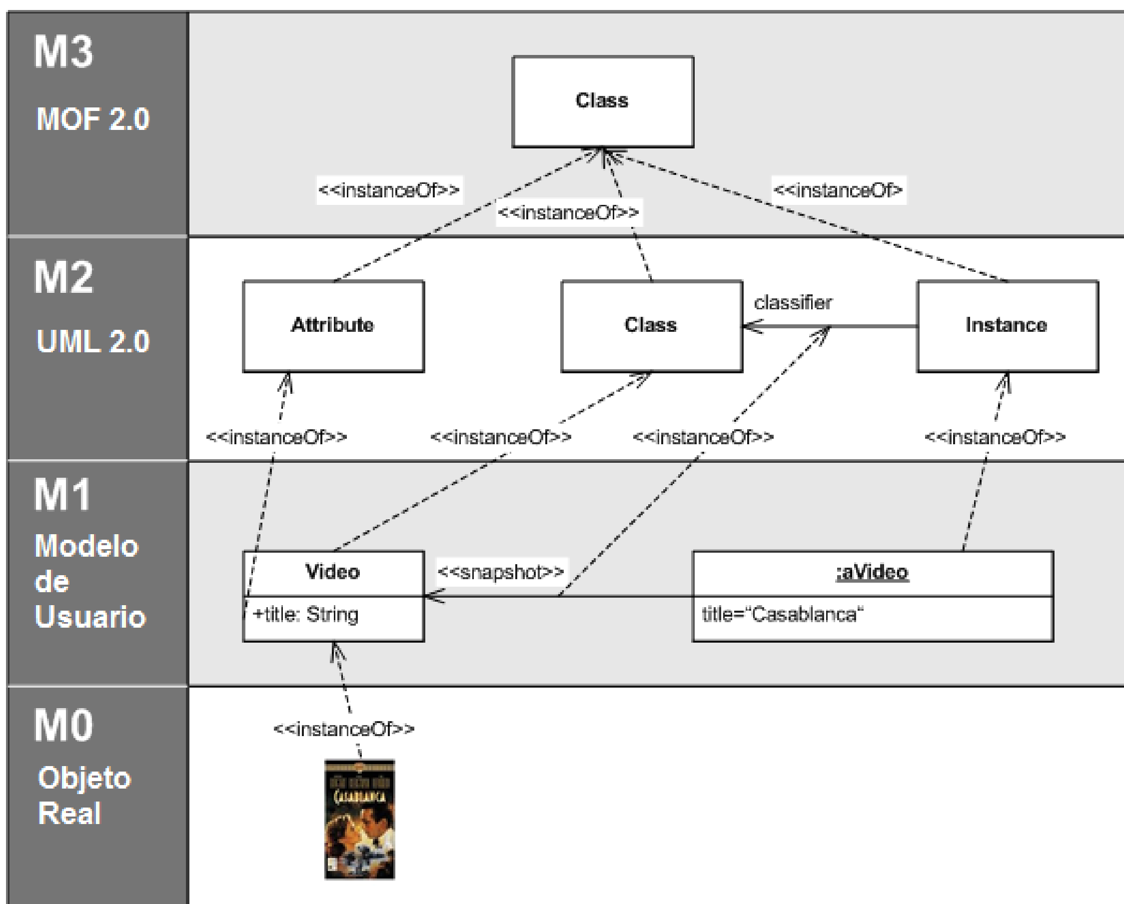

Figura 2. Ejemplo de niveles MOF.

Una característica específica del metamodelado es la capacidad de definir *lenguajes y modelos reflexivos*, es decir lenguajes que pueden ser usados para definirse a sí mismos y modelos que

pueden ser instancias de sí mismos [22]. La *InfrastructureLibrary*[19], es un ejemplo de modelo reflexivo, ya que contiene todas las metaclases requeridas para autodefinirse. MOF es auto-reflexivo ya que se basa en la *InfrastructureLibrary*. Por esta razón no hay metacapas superiores a la capa en que está definida MOF. [23]

1.1.2.2 *Essential* MOF (eMOF) y *Complete* MOF (cMOF)

MOF ha servido como base a diversos marcos de metamodelado, los cuáles, en la práctica, se limitan a implementar un subconjunto relevante de MOF. Actualmente el más influyente de estos marcos es *Eclipse Modeling Framework* (EMF) y su meta-metamodelo eCore [24]. La implementación de EMF, a su vez, ha influido en el proceso de estandarización de MOF 2.0. Como consecuencia de dicha influencia, OMG identificó un subconjunto de MOF llamado *essential* MOF (EMOF), durante la estandarización de MOF 2.0 en 2003, como subconjunto suficiente para la mayoría de las realizaciones del estándar[25]. Como consecuencia, el meta-metamodelo eCore de EMF es ahora conforme con el estándar EMOF del OMG.

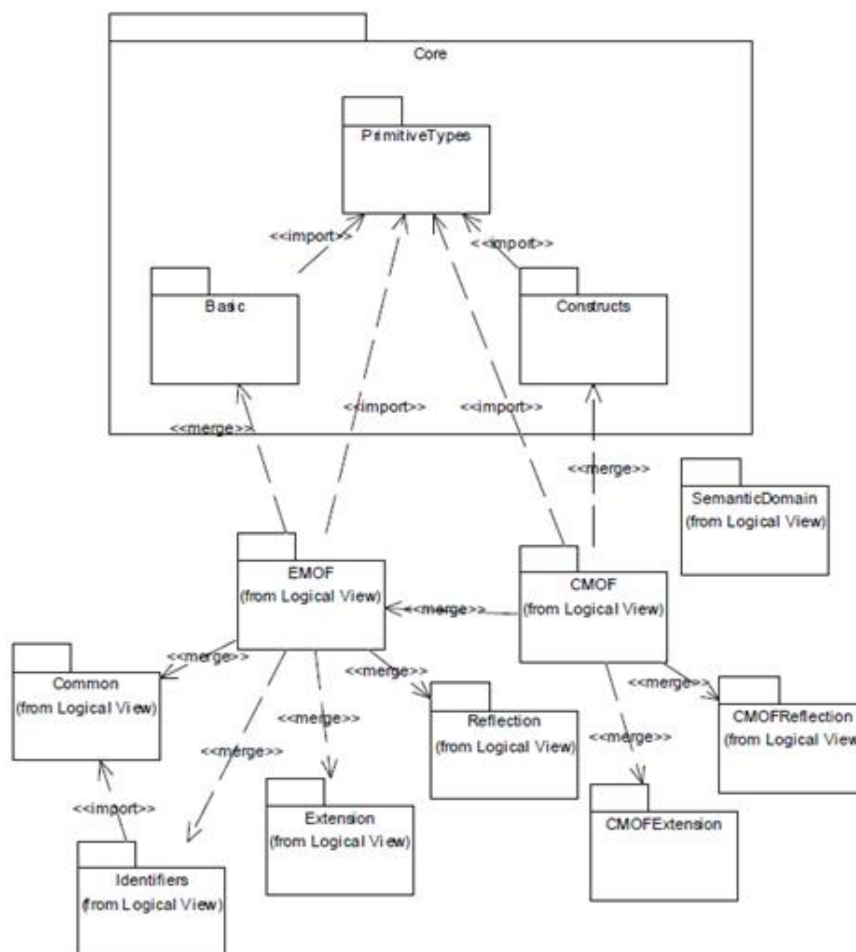

Figura 3. Paquetes de EMOF y CMOF.

El OMG también ha definido CMOF (*Complete* MOF), el cual se utiliza para la definición de metamodelos más complejos, como UML, y que además posee características introspectivas más poderosas [25], debido a que está construido a partir de EMOF, agregando nuevas funcionalidades, entre ellos paquetes de reflexión y los constructores del núcleo (*core*) de la *InfrastructureLibrary* (UML 2), como muestra la figura 3.

1.1.2.3 *Object Constraint Language* (OCL)

El *Object Constraint Language* (OCL) es un lenguaje declarativo para la definición de restricciones, reglas y consultas para los lenguajes de modelado basados en MOF[23]. El lenguaje en sí permite también enriquecer los modelos con información adicional sobre la validez de las instancias de los modelos. El ámbito de uso de OCL se restringe, normalmente, a los niveles M1 y M2 del esquema de capas MOF.

1.1.2.4 XML M*etadata Interchange* (XMI)

XMI define convenios de representación XML para MOF [26]. Actualmente es la base para la interoperabilidad entre diferentes herramientas MDA. Desde la versión 2.0, XMI también permite la serialización de diagramas (las representaciones visuales de los modelos), lo cual es obligatorio en un intercambio de modelos práctico y útil entre las herramientas de modelado, y no sólo para la generación de código. La versión 2.1 definió también una codificación simplificada para EMOF, con el fin de proporcionar mejor interoperabilidad entre herramientas.

XMI establece dos tipos de convenios de representación:

- El primero es uno completamente genérico que establece una transformación de cualquier metamodelo MOF mediante un DTD (*Document Type Definition*) general [16]. De esta forma, cada modelo se representa como un documento que sigue la DTD. La ventaja es que este convenio se puede aplicar de forma homogénea a todos los metamodelos MOF. No obstante, debido a que el convenio de codificación se define de manera genérica, al nivel de MOF, los documentos resultantes se vuelvan poco concisos. Como consecuencia, estos documentos resultan difíciles de leer para los humanos, y no son totalmente adecuados para servir como base a transformaciones basadas en XSLT (*Extensible* S*tylesheet* L*anguage* T*ransformations*) [27].

- El segundo convenio se basa en utilizar un esquema específico para cada metamodelo MOF. Como consecuencia, cada tipo de modelos se codificará mediante un tipo específico de documento. Por tanto, los archivos resultantes serán más compactos y concretos, porque las estructuras se representarán a nivel de metamodelo y no de meta-metamodelo. El precio a pagar es una pérdida evidente de generalidad.

Obviamente la mayoría de herramientas utilizan el primer tipo, dada su generalidad.

1.2 Revisión de lenguajes de transformación de modelos

Como se ha visto en el apartado anterior, la transformación de modelos es una tecnología fundamental en los diversos enfoques del desarrollo de software dirigido por modelos. Esta tecnología permite caracterizar artefactos capaces de traducir automáticamente modelos que se ajustan a un determinado metamodelo origen, en modelos que se ajustan a un determinado metamodelo objetivo (transformaciones *modelo a modelo*). Así mismo, estas tecnologías permiten también realizar transformaciones entre modelos y otros formatos de representación (p.ej., entre modelos y formatos textuales, dando lugar a las denominadas transformaciones de *modelo a texto* y de *texto a modelo*).

En los últimos años se han definido una gran cantidad de lenguajes de transformación, cada uno con diferentes características. Sin embargo, aún es necesaria una mejor comprensión de la naturaleza de las transformaciones de modelos, así como seguir investigando para encontrar las propiedades deseables de un lenguaje de transformación de modelos. Este proyecto se une a dicho esfuerzo. De esta forma, como paso previo se considera fundamental analizar las propuestas existentes con el fin de fundamentar las propuestas realizadas en el mismo.

El propósito de este apartado es, por tanto, realizar una revisión de los lenguajes más populares y profundizar en algunas propuestas por las características especiales de su implementación. En concreto, en este apartado se analizarán con mayor detalle: QVT[28] el estándar presentado por el OMG, ATL[29] por ser una implementación muy cercana al estándar QVT que realiza transformaciones de Modelo a Modelo, MT *model transformation language*[30] el cual está implementado como un lenguaje especifico de dominio (DSL) embebido o interno, MOFScript [31] que es un lenguaje de transformación de modelos a texto, y UMLX[32] que es un lenguaje gráfico. Cabe aclarar que no se pretende llevar a cabo una exposición detallada de cada uno de los aspectos de estos lenguajes, sino resaltar sus características más importantes y ejemplificar su utilización.

1.2.1 Compendio lenguajes de transformación de modelos

La Tabla 1 es una muestra de la diversidad de lenguajes de transformaciones que se pueden encontrar en la actualidad, una actualización de la presentada en [33]. La intención de dicha tabla es mostrar el gran número de propuestas existentes, que evidencia el gran interés que se ha despertado por este aspecto del paradigma del desarrollo de software dirigido por modelos, así como la enorme actividad de investigación surgida alrededor del mismo. De todos estos lenguajes, los más representativos se encuentran descritos con mayor detalle posteriormente.

Tabla 1. Lenguajes de transformación de modelos e implementaciones más notables.

Lenguaje/Implementación	Breve descripción
ATL (Atlas Transformation Language) [29]	Lenguaje de transformación de modelos y herramienta para Eclipse desarrollada por el Atlas Group (INRIA).
BOTL(Basic Object-Oriented Transformation Language) [34]	Propuesta gráfica de la Universidad Técnica de München.
ETL (Epsilon Transformacion Language) [35]	Lenguaje definido sobre la plataforma Epsilon, plataforma desarrollada como un conjunto de plug-ins (editores, asistentes, pantallas de configuración, etc) sobre Eclipse. Dicha plataforma presenta el lenguaje de metamodelado independiente *Epsilon Object Language* que se basa en OCL, y puede ser utilizado como lenguaje de gestión de modelos o como infraestructura a extender para producir nuevos lenguajes específicos de dominio, como el citado *Epsilon Transformation Language* (ETL). Además de ETL, *Epsilon* soporta a *Epsilon Comparison Language* (ECL) y a *Epsilon Merging Language* (EML), orientados a la comparación y composición de modelos, respectivamente.
GreAT (Graph Rewriting and Transformations) [36]	Basado en la transformación de grafos. Es la propuesta de una organización independiente: ESCHER Research Institute (The Embedded Systems Consortium for Hybrid and Embedded Research).
JMI (Java Metadata Interface) [37]	Propuesta de Sun basado en MOF que permite manipulación de archivos XMI.
Kent o KMTL(Kent Model Transformation Language)[38]	Propuesta realizada por la Universidad de Kent - Reino Unido. Es la evolución de dos propuestas anteriores de la misma universidad. Es un lenguaje de especificación declarativo, con la opción de proveer partes constructivas. Se basa en los principios de QVT.
Kermeta[39]	Desarrollado por Triskell, que es un equipo de investigación de IRISA (que reúne investigadores de CNRS, Université Rennes, INRIA e INSA). Es un lenguaje que permite la definición de metamodelos y modelos, como también la definición de acciones, consultas, vistas y transformaciones. Es una extensión de EMOF (Essential Meta-Object Facilities) 2.0 para dar soporte a la definición del comportamiento.
M2T (Model to Text project) [40]	Se focaliza en la generación de artefactos textuales a partir de modelos. Consta de un conjunto de herramientas desarrolladas para la plataforma Eclipse.
Mod-Transf [41]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el DART team (INRIA).
MOFScript [31]	Lenguaje de transformación de modelo a texto implementado como un plugin para Eclipse.
MOLA (MOdel transformation LAnguage)[42]	Lenguaje gráfico para describir transformaciones propuesto por la Universidad de Letonia.
MT model transformation language [30]	Basado en QVT y desarrollado como DSL (Domain Specific Language)

	embebido por L. Tratt del King's College de Londres.
MTF (Model Transformation Framework)[43]	Es un conjunto de herramientas desarrollado por IBM que permiten hacer comparaciones, comprobar la consistencia y ejecutar transformaciones entre modelos EMF.
MTRANS [44]	Proyecto de la Universidad de Nantes - Francia. Es un framework que permite expresar transformaciones de modelos.
MTL (Model Transformation Language) [45]	Lenguaje de transformación de modelos y herramienta en Eclipse desarrollada por el grupo Triskell.
QVT (Query/View/ Transformation) [28]	Especificación estándar de OMG. Está basado en MOF (Meta Object Facility) para lenguajes de transformación en MDA. Implementaciones [46]: • QVT-Operational: o QVT Operacional de Borland© o SmartQVT o Eclipse M2M • QVT-Core: o OptimalJ (Descontinuado) o QVT-Relations: o MediniQVT o Eclipse M2M Declarative QVT o ModelMorf
RubyTL [47]	Lenguaje de transformación híbrido definido como un lenguaje específico del dominio embebido en el lenguaje de programación Ruby, y diseñado como un lenguaje extensible en el que un mecanismo de plugins permite añadir nuevas características al núcleo de características básicas.
Stratego [48]	Lenguaje de descripción de transformaciones de programas. La herramienta desarrollada como soporte del lenguaje es Stratego/XT.
Tefkat [49]	Lenguaje declarativo basado en MOF y QVT. Es el aporte de la Universidad de Queensland.
UMLX [32]	Lenguaje gráfico que extiende a UML y también a QVT.
VIATRA[50]	Forma parte del framework VIATRA2, que ha sido escrito en Java y que se encuentra integrado en Eclipse. Proporciona lenguajes textuales para describir modelos y metamodelos, y transformaciones llamados VTML y VTCL respectivamente. No se basa en los estándares MOF y QVT. No obstante, pretende soportarlos en un futuro mediante mecanismos de importación y exportación integrados en el framework.
xUML (eXecutable UML) [51]	Propuesta basada en UML para la construcción de modelos de dominio ejecutables y sus transformaciones.

1.2.2 QVT (Query/View/Transformation)

QVT (Query/View/Transformation) es un estándar para la transformación de modelos definido por el OMG. Este estándar es un híbrido de naturaleza declarativa e imperativa, donde la parte declarativa se divide en una arquitectura de dos niveles (Figura 4) constituida por un metamodelo y un lenguaje llamado *Relations*, el cual soporta concordancia de patrones de objetos complejos y plantillas de creación de objetos, y un metamodelo y un lenguaje llamado *Core*, definido usando mínimas extensiones de EMOF (Essential MOF) y OCL (Object Constraint Language):

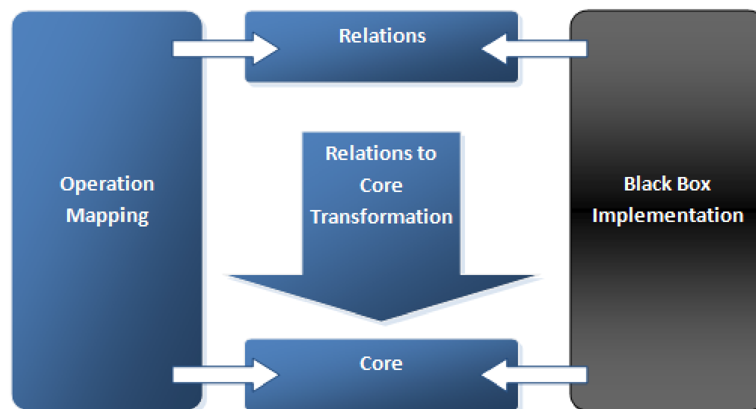

Figura 4. Arquitectura QVT

- *Relations* es una especificación declarativa de las relaciones entre modelos MOF, que ofrece la concordancia de patrones de objetos complejos, así como mecanismos para la creación implícita de clases de trazas (*registros*) y de sus instancias, con la finalidad de registrar lo que ocurre durante la ejecución de una transformación.

- *Core* es un pequeño Lenguaje/Modelo que sólo soporta concordancia de patrones con respecto a un conjunto plano de variables mediante la evaluación de condiciones definidas con esas variables y respecto de un conjunto de modelos. Este lenguaje trata todos los elementos del modelo (fuente, destino y registro) simétricamente. Tiene un potencial equivalente al del lenguaje de relaciones (como sugiere la Figura 4), pero su definición semántica es más simple. Por otra parte, los modelos de registro deben ser explícitamente definidos, no siendo deducidos implícitamente de la descripción de la transformación, como en el caso de *Relations*.

Existen dos mecanismos para invocar implementaciones imperativas de transformaciones desde *Relations* y *Core*: un lenguaje estándar llamado *Operational Mappings* y una implementación no estándar llamada *Black-box* MOF *Operation*:

- *Operational Mappings* constituye una manera estándar de proveer una implementación imperativa, la cual puebla los modelos de los mismos registros que el lenguaje *Relations*. Proporciona extensiones de OCL que permiten utilizar un estilo más procedimental y una sintaxis concreta que resulta más familiar a los programadores de los lenguajes imperativos más habituales (C++, Java, etc.). Se utiliza para implementar una o más relaciones, a partir de una especificación de relaciones, cuando la definición declarativa de éstas resulta complicada o no natural.

- *Black-box* MOF *Operation* aprovecha que *Relations* hace posible ensamblar cualquier implementación a una operación MOF si se mantiene la signatura de dicha operación. Esta característica resulta muy conveniente, ya que permite la codificación de algoritmos complejos en cualquier lenguaje de programación que tenga un puente a MOF (o que pueda ser ejecutado dentro de un lenguaje que lo tenga), permite la utilización de bibliotecas específicas de dominio para calcular los valores de las propiedades de un modelo cuando dichos cálculos resultan difíciles de expresar en OCL, y permite que las implementaciones de algunas partes de una transformación sean opacas. Sin embargo esta implementación no está carente de riesgos, ya que tiene acceso a referencia de objetos en los modelos, pudiendo realizar acciones con estos objetos que potencialmente pueden romper la encapsulación de los mismos. Además, las cajas negras introducidas por *Black-box* no tienen una vinculación implícita con *Relations*, de tal forma que cada una de estas cajas negras es responsable de mantener explícitamente las trazas entre elementos de los modelos relacionados por la implementación de la operación.

1.2.2.1 Ejemplo de transformación con QVT

Este ejemplo, tomado de [28], lleva a cabo una transformación de las clases persistentes de un modelo simple UML a tablas de un modelo simple de RDBMS (*Relational Database Management System*). La Figura 5 muestra el metamodelo UML simplificado utilizado en el ejemplo, mientras que la Figura 6 muestra el metamodelo simplificado de los modelos de datos soportados por el RDBMS.

Figura 5. Metamodelo Simple UML.

El metamodelo simple de UML (ver Figura 5) consiste en elementos que heredan de *UMLModelElement* que tiene como atributos un tipo (*kind*) que indica si el elemento es persistente y un nombre (*name*). Los paquetes (*Package*) están formados por elementos (*PackageElement*) que pueden ser asociaciones (*Association*) y clasificadores (*Classifiers*), estos últimos pueden ser clases (*Class*) o datos primitivos (*PrimitiveDataType*). Las clases están conformadas por atributos (*Atribute*) que a su vez está definido con algún tipo de clasificador (tipo de dato primitivo o clase). Las asociaciones presentes en UML son modeladas a través del elemento *Association* y la generalización a través de la conexión *general* del elemento *Class* consigo mismo.

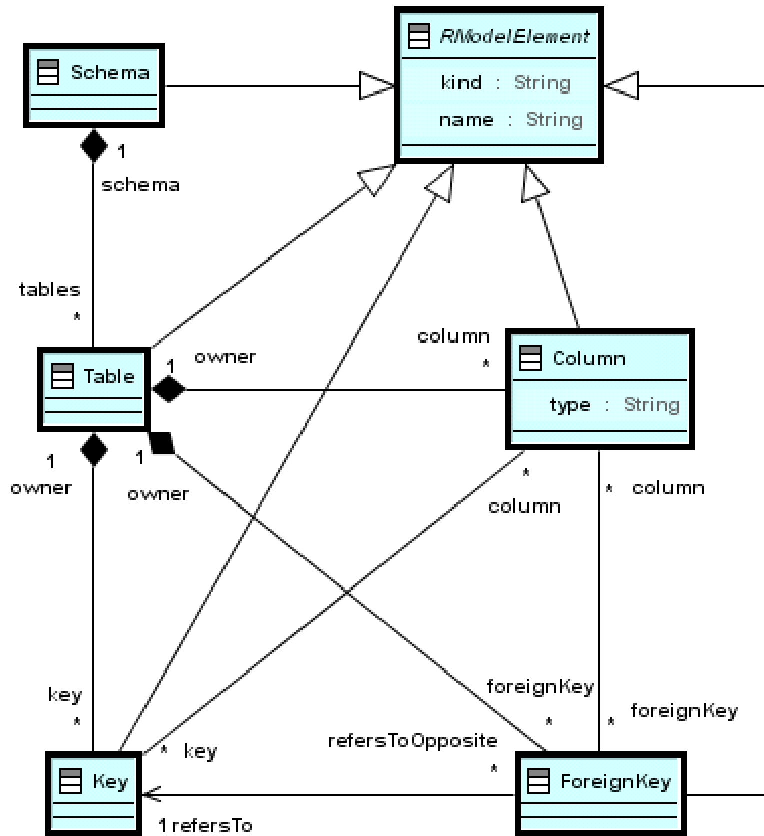

Figura 6. Metamodelo Simple RDBMS

El metamodelo simple RDBMS (ver Figura 6) consiste en elementos que heredan de *RModelElement* que tiene como atributos un tipo (*kind*) y un nombre (*name*). El elemento esquema (*Schema)* está conformado por tablas (*Table*), que a su vez se compone de columnas (*Column*) y se asocia con claves (*key*) y claves externas (*Foreignkey*). Las columnas poseen un tipo (type) y además pueden estar relacionadas con una clave y/o clave externa. Las claves externas referencian a una clase.

La descripción informal de las reglas de transformación es la siguiente:

- Cada clase persistente se mapea a: una tabla, una clave primaria y una columna identificadora.
- Los atributos de la clase persistente se mapean a columnas de la tabla.
- Cada atributo con un tipo de datos primitivos se mapea a una sola columna.
- Los atributos heredados de acuerdo con la jerarquía de clases se mapean también a columnas.
- Cada asociación entre dos clases persistentes se mapea a una relación entre las tablas correspondientes representada a través de claves externas.

En la figura 7 se muestra un fragmento de la transformación propuesta codificada en QVT. En ésta se pueden observar en la primera línea cómo se nombran los metamodelos que participaran en la transformación (*uml:SimpleUML y rdbms:SimpleRDBMS*). De igual manera, se encuentran expresadas dos relaciones o reglas: la primera realiza el mapeo de clases a tablas (*ClassToTable* – líneas 5 a 32) y la segunda realiza el mapeo de atributo a columna (*AttributeToColumn* – líneas 34 a 44).

```
1.      transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
2.      {
3.         ...
4.
5.            top relation ClassToTable
6.            {
7.                   cn, prefix: String;
8.                   checkonly domain uml c:Class {
9.                         namespace=p:Package {},
10.                        kind='Persistent',
11.                        name=cn
12.                  };
13.                  enforce domain rdbms t:Table {
14.                        schema=s:Schema {},
15.                        name=cn,
16.                        column=cl:Column {
17.                            name=cn+'_tid',
18.                            type='NUMBER'
19.                        },
20.                        key=k:Key {
21.                            name=cn+'_pk',
22.                            column=cl
23.                        }
24.                  };
25.                  when {
26.                         PackageToSchema(p, s);
27.                  }
28.                  where {
29.                         prefix = '';
30.                         AttributeToColumn(c, t, prefix);
31.                  }
32.           }
33.
34.           relation AttributeToColumn
35.           {
36.                  checkonly domain uml c:Class {};
37.                  enforce domain rdbms t:Table {};
38.                  primitive domain prefix:String;
39.                  where {
40.                         PrimitiveAttributeToColumn(c, t, prefix);
41.                         ComplexAttributeToColumn(c, t, prefix);
42.                         SuperAttributeToColumn(c, t, prefix);
43.                  }
44.           }
45.        ...
46.     }
```

Figura 7. Fragmento de la transformación para el ejemplo QVT.

A partir de una clase persistente seleccionada en las líneas 8 a 12, se puede observar en las líneas 13 a 24 la creación de una (descripción de) tabla a partir de dicha clase. La tabla poseerá una columna de tipo *'NUMBER'*, que tiene el mismo nombre de la clase concatenado con *'_tid'*

y una clave relacionada con dicha columna, que tiene el mismo nombre de la clase concatenado con *'_pk'.*

Cabe recalcar el comportamiento de las clausulas *when* y *where*, que pueden ser vistas como precondición y post-condición de la regla. La primera especifica una condición bajo la cual una relación necesita mantenerse. Por ejemplo en las líneas 25 a 27 se indica que para realizar el mapeo de clase a tabla es necesario que el paquete al que pertenece la clase sea mapeado al esquema que contiene la tabla, lo cual se realiza a través de la invocación a la relación *PackageToSchema*. La clausula *where* específica las condiciones que deben ser satisfechas por todos los elementos participantes en la relación. Por ejemplo en las líneas 28 a 30 se indica que los atributos de la clase deben ser mapeados a columnas, llamando a la relación *AttributeToColumn*; ésta ultima a su vez realiza la invocación de otros mapeos, en las lineas 39 a 43, que realizan la transformación dependiendo el tipo de atributo (simple, complejo o heredado), las cuales se encuentran fuera del fragmento mostrado. Se puede encontrar el codigo completo del ejemplo en [28].

Otro aspecto importante para observar el orden de ejecución de las reglas, es que el nivel de las reglas está determinado por la palabra clave *top*. La ejecución de una transformación requiere que las relaciones se encuentren al nivel *top*, mientras que las que no se encuentren a dicho nivel sólo se ejecutan cuando son directa o indirectamente invocadas desde una clausula *where*. Se puede observar que la relación *ClassToTable* se encuentra al nivel *top*, a diferencia de *AttributeToColumn.*

Por otra parte, QVT también propone una sintaxis grafica. La figura 8 muestra una parte de la solución del ejemplo en esta notación, donde se realiza la transformación entre clase y tabla, lo que equivaldría a la regla *ClassToTable* anteriormente expuesta.

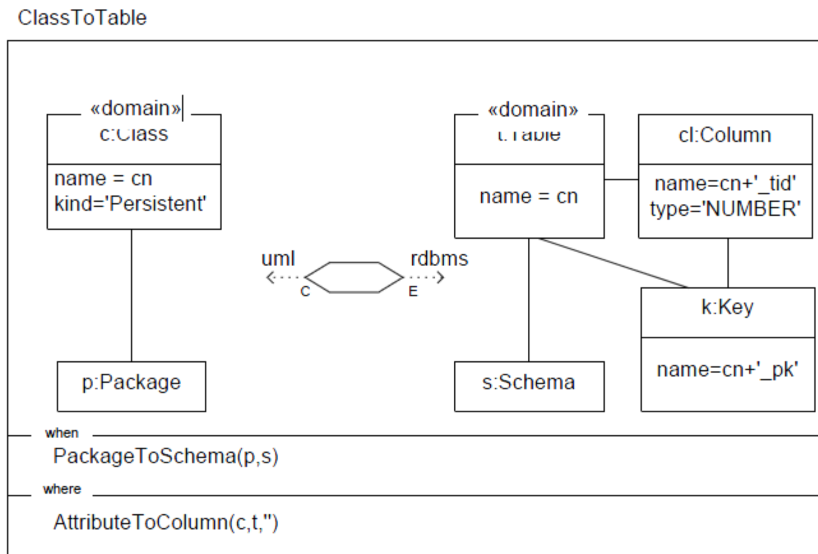

Figura 8. Relación Clase a Tabla para el ejemplo QVT.

1.2.3 ATL (Atlas Transformation Language)

ATL es un lenguaje de transformación de modelos híbrido que permite, en su definición de transformaciones, especificar construcciones declarativas e imperativas. La propuesta es del ATLAS Group del INRIA & LINA, de la Universidad de Nantes y fue desarrollada como parte de la plataforma AMMA (ATLAS Model Management Architecture).

ATL es compatible con los estándares de la OMG: es posible describir transformaciones modelo a modelo, modelos que deben conformar metamodelos definidos con MOF. Además, el lenguaje se basa en QVT (en [52] se discute el nivel de compatibilidad de ATL y QVT; tal discusión excede el alcance de este documento). También permite definir pre y postcondiciones en OCL, resultando sumamente expresivo y de fácil escritura para sus usuarios.

En cuanto a las reglas de transformación, ATL define, tal y como se ha sugerido, un dominio (metamodelo) para el origen (*source*) y otro dominio para el destino (*target*), siendo ambos instancias de MOF y con direcciones *in* y *out* respectivamente. Los metamodelos *source* y *target* pueden tener iguales o diferentes dominios, pero, aún siendo iguales, ambos deben ser claramente identificados. Si bien las transformaciones son unidireccionales, ATL permite la definición de transformaciones bidireccionales mediante la implementación de dos transformaciones, una para cada dirección.

Como características particulares del lenguaje, podemos mencionar las estructuras que define este lenguaje. En primer lugar, la definición de transformaciones forman módulos (*modules*) que contienen las declaraciones iniciales y un número de *helpers* y reglas de transformación.

Los *helpers* son una estructura intermedia dentro de las transformaciones que permiten definir operaciones y tuplas OCL, componentes que facilitan la navegación, la modularización y la reutilización. Existe también una construcción llamada *called rule* que permite representar procedimientos que pueden contener argumentos y ser invocados por nombre.

La aplicación de las reglas se realiza de forma no determinista, por equiparación de patrones, no habiéndose provisto ninguna construcción o clausula que permita aplicar en forma condicional las reglas. Cabe mencionar que la invocación de *called rules* es determinista. Esta invocación y la utilización de parámetros permiten soportar recursión. ATL además provee modularización basada en procedimientos (las ya citadas *called rules*), así como mecanismos de reutilización, ya que las reglas se pueden heredar y sus módulos pueden incluir a otros módulos.

En cuanto a la trazabilidad, este lenguaje crea un enlace de trazabilidad (*traceability link*) con la ejecución de cada regla, que se guarda en el motor de la transformación. Este enlace relaciona tres elementos: la regla, los elementos originales (*source*) y los elementos creados (*target*).

1.2.3.1 Ejemplo de Transformación ATL

El siguiente ejemplo, tomado de [53], muestra la transformación de otro modelo simplificado de UML (concretamente, el metamodelo simplificado para los modelos de clases; véase la figura 9), a otro modelo simplificado de una base de datos relacional (Figura 10).

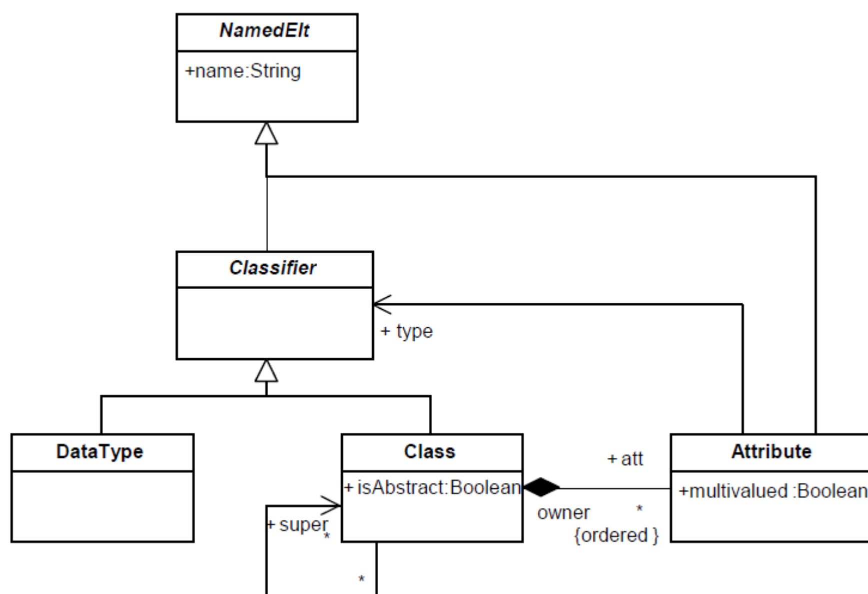

Figura 9. Metamodelo de clases.

El metamodelo de clases consiste en clases que poseen un nombre heredado de la clase abstracta *NamedElt*. La clase principal es *Class*, la cual contiene un conjunto de atributos del tipo *Attribute*. La clase *Datatype* modela el tipo de datos primitivo. *Class* y *DataType* heredan de *Classifier,* el cual sirve para declarar el tipo de *Attribute*. Los atributos pueden ser multivaluados, lo cual tiene un importante impacto en la transformación ejemplificada en esta sección.

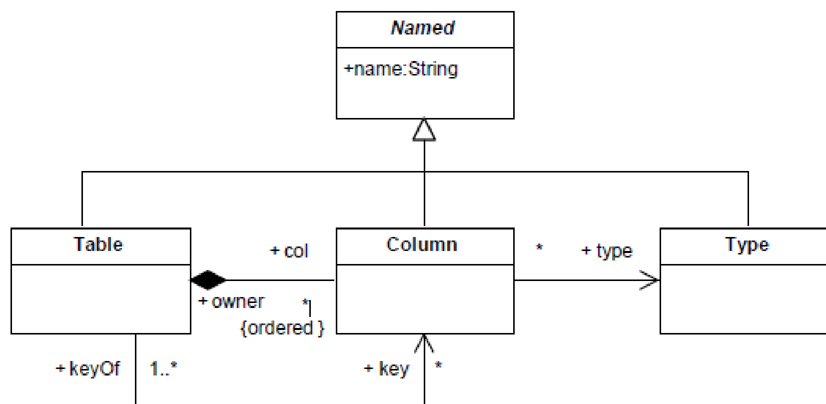

Figura 10. Metamodelo de la base de datos relacional.

El metamodelo de la base de datos relacional consta de clases que tienen un nombre, por lo que heredan de la clase abstracta *Named*. La clase principal *Table* contiene un conjunto de columnas y hace referencia a sus claves. La clase *Column* apunta a la tabla a la cual pertenece por medio de *owner* y a la tabla de la cual es completa o parcialmente la clave por medio de *keyOf*. Finalmente la columna hace referencia a un tipo.

La descripción informal de las reglas de transformación es la siguiente:

- Para cada instancia de *Class*, se debe crear una instancia de una tabla.
  - Sus nombres deben corresponder.
  - La referencia *col* debe contener todas las columnas que hayan sido creadas por cada atributo simple y también las claves.
  - Se debe crear una *key* que referencia a una columna creada con el nombre de '*objectId'* y será de tipo '*Integer'*.
- Para cada instancia de *DataType*, debe crearse una instancia de *Type* con el mismo nombre.
- Para cada instancia de atributos simples del tipo *DataType*, debe crearse una instancia de columna con el mismo nombre, y el tipo debe corresponder.
- Para cada instancia de atributos multivaluada del tipo *DataType*, debe crearse una instancia de Tabla.

    - El nombre de la Tabla es el nombre de la Clase a la que pertenece el atributo concatenado con '_' y el nombre del atributo.
    - *col* debe referenciar a las siguientes dos columnas:
        - Se debe crear una columna identificadora con el nombre de la clase del atributo concatenado con 'Id' y su tipo *'Integer'.*
        - Se debe crear una columna con el valor del atributo. Su nombre y tipo deben corresponder.
- Para cada instancia de atributos simples del tipo *Class,* una nueva columna debe ser creada, donde su nombre es el nombre del atributo concatenado con 'id' y su tipo 'Integer'.
- Para cada instancia de atributos multievaluado del tipo *Class*, debe crearse una nueva tabla:
    - El nombre de la tabla es el nombre de la clase a la cual pertenece el atributo concatenado con '_' y el nombre del atributo.
    - *col* debe referenciar a las siguientes dos columnas:
        - Se debe crear una columna identificadora con el nombre de la clase del atributo concatenado con 'Id' y su tipo *'Integer'.*
        - Se debe crear una instancia de columna como clave externa. Su nombre debe ser el nombre del atributo, concatenada con 'Id' y de tipo *'Integer'.*
- No se tienen en cuenta herencia ni el clasificador *isAbstract*.

En la figura 11 se puede apreciar un fragmento de la codificación de la transformación de ejemplo utilizando ATL. Este incluye en las líneas 1 a 3 el nombre del modulo *Class2Relational*, los metamodelos de entrada (*IN : Class*) y salida (*OUT : Relational*) de la transformación y la manera de importar ciertos datos básicos, en este caso *strings* (cadenas de caracteres). También se puede observar la definición de un *helper* (líneas 5 a 7) y dos reglas, la primera para transformar Clases a Tablas (*Class2Table* – líneas 9 al 23) y la segunda para transformar atributos simples de un tipo de dato básico (*DataType*) en columnas (SingleValuedDataTypeAttribute2Column – líneas 25 a 37).

El *helper* llamado *objectIdType* se utiliza para seleccionar el tipo del identificador de cada objeto que es creado como una clave. En el ejemplo se selecciona el tipo *Integer* mediante instrucciones OCL*,* asumiendo que este tipo está definido en el modelo fuente (*Class*). En la línea 21 se puede observar su utilización.

En cuanto a la regla *Class2Table*, primero debe crear la clave (*key)* en las líneas 19 y 22, con nombre '*objectId*' y el tipo definido por el *helper* anteriormente comentado. Luego se puede crear la tabla (líneas 13 a 18), con el mismo nombre de la clase, agregando la columna *key* creada anteriormente, sumándole todos los atributos no multivaluados, los cuales deben ser transformados implícitamente por la regla *SingleValuedData-*

TypeAttribute2Column o la regla *ClassAttribute2Column* (la cual no aparece en el fragmento) que es la encargada de transformar los atributos que son clases en columnas. Finalmente agrega la columna *key* como una clave (línea 17).

```
1.        module Class2Relational;
2.        create OUT : Relational from IN : Class;
3.        uses strings;
4.
5.        helper def: objectIdType : Relational!Type =
6.                Class!DataType.allInstances()
7.                        ->select(e | e.name = 'Integer')->first();
8.
9.        rule Class2Table {
10.               from
11.                       c : Class!Class
12.               to
13.                       out : Relational!Table (
14.                               name <- c.name,
15.                               col <- Sequence {key}->
16.                               union(c.attr->select(e | not e.multiValued)),
17.                               key <- Set {key}
18.                       ),
19.                       key : Relational!Column (
20.                               name <- 'objectId',
21.                               type <- thisModule.objectIdType
22.                       )
23.       }
24.       ...
25.       rule SingleValuedDataTypeAttribute2Column {
26.               from
27.                       a : Class!Attribute (
28.                               a.type.oclIsKindOf(Class!DataType) and not
29.                               a.multiValued
30.                       )
31.               to
32.                       out : Relational!Column (
33.                               name <- a.name,
34.                               type <- a.type
35.                               owner <- [Class2Type.key]a.owner
36.                       )
37.       }
38.       ...
```

Figura 11. Fragmento de la transformación para el ejemplo ATL.

En la regla *SingleValuedDataTypeAttribute2Column* se puede destacar la definición de una precondición a través de OCL (líneas 28 y 29), donde se seleccionan únicamente los atributos que sean de tipo *DataType* y no sean multivaluados, para que la regla sea aplicada. También mencionar que la línea 35 en el ejemplo original se encuentra comentada y se dice que es la manera explícita de usar los enlaces de rastreo implícitos (*implicit tracking links*); intuitivamente es necesaria para relacionar la columna con la tabla a la cual pertenece, de igual manera que el atributo pertenece a una clase.

El ejemplo en su totalidad posee otras reglas similares a *SingleValuedDataType-Attribute2Column* para transformar los atributos que son del tipo *Class* y son multievaluados. Puede encontrar todo el código del ejemplo en [53].

1.2.4 MT model transformation language

MT model transformation language es un lenguaje de transformaciones de modelos implementado como un DSL *embebido* [54] en el lenguaje de programación Converge, una derivación del lenguaje Python con una sintaxis extensible que facilita la creación de este tipo de DSLs embebidos. Fue desarrollado por Laurence Tratt del King's College London. Los modelos utilizados en las transformaciones por MT se pueden escribir en el DSL TM, creado por el mismo autor.

MT en parte es una derivación del enfoque QVT-Partners [55]. Este enfoque se basa principalmente en el uso de patrones (en concreto, patrones equivalentes a las expresiones regulares textuales, pero definidos sobre modelos). Sin embargo los patrones de QVT-Partners son pobres en expresividad, puede sólo equipararse contra un número fijo de elementos, y contienen defectos en los alcances de las reglas. Así mismo, QVT-Partners utiliza un lenguaje imperativo complejo para el cuerpo de las reglas. Se pueden encontrar detalles de estos inconvenientes en [56].

MT se integra de manera natural con Converge, usando patrones declarativos para realizar las correspondencias entre los elementos del modelo de una manera concisa pero poderosa, y permitiendo, a la vez, embeber dentro de las reglas código normal imperativo escrito en Converge. El hecho de que MT sea un DSL embebido, ha permitido que su diseño se haya mejorado rápidamente, haciendo posible una rápida experimentación y retroalimentación, de tal manera que se ha logrado un mayor grado de comprensión de las transformaciones de modelos y a su vez se ha conseguido un lenguaje de patrones más sofisticado y maneras más adecuadas de visualización de dichas transformaciones.

En cuanto a la modularización del lenguaje, todas las reglas se encuentran en un solo espacio de nombres, no existiendo la notación para el manejo de módulos. Sin embargo, en futuras versiones el autor tiene previsto implementar esta característica. En cuanto a la trazabilidad, MT mantiene registros que se crean automáticamente conforme se ejecutan las reglas de transformación. Cada regla ejecutada agrega una traza, que es una tupla que almacena el elemento fuente (*source*) y el elemento resultante (*target*). Sin embargo, por omisión, sólo se registran aquellos elementos equiparados por patrones de elementos no anidados. Esto significa que los elementos fuente que son almacenados en la información de traza no constituyen necesariamente el universo entero de elementos pasados como parámetros a las transformaciones.

1.2.4.1 Ejemplo de Transformación con MT

El siguiente ejemplo, tomado de [30] , muestra la transformación de un tercer metamodelo de clases UML simplificado a un tercer modelo simplificado de bases de datos relacional. En la figura 12, se muestra un modelo mucho mas simplificado de UML, diferenciado de los

anteriormente presentados porque no posee paquetes dentro de su modelo, generalización de clases, ni un elemento superior al clasificador (*Classifier*). Por otra parte, únicamente las clases poseen la propiedad *is_persistent,* las cuales serán las únicas transformadas a tablas. Así mismo, los atributos poseen una marca llamada *is_primary* que permite identificarlos como clave primaria en la transformación.

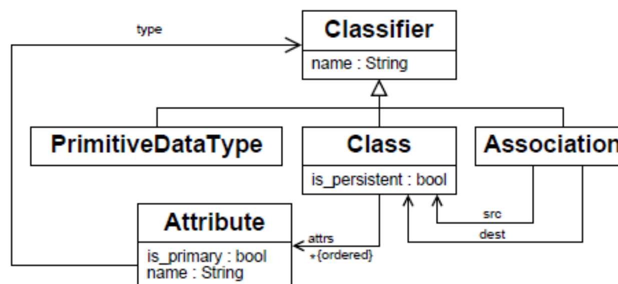

Figura 12. Metamodelo UML Simplificado.

En la figura 13 se muestra un metamodelo simplificado de bases de datos que sólo posee tablas y columnas. La tabla posee como atributos un nombre (*name*) y un conjunto de claves externas (*fkeys*). Por otra parte las columnas tienen como atributos un nombre (*name*) y un tipo (*type*) de tipo cadena de texto (*String).* La pertenencia de columnas a las tablas están dadas por la relación *col*, y una columna es designada como clave a través de la relación *pkey.*

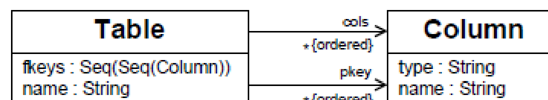

Figura 13. Metamodelo Base de datos relacional simplificado.

En contraposición con las ya vistas, esta transformación tiene las siguientes características:

- Las clases que poseen la propiedad *is_persistent,* serán transformadas a tablas.
- Las asociaciones y atributos serán transformados a columnas.
- Los atributos, cuya propiedad *is_primary* sea verdadera, serán referenciados como clave primaria en el modelo resultante; además serán referenciados como claves foráneas en las tablas relacionadas.
- Las clases no pueden ser transformadas aisladamente: todas las asociaciones para las cuales una clase es fuente deben ser consideradas en orden, de tal manera que la tabla resultante contenga todas las columnas necesarias.
- Las claves foráneas y primarias son columnas referenciadas; es importante que las columnas no se dupliquen, pero mantengan la información del modelo.

En la figura 14 se presenta un fragmento de la codificación del ejemplo utilizando el lenguaje MT. En este se muestra cómo se realiza la especificación dentro del lenguaje *Converge*. Este fragmento contiene la regla que realiza la transformación de clase a tabla (*Persistent_-Class_To_Table* - líneas 4 a 24), la regla que realiza la transformación de atributos no primarios de tipos primitivos a columnas (*Non_Primary_Primitive_Type_Attribute_To_Columns* – líneas 26 a 37 *)* y una regla por defecto (líneas 39 a 43).

```
1.          $<MT.mt>:
2.                 transformation Classes_To_Tables
3.
4.          rule Persistent_Class_To_Table:
5.                 srcp:
6.                        (Class, <c>)[name == <n>, attrs == <attrs>, \
7.                        is_persistent == 1]
8.                        (Association, <assoc>)[src == c] : * <assocs>
9.
10.                tgtp:
11.                       (Table)[name := n, cols := cols, pkey := pkeys, \
12.                       fkeys := fkeys]
13.
14.                tgt_where:
15.                       cols := []
16.                       pkeys := []
17.                       fkeys := []
18.                       for aa := (attrs + MT.mult_extract(assocs, "assoc")) \
19.                       .iterate():
20.                          a_cols, a_pkeys, a_fkeys := self.transform([""], [aa])
21.                          cols.extend(a_cols)
22.                          pkeys.extend(a_pkeys)
23.                          fkeys.extend(a_fkeys)
24.         ...
25.         rule Non_Primary_Primitive_Type_Attribute_To_Columns:
26.                srcp:
27.                       (String, <prefix>)[]
28.                       (Attribute)[name == <attr_name>, type == \
29.                       (PrimitiveDataType)[name == <type_name>], \
30.                       is_primary == 0]
31.
32.                tgtp:
33.                       [(Column)[name := concat_name(prefix, attr_name), \
34.                              type := type_name]]
35.                       []
36.                       []
37.         ...
38.         rule Default:
39.                srcp:
40.                       (MObject)[]
41.
42.                tgtp:
43.                       null
```

Figura 14. Fragmento de la transformación para el ejemplo MT.

En la primera regla *Persistent_Class_To_Table* se pueden observar los bloques que pueden conformar una regla escrita en MT: el demarcado por *scrp,* donde se especifica la fuente de la regla – en este caso las clases persistentes y las Asociaciones (líneas 5 a 8) -, el demarcado por *tgtp,* donde se especifica el objetivo – en este caso las tablas (líneas 10 a 12) -, y el demarcado por *tgt_where,* donde se incluyen las post-condiciones; en el ejemplo se invocan las

transformaciones de atributos y asociaciones a columnas, claves primarias y externas (líneas 14 a 23). También puede existir un cuarto bloque en las reglas, que en el ejemplo no es utilizado, marcado de igual forma con *srcp_when*, donde se podrían incluir precondiciones. Se puede evidenciar también el uso del lenguaje *converge* dentro de las reglas de manera natural; claro ejemplo de este uso es la iteración presente en el bloque *tgt_where* mediante un ciclo *for* (línea 18), que en *converge* adopta la forma usual utilizada en los lenguajes imperativos.

En la regla *Non_Primary_Primitive_Type_Attribute_To_Columns*, no se tienen precondiciones ni post-condiciones. Nótese cómo se realiza su invocación en la línea 20, con la variable *'aa'* como atributo y un prefijo "", de tal manera que dicho atributo ejecutará la regla si éste no es primario (línea 30). Del mismo modo se intentan ejecutar las otras reglas que aparecen en el ejemplo. La codificación completa se encuentra en [30] .

Se puede también recalcar el uso de la regla por defecto (*rule Default*), que toma los elementos que no han encajado en ninguna otra regla y los transforma en elementos nulos. Esta regla puede ser útil para realizar información de trazado y evitar posibles excepciones en la transformación.

1.2.5 MOFScript

MOFScript es un lenguaje de transformación de modelo a texto presentado actualmente como candidato en el proceso OMG RFP. Ha sido desarrollado por la comunidad de desarrollo SINTEF, soportado y probado por el proyecto europeo de integración MODELWARE. Este lenguaje presta particular atención a la manipulación de texto y al control e impresión de salida de archivos. Es especialmente útil para realizar implementaciones dependientes de plataforma, como, por ejemplo: (i) la creación de código fuente en lenguajes como Java o C# a partir de modelos UML, o (ii) la generación automática del código de creación de una base de datos mediante SQL a partir de un Modelo de base de datos relacional. También puede ser utilizado para crear documentación o transformaciones de modelos a lenguajes textuales como XML o HTML.

MOFScript se basa en otros estándares de la OMG: es compatible con QVT y MOF para el modelo de entrada (el modelo objetivo –*target*- siempre es texto). Para las reglas de transformación se define un metamodelo de entrada (*source*) sobre el cual operarán las reglas. El target es generalmente un archivo de texto (o salida de texto por pantalla). Existe una implementación como un plugin para eclipse: MOFScript tools.

Las transformaciones son siempre unidireccionales, y no es posible definir pre y post condiciones para ellas. La separación sintáctica resulta clara por la misma definición de reglas, lo que hace al lenguaje legible. No provee estructuras intermedias, pero sí la parametrización necesaria para la invocación de reglas.

En cuanto a la aplicación de reglas, éstas se aplican en forma determinista y en orden secuencial. Se provee aplicación condicional e iteración de reglas. Las condiciones de aplicación se expresan en cláusulas *when*, como definición de guardas. La iteración se realiza mediante los bucles *for each* y *while*.

MOFScript no organiza las reglas en módulos propiamente dichos. Ahora bien, en la definición de una regla se pueden invocar a otras reglas, utilizando incluso parámetros, con lo cual las reglas en sí pueden entenderse como mecanismos básicos de modularización. También es posible definir jerarquías de transformaciones.

Finalmente, para trazabilidad, MOFScript define (pero no implementa aún) un conjunto de conceptos para relacionar elementos fuente con sus ubicaciones en los archivos de texto generados en el target. En cuanto a la composición de transformaciones (combinación de reglas para obtener una nueva regla), no se contempla expresamente.

1.2.5.1 Ejemplo de transformación con MOFScript

Este ejemplo, tomado de [57], lleva a cabo una transformación simplificada de un modelo en UML a código en lenguaje Java: el resultado esperado es una serie de ficheros con las clases codificadas.

En la figura 15, se encuentra un fragmento del código que realiza la transformación descrita. Este incluye en la primera línea la declaración del nombre del modulo y el metamodelo de entrada (*uml:uml2*). Luego incluye la declaración de algunas constantes (líneas 3 a 6), el método principal de la transformación que indica el inicio de la ejecución de la transformación (8 a 12), un método que realiza el mapeo de paquetes (líneas 14 a 20) y uno que lo hace para las clases (22 a 51).

Las constantes utilizadas para la transformación son la ubicación raíz donde serán almacenados los ficheros, el nombre del paquete y la ruta del mismo, y la extensión de los ficheros, respectivamente en orden de aparición.

La regla principal (*main*) hace un recorrido por todos los paquetes que existan en el modelo, e invoca a la regla que realiza su transformación *mapPackage*; nótese que ésta se define para los paquetes del metamodelo UML (*uml_package* – línea 14) y su invocación se realiza a través de elementos de este tipo (líneas 10 y 18). De igual manera se realiza la definición (línea 22) y la invocación a la regla *mapClass* (línea 16) definida para las clases del metamodelo UML (*uml_Class*).

La regla *mapPackage* lleva a cabo un recorrido por todas las clases que contiene el paquete e invoca a la regla que realiza la transformación de éstas (*mapClass*). Luego, recursivamente, recorre los paquetes que dicho paquete contiene, y ejecuta este mismo método de mapeo (línea 18).

```
1.    textmodule UML2Java (in uml:uml2)
2.
3.    property rootdir = "c:/tmp2/"
4.    property package_name = "org.sintef.no"
5.    property package_dir = "org/sintef/no/"
6.    property ext = ".java"
7.
8.    uml.Model::main () {
9.            self.ownedMember->forEach(p:uml.Package) {
10.                   p.mapPackage()
11.           }
12.   }
13.
14.   uml.Package::mapPackage (){
15.           self.ownedMember->forEach(c:uml.Class)
16.                   c.mapClass()
17.           self.ownedMember->forEach(p:uml.Package) {
18.                   p.mapPackage()
19.           }
20.   }
21.
22.   uml.Class::mapClass(){
23.           file (rootdir + package_dir + self.name + ext)
24.           self.classPackage()
25.           self.standardClassImport ()
26.           self.standardClassHeaderComment ()
27.           <% public class %> self.name <% extends Serializable { %>
28.           self.classConstructor()
29.           <%
30.           /*
31.           * Attributes
32.           */
33.           %>
34.           self.ownedAttribute->forEach(p : uml.Property | p.association = null) {
35.                   p.classPrivateAttribute()
36.           }
37.           nl(2)
38.           self.ownedAttribute->forEach(p : uml.Property | p.association != null){
39.                   p.classPrivateAssociations()
40.           }
41.           nl(2)
42.           self.ownedAttribute->forEach(p : uml.Property | p.association = null)
43.                   p.attributeGettersSetters()
44.           nl(2)
45.           self.classAssociationMethods()
46.                   self.ownedAttribute->forEach(assoc : uml.Association) {
47.           }
48.
49.           <%
50.           } // End of class %> self.name
51.   }
52.   ...
```

Figura 15. Fragmento de la transformación para el ejemplo MOFScript.

En cuanto a la regla *mapClass*, se puede resaltar la invocación a la función *file* con el nombre del fichero a donde se destinará el resultado de la transformación (línea 23), además de la utilización de las marcas <% … %> para escribir directamente en el archivo frases textuales; por ejemplo, en la línea 27 se escribe una línea de texto con la sintaxis de especificación de una clase Java, combinando las palabras reservadas (*public, class, etc.)* con el nombre de la clase UML especificada en el modelo de entrada de la transformación. En el ejemplo también se

utilizan estas marcas para escribir comentarios o documentación del código generado. Otro comando común es *nl,* el cual realiza tantos saltos de línea como el parámetro que recibe.

Por otra parte, dentro de esta ultima regla, se invocan a otras reglas definidas para elementos UML que no se encuentran en el fragmento; por ejemplo la escritura de los atributos utilizando las propiedades de las clases del metamodelo UML (*uml_Property*::*class-PrivateAttribute* – Linea 35) y la creación de sus métodos accesores/modificadores (*uml_Property*::*attributeGettersSetters* – Linea 43). Así mismo, también se invocan otras reglas definidas para el mismo módulo mediante el elemento *self*, lo que proporciona orden y modularidad al código; por ejemplo, se puede observar la invocación de la función *classPackage* (línea 24), encargada de llevar a cabo la escritura en el fichero para definir el paquete al cual pertenece la clase. Puede encontrase el código completo del ejemplo en [57].

1.2.6 UMLX

UMLX es un lenguaje gráfico de transformaciones entre modelos (M2M) y que se basó en extensiones mínimas a UML. Es una propuesta de E. Willink, del GMT Consortium. Su implementación se ha realizado sobre Eclipse. En un último avance, se ha anunciado que la transformación textual que la herramienta UMLX genera desde la notación gráfica, puede también editarse. Como UMLX se traduce, en gran medida, a lenguaje OCL, la notación textual resulta muy conveniente para aquellos que conocen este lenguaje, omitiendo la necesidad de aprender uno nuevo.

La definición de este lenguaje se basó en QVT. Asimismo, los modelos participantes en las transformaciones deben ser instancias de MOF. Cada uno de estos juega un rol diferente (uno de entrada *in* y otro de salida *out*), aunque se trate de los mismos modelos. Para estos no pueden especificarse pre o post condiciones.

Figura 16. Interfaces de entrada y salida de las transformaciones.

En cuanto a las reglas de transformación, UMLX utiliza diferentes íconos gráficos para las transformaciones, reglas y relaciones de creación, preservación y eliminación de elementos. Cabe destacar que la semántica de los íconos del lenguaje no se ha definido formalmente, y aunque es gráficamente intuitivo, para algunas relaciones, no queda claro cuál es su significado.

En la figura 16 se muestra un ejemplo que asocia un puerto de entrada con el nombre *from* con una instancia de *Class* llamada input, y un puerto de salida con el nombre *to* con una instancia de *Class* llamada output.

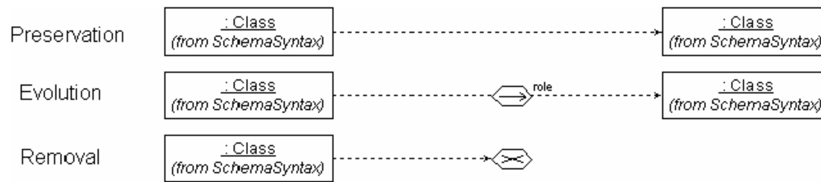

Figura 17. Operaciones de transformación.

Las operaciones definidas en UMLX son: *Evolution*, que crea una nueva instancia en el modelo resultante (RHS - Right Hand Side) de la transformación independiente del original (LHS - Left Hand Side); *Preservation*, que indica que la jerarquía de LHS es preservada en RHS; y *Removal*, que elimina de RHS la instancia representada en LHS.

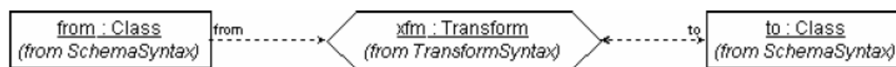

Figura 18. Invocación de una transformación.

Las estructuras que define este lenguaje son bastante simples e intuitivas, pero carecen de una semántica bien definida. No hay módulos, estructuras intermedias ni cláusulas de iteración o condición. Sin embargo, sí introduce el concepto de *capas*, que permite agrupar un diagrama completo en un único símbolo, logrando capas más abstractas y facilitando la reutilización. Por otra parte, también introduce herencia, lo que facilita que unos diagramas puedan usar comportamiento de otros. Existen variables locales, para indicar que se hace referencia a una instancia en particular; por ejemplo, si se define una regla para todas las clases, y se define una regla también para cada atributo, al hacer referencia a una instancia de clase particular (cl, por ejemplo), se pueden mencionar a todos los atributos de cl, notándolo con @cl.

La aplicación de las reglas se realiza de forma no determinista, por concordancia de patrones de reglas, y no se ha provisto ninguna construcción o cláusula que permita aplicar en forma condicional las reglas. El lenguaje tampoco contempla mecanismos de trazabilidad.

1.2.6.1 Ejemplo de transformación con UMLX

Este ejemplo, tomado de [58], se basa, nuevamente, en la transformación de un modelo expresado usando la sintaxis del diagrama de clases UML (ver Figura 19) hacia un esquema de base de datos (ver Figura 20) usando tablas y columnas, similares a los presentados en el ejemplo de QVT.

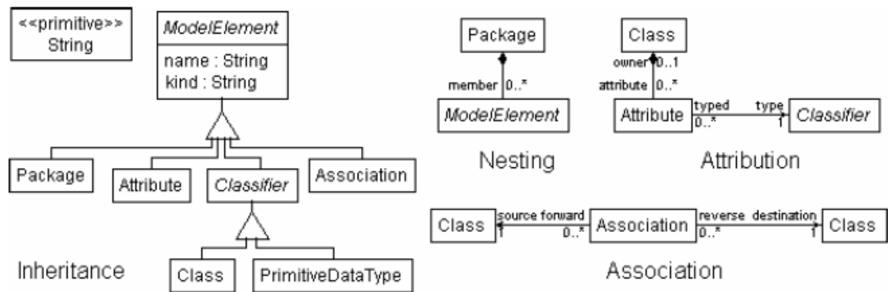

Figura 19. Metamodelo UML simplificado.

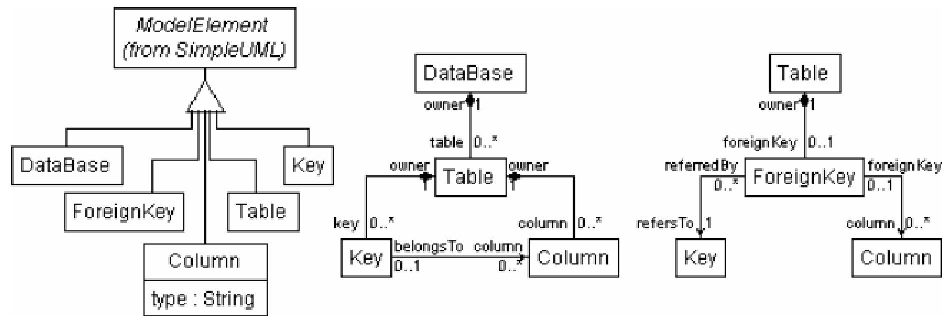

Figura 20. Metamodelo RDBMS simplificado.

Esta transformación tiene las siguientes características:

- Las clases se mapean a tablas.
- Los atributos de tipos primitivo se mapean a columnas de tablas.
- Las asociaciones se mapean a claves externas de las tablas correspondientes al origen de dichas asociaciones.
- Las claves externas referencian a una clave primaria en la tabla correspondiente al destino de la asociación.

La solución está comprendida por 11 transformaciones/reglas, de las cuales dos generalizan (heredan) de una transformación abstracta como se muestra en el paquete Uml2Rdbms (ver Figura 21).

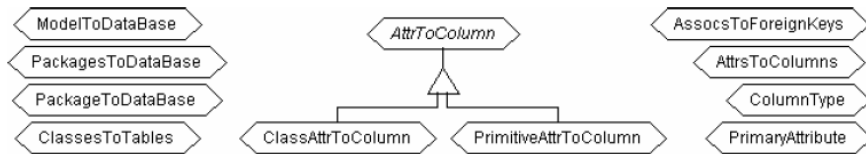

Figura 21. Paquete Uml2Rdbms

En la figura 22 se presenta la regla que transforma clases a tablas. Se puede observar la invocación a la regla que realiza la transformación de atributo a columna (*AttrsToColums*) utilizando el parámetro *prefix* con valor 'c'. También se muestra la creación de una tabla por clase (*tableForClass*) y una clave por clase (*keyForClass*) utilizando el conector/símbolo de evolución. También es notorio, en el LHS, cómo se realiza la selección únicamente de las clases persistentes (*kind = 'persistent'*). En el RHS se puede observar cómo se construyen los nombres de las tablas resultantes, concatenado un prefijo concatenado con '_' y el nombre de la clase (*name = prefix + '_' + class.name*), y de las claves concatenando 'k_' con el nombre de la clase (*name = 'k_' + class.name*). Nótese que la entrada a la regla es un paquete (*Package*), pero la Base de datos (*DataBase*) también se obtiene a partir de la invocación a la regla, y fue creada en una capa superior (*Uml2Rdbms.ModelToDataBase*).

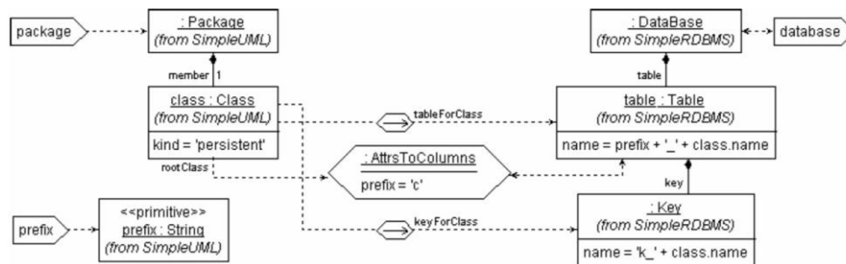

Figura 22. Uml2Rdbms.ClassesToTables

En la figura 23 se muestra la invocación a la regla genérica *AttrToColumn* para transformar los atributos en columnas. Dependiendo del tipo de atributo se ejecutarán las reglas que son una generalización de ésta: *ClassAttrToColumn* si el atributo es una clase o *PrimitiveAttrToColumn* si el atributo es un dato de tipo primitivo. Nótese, también, que la tabla fue creada en la capa superior (ver Figura 22. Uml2Rdbms.ClassesToTables). Puede encontrarse todas las reglas del ejemplo en [58].

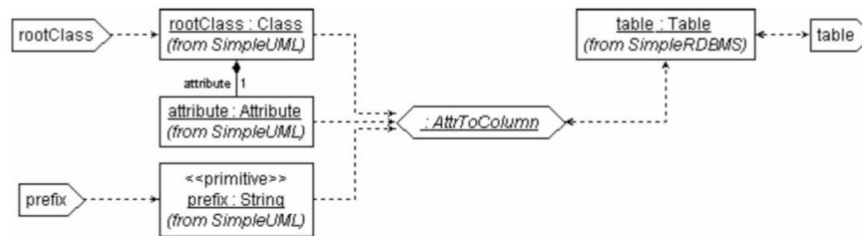

Figura 23. Uml2Rdbms.AttrsToColumns.

1.2.7 Comparativa y discusión

En la tabla 2 se puede encontrar una comparativa de las principales características de los lenguajes de transformación de modelos analizados.

Como se ha podido apreciar en este apartado, existe una variedad muy amplia de lenguajes de transformación, donde cada uno posee características deseables al igual que inconvenientes: los lenguajes basados en QVT como ATL tienen la ventaja de ceñirse a un estándar, lo cual los hace mas compatibles con otras tecnologías y comprensibles, dada la formalidad de su especificación, pero pueden carecer de expresividad y verse limitados por su naturaleza cerrada. En cuanto a los lenguajes implementados como DSLs embebidos, como MT, ofrecen la ventaja de aprovechar todo el potencial del lenguaje *host*, sumándole la expresividad y sofisticación del aparato transformacional. No obstante, esta característica también los hace poco compatible con otras herramientas que se basen en el los estándares de la OMG (QVT, OCL, MOF, etc). Por otra parte, los lenguajes gráficos como UMLX son sumamente intuitivos pero a la vez limitados por la misma conformación de la simbología del lenguaje. Conjuntamente, la descripción de transformaciones demasiado complejas puede resultar engorrosa para su utilización. Además suelen incurrir en costos computacionales más elevados en comparación con lenguajes textuales, debido especialmente a que su representación es más compleja de procesar (por su naturaleza grafica) y sus estructuras internas, que también son grafos, son más complejos de tratar que los árboles, estructura interna manejada comúnmente durante el procesamiento de los lenguajes textuales. Finalmente los lenguajes de transformación de Modelo a Texto son bastante útiles para realizar implementaciones específicas de plataforma a partir de un modelo independiente de plataforma. Sin embargo, dejan de lado la transformación entre modelos, parte fundamental del desarrollo de software dirigido por modelos. En cualquier caso, actualmente muchas de las propuestas analizadas se encuentran en etapa de desarrollo, lo cual seguramente permitirá observar, en el futuro, mejoras en cada una de ellas.

Tabla 2. Resumen Comparativo de los lenguajes estudiados.

Características	ATL	MOFScript	MT	UMLX
Tipo	Textual/ M2M	Textual/ M2Text	Textual/ M2M	Grafico/ M2M
Estilo	Declarativo y operacional.	Declarativo y operacional.	Declarativo y operacional.	Declarativo y operacional.
Pre y post condiciones	Si (OCL)	No	Si (Converge)	No
MOF/QVT compatible	Si	Si	Parcial	Si
Orden Aplicación de reglas deterministico.	No	Si	-	No
Modularización	Si	No	No	Si (Capas)
Trazabilidad	Si	Si	Si	No
Reusabilidad de Reglas.	Si	Si	Si	Si

1.3 Enfoques gramaticales a la transformación de modelos

Algunas propuestas que realizan transformación de modelos han utilizado formalismos provenientes del campo de las gramáticas formales, en especial las gramáticas de grafos y las gramáticas de atributos. En este capítulo se dará una descripción de estos formalismos y se presentará su utilización dentro de diferentes propuestas.

1.3.1 Conceptos básicos

La Gramática es definida comúnmente como el estudio de las reglas y principios que regulan el uso de las lenguas y la organización de las palabras dentro de una oración. Las gramáticas formales aparecen en lingüística computacional primordialmente para definir la sintaxis de cada lenguaje de programación; consecuentemente han sido utilizadas para guiar la construcción de traductores de lenguajes computacionales, en especial compiladores [6]. En teoría de la informática y en matemática, las gramáticas formales definen lenguajes formales [59]. Es evidente, por tanto, que en la creación de un lenguaje formal de transformación de modelos se pretenda utilizar este concepto, ya que los modelos fuente y resultante de una transformación pueden verse como sentencias o frases de lenguajes formales (caracterizados por los respectivos metamodelos)

En este apartado se presentará la noción de gramática formal y se resaltarán las principales características de los conceptos de gramática de atributos y gramática de grafos, los cuales han sido utilizados satisfactoriamente en las transformaciones de modelos.

1.3.1.1 Gramáticas

Las gramáticas consisten, principalmente, de un conjunto de *reglas de producción*, con las cuales se puede crear o producir cualquier expresión dentro de un lenguaje (por eso el nombre de *producción*). A modo de ejemplo, considérese la gramática mostrada en la figura 24, esta es una gramática simplificada para un subconjunto del lenguaje español, en la cual *FNominal* denota una frase nominal y *FVerbal* una frase verbal, y los corchetes [ ] indican partes opcionales.

```
Oración -> FNominal FVerbal [Complemento]
FNominal -> [Articulo] Sustantivo [Adjetivo]
FVerbal -> Verbo [Adverbio]
```

Figura 24. Ejemplo de una gramática.

El conjunto de reglas de producción de una gramática también puede usarse para determinar si una expresión pertenece al lenguaje especificado por la gramática. Esto se realiza encontrando una regla que pueda ser aplicada a una parte de la expresión y luego tratando de encontrar reglas que puedan coincidir con el resto. Para el ejemplo anterior se puede observar que la oración *"El Sol brilla en el cielo"* puede ser reconocida como *"Articulo Sustantivo Verbo Complemento"*, cadena que, a su vez, puede transformarse a *"FNominal FVerbal Complemento"*, y, finalmente, a *"Oración".* Este proceso de reconocimiento se denomina *análisis sintáctico* (*parsing)*.

Formalmente, una gramática es una tupla $(\sum_N, \sum_T, P, S)$, donde:

- $\sum_N$: Es un conjunto finito de símbolos no terminales o variables.
- $\sum_T$: Es el alfabeto o conjunto de símbolos terminales.
- P : Es una relación binaria entre elementos de $(\sum_T \cup \sum_N)^*$, el conjunto de todas las posibles cadenas de terminales y no terminales (incluyendo la cadena vacía). Cada par en esta relación se corresponde con una regla de producción. Comúnmente se utiliza la notación E1->E2 para representar la regla (E1,E2) en P.
- $S : S \in \sum_N$, es el símbolo de inicio o *axioma* de la gramática.

El lingüista Noam Chomsky [60] ha identificado cuatro clases de gramáticas, donde cada clase aplica restricciones en la forma de las reglas de producción. La clasificación es la siguiente:

1. Gramáticas recursivamente enumerables: No pone restricciones en las reglas de producción.
2. Gramáticas sensibles al contexto: La restricción en las reglas de producción es que el número de elementos en la cadena de la izquierda debe ser menor o igual que el número de elementos en la cadena del lado derecho (por ejemplo la regla *xAzB -> yxyzB*, donde *A* y *B* son no terminales y *x,y,z* son terminales).

3. Gramáticas libres de contexto: Todas las reglas de producción tienen un único no terminal en el lado izquierdo (por ejemplo la regla *A -> xyBz*).
4. Gramáticas regulares: Todas las reglas de producción deben ser de la forma *A->wB* o *A->Bw,* donde *A* y *B* son no terminales, y *w* es una cadena de terminales, incluida la cadena vacía.

Se puede observar que cada tipo de gramática incluye a las posteriores. Por ejemplo todas las gramáticas regulares son también sensibles a contexto, pero no al contrario, no todas las gramáticas sensibles de contexto son regulares.

De los tipos de gramáticas introducidos anteriormente, las gramáticas libres de contexto son especialmente relevantes. Estas gramáticas permiten estructurar las expresiones de un lenguaje en forma de árboles: los *árboles de análisis sintáctico*. Un árbol de análisis sintáctico o árbol de derivación es aquel cuyos nodos representan terminales y no terminales de la gramática, donde el nodo raíz es el símbolo de inicio y los hijos de cada nodo no terminal son los símbolos que reemplazan a ese no terminal en la derivación. Este posee las características – como todo árbol- de ser un grafo dirigido acíclico en el cual cada nodo se conecta con un nodo distinguido, llamado nodo raíz, mediante un único camino. En un árbol de análisis sintáctico ningún símbolo terminal puede ser nodo interior del árbol, ni ningún símbolo no terminal puede ser una hoja. El anteriormente citado proceso de análisis sintáctico (*parsing*), cuando está guiado por una gramática libre de contexto, es el encargado de convertir la cadena de entrada en dicha estructura, que es más útil para el posterior procesamiento, ya que explicita la jerarquía implícita de la entrada. En la figura 25 se muestra el árbol de análisis sintáctico para el ejemplo presentado anteriormente y la gramática que aparece en la figura 24.

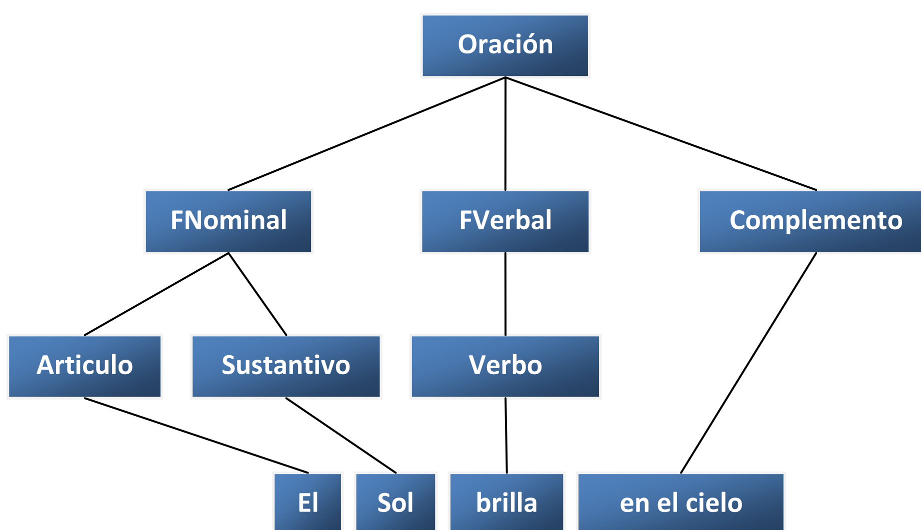

Figura 25.Ejemplo árbol de análisis sintáctico. Las aristas deben considerarse arcos dirigidos, que parten del nodo en el nivel superior (el nodo padre), y van a los nodos en el nivel inferior (los nodos hijo). Los nodos hijo se suponen, así mismo, ordenados, por orden de aparición de izquierda a derechas.

Aunque las gramáticas conciben lenguajes como conjuntos de cadenas de símbolos concatenados unos detrás de los otros, es posible utilizar formalismos gramaticales para caracterizar otro tipo de lenguajes. De estos, quizá los más relevantes son los de las *gramáticas de grafos*, que se describen brevemente en la sección *1.3.1.3.*

1.3.1.2 Gramática de Atributos

Las gramática de atributos propuestas a fines de los 60 por Knuth [61] presenta la forma de especificar la sintaxis de los lenguajes en términos de gramáticas libres del contexto, mientras que la semántica se expresa asociando *atributos* con los símbolos gramaticales, y *reglas semánticas* con las producciones que, convenientemente aplicadas, permiten computar los valores de dichos atributos en los árboles de derivación. El significado de las cadenas se obtiene, entonces, a partir de los valores de los atributos en las raíces de sus árboles de derivación y el cómputo de los valores para los atributos en los nodos de dicho árbol, mediante las reglas semánticas apropiadas.

Los atributos asociados a un símbolo se clasifican en *atributos sintetizados* y *atributos heredados*. La diferencia radica en que los atributos sintetizados de un nodo se computan a partir de atributos en los nodos hijo y los atributos heredados se computan a partir de atributos en el nodo padre y/o en los hermanos. Los atributos además pueden tener cualquier estructura: arboles, grafos o estructuras aún más complejas.

Las gramáticas de atributos han sido utilizadas para describir transformaciones dirigidas por la sintaxis, y para especificar y establecer la semántica de lenguajes de programación. Así mismo, han tenido importantes aplicaciones en entornos de programación y compiladores [62].

1.3.1.2.1 Evaluación de Atributos

Durante la definición de una gramática de atributos no es necesario especificar explícitamente en qué orden tienen que aplicarse las ecuaciones semánticas para calcular los valores de los atributos en los árboles sintácticos (es decir, para *evaluar* dichos atributos). El orden de evaluación se deriva de las *dependencias* entre los atributos introducidas por las ecuaciones semánticas. El método de evaluación en sí puede ser *estático* o *dinámico*. Los métodos estáticos analizan la gramática durante la generación del evaluador para encontrar un orden de evaluación que funciona para cualquier sentencia. Los métodos dinámicos deciden el orden de evaluación para cada sentencia particular, de tal manera que aceptan una clase más amplia de gramáticas de atributos que los estáticos, aun que consta de una ligera pérdida de eficiencia.

Los métodos de evaluación dinámicos se basan, normalmente, en el análisis de las interdependencias entre los atributos sintetizados y heredados en los nodos de cada árbol de análisis sintáctico concreto[59]. Dichas interdependencias se pueden representar mediante un grafo dirigido llamado *grafo de dependencias.* En el caso más simple, los grafos de

dependencias de atributos no contienen ciclos y el orden de evaluación es sencillamente cualquier orden topológico del grafo. Sin embargo una gramática de atributos puede generar grafos de dependencias de atributos cíclicos: este tipo de gramáticas se denominan *circulares*.

Un método de evaluación simple que funciona para gramáticas no circulares arbitrarias, y que también puede adaptarse para manejar gramáticas circulares (aunque dicha adaptación no la consideraremos aquí), es el denominado método de evaluación *bajo demanda* [63]. En dicho método, cada atributo en el árbol de análisis tiene asociado un campo para contener su valor, así como un *flag* que indica si el atributo ha sido o no computado previamente. De esta forma, cuando se solicita el valor de un atributo:

- Si éste ha sido ya computado, se devuelve directamente.
- En otro caso, se solicitan los valores de los atributos de los que éste depende, se calcula el valor aplicando la expresión especificada en la correspondiente ecuación semántica, se almacena dicho valor, de forma que pueda devolverse en futuras solicitudes, se activa el *flag* de indicación de que el atributo ya ha sido computado, y se devuelve el valor.

1.3.1.2.2 Ejemplo de una Gramática de atributos

Para ejemplificar el uso del formalismo, se presenta a continuación una gramática de atributos que es capaz de calcular el número de listas que aparecen en una sentencia. Una lista consta de un paréntesis de apertura, seguida por una secuencia de 0 o más listas, seguida por un paréntesis de cierre, Así, por ejemplo:

- ( ) , contiene una lista
- ((( ))), contiene 3 listas
- (( ) (( ))), contiene 4 listas
- (( ) ( ) ( ) ( )), contiene 5 listas

En la figura 26 se presenta una posible gramática de atributos que formaliza la tarea planteada. La figura 27 muestra el árbol de análisis sintáctico y el grafo de dependencias asociado para la sentencia (() (())). Se puede observar la utilización de los atributos en los símbolos no terminales: <*lista*> posee el atributo sintetizado *num_listas_totales* y <*lista_interna*> posee el atributo heredado *num_listas_antes* y el sintetizado *num_listas_despues.* Estos atributos se utilizan en las reglas semánticas que acompañan a cada regla de producción, reglas que se encierran entre llaves *"{}"*. Dichas reglas semánticas indican, de esta forma, los cálculos necesarios a realizar para calcular los valores de los atributos sintetizados de la parte izquierda de la producción, y de los atributos heredados de los símbolos en la parte derecha. Nótese la utilización de subíndices para distinguir entre las diferentes ocurrencias de un mismo tipo de no terminal. Más concretamente:

- La gramática libre de contexto subyacente establece que: (i) una lista está formada por un paréntesis de apertura, seguido de una *lista interna*, seguida de un paréntesis de cierre (producción en línea 1); (ii) una lista interna está formada por otra lista interna, seguida de una lista (línea 5), o bien por nada (línea 12; el símbolo λ representa la cadena vacía).

- El atributo *num_listas_antes* de <*lista_interna*> lleva cuenta del número de listas contabilizadas hasta el momento. De esta forma: (i) a nivel de la lista global interna de la lista se habrá contabilizado ya una lista (la propia lista global; línea 2), (ii) para listas internas formadas por una lista interna *li*, seguida de una lista *l*, a nivel de *li* se habrán contabilizado el número de listas contabilizadas a nivel de la lista interna global (línea 6), mientras que a nivel de *l* habrá que sumar 1 a las listas contabilizadas tras de procesar *li* (líneas 7 y 8).

- Por su parte, cuando la lista interna no tiene elementos (línea 12), el valor de *listas_despues* será, directamente, el valor de *listas_antes* (línea 13). Por su parte, para listas internas no vacías, el valor de *listas_despues* será el valor de *listas_despues* en la última lista (líneas 9 y 10).

```
1.  <lista> -> ( <lista_interna> ) {
2.     <lista_interna>.num_listas_antes = 1
3.     <lista>.num_listas_totales=<lista_interna>.num_listas_despues
4.     }
5.  <lista_interna> ->   <lista_interna> (<lista_interna>) {
6.    <lista_interna>1.num_listas_antes = <lista_interna>0.num_listas_antes
7.    <lista_interna>2.num_listas_antes=
8.                      <lista_interna>1.num_listas_despues+1
9.     <lista_interna>0.num_listas_despues=
10.                  <lista_interna>2.num_listas_despues
11.   }
12. <lista_interna> -> λ {
13.    <lista_interna>.num_listas_despues=<lista_interna>.num_listas_antes
14. }
```

Figura 26. Ejemplo gramática de atributos.

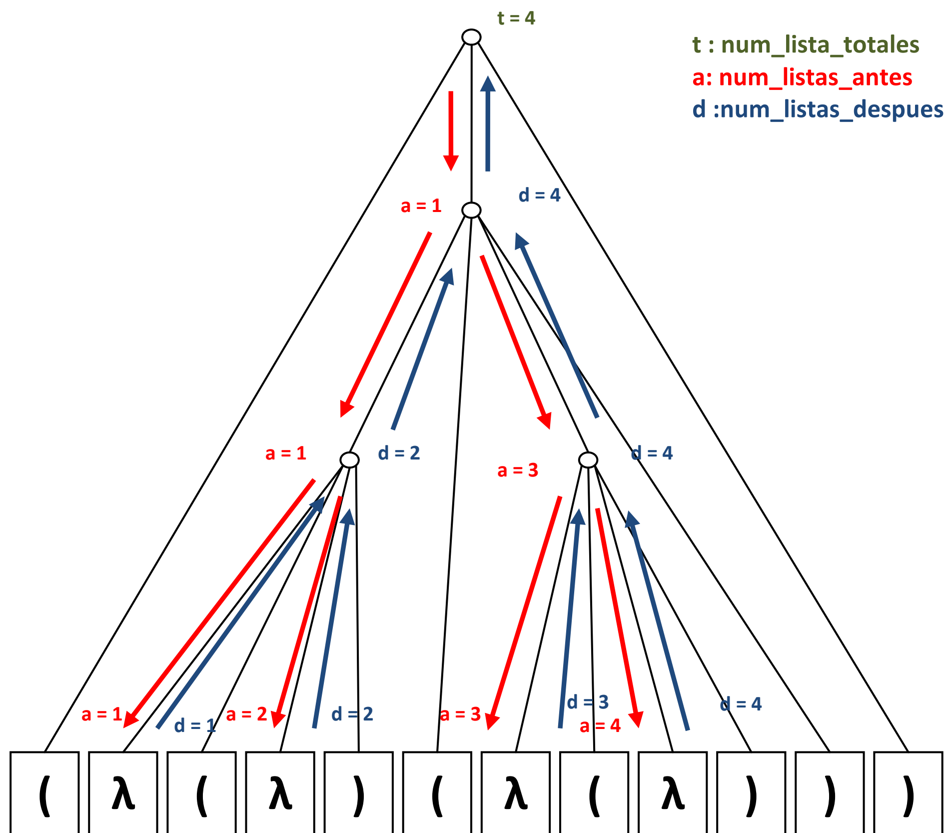

Figura 27. Árbol de análisis de ejemplo y grafo de dependencias asociado

1.3.1.3 Gramática de grafos

La gramáticas de grafos son del tipo de gramáticas sensibles al contexto, la cuales han sido usadas desde los años 70's para afrontar los problemas de especificación de lenguajes gráficos. Una gramática de grafos es una colección de reglas que especifica cómo crear un conjunto de gráfos para el cual la gramática completa es una descripción valida. En las gramáticas de grafos, cada lado de las reglas utilizadas consiste en un grafo (un conjunto de elementos conectados), donde cada elemento puede ser transformado por la aplicación de las reglas. De esta forma, cada regla es un par de sub-grafos formados por el lado izquierdo (*LHS - left hand side*) y el lado derecho (*RHS - left hand side*). El proceso de análisis busca patrones de nodos y aristas que coincidan con el *LHS* de la regla; si el patrón se encuentra en el grafo analizado, entonces se remplaza por el RHS.

Las gramáticas de grafos son mucho más complejas que las tradicionales gramáticas de cadenas. Efectivamente, los diagramas en sí mismo se representan como grafos dirigidos. En estos, las aristas representan las relaciones entre las partes del diagrama, y cada atributo o

etiqueta de las aristas o nodos simboliza la representación concreta de los elementos. Por otra parte, la aplicación de las reglas gramaticales en la creación o reconocimiento de un diagrama especifico forma una estructura de análisis (el equivalente al árbol de derivación en una gramática libre de contexto), que también es representada como un grafo. Las gramáticas de grafos se analizan con detalle en [64].

1.3.2 Transformación de modelos utilizando gramáticas de grafos

Como ya se ha discutido anteriormente, generalmente los metamodelos son representados en formalismos gráficos. Por ejemplo, para MDA comúnmente se utiliza los diagramas de clases UML (*InfrastructureLibrary*). Como resultado, los modelos pueden ser vistos como grafos, lo que sugiere directamente la utilización de gramáticas de grafos para expresar las transformaciones de modelos.

Una propuesta que utiliza este concepto es AToM^3 (*A Tool for Multi-formalism, Meta-Modelling*)[65]. En esta herramienta, las reglas se ejecutan de acuerdo a una prioridad dada, de manera descendente, lo que convierte a su lenguaje de especificación en lenguaje determinista. Por otra parte los autores son consientes del alto coste de la utilización de gramáticas de grafos, que han subsanado con el uso de pequeños subgrafos en las partes izquierdas de las reglas, añadiendo el tipado de nodos y aristas, más la utilización de atributos, de tal manera que se pueda reducir en gran medida el espacio de búsqueda. Como ventajas de la herramienta, los autores recalcan su alto nivel de abstracción, que es declarativo, natural y visual, lo que lo hace intuitivo, además de poseer una notación formal. AToM^3 actualmente también soporta gramáticas de grafos triples [66], las cuales son estructuras formadas por tres grafos llamados fuente (*source*), objetivo (*target*) y correspondencia (*correspondence*). Este último mantiene la relación entre los elementos de los otros dos grafos. Esto es útil para expresar la evolución de dos modelos diferentes, relacionados por uno intermedio. Otras propuestas que utiliza este principio son la de Alexander Königs [67] y la de Lars Grunske et ál [68], los cuales han obtenido resultados similares.

La utilización de las gramáticas de grafos implica, no obstante, los inconvenientes anteriormente expuestos: una estructura y representación compleja, además de un alto costo computacional en la reescritura de grafos y la dificultad de especificación y manejo de transformaciones de gran tamaño. Así mismo, desde un punto de vista personal, el uso de gramáticas de grafos en transformación de modelos pervierten, en cierto sentido, el espíritu tradicionalmente dado a las gramáticas en el dominio de los lenguajes informáticos: mecanismos que describen lenguajes, en lugar de mecanismos que expresan transformaciones.

1.3.3 Transformación de modelos utilizando gramáticas de atributos

La motivación para el uso de gramática de atributos para la transformación de modelos surge debido a su formalidad y buen rendimiento en sus implementaciones, por ejemplo como técnica de compilación, donde satisfactoriamente puede realizar transformaciones de programas desde un leguaje de programación fuente a un lenguaje objeto.

La propuesta de May Dehayni y Louis Féraud [1] consiste en representar los metamodelos como lenguajes textuales, y los modelos como frases de dichos lenguajes. De esta forma, es posible definir gramáticas de atributos convencionales que caractericen las transformaciones. (Figura 28).

Figura 28. Arquitectura del sistema de transformación de modelos de Dehayni y Féraud.

Los lenguajes textuales asociados con los metamodelos pueden concebirse como una sintaxis abstracta de los mismos. Para ello, se usa los conceptos de *phyla*[1] y operadores, donde *phylum* es el análogo a un no terminal y un operador es similar a la parte derecha de una regla sintáctica. De esta forma, las reglas de producción tienen el formato:

$$phy_0 : Operador(phy_1, phy_2, \dots, phy_n)$$

donde phy_i son *phyla* y ":" es el análogo a "->".

Las reglas semánticas están expresadas en términos de SSL (*Synthesizer Specification Language*), un lenguaje de especificación basado en algebra de términos y gramática de atributos, el cual fue adoptado en el sistema *Cornell Synthetizer Generator* [69], el cual puede

[1] Plural de *phylum.*

procesar gramáticas de atributos ordenadas (OAG - Ordered Attribute Grammars), en las cuáles es posible determinar el orden de evaluación de atributos a partir de un orden total de los atributos en cada producción [70]. Su selección fue debida al buen equilibrio entre poder expresivo y buen rendimiento en tiempo de ejecución del evaluador.

Para representar los (meta) modelos los autores han seleccionado HUTN (*Human-Usable Textual Notation*) [71], un estándar de la OMG para la representación de modelos MOF más legible y menos verboso que XMI, que, así mismo, es fácilmente convertible a XMI.

Utilizando los conceptos anteriores se pueden crear una gramática, que representa la sintaxis textual deseada para la producción de un metamodelo en HUTN. Dicho proceso se basa en el establecimiento de un metamodelo genérico que considera los conceptos básicos de MOF (paquete, clase, atributo-oo[2], referencia y asociación), y se asocia una *phylum* a cada concepto.

La transformación de modelos consiste en computar las propiedades de las entidades (atributos-oo y referencias). Para especificar las entidades objeto, los autores asocian a cada phylum, que representan un concepto origen, un conjunto de atributos que definen sus componentes: nombre, atributos-oo, referencias, sub-entidades y añaden un atributo sintetizado llamado “objetivo”, el cual especifica la entidad objetivo en concordancia con la relación entre los metamodelos origen y objeto. Los valores de atributos-oo y las referencias de las entidades objetivo pueden computarse desde los valores de la entidad origen, o pueden inicializarse con valores por defecto.

La propuesta incide, así mismo, en las ventajas que supone la utilización de gramáticas de atributos en la transformación de modelos. La principal radica en que los usuarios pueden concentrarse en especificar las reglas que serán utilizadas en la evaluación de los atributos, abstrayéndose del orden de ejecución de dichas reglas. Efectivamente, dicho orden queda implícitamente determinado por la relación de dependencia entre los atributos. La elección de sintaxis textuales basadas en lenguajes de términos simplifica, así mismo, la complejidad de los arboles sintácticos sobre los que se lleva a cabo la evaluación de atributos. Por último, la utilización de gramáticas de atributos garantiza precisión, dado su origen teórico. Las reglas semánticas, estando declaradas de manera declarativa en la estructura sintáctica, permiten que las transformaciones sean expresadas de forma concisa y clara. La modularidad de una especificación escrita con una gramática de atributos se garantiza por su estructura en bloques, derivada de las reglas de producción. Además, la gramática de atributos constituye un método ejecutable de especificación, que puede ser efectiva y sencillamente implementado utilizando un evaluador existente. Finalmente en comparación con otras propuestas, normalmente sujetas a modelado basado en lenguajes tipo UML, la utilización de gramática de atributos se puede ajustar fácilmente a otros estilos de modelado.

---

[2] Atributo-oo (orientado a objetos) hace referencia al concepto de atributo dentro de la arquitectura MOF con la intención de diferenciarlo del concepto de *atributo* en la gramática de atributos.

Mientras que el presente trabajo se adhiere a este enfoque basado en gramáticas de atributos, la principal diferencia con el mismo será evitar explícitamente en pasar a formatos textuales en los que expresar los modelos, concentrándose directamente el proceso de transformación en las estructuras gráficas de dichos modelos.

1.4 A modo de conclusión

Es evidente el interés que ha despertado el desarrollo de software dirigido por modelos en el moderno escenario de desarrollo de software, debido al potencial que proporciona en la creación y mantenimiento de software de manera más eficaz y eficiente. Este interés se muestra en las propuestas que han surgido de diferentes organizaciones, entre las que podríamos destacar las ya citadas MDA de la OMG o *Software Factories* de Microsoft entre otras.

Dentro del MDSD, la transformación de modelos constituye un componente fundamental, lo cual se ve reflejado en la cantidad de propuestas para llevar a cabo dicha tarea. Durante el análisis realizado en este capítulo, se han encontrado más de 25 propuestas que utilizan diferentes enfoques: basados en transformaciones de grafos dentro de los cuales se puede incluir la utilización de gramática de grafos, manipulación directa vía API, basados en XSLT, propuestas relacionales, declarativas e híbridas, y basadas en gramática de atributos. Muchas de estas propuestas siguen aún en evolución, y seguramente el número aumentará con la utilización de técnicas que aún no se han tenido en cuenta, con la combinación de propuestas y con la utilización de nuevas herramientas, lo que indica que aún no se ha encontrado una propuesta lo suficientemente satisfactoria, o, simplemente, que dicha propuesta depende de cada clase de dominio de aplicación, sugiriendo un amplio campo de investigación y refinamiento de las técnicas anteriormente nombradas.

Dentro de dichas propuestas se ha encontrado que las basadas en transformaciones de grafos, y, en especial las que utilizan las gramáticas de grafos, se encuentran teóricamente muy bien fundamentadas, y que, además, se ajusta coherentemente con los lenguajes gráficos comúnmente utilizados para modelado. Sin embargo, son muy sensibles a la complejidad resultante del carácter no determinista de la ordenación y estrategia de aplicación de reglas, a lo que debe agregarse el costo computacional del tratamiento de grafos y el evidente esfuerzo que conlleva la especificación y manipulación de transformaciones de modelos de gran tamaño.

La transformación directa mediante la utilización de APIs escritos en lenguajes imperativos como C++ o Java, posee un buen rendimiento debido a que los lenguajes en sí son bastante eficientes en tiempo de ejecución. Sin embargo el procedimiento debe ser descrito y ejecutado en su mayoría por el usuario, utilizando explícitamente sentencias imperativas.

Con XMI se puede expresar los modelos en formato XML, por lo que, en principio, resultaría factible utilizar XSLT como lenguaje de transformación. Sin embargo los tipos de transformaciones abordables son limitados, lo que restringe el alcance de la información que puede ser computada durante el proceso de transformación. Además en las DTDs y otros tipos de gramáticas documentales, se mezcla la sintaxis y la semántica intencional de los documentos XML. Como consecuencia las reglas de transformación deben tratar con ambas.

Las propuestas híbridas son las que más popularidad y aceptación han tenido (en especial QVT y los lenguajes basados en éste para la transformación de modelos dentro de MDA). Estas combinan la utilización de lenguajes declarativos e imperativos. Generalmente los primeros se utilizan para la definición y selección de las transformaciones que pueden ser aplicadas, mientras que la parte imperativa se utiliza para describir la estrategia para el control de flujo de la ejecución de las reglas, y por lo tanto de la ejecución de la transformación. En estas propuestas, las reglas de transformación deben manejar la sintaxis y la semántica combinadas. Además permiten al usuario combinar diferentes conceptos y paradigmas (por ejemplo la utilización de OCL dentro de QVT), lo cual puede conllevar un esfuerzo mayor de aprendizaje. No obstante, los lenguajes híbridos suelen ser más declarativos que imperativos, lo que mitiga en gran medida dicho impacto. Otro inconveniente de este tipo de propuestas, especialmente QVT, es que no se encuentran definidas teóricamente, por lo cual han surgido aproximaciones que intentan darle dicha formalidad (por ejemplo la propuesta descrita en [72]).

Finalmente, atendiendo los resultados exitosos obtenidos por la propuesta de May Dehayni y Louis Féraud [1] con la utilización de gramáticas de atributos y su eficiencia comprobada en problemáticas similares, como su uso en compiladores, además de su formalismo y su naturaleza declarativa, indica que es un opción viable, de fácil utilización a nivel de usuario, eficaz y eficiente para realizar transformación de modelos. En el siguiente capítulo partiremos de esta hipótesis para formular una propuesta que combina el estilo de procesamiento dirigido por la sintaxis de las gramáticas de atributos, con el enfoque respetuoso con la naturaleza gráfica de los modelos de las gramáticas de grafos.

2 EL MARCO DE TRANSFORMACIÓN AGT

2.1 Introducción

En este proyecto de máster se ha explorado la factibilidad de utilizar especificaciones basadas en gramáticas de atributos para describir declarativamente transformaciones de modelo a modelo. Para ello, se ha realizado una implementación de los conceptos necesarios para que el procesamiento dirigido por la sintaxis impuesto por una gramática de atributos pueda realizar transformaciones de modelos, de tal manera que se pueda evidenciar la citada factibilidad. De esta forma, hemos definido, diseñado e implementado AGT (*Attribute Grammar Transformer*), un marco para la transformación de modelos basado en gramáticas de atributos y escrito en Java. Este capítulo esboza dicho marco.

La arquitectura de AGT es similar a la presentada por May Dehayni y Louis Féraud *(ver apartado* 1.3.3*),* pero se diferencia en que la sintaxis abstracta de los modelos está representado mediante clases y sus respectivos atributos -particularmente en el lenguaje Java, de tal manera que pueda representarse directamente modelos en términos orientados a objetos convencionales, sin necesidad de *descender* a sintaxis textuales auxiliares. De esta forma, los modelos origen y objetivo no son más que instancias de dichas clases, como se sugiere en la figura 29.

Más concretamente, AGT consta de un metamodelo (especificado en Java) que permite representar las transformaciones basadas en gramáticas de atributos, así como de un transformador que, dirigido por las transformaciones, opera sobre un modelo fuente para producir un modelo objetivo. Dichos modelos (fuente y objetivo) se representan mediante instancias de metamodelos codificados también en Java. El motivo para haber adoptado dicho convenio de representación es, por una parte, aislar la experiencia de las complejidades colaterales de las propuestas de metamodelado, y, por otra, independizar el sistema de dichas propuestas, que, en el futuro, podrán incorporarse mediante componentes conectores adecuados.

Por último, AGT incluye también un lenguaje textual de especificación: AGTL (*Attribute Grammar Transformation Language*). AGTL es un lenguaje textual de especificación de transformaciones basado en gramáticas de atributos, similar a los lenguajes tradicionales para la representación de este tipo de gramáticas. AGTL se ha implementado también en Java. Su

implementación consta de un analizador léxico construido con *JFlex*[73] y un analizador sintáctico y semántico construido con *CUP*[74], para dar lugar a un traductor que convierte las especificaciones textuales en representaciones Java en términos del metamodelo de AGT. Dichas representaciones pueden, entonces, guiar la operación del componente transformador, que utilizará también la instancia de una *clase semántica* que implemente, como métodos, las funciones semánticas referidas en la especificación. Dichos métodos serán utilizados durante la creación/instanciación de los objetos que representan el modelo resultante

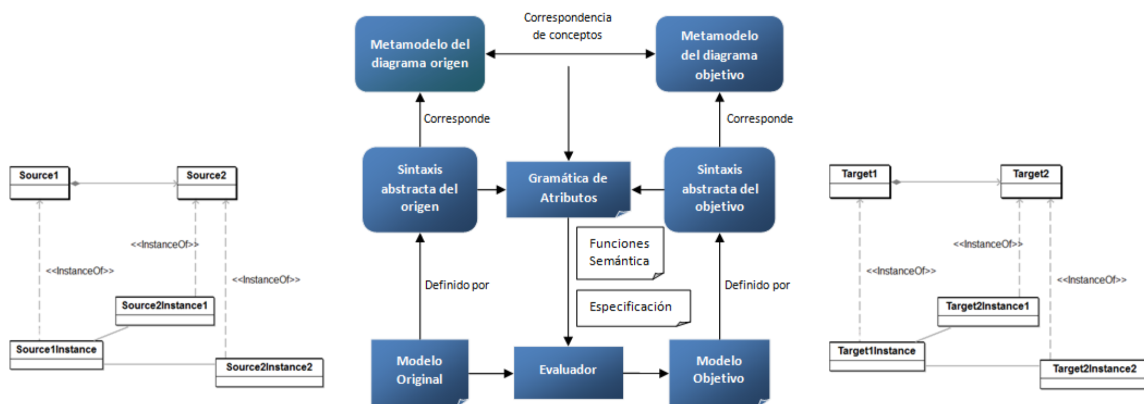

Figura 29. Arquitectura del marco de transformación AGT.

En este capítulo se describe el marco AGT. La sección 2.2 introduce un ejemplo de transformación especificada en AGTL. La sección 2.3 describe el lenguaje de especificación AGTL en sí. La sección 2.4 detalla el metamodelo de AGT. La sección 2.5 introduce el concepto de *clase semántica* en AGT. La sección 2.6 describe el traductor de AGTL a los modelos de transformaciones (las instancias del metamodelo AGT). La sección 2.7 detalla el transformador. Por último, la sección 2.8 cierra el capítulo con los resultados obtenidos por la implementación.

2.2 Un ejemplo de transformación en AGT

Como ejemplo de transformación en AGT, que sirva para discutir los distintos aspectos del marco, utilizaremos el ejemplo clásico de transformación de un modelo de clases en un esquema de base de datos relacional ya utilizado en el capítulo anterior. La figura 30 muestra la estructura del metamodelo origen. La figura 31 muestra, por su parte, la estructura del metamodelo objetivo.

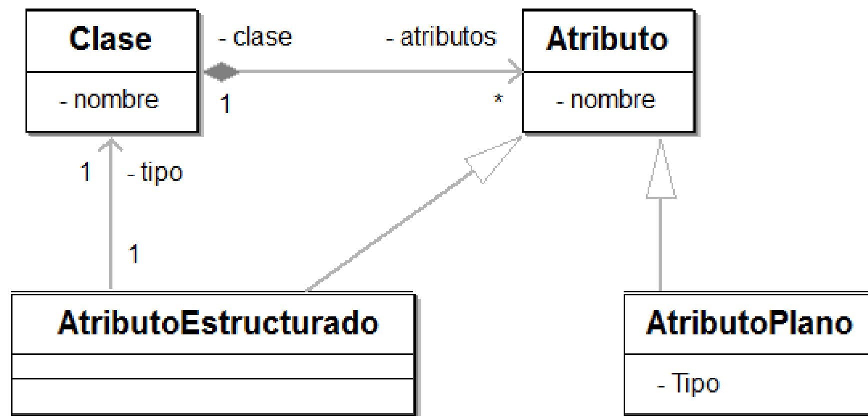

Figura 30. Metamodelo origen.

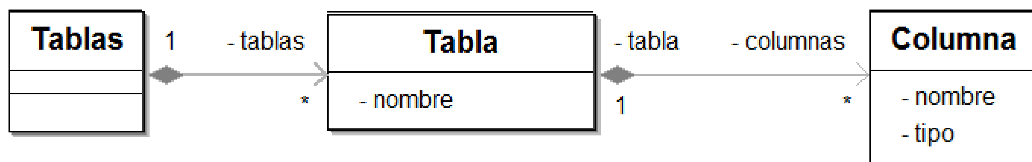

Figura 31. Metamodelo destino.

Las figura 32 por su parte, muestra la transformación especificada en AGTL. En dicha transformación cada clase en el modelo de clases se transforma en una tabla del esquema relacional. El nombre de la tabla coincide con el nombre de la clase. Los nombres de las columnas coinciden con los nombres de los atributos de la clase. Los tipos de las columnas dependen de la clase de atributo involucrada:

- Para atributos *planos* (atributos cuyos tipos son tipos primitivos: boolean, integer y char), el tipo del atributo será el tipo de la columna.
- Para atributos *estructurados* (atributos cuyos tipos son referencias a clases), el tipo del atributo será “fk” (de *foreign key*), seguido del nombre de la clase.

A modo de ejemplo, la figura 33 muestra un diagrama de clases (instancia del metamodelo de la figura 30), y la 34 muestra el esquema relacional asociado (instancia del metamodelo de la figura 31).

```
nt(<Root>,[],[tablas]).
nt(<Clase>,[tablash],[tablas,nombre]).
nt(<Atributos>,[tablash],[tablas,cols]).
nt(<Atributo>,[tablash],[tablas,col]).


<Root> ::= Clase: <Clase> {
               <Clase>.tablash =  mkEmptySetOfTables(),
               <Root>.tablas = mkTables(<Clase>.tablas)
           }.


<Clase> ::= Clase { atributos: <Atributos>  }
    {
      <Atributos>.tablash = addTable(<Clase>.tablash,
                                     mkTable(@nombre,<Atributos>.cols)),
      <Clase>.tablas = <Atributos>.tablas,
      <Clase>.nombre = @nombre
    }
    {
      <Clase>.tablas = <Clase>.tablash,
      <Clase>.nombre = @nombre
    }.

<Atributos> ::= com.agt.core.Array {first: <Atributo>,
                                    butfirst: <Atributos>} {
      <Atributo>.tablash = <Atributos>(0).tablash,
      <Atributos>(1).tablash = <Atributo>.tablas,
      <Atributos>(0).tablas = <Atributos>(1).tablas,
      <Atributos>(0).cols = addAttribute(<Atributos>(1).cols,
                                         <Atributo>.col)
    }.

<Atributos> ::= null {
      <Atributos>.tablas = <Atributos>.tablash,
      <Atributos>.cols = mkEmptyListOfAttributes()
 }.

<Atributo> ::= AtributoPlano {}{
      <Atributo>.tablas = <Atributo>.tablash,
      <Atributo>.col = mkAttr(@nombre,@tipo)
 }.

<Atributo> ::= AtributoEstructurado {tipo: <Clase>}{
      <Clase>.tablash = <Atributo>.tablash,
      <Atributo>.tablas = <Clase>.tablas,
      <Atributo>.col = mkAttr(@nombre, mkForeignKey(<Clase>.nombre))
 }.
```

Figura 32. Especificación AGT para transformar diagramas de clases en esquemas de bases de datos relacionales

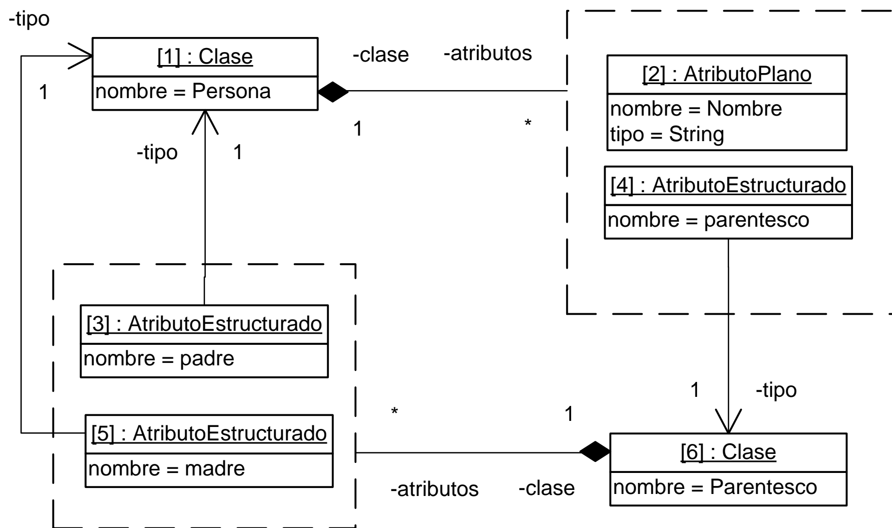

Figura 33. Modelo origen.

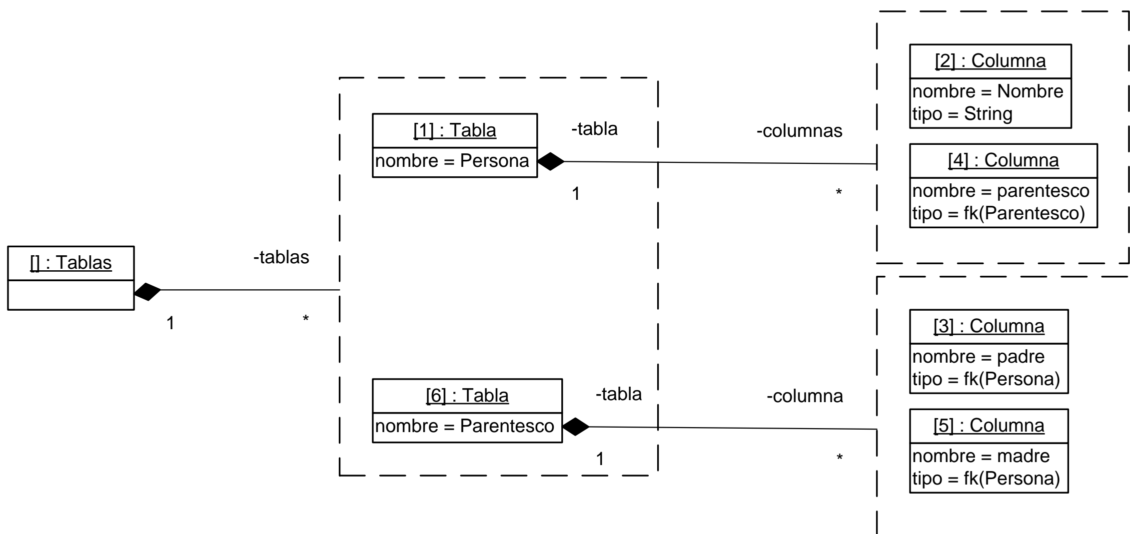

Figura 34. Modelo destino

En las siguientes secciones se analizará con detalle el significado de la transformación AGTL, así como la manera en la que el marco AGT realiza la transformación.

2.3 El lenguaje de especificación AGTL

AGTL es el lenguaje de especificación textual de transformaciones del marco AGT. En esta sección se describe la sintaxis del lenguaje (apartado 2.3.1), así como se proporciona una descripción informal de su semántica (apartado 2.3.2).

2.3.1 Sintaxis de AGTL

Una especificación AGTL consta de dos bloques principales:

- La declaración de los no terminales y de sus atributos asociados, tanto sintetizados como heredados.
- La especificación de las reglas de producción, que incluye un conjunto de reglas gramaticales, donde cada una relaciona un no terminal con una clase del modelo origen. En dichas reglas se especifica también la relación de los campos de dicha clase con otros no terminales, y un conjunto de ecuaciones que realizan el cálculo de los atributos sintetizados de la parte izquierda de la regla, y de los atributos heredados de la parte derecha de la misma. Así mismo, la regla puede incluir un segundo conjunto de ecuaciones, que se utiliza, como se verá más adelante, cuando ésta se aplica sobre un objeto ya analizado anteriormente.

La definición de un no terminal se realiza de la siguiente forma:

nt(<_nombre_noterminal_**>,[**_atributos_heredados_**],[**_atributos_sintetizados_**]).**

donde *nombre_noterminal* es un identificador del no terminal, *atributos_heredados* es una lista de identificadores separados por coma, que representan los atributos heredados pertenecientes al no terminal, y *atributos_sintetizados* es una lista de identificadores también separados por comas, que representan los atributos sintetizados del no terminal. Estas listas pueden ser vacías, lo que indicará que el no terminal no tiene atributos de la clase correspondiente. La especificación de un no terminal no puede contener atributos duplicados. Por su parte, tanto los nombres de los no terminales como los nombres de los atributos se componen de secuencias de caracteres alfanuméricos y el carácter *underline* (_), empezando siempre por un carácter alfabético.

La figura 35 ejemplifica la declaración de no terminales con el caso de estudio presentado en la sección 2.2. El no terminal *Root* tiene asociado únicamente un atributo sintetizado *tablas*. Por su parte, el no terminal *Clase* tiene asociado un atributo heredado *tablash* y dos atributos sintetizados: *tablas* y *nombre*.

```
nt(<Root>,[],[tablas]).
nt(<Clase>,[tablash],[tablas,nombre]).
```

Figura 35. Ejemplo de definición de no terminales.

Cada regla puede ser especificada de las tres formas siguientes:

- **<**_nombre_noterminal_**>::=**_nombre_clase_**{**_especificación_campos_**}{**_ecuaciones_**}**(**{**_ecuaciones_ **}**)?. Estas reglas se demominan *reglas de clase*.
- **<**_nombre_noterminal_**>::=null{**_ecuaciones_**}.** Estas reglas se denominan *reglas nulas*.

- <*nombre_noterminal*>::=*nombre_clase*: <*nombre_noterminal*>{*ecuaciones*}. Estas reglas se denominan *reglas puente*.

En cada uno de estos tres tipos de reglas, *nombre_noterminal* es un identificador del no terminal y *nombre_clase* es el nombre de una clase Java, el cual puede contener puntos para especificar el paquete al cual pertenece. *especificación_campos* es una lista de elementos separados por comas, posiblemente vacía, donde cada entrada indica la relación de los atributos de la clase con otros no terminales, especificados del siguiente modo:

nombre_campo **:<***nombre_noterminal***>**

donde *nombre_campo* es el nombre de un campo perteneciente a la clase especificada en la regla y *nombre_noterminal* es el identificador del no terminal que se relaciona con dicho atributo. Obsérvese que, mediante una regla nula, puede especificase que un no terminal se relacione con referencias nulas mediante la palabra reservada **null**; en este caso no tiene sentido especificar lista de campos alguna. También, mediante una regla puente, puede especificarse que un no terminal se relacione directamente con un no terminal; en este caso, es necesario especificar también el nombre de la clase para el cual la regla es aplicable. Este último tipo de reglas son útiles para ligar los no terminales asociados a los campos en las reglas de clase con las partes izquierdas de otras reglas de clase, evitando la proliferación de reglas de clase redundantes.

Por otra parte, en la especificación de una regla, *ecuaciones* es una lista de ecuaciones separadas por coma, posiblemente vacía, donde cada ecuación posee la siguiente sintaxis:

<*nombre_noterminal***>(***###***)**.*nombre_atributo* **=** *expresión*

Aquí, *nombre_noterminal y nombre_atributo* son los identificadores de un no terminal presente en la regla y el identificador de un atributo perteneciente a dicho no terminal, respectivamente. (*###)* es un índice que indica un no terminal especifico, en el caso de repeticiones en la regla de un mismo no terminal, según su orden de aparición en la parte izquierda de la regla y en la especificación de los atributos. Por omisión, este será (0). El atributo especificado tomara como valor la evaluación de la *expresión*, que se puede especificar de las siguientes maneras:

- **@***nombre_campo*
- **<***nombre_noterminal***>(***###***)**.*nombre_atributo*
- *Nombre_funcion***(***expresiones***)**

El primer caso, *nombre_campo* precedido de una @, es el nombre de un *campo*[3] de la clase especificada en la regla. En concreto, se asume que toda clase tiene un campo **this**, cuyo valor, en las instancias de dicha clase, serán referencias a las propias instancias. El segundo caso, es un atributo de un no terminal especificado de la misma manera que la parte izquierda de una ecuación. Finalmente, *Nombre_funcion* es el nombre de una función semántica que debe estar implementada como un método Java de la clase semántica que es utilizada por el transformador para ejecutar las especificaciones. Por su parte, *expresiones* son los parámetros reales de la función, siendo una lista de expresiones, separadas por comas, que puede ser vacía.

Obsérvese que, en las reglas asociadas con clases, pueden existir dos secciones de ecuaciones. Las ecuaciones en la primera sección, que también aparece en las reglas nulas y en las puente, deben respetar, así mismo, las siguientes restricciones:

- Las partes izquierdas de las ecuaciones deben corresponderse siempre con atributos sintetizados de la parte izquierda de la regla, o con atributos heredados de los no terminales en la parte derecha.

```
<Clase> ::= Clase { atributos: <Atributos>  }
    {
      <Atributos>.tablash = addTable(<Clase>.tablash,
                                    mkTable(@nombre,<Atributos>.cols)),
      <Clase>.tablas = <Atributos>.tablas,
      <Clase>.nombre = @nombre
    }
    {
      <Clase>.tablas = <Clase>.tablash,
      <Clase>.nombre = @nombre
    }.

<Atributos> ::= null {
      <Atributos>.tablas = <Atributos>.tablash,
      <Atributos>.cols = mkEmptyListOfAttributes()
 }.

<Root> ::= Clase: <Clase> {
               <Clase>.tablash =  mkEmptySetOfTables(),
               <Root>.tablas = mkTables(<Clase>.tablas)
           }.
```

Figura 36. Ejemplo de especificación de reglas

- Debe haber *exactamente* una ecuación para cada atributo sintetizado de la cabeza, así como para cada atributo heredado de los no terminales asociados a los campos en la parte derecha.

[3] A fin de no confundir los atributos de las clases con los atributos de las gramáticas, en la terminología AGT se denomina *campos* a los primeros, reservando el término *atributos* para los segundos.

- En la parte derecha de las ecuaciones únicamente es posible utilizar atributos heredados de la parte izquierda, y atributos sintetizados de los no terminales de la parte derecha.

La segunda sección, denominada *sección de ecuaciones de corte,* contiene las ecuaciones que deben aplicarse cuando, en el proceso de análisis, se descubre un ciclo en el modelo origen. En este caso, únicamente deben especificarse ecuaciones para los atributos sintetizados de la cabeza, y no deben referirse atributos sintetizados de los símbolos del cuerpo (ya que el proceso de análisis se interrumpirá en dicho punto).

A fin de ejemplificar la especificación de las reglas, considérese las tres reglas del caso de estudio mostradas en la figura 36. La primera regla es aplicable sobre instancias de la clase *Clase*. Así mismo, indica que el valor del campo *atributos* está asociado al no terminal *Atributos* (lo que, a su vez, implica que debe seguir alguna de las reglas asociadas a dicho no terminal). La regla introduce las siguientes ecuaciones:

- Una ecuación que indica que el valor del atributo heredado *tablash* de *Atributos* ha de obtenerse añadiendo la tabla correspondiente al conjunto de tablas *tablash* del no terminal *Clase* (conjunto de tablas construido hasta el momento).
- Una ecuación que indica que el conjunto de tablas *tablas* de *Clase* es el valor del correspondiente atributo *tablas* de *Atributos*.
- Una última ecuación que indica que el atributo *nombre* de *Clase* es el valor del campo *nombre* en la correspondiente instancia de *Clase* a la que se aplica la regla.

La regla introduce también una sección de ecuaciones de corte, que indican cómo computar los valores de los atributos sintetizados *tabla* y *nombre* de la parte izquierda *Clase* cuando, al aplicar la regla, se descubre un ciclo en el modelo origen:

- La primera ecuación indica que el conjunto de tablas resultante (*tablas*) es el que se recibe como entrada (*tablash*).
- La segunda ecuación determina el valor del atributo *nombre* como en la ecuación análoga en la sección de ecuaciones convencional de la regla.

Por su parte, la segunda regla indica que el no terminal *Atributos* puede corresponderse con una referencia nula; en este caso, las ecuaciones indican que el valor de su atributo sintetizado *tablas* será el valor de su atributo heredado *tablash* (primera ecuación), mientras que la lista de columnas asociada será la lista vacía (segunda ecuación).

Por último, la tercera regla representa un ejemplo de regla puente, que indica que el análisis respecto al no terminal *Root* puede proceder según el no terminal *Clase*, siempre y cuando el objeto sea de tipo *Clase*. El valor del atributo *tablash* para el no terminal *Clase* se fija al conjunto vacío de tablas, mientras que el valor de *tablas* para *Root* se toma del de *Clase*.

La figura 37 resume la sintaxis del lenguaje de transformación AGTL.

```
<Grammar> ::=  <NTDec>+ <Rule>+
<NTDec>  ::= nt(nonTerminalName,[<AttList>],[<AttList>]).
<AttList> ::= (attName (, attName)*)?
<Rule> ::= nonTerminalName '::='
             (className {<FieldSpecList>} {<EquationList>} ({<EquationList>})? |
              null {<EquationList>} |
              className: nonTerminalName {<EquationList>}) .
<FieldSpecList> ::= (<FieldSpec> (, <FieldSpec>)*)?
<FieldSpec> ::= fieldName : nonTerminalName
<EquationList> ::= (<Equation> (, <Equation>)*)?
<Equation> ::= <AttrRef> = <SemExp>
<AttrRef> ::= nonTerminalName( '(' num ')' )?.attname
<SemExp> ::= <AttrRef>|funName ( <ArgList> )|@fieldName
<ArgList> ::= (<SemExp> (,<SempExp>)*)?
```

Figura 37. Gramática EBF que resume la sintaxis del lenguaje de transformación AGTL.

2.3.2 Semántica de AGTL

La semántica de AGTL se basa en el modelo de ejecución clásico de las gramáticas de atributos. Dicho modelo es, como ya se ha discutido en el capítulo anterior, un modelo de traducción dirigida por la sintaxis. Este modelo de procesamiento estructura el procesamiento en dos fases [59]:

- En la primera fase, las reglas sintácticas se utilizan para explicitar la estructura sintáctica del objeto de entrada (en el caso de lenguajes textuales, de la frase de entrada; en el caso de AGTL, del modelo origen). En esta fase es posible, también, rechazar dicho objeto como *sintácticamente inválido*.
- En la segunda fase, las ecuaciones semánticas se utilizan para encontrar los valores de los atributos en los nodos de la estructura sintáctica explicitada.

2.3.2.1 La fase de análisis

El objetivo de la fase de análisis es *superponer* sobre el modelo origen su *estructura sintáctica*. Dicha estructura es un *árbol de análisis* que *recubrirá* los objetos del modelo. Los nodos de este árbol estarán etiquetados con no terminales y asociados con los objetos del modelo origen, y sus arcos estarán etiquetados con nombres de campos de dichos de objetos. Así mismo, los nodos se clasificarán en *nodos corrientes* y *nodos de corte*. Dicho árbol de análisis será el análogo al árbol de análisis sintáctico para una frase analizada con respecto a una gramática incontextual convencional.

La fase de análisis, en sí comienza eligiendo un objeto distinguido del modelo, denominado *objeto raíz*, así como un no terminal distinguido, denominado *no terminal de inicio* (la elección del objeto raíz y del no terminal de inicio se deja a cargo del cliente que invoca al procesador de AGTL), y procediendo a *analizar* el objeto raíz con el no terminal de inicio, tomando, además, el conjunto formado por el objeto raíz como conjunto de *objetos analizados*. El

proceso de análisis procede, entonces, de forma recursiva, de acuerdo con las siguientes reglas:

- Si se trata de analizar un objeto *o* de acuerdo con un no terminal *n*, y dicho objeto *o* aparece en el conjunto de objetos analizados, entonces se asocia con *o* un *nodo de corte* etiquetado por *n*, y el proceso de análisis finaliza, en este punto, con éxito. Nótese que esta regla se corresponde con la detección de un ciclo en el modelo origen.

- Si el objeto *o* se trata de analizar de acuerdo con el no terminal *n*, y dicho objeto *o* no aparece en el conjunto de objetos analizados, se busca la regla *más específica* para *n aplicable* a *o*:
    - Si dicha regla existe, y es una regla puente, se procede a analizar *o* con respecto al no terminal *n′* indicado en la parte derecha, sin modificar el conjunto de objetos analizados. Si el análisis finaliza con éxito, se crea un nodo corriente etiquetado por *n*, y unido por un arco al nodo resultante de analizar *o* respecto a *n*', y se devuelve dicho nodo.
    - En caso de que la regla sea una regla de clase, se procede a analizar los valores de los campos de *o* de acuerdo con las especificaciones contenidas en la lista de campos de la regla, analizando cada valor de acuerdo con el correspondiente no terminal, e incluyendo *o* en el conjunto de objetos analizados utilizados en cada subproceso de análisis En caso de que se descubra un campo en la regla que no esté presente en el objeto *o,* el modelo se descarta como sintácticamente inválido. En otro caso, se crea un nodo corriente etiquetado por *n* y asociado a *o*, y se crean arcos apropiados desde dicho nodo hacia los nodos resultantes del análisis de los campos.
    - En otro caso, el modelo origen se rechaza como sintácticamente inválido.

  En este punto, es necesario precisar el concepto de *regla aplicable* a *o*: será una regla cuya clase asociada sea la de *o* o una superclase de la de *o*. Así mismo, de todas las reglas aplicables, la más específica es la que se corresponde a la clase más específica (en caso de que haya más de una, se toma siempre la que aparece en primer lugar).

- Si trata de analizarse una referencia nula respecto al no terminal *n*, se busca la primera regla para *n* que tenga a **null** en su parte derecha:
    - Si dicha regla existe, el proceso de análisis finaliza en este punto con éxito.
    - En otro caso, el modelo se descarta como sintácticamente inválido.

Durante este proceso es importante, así mismo, tener en cuenta que el transformador puede realizar ciertas *transformaciones* de bajo nivel que faciliten la posterior fase de síntesis. En concreto, la versión actual de AGT transforma cada objeto de tipo *array* en un objeto de tipo *com.agt.core.Array,* el cual *envuelve* el array y ofrece una vista del mismo en base a cuatro campos: *first* (primer elemento del array), *last* (último elemento del array), *butFirst* (el mismo

array, pero sin el primer elemento, o null en caso de arrays con un único elemento) y *butLast* (el mismo array, pero sin el último elemento, o null en caso de arrays con un único elemento). Esta adaptación permite ver los arrays, desde el punto de vista de AGTL, como árboles de objetos de tipo *com.agt.core.Array* (véase la figura 38), lo que permite tratarlos homogéneamente utilizando el formalismo gramatical básico de AGTL.

Figura 38. Ejemplo de *com.agt.core.Array* como *envoltorio* de una lista de atributos.

En la figura 38 se ilustra como un objeto *com.agt.core.Array envuelve* una lista de atributos perteneciente al modelo de ejemplo presentado en la figura 33.

Con estas consideraciones, es fácil *leer* la parte sintáctica de la especificación AGTL en el caso de estudio:

- La regla primera regla (<Root> ::= Clase: <Clase>) puede entenderse como que *el objeto raíz del modelo origen ha de ser tipo tipo* Clase, y seguir la estructura impuesta por el no terminal *Clase*.
- La segunda regla (<Clase> ::= Clase { atributos: <Atributos> }) indica que *una* Clase *es un objeto de tipo* Clase *que exhibe un campo* atributos *cuyo valor es una secuencia de* Atributos.
- La tercera regla (<Atributos> ::= com.agt.core.Array {first: <Atributo>, butfirst: <Atributos>}), que utiliza el envoltorio AGT para los arrays, se lee como sigue: *una secuencia de* Atributos *puede ser un array cuyo primer elemento es un* Atributo *y en el que el resto de los atributos forman una secuencia de* Atributos.
- La cuarta regla (<Atributos> ::= null), contempla el caso de una lista de atributos vacía.
- La quinta regla (<Atributo> ::= AtributoPlano {}), contempla el caso de que un atributo sea de tipo *AtributoPlano*.

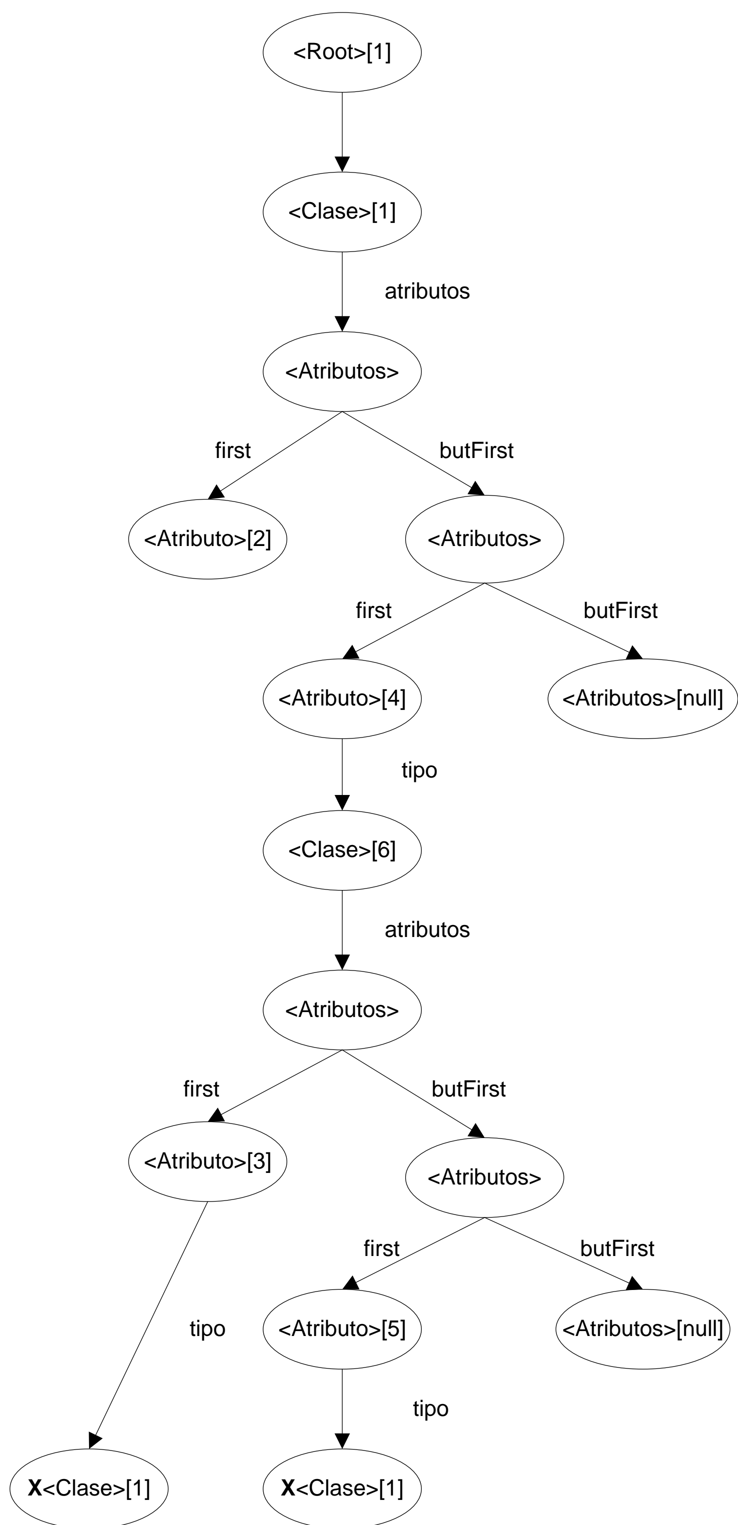

Figura 39. Árbol de análisis del modelo de ejemplo.

- Por último, la sexta regla (<Atributo> ::= AtributoEstructurado {tipo: <Clase>}) considera el caso de atributos estructurados; en este caso, el valor del campo *clase* ha de seguir la estructura dictada para las clases.

La figura 39 muestra el árbol de análisis resultante de analizar el modelo de ejemplo (ver figura 33) con respecto a estas reglas sintácticas. Para mayor claridad, en el árbol se ponen referencias a los objetos del modelo asociados con los nodos, pero se obvian las referencias a los objetos '*Array*' intermedios, ya que estos no están explícitamente representados en el modelo. Los nodos de corte se marcan con **X**.

2.3.2.2 La fase de evaluación

Durante esta fase, las ecuaciones semánticas se utilizan para calcular los valores de los atributos de los nodos del grafo sintáctico. De esta forma, los atributos y ecuaciones semánticas tienen la misma función que en una gramática de atributos convencional. Esto permite *leer* la parte semántica de la especificación en el caso de estudio como sigue:

- El papel de los distintos atributos es el siguiente:
    - El atributo heredado *tablash* se utiliza para almacenar las tablas obtenidas hasta el momento.
    - El atributo sintetizado *tablas* se usa para almacenar el conjunto total de tablas obtenido.
    - El atributo *resul* de *Root* se usa para almacenar una referencia al nodo raíz del modelo objetivo obtenido. El atributo *nombre* de *Class* se utiliza para almacenar el nombre de la clase. El atributo *cols* de *Atributos* almacena la lista de columnas de la tabla para la clase, mientras que el atributo *col* de *Atributo* almacena la columna correspondiente.

- En la primera regla *(<Root> ::= Clase*: <Clase>*)*, se *pasa* a *Clase* el conjunto vacío como valor de *tablash* (ecuación 1). Así mismo, el modelo resultante (valor de *resul* para *Root*) se construye a partir del conjunto de tablas resultante de procesar *Clase* (ecuación 2).

- En la segunda regla *(<Clase> ::= Clase { atributos: <Atributos> })*, en los nodos corrientes, el conjunto de tablas construido cuando se procesa *Atributos* será el conjunto de tablas construido antes de comenzar a procesar la clase, junto con la tabla asociada a la clase (ecuación 1). El conjunto final de tablas es el que resulta de *Atributos* (ecuación 2). Por último, el nombre de la clase será el valor del campo *nombre* (ecuación 3). Por su parte, en los nodos de corte, el conjunto de tablas final se toma directamente como el conjunto de tablas construido (ecuación 4), y el nombre se computa igual que antes (ecuación 5).

- En la tercera regla *(<Atributos> ::= com.agt.core.Array {first: <Atributo>, butfirst: <Atributos>})*, las tablas construidas antes de procesar *Atributo* serán las tablas construidas al inicio del procesamiento dictado por la regla (ecuación 1). Por su parte, las tablas que resultan de procesar *Atributo* serán las tablas construidas en el momento de procesar el valor del campo *butFirst* (ecuación 2). De esta forma, las tablas finalmente construidas vendrán dadas por el valor del atributo *tablas* de la ocurrencia de *Atributos* en el cuerpo de la regla (ecuación 3). Por su parte, las columnas en las que se transforma la lista de atributos se obtendrán añadiendo la columna asociada al primer atributo a la lista de columnas asociada al resto (ecuación 4).

- En la cuarta regla *(<Atributos> ::= null),* las tablas finales serán las construidas hasta este momento (ecuación 1), mientras que la lista de columnas será la lista vacía (ecuación 2).

- En la quinta regla *(<Atributo> ::= AtributoPlano {}),* las tablas finales serán también las construidas hasta ese momento (ecuación 1), mientras que la columna se construirá directamente a partir del nombre y el tipo del atributo (ecuación 2).

- Por último, en la sexta regla *(<Atributo> ::= AtributoEstructurado {clase: <Clase>})*, *tablas* de *Atributo* se sintetiza lo mismo que en la regla anterior (ecuación 1), mientras que *col* se sintetiza a partir del nombre del atributo y del nombre de la clase (ecuación 2).

En AGT el proceso de evaluación se implementa como un proceso de evaluación bajo demanda (véase la sección 1.3.1.2.1 del capítulo anterior). La sección 2.7 proporciona más detalles sobre dicho proceso.

2.4 El metamodelo AGT

En la figura 40 se presenta el metamodelo AGT, el cual permite modelar una gramática de atributos específica para describir una transformación de modelo a modelo, y a su vez permite la construcción de las estructuras necesarias para realizar el procesamiento de transformación.

Este metamodelo consta de los elementos característicos de la gramática contemplados por AGTL. De esta forma, el metamodelo puede considerarse una caracterización de la *sintaxis abstracta* de AGTL.El metamodelo en sí está constituido por las clases:

- *Grammar:* Es la clase que representa en su totalidad una gramática especifica. Está compuesta por un conjunto de reglas (*rules*) y un conjunto de no terminales (*nonTerminalDescriptions*) indexados por sus nombres (*nonTerminalName*) para permitir un acceso y manipulación más eficiente.

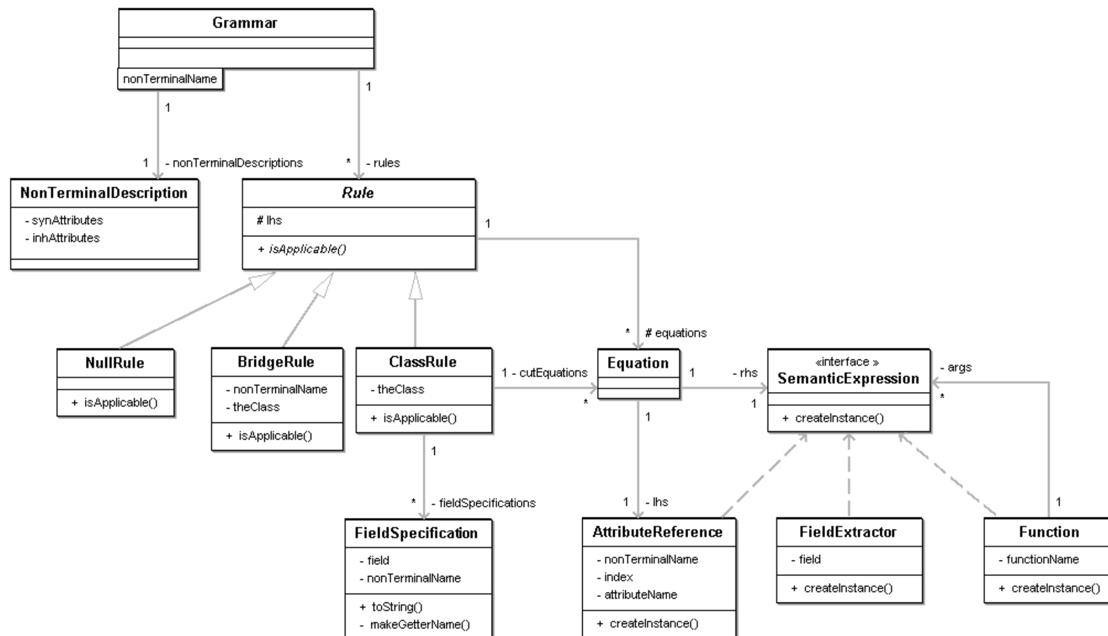

Figura 40. El Metamodelo de AGT.

- *NonTerminalDescription:* Representa la especificación de un no terminal, el cual está compuesto por una lista de atributos sintetizados (*synAttributes*) y heredados (*inhAttributes*), donde cada uno de estos es simplemente el nombre del atributo (una cadena de caracteres).

- *Rule:* Es una clase abstracta que representa una regla, compuesta por un atributo que almacena el nombre del no terminal (una cadena de caracteres) que aparece al lado izquierdo de una regla (*lhs*), y un conjunto de ecuaciones (*equations*). Además posee un método abstracto llamado *isAplicable* que permite conocer si una regla es aplicable para un objeto y un no terminal especifico, que debe ser implementado por las clases que realizan la especialización de ésta:

    - *NullRule:* Representa las reglas que se ejecutan cuando el objeto que se intenta transformar es nulo.
    - *ClassRule:* Representa las reglas que relacionan el no terminal dictado por *lhs* con el atributo *theClass*. Dicho atributo almacena instancias de la clase *Class* que representa clases en una aplicación Java. Así mismo, esta clase tiene asociado un conjunto de especificaciones de campos (*fieldSpecifications*), así como un conjunto (posiblemente vacío) formado por las ecuaciones de la sección de corte.

- *BridgeRule*: Representa las reglas que relacionan un no terminal con otro terminal.

- *FieldSpecifications:* Representa la relación entre un campo de la clase con un no terminal. El campo *field* es almacenado como un método (una instancia de *Method,* parte del paquete de reflexión que Java ofrece) que es el método lector (*getter*) que permite acceder al valor de un atributo de un objeto especifico en ejecución. El no terminal (*nonTerminalName*) simplemente queda referenciado por su nombre.

- *Equation:* Representa cómo se calcula el valor de un atributo de un no terminal (*lhs*), que será una instancia de *AttributeReference,* a través de una expresión semántica (*rhs*), que será alguna instancia de una clase que implemente la interfaz *SemanticExpression*.

- *SemanticExpression:* Interfaz que representa las diferentes formas de calcular el valor de un atributo. Cualquier clase que implemente esta interfaz debe proveer la manera de *instanciar* una estructura que represente dicha expresión semántica, para su posterior proceso, a través del método *createInstance*. Implementan esta interfaz:

  - *AttributeReference:* Clase que representa una referencia a un atributo de un no terminal especifico mediante el nombre del atributo (*AtributeName*), el nombre del no terminal (*nonTerminalName*) y el índice de dicho no terminal (*index*) que indica el orden de aparición dentro de la regla, de la misma forma que se establece en AGTL (véase la sección 2.3.1).
  - *FieldExtractor:* Representa el valor de un campo de una clase. En *field* almacena el método lector correspondiente (una instancia de *Method*).
  - *Function:* Almacena el nombre de una función semántica (*functionName*) y un conjunto de argumentos (*args*) que son clases que implementan una *SemanticExpression.*

En el marco AGT, todas estas clases son parte del paquete *com.agt.core*, el cual además contiene la clase anteriormente mencionada *Array*.

En la figura 41 se puede observar un fragmento de la representación de la gramática de atributos que se ha utilizado en todo el capitulo como ejemplo en términos del metamodelo presentado. Más concretamente, se detallan tres reglas: <*Clase*> ::= *Clase*{atributos: <*Atributos*>}, <*Root*> *::= Clase*:<*Clase*>, y <*Atributos*> *::= null*. La primera se muestra completamente y la segunda y tercera sólo se presenta hasta el nivel de sus ecuaciones. En la notación empleada, mediante *Class*[*nombre*] y *Method*[*nombre*] se representan instancias de este tipo de clases correspondientes con los nombres indicados, que las identifican

intuitivamente. También se puede observar la especificación de los no terminales: *“Root”, “Clase”* y *“Atributos”*.

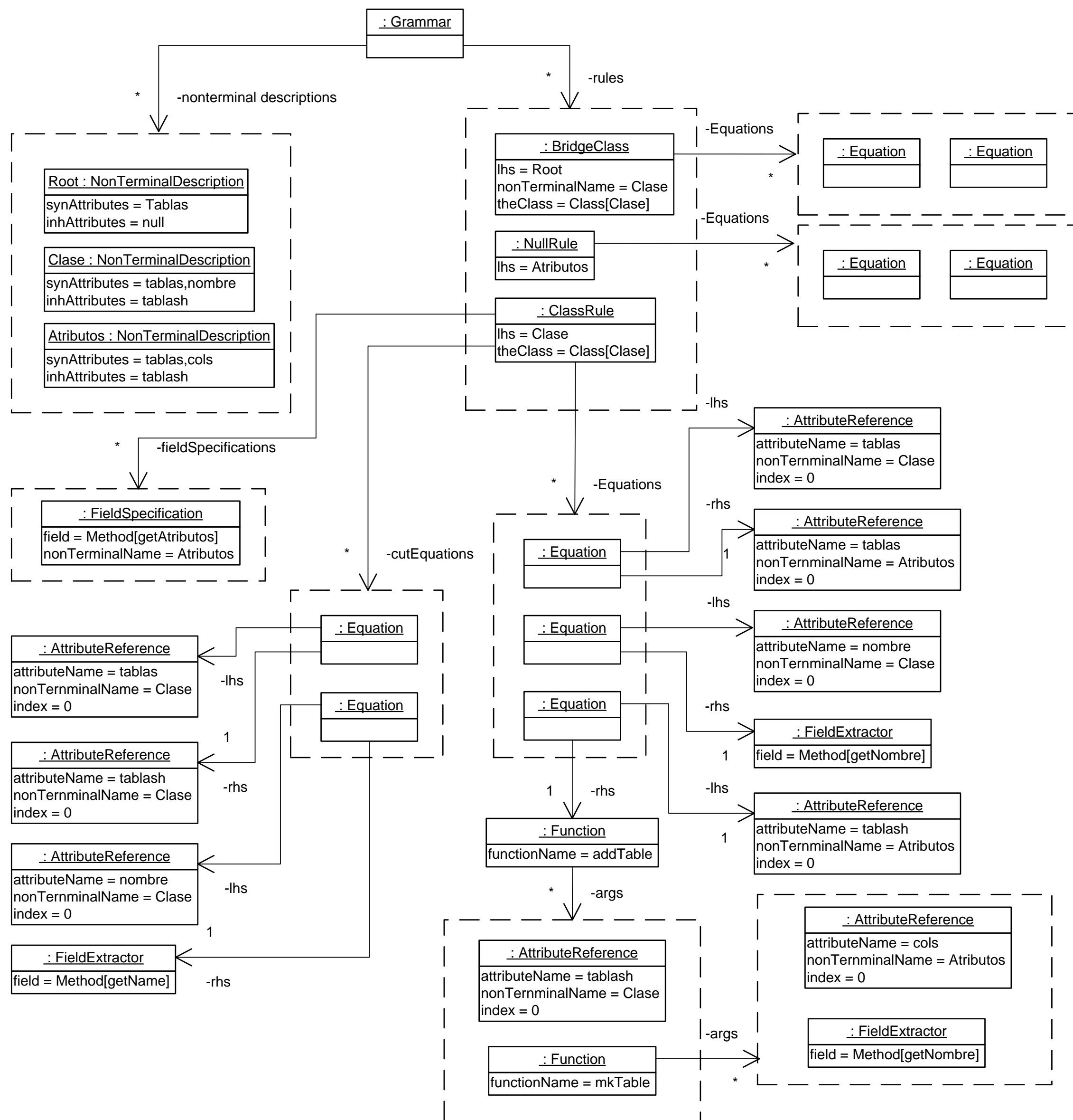

Figura 41. Fragmento de la representación de la gramática de ejemplo mediante el metamodelo de AGT.

2.5 La clase semántica

La clase semántica es una clase programada en Java que implementa las funciones semánticas definidas en la gramática de atributos. Cada transformación podrá tener su propia clase semántica, y, de hecho, dicha clase semántica podrá variarse entre aplicaciones de la

transformación (éste es el motivo por el qué en las instancias de la clase *Function* en el metamodelo no se almacena un objeto de tipo *Method*, sino directamente el nombre de la función semántica). La clase semántica, al poder se programada en Java, otorga una completa flexibilidad al usuario a la hora de especificar las operaciones que se realizarán en el proceso de transformación. Como restricción, se impone que la clase no puede tener dos o más funciones con el mismo nombre, ya que esto supondría diferenciarlas no sólo por su nombre sino también por el tipo de sus parámetros, lo que en esta propuesta no está contemplado, al no estar AGTL estáticamente tipado.

2.6 El Traductor

El traductor es el encargado de crear un modelo basado en el metamodelo AGT a partir de una especificación realizada en AGTL, de tal manera que se pueda realizar posteriormente la transformación de modelos deseada. Este componente integra dos etapas: una de análisis léxico, y otra que amalgama el análisis sintáctico y semántico de AGTL, así como las acciones de instanciación del metamodelo necesarias para construir un modelo AGT correcto.

2.6.1 El analizador léxico

Como es habitual en la construcción de procesadores de lenguajes textuales, la primera fase en la traducción es realizar un análisis lexicográfico [59]. Este análisis se encarga de localizar los componentes léxicos o palabras que constituyen el lenguaje que se está describiendo. Para ello, dicho análisis se basa, como es también habitual, en la descripción de dichos componentes mediante expresiones regulares. La entrada de este analizador es una secuencia de caracteres, la cual el analizador convierte en una secuencia de componentes léxicos que utiliza el analizador sintáctico. Para ello, el analizador léxico funciona en modo *pull*: cuando el analizador sintáctico requiere el siguiente componente léxico (token), el analizador léxico avanza en el proceso de la entrada hasta encontrarlo, y entonces se lo pasa al analizador sintáctico según un formato convenido.

Desde un punto de vista general, la estructura de la especificación de un analizador léxico es un conjunto de patrones basados en expresiones regulares, que tienen asociados acciones a ejecutar, las cuáles normalmente suele finalizar con la devolución de componentes léxicos o *tokens* asociados a dicho patrones. Dichos componentes identifican la *clase* de componente léxico reconocido (la *categoría léxica* del token), y, dependiendo de su tipo, pueden incluir información adicional (*atributos léxicos*) para las posteriores etapas de análisis.

Además de esta función, el analizador léxico nos permite, sobre el análisis del fichero de texto:

- Ignorar los comentarios.
- Ignorar espacios en blanco, tabuladores, retorno de carro.
- Reconocer las palabras reservadas del lenguaje.

- Llevar la cuenta del número de línea y carácter que va leyendo, por si se produce algún error, dar información acerca del lugar donde se ha producido.
- Avisar de errores léxicos, por ejemplo caracteres no permitidos.

El analizador léxico de AGT se ha desarrollado, como ya se ha indicado, con ayuda de JFlex. JFlex es un generador de analizadores lexicográficos realizado por Grewin Klein como extensión a la herramienta JLex desarrollada en la Universidad de Princeton. Está programado en Java, y genera código Java a partir de una especificación realizada en un lenguaje análogo al utilizado en Lex, un generador de analizadores léxico clásico incluido en los sistemas tipo Unix e implementado inicialmente en lenguaje C [75].

En AGT, la especificación del analizador y el fichero java generado automáticamente por JFlex se encuentran en el paquete *com.agt.parser*. En la distribución fuente del marco se denominan *Scanner.lex* y *Scanner.java* respectivamente. En la figura 42 se presenta un fragmento de la especificación del analizador léxico utilizado en la implementación.

```
        ...
1.      [\t\r\n ] {}
2.      #[^\n]*   {}
3.      nt   {return sf.newSymbol("NT",ParserSym.NT,yyline,yycolumn);}
4.      \(   {return sf.newSymbol("(",ParserSym.OPAR,yyline,yycolumn);}
5.      \)   {return sf.newSymbol(")",ParserSym.CPAR,yyline,yycolumn);}
6.      \[   {return sf.newSymbol("[",ParserSym.OSPAR,yyline,yycolumn);}
7.      \]   {return sf.newSymbol("]",ParserSym.CSPAR,yyline,yycolumn);}
8.      \{   {return sf.newSymbol("{",ParserSym.OCPAR,yyline,yycolumn);}
9.      \}   {return sf.newSymbol("}",ParserSym.CCPAR,yyline,yycolumn);}
10.     \,   {return sf.newSymbol(",",ParserSym.COLOM,yyline,yycolumn);}
11.     \.   {return sf.newSymbol(".",ParserSym.DOT,yyline,yycolumn);}
12.     \:   {return sf.newSymbol(":",ParserSym.DOTS,yyline,yycolumn);}
13.     \=   {return sf.newSymbol("=",ParserSym.EQ,yyline,yycolumn);}
14.     \:\:\=   {return sf.newSymbol("::=",ParserSym.IMP,yyline,yycolumn);}
15.     @        {return sf.newSymbol("@",ParserSym.AT,yyline,yycolumn);}
16.     null     {return sf.newSymbol("null",ParserSym.NULL,yyline,yycolumn);}
17.     {nonTerminalName}  {return
        sf.newSymbol("NonTerminalName",ParserSym.NON_TERMINAL_NAME,yyline,yycolumn,yyt
        ext());}
18.     {iden}  {return sf.newSymbol("IDEN",ParserSym.IDEN,yyline,yycolumn,yytext());}
19.     {Num}   {return sf.newSymbol("NUM",ParserSym.NUM,yyline,yycolumn,yytext());}
        …
```

Figura 42. Fragmento especificación del analizador léxico de AGTL.

Cada línea de la figura 42 es la especificación de un patrón y la respectiva acción, marcada entre corchetes '{}', que se debe realizar cuando se reconoce dicho patrón. Se puede observar en la primera línea cómo son ignorados los tabuladores, retorno de carro y símbolos de salto de línea. En la línea dos son ignorados los comentarios, que en AGLT son todo texto comprendido entre la almohadilla *(#)* y el final de la línea. Se puede apreciar, en las líneas restantes, las acciones asociadas con el reconocimiento de los diferentes componentes léxicos:

en dichas acciones se devuelve el token correspondiente, como un objeto *Symbol* que contiene la información necesaria para el posterior análisis:

- El código de la categoría léxica asociada al token. Dichos códigos son fijados por el analizador sintáctico a través de constantes definida en la clase *ParserSym*.
- La línea donde fue reconocido el patrón: *yyline*.
- La posición del carácter desde el inicio de línea donde fue reconocido el patrón: *yycolum*.
- Para los componentes léxicos para los que proceda, *yytext* el texto reconocido (el *lexema* del token).

Como ejemplo se pueden observar las líneas 3 y 16 que reconocen las palabras reservadas *nt* y *null* respectivamente o la línea 19 que reconoce un número mediante una expresión regular asociada con el identificador *Num*, que no se encuentra especificada en el fragmento expuesto.

2.6.2 El analizador sintáctico – analizador semántico – traductor

En la fase de análisis sintáctico se realiza el procesamiento de la secuencia de los *tokens* obtenidos en el análisis léxico, basándose en una gramática especifica del lenguaje a procesar. En AGT se sigue la organización típica de traducción en una pasada, según la cual el analizador sintáctico dirige el proceso de traducción, de manera que las otras fases, como el análisis semántico y la traducción, van evolucionando a medida que el sintáctico va reconociendo y validando la secuencia de tokens entrada [59]. De esta manera, el analizador sintáctico incorpora acciones semánticas que son las que comprueban las restricciones semánticas del lenguaje, informan de los errores que se van encontrando, y realizan la traducción.

El analizador sintáctico – semántico – traductor de AGT se ha desarrollado utilizado el generador de analizadores sintácticos CUP. CUP es un generador automático de analizadores sintácticos a partir de la especificación de gramáticas del tipo LALR(1) [59], que fue desarrollado en el Instituto de Tecnología de Georgia (EE.UU.). CUP genera código Java y permite incluir, en la especificación de la gramática, llamadas a acciones semánticas escritas en dicho lenguaje. Dichas llamadas se invocan durante la *reducción* de las reglas a las que están asociadas (véase [59] para más detalles sobre el funcionamiento interno de los analizadores del tipo de los generados por CUP). Las implementaciones de estas acciones semánticas son las que construyen el modelo AGT y realizan la validación semántica. Las acciones semánticas en el analizador de AGT invocan, fundamentalmente, métodos contenidos en la clase *ModelConstructor*, que también almacena información de estado útil para relacionar entre sí las invocaciones de distintas acciones, evitando la farragosa propagación de atributos heredados en analizadores ascendentes. En la distribución fuente de AGT, la gramática utilizada para construir el analizador sintáctico se encuentra especificada en el fichero *Parser.cup* y el código generado por *Cup* en los ficheros *ParserCup.java* y *ParserSym.java*

El diagnóstico de errores es una componente fundamental dentro del proceso de traducción, ya que permite identificar y localizar los errores presentes en la especificación. En AGT, el manejo de errores lo realiza una sola clase llamada *ErrorManager*. Dicho manejador informa de los siguientes tipos de errores:

- Errores léxicos. Se corresponden con el descubrimiento de caracteres que no puede formar parte de una especificación AGTL.
- Errores sintácticos. Se corresponden con estructuras sintácticas mal formadas. Dentro de estos errores sintácticos se encuentra por ejemplo, la detección de paréntesis mal balanceados o que no aparezca un determinado símbolo de puntuación cuando se esperaba dicho símbolo.
- Errores semánticos. Son errores que violan las restricciones semánticas de AGTL: por ejemplo la redefinición de un atributo dentro de un mismo no terminal, o que se haga referencia a un no terminal que no se encuentra especificado.

En el traductor de AGT, se lleva a cabo un tratamiento simple de todos estos errores: si se encuentra un error, el proceso se aborta y se informa de la posición dentro del fichero de entrada donde ocurrió el error, así como se proporciona un mensaje significativo para ayudar a diagnosticar el mismo.

Como ejemplo, en la figura 43 se puede observar un fragmento de la gramática que define el analizador sintáctico – analizador semánico – traductor de AGT. Se observa en la primera línea la definición de un no terminal *Index*. Nótese, además, que se declara dicho no terminal como de *tipo entero*. Esto permitirá asociar con dicho no terminal un único atributo sintetizado de dicho tipo (en traducción ascendente, el manejo de atributos sintetizados no reviste ningún problema). Las reglas asociadas a dicho no terminal están dadas a continuación:

- En la línea 2 se incluye una regla que contempla el caso en el que no se especifica el índice del no terminal en la ecuación. La acción semántica indica que, en este caso, se toma el valor por omisión 0.

- En la línea 3 se incluye la regla que contempla el caso en el que se examina el índice: se reconocen los *tokens* seguidos *OPAR* (paréntesis de apertura ‘(‘), *NUM* (un numero) y *CPAR* (paréntesis de cierre ‘)’). En este caso, se invoca al método del *modelConstructor makeIndex*, que tiene como parámetro el valor que tenga el *token* NUM. La implementación de este método se incluye en la figura 44, donde se ve que su único propósito es convertir el texto en una instancia de un entero. Cabe resaltar, así mismo, que, según los convenios utilizados en CUP, *RESULT* es el valor sintetizado que toma el no terminal *Index* y que posteriormente será procesado como el índice del no terminal dentro del contexto de AGT.

- Las líneas 4 y 5 introducen producciones especiales orientadas a capturar errores sintácticos: si después de *OPAR* no aparece NUM se invoca al *ErrorManager* con el código de error *NUM_EXPECTED*, o si después de OPAR NUM no viene un *CPAR*, se invoca dicho componente con el código de error CPAR_EXPECTED.

```
   …
1. non terminal Integer Index;
   …

2. Index ::= {: RESULT=0; :};
3. Index ::= OPAR NUM:num CPAR
                {: RESULT=modelConstructor.makeIndex(num);  :};
4. Index ::= OPAR error
                {: errorManager.error(ErrorManager.NUM_EXPECTED);
                RESULT=-1; :};
5. Index ::= OPAR NUM error
                {:errorManager.error(ErrorManager.CPAR_EXPECTED);
                RESULT=-1; :};

6. Null ::= NULL {: modelConstructor.makeNullRule(); :};
```

Figura 43. Fragmento de la gramática que define el analizador sintáctico.

```
public int makeIndex(String index) {
  try {
     return Integer.valueOf(index).intValue();
  }
  catch(NumberFormatException e) {
    errorManager.error(ErrorManager.BAD_INDEX_FORMAT);
    return -1;
  }
}

public void makeNullRule() {
    rule = new NullRule(lhs);
    grammar.addRule(rule);
}
```

Figura 44. Fragmento de las funciones semánticas de la clase *ModelConstructor*.

También se puede apreciar en la línea 6 de la figura 43 cómo se procesa, a nivel sintáctico, el *token* NULL. Como se muestra en la figura 42, este *token* se emite cuando se reconoce la palabra reservada *null*, la cual, dentro de la sintaxis de AGTL, es útil para especificar las reglas que relacionan un no terminal con un objeto nulo. Se observa que, al ser sintetizada esta palabra, se invoca al método *makeNullRule* del *ModelConstructor*. La implementación de este método se incluye también en la figura 44, en la cual se crea una *NullRule* del metamodelo AGT y se agrega a la gramática en construcción. Estos objetos se conservan en campos

internos del *ModelConstructor* para que posteriormente puedan ser modificados según se vaya procesando la gramática de entrada (en este caso, podrá agregarse las ecuaciones que se reconozcan posteriormente). Esta técnica se utiliza de forma extensiva en la construcción del modelo AGT según se va procesando la especificación AGTL.

2.7 El Transformador

En AGT, la transformación de modelos está guiada por la especificación AGTL, representada en términos de una instancia adecuada del metamodelo AGT, y, de acuerdo con la semántica de AGTL, consta de dos etapas:

- Una fase de análisis, donde se lleva a cabo un recorrido del modelo origen, dirigido por el modelo AGT de la transformación, durante el cual se crean las estructuras necesarias para realizar la transformación. Dichas estructuras consisten, básicamente, en el grafo de dependencias entre los atributos del árbol de análisis, donde cada atributo se encuentra debidamente decorado con el campo *valor*, el *flag* que indica si el atributo ha sido o no ha sido computado, y con una *instancia* de la expresión semántica que permite computar el valor del atributo. De esta forma, en AGT se optimiza el modelo conceptual de procesamiento definido en la sección 2.3.2, al evitar construir explícitamente el árbol de análisis.

- Una fase de evaluación, donde se obtiene el valor del atributo del no terminal raíz del árbol de análisis asociado al modelo que contendrá una referencia al objeto de inicio del modelo objetivo deseado.

En las siguientes secciones se presentaran en detalle estas dos etapas.

2.7.1 La fase de análisis

Dado que, en el modelo conceptual de transformación asociado a AGTL, el árbol de análisis se utiliza únicamente como base para llevar a cabo el cómputo de los atributos, una vez que se dispone del grafo de dependencias entre atributos, dicho árbol deja de ser necesario. Como dicho grafo puede construirse superponiendo los subgrafos asociados a los padres e hijos en el árbol, y como el análisis de los modelos se lleva a cabo siguiendo una estrategia recursiva descendente (el modelo origen se explora de izquierda a derechas, preferente en profundidad), no es necesario construir explícitamente el árbol de análisis, sino que puede construirse directamente el grafo de dependencias entre atributos. La fase de análisis en AGT está precisamente orientada a construir dicho grafo. En este proceso, los nodos del árbol de análisis se utilizan como meros contenedores de atributos, de manera temporal mientras se explora cada regla. La representación involucrada se lleva a cabo mediante las clases que se presentan en la figura 45, y que pertenecen al paquete *com.agt.structure* en el marco AGT.

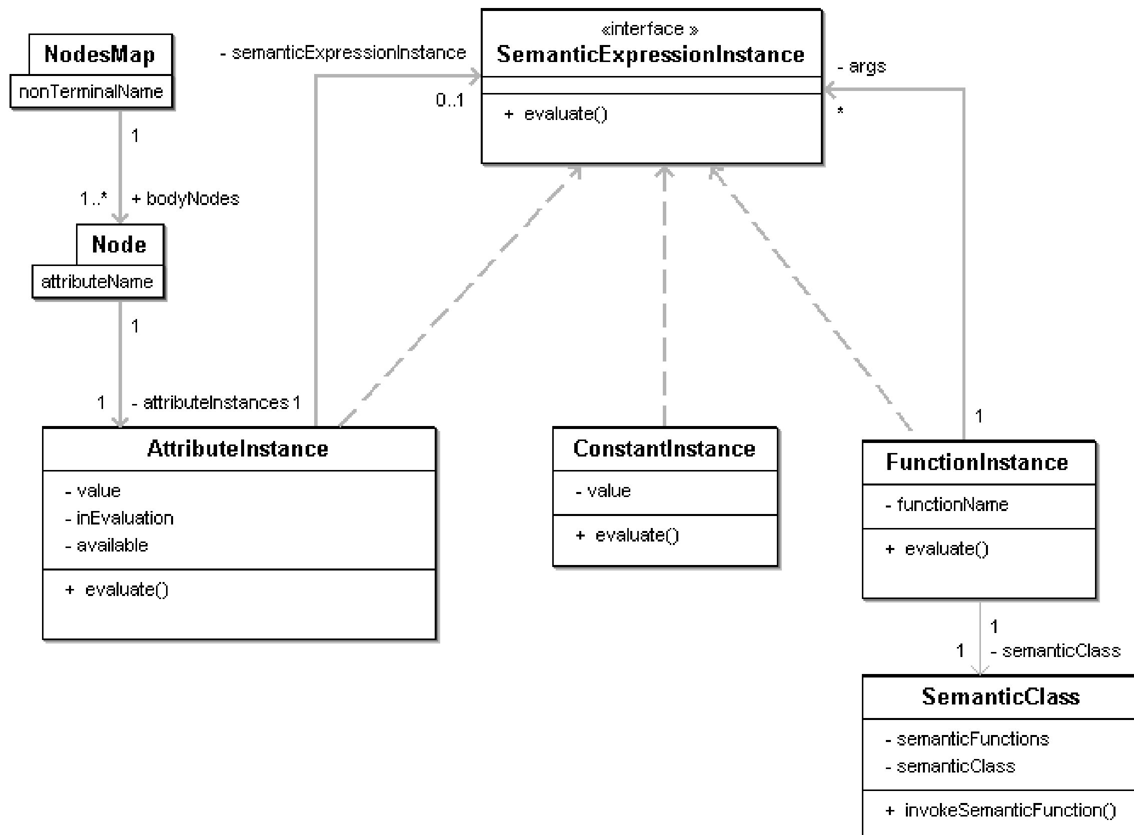

Figura 45. Diagrama de clases de los elementos que soportan las estructuras de análisis y evaluación.

La fase de análisis se inicia a partir de un objeto raíz del modelo origen, y un *axioma* (que es el identificador de un no terminal), y está dirigido por el modelo de la gramática AGT construido previamente, que como ya se ha indicado contiene la definición de los no terminales y de las reglas. El proceso en sí se ajusta fielmente al modelo conceptual descrito en la sección 2.3.2.1:

- El estado de análisis se caracteriza por: un objeto del modelo origen (o bien **null**), el no terminal respecto al que se desea llevar a cabo el análisis, y el conjunto de objetos *activos* que se utiliza para detectar ciclos (este conjunto se representa mediante *NodesMap*). Inicialmente, el estado de análisis viene dado por el objeto raíz *o*, por el axioma *A*, y por el conjunto vacío de objetos activos $\emptyset$.
- Dado un estado de análisis <*o*, *A*, Φ>:
    - Si *o* es un array Java, se crea un objeto *o*’ de tipo *com.agt.core.Array* que lo envuelva, y se realiza el análisis con respecto a <*o*’, *A*, Φ>.
    - En otro caso, se busca la primera regla aplicable, dado *o* y *A*, siguiendo los convenios ya establecidos anteriormente. En caso de no encontrar tal regla, el modelo es rechazado como sintácticamente inválido, señalándose la correspondiente excepción.

- En caso de haber encontrado la regla:
    - Se crea un nodo contenedor de atributos *n* (instancia de *Node*).
    - Se consulta en el modelo AGT los atributos sintetizados y heredados de *A*, y se crean dichos atributos en el nodo (instancias de la clase *AttributeInstance*).
    - Si $o \in \Phi$, el proceso detecta un ciclo. En este caso, si la regla obtenida tiene una sección de ecuaciones de corte, se crea una instancia de la expresión semántica asociada con cada ecuación (objeto de tipo *SemanticExpressionInstance*), y dicha instancia se asocia con el correspondiente atributo sintetizado en el nodo *n*. En caso de que la regla no tenga sección de ecuaciones de corte, el proceso finaliza señalando el error mediante una excepción.
    - Si $o \notin \Phi$:
        - Si la regla aplicable es una regla de clase:
            - Para cada no terminal *B* asociado con un campo *c* de la clase en la regla:
                - Se consulta el valor *o'* del campo *c* en *o*, apelando al método lector correspondiente. Dicho método debe tener, por convenio, el nombre get*C*, donde *C* es el nombre del campo *c* con la primera letra puesta en mayúsculas.
                - Se lleva a cabo un análisis respecto al estado $<o', A, \Phi \cup \{o\}>$
            - Utilizando la secuencia Ω de los nodos contenedores resultantes de dichos análisis, y el nodo *n* creado anteriormente, se instancian las expresiones semánticas de las ecuaciones, fijando las expresiones de los atributos sintetizados en *n*, y la de los atributos heredados en los correspondientes nodos contenedores de Ω.
        - Si la regla es una regla puente:
            - Se lleva a cabo el análisis respecto al estado $<o, B, \Phi>$, siendo *B* el no terminal en la parte derecha de la regla.
            - Utilizando el nodo *n'* resultante de dicho análisis, y el nodo *n* creado anteriormente, se instancian las expresiones semánticas de las ecuaciones, y se fijan en los correspondientes atributos.

        - Si la regla es una regla nula, se instancian las correspondientes expresiones semánticas y se fijan en los correspondientes atributos sintetizados de *n*.
    - Por último, se devuelve *n* como resultado del análisis.

En este proceso, el mecanismo de instanciación de las expresiones semánticas del modelo AGT merece un mayor detalle. Para facilitar dicho proceso, durante la aplicación de cada regla se crea un *map* de listas de nodos, llamado *NodeMap,* e indexado por el nombre de los no terminales. Dado un no terminal *B*, el nodo *i*-esimo en la lista asociado al mismo representa el nodo asociado con la ocurrencia i-esima de *B* en la producción. Esta estructura, por tanto, es la base para llevar a cabo la instanciación de las expresiones semánticas. Efectivamente, las expresiones semánticas del modelo AGT brindan la facilidad de obtener las instancias requeridas con el método *createInstance*, tal y como se menciono en el apartado 2.4. Dicho método toma como argumento el *NodeMap*, así como el objeto en el estado de análisis, y la instancia de la clase semántica. De esta forma:

- Las expresiones de tipo *FieldExtractor* se instancian creando objetos de tipo *ConstantInstace*. En este caso, como se dispone del objeto al cuál se le está aplicando la regla, es posible consultar el valor del campo en dicho objeto, valor que se pasa como argumento al constructor de *ConstantInstance*.
- Las expresiones de tipo *Function* se instancian creando instancias de la clase *FunctionInstance*. Para ello, primeramente instancian sus argumentos, y, a partir de dicha lista de argumentos, el nombre de la función, y la instancia de la clase semántica, producen la correspondiente instancia de *FunctionInstance*.
- Las expresiones de tipo *AttributeReference* utilizan el *NodeMap* para encontrar el nodo correspondiente al no terminal referido, y recuperan de él el correspondiente atributo (instancia de *AttributeInstance*), utilizando dicho valor como resultado de la instanciación. La consecuencia de este último proceso, de hecho, establece los enlaces entre las instancias de los atributos, reflejando de manera implícita el grafo de dependencias. De hecho, en esta representación, los arcos del grafo se expanden en términos de las instancias de las expresiones semánticas de las ecuaciones que lo inducen.

Para ilustrar las estructuras que se crean en el proceso de análisis, en la figura 46 se observan las instancias creadas al analizar la regla *(<Atributos> ::= com.agt.core.Array).* Este proceso, en esencia, crea instancias de los atributos especificados en la regla, junto con instancias de otras expresiones semánticas, que en este caso sólo supone la instanciación de la función semántica *addAtribute*. Los atributos y otras expresiones semánticas se encadenan según lo dictan las ecuaciones.

Figura 46. Fragmento de las instancias creadas en la fase de análisis.

Una vez creada la estructura descrita, es suficiente quedarse con el nodo que representa el axioma, el cual referencia el atributo (*AttributeInstance*) que contendrá el resultado de la transformación. Este atributo ya se encuentra encadenado a toda la información necesaria para ser calculado, lo que evita la necesidad de recorrer el árbol de análisis; en su lugar, se usa directamente el grafo de dependencias construido, que está dado por el mismo encadenamiento de los atributos. De esta forma, este nodo, junto con el atributo, se pasa a la fase de evaluación.

En esta etapa también se valida la clase semántica, la cual se envuelve en un objeto de tipo *SemanticClass*. Esta clase provee un acceso por nombre eficiente a los métodos, para facilitar

su invocación dinámica. La clase semántica no puede tener métodos con el mismo nombre; en caso contrario, se lanza la excepción *RedefinedSemanticMethodException*.

2.7.2 La fase de evaluación

Aplicando la estrategia de evaluación por demanda, la fase de evaluación es directa, y simplemente se desencadena consultando el método *evaluate* del atributo que, finalmente, contendrá el resultado de la transformación. Dicha estrategia puede entenderse mejor examinando el código que implementa dicho método, el cual se muestra en la Figura 47. Dicha implementación evidencia la estrategia de evaluación bajo demanda:

- Si el valor está disponible, el método lo devuelve.
- Si no, el método invoca el método *evaluate* de su expresión semántica. Tal y como se muestra en la Figura 47, la evaluación de la expresión semántica desencadenará, en última instancia, la invocación de los métodos *evaluate* de los atributos de los que se depende, con idéntico comportamiento.

```
//AttributeInstance
	public Object evaluate() throws EvaluationException {
		if (available) {
			return value;
		} else {
			if (inEvaluation) {
				throw new CircularDependencyException(this);
			} else {
				inEvaluation = true;
				value = semanticExpressionInstance.evaluate();
				available = true;
				inEvaluation = false;
				return value;
			}
		}
	}

// ConstantInstance
	public Object evaluate() {
		return value;
	}

//FunctionInstance
	public Object evaluate() throws EvaluationException {
		Object[] vals = new Object[args.size()];
		int i = 0;
		for (SemanticExpressionInstance semExp : args) {
			vals[i++] = semExp.evaluate();
		}
		return semanticClass.invokeSemanticFunction(functionName, vals);
	}
```

Figura 47. Funciones de evaluación que implementa cada instancia de una expresión semántica.

Obsérvese, así mismo, que el mecanismo incluye de forma directa un chequeo de circularidad. Efectivamente, si, cuando se invoca la evaluación de un atributo, éste ya se encuentra evaluándose, se detecta un error y se lanza la excepción *CircularDependencyException*.

2.8 A modo de conclusión

En este capítulo se ha descrito el marco de transformación AGT, un marco basado en gramáticas de atributos para la realización directa de transformaciones modelo a modelo. De esta forma, este capítulo constata la factibilidad de basar este tipo de transformaciones en el formalismo de gramáticas de atributos, evitando, al mismo tiempo, tener que recurrir a sintaxis textuales auxiliares, como ocurre con el trabajo de May Dehayni y Louis Féraud.

La principal característica de AGT es aplicar principios ampliamente utilizados en el dominio de la construcción de procesadores de lenguaje a la transformación entre modelos. Esto permite aplicar técnicas sistemáticas propias del modelo de *análisis – síntesis* dirigido por la sintaxis, utilizado en la construcción de procesadores de lenguaje. La sistemática resultante ofrece un método perfectamente definido para la escritura de transformaciones:

- Inicialmente el escritor de la transformación se concentra en la estructura del modelo origen, concibiendo dicha estructura como un árbol que recubre dicho modelo, y caracterizándola mediante un conjunto de reglas sintácticas.

- Seguidamente, el escritor se concentra en cómo llevar a cabo la transformación, concibiendo este proceso como un proceso de cómputo de valores de atributos semánticos asociados con los nodos de la estructura impuesta sobre el modelo origen. Para ello introduce atributos apropiados, así como ecuaciones para computar el valor de dichos atributos. Así mismo, el escrito analiza en qué puntos del proceso de análisis del modelo pueden descubrirse ciclos, e introduce ecuaciones de corte en las reglas correspondientes.

- Por último, el escritor puede programar las funciones semánticas utilizadas en dichas ecuaciones directamente en Java, lo que permite disponer de todo el poder expresivo de dicho lenguaje en la caracterización de las operaciones básicas aplicadas durante el proceso de transformación.

El propósito del desarrollo de AGT no ha sido producir un nuevo marco de transformación que *compita* con los analizados en el capítulo anterior, sino probar la ya citada factibilidad de basar la especificación de transformaciones directas modelo a modelo en el paradigma de las gramáticas de atributos. De esta forma, AGT debe tomarse más como una prueba de concepto experimental que como un sistema listo para su utilización en producción. Efectivamente, AGT carece de muchas características de lenguajes y marcos de transformación más maduros, como, por ejemplo, mecanismos de trazabilidad y soporte para formatos estándares de metamodelado. De hecho, AGT, en su versión actual, no es más que una herramienta de transformación de redes de objetos Java en redes de objetos Java. Sin embargo, la versión operativa de AGT desarrollada en este proyecto permite especificar declarativamente, de manera bastante natural, transformaciones relativamente complejas, como la ilustrada en este

capítulo, o como el caso de estudio que se analizará en el capítulo siguiente, con el que se pretende demostrar que AGT, aún encontrándose aún en una fase muy inicial de su concepción, ofrece un paradigma de transformación de aplicación práctica a casos reales.

3 CASO DE ESTUDIO

3.1 Introducción

En esta sección se presenta un caso de estudio en el dominio de e-Learning que permite llevar a cabo una evaluación cualitativa inicial sobre el uso del marco de transformación AGT en casos reales y más complejos que los considerados en los capítulos previos. Más concretamente, el objetivo de este caso de estudio es realizar una transformación de modelos entre un metamodelo de alto nivel para la descripción de tutoriales interactivos y el metamodelo del *framework* de <e-tutor>, una aplicación experimental desarrollada en el grupo de Ingeniería del Software aplicada al e-Learning del Dpto. de Ingeniería del Software e Inteligencia Artificial de la Facultad de Informática de la UCM que permite reproducir este tipo de tutoriales[76].

Este caso de estudio representa varios retos, debido a la complejidad de los metamodelos y, en particular, la forma de construcción de los elementos del metamodelo destino, los cuales obviamente no pueden ser modificados. De esta forma, el caso de estudio será apropiado para poner a prueba las ventajas y desvelar las posibles desventajas que pueden surgir en la utilización de la propuesta realizada en este trabajo de investigación.

La estructura del capítulo es como sigue. La sección 3.2 describe brevemente el sistema <e-Tutor>. La sección 3.3 propone un metamodelo de alto nivel para la descripción de tutoriales interactivos del tipo de los reproducibles en <e-Tutor>. La sección 3.4 aborda la implementación AGT de la transformación de los modelos de alto nivel en los modelos reproducibles por <e-Tutor>. Por último, la sección 3.4.4 concluye el capítulo.

3.2 <e-Tutor>

La utilidad del sistema <e-Tutor> es el desarrollo de tutores socráticos basados en los trabajos de Alfred Bork y su equipo durante los años 80 [77]. Este tipo de tutores permite presentar contenidos de aprendizaje con preguntas interactivas y respuestas que dependen de las interacciones que el usuario realiza conforme avanza en la aplicación.

Un tutorial socrático analiza las respuestas del estudiante a las preguntas planteadas, proporcionando a la vez una realimentación apropiada, y decide el próximo paso que se realizará en el proceso de aprendizaje. Tal realimentación puede adaptar distintos itinerarios

de aprendizaje. La adaptación más general podría depender de la historia completa de la interacción del estudiante con el sistema. No obstante, <e-Tutor> adopta un mecanismo más simple basado en contadores, al estilo de los utilizados por Bork en sus trabajos [77]. De esta forma, el sistema asocia contadores a cada posible respuesta de cada pregunta, de tal manera que cada vez que el estudiante da una respuesta se incrementa el contador asociado. Así, la realimentación y el siguiente paso a dar pueden depender del valor de dichos contadores. Este hecho se ejemplifica en la figura 48, donde se puede apreciar un fragmento de un tutorial socrático de ejemplo, que consiste en una pregunta (¿*Cual es la capital de Brasil?*), una clasificación de tres posibles respuestas dadas por el usuario (intuitivamente en *Otra* encaja cualquier respuesta diferente de las otras dos posibilidades –*Brasilia y Rio de Janeiro*-), y las posibles realimentaciones. En esta figura se muestra claramente que, dependiendo de la respuesta y de la cantidad de intentos de responderla (marcada por el número entre paréntesis) se producen diferentes realimentaciones y se define el flujo que seguirá el tutorial.

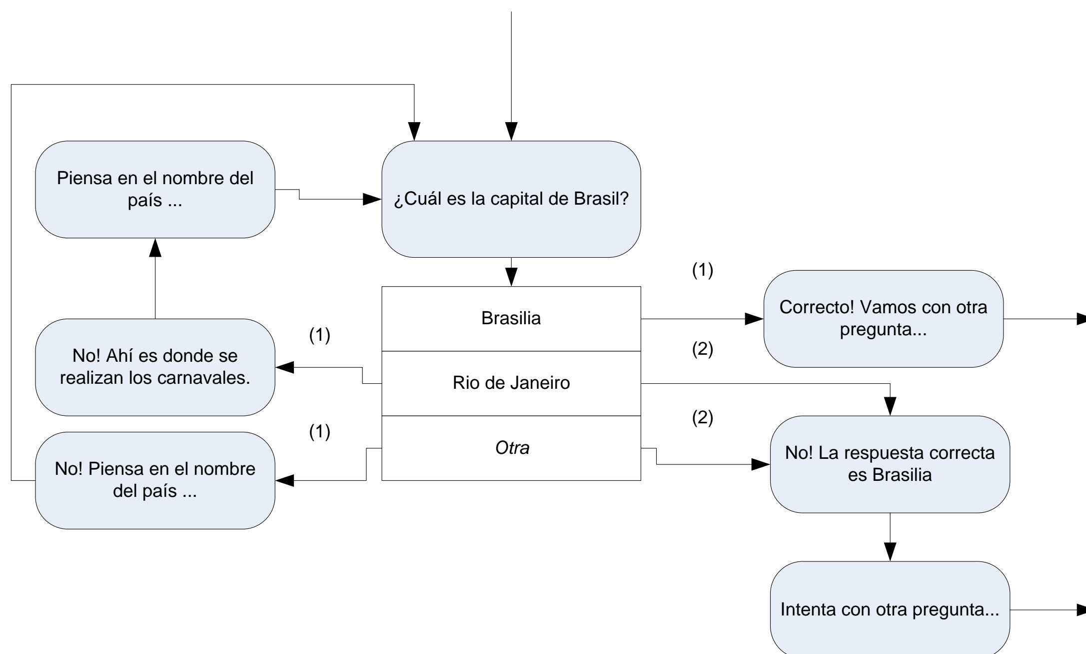

Figura 48. Fragmento de un tutorial socrático (ejemplo tomado de[78]).

La Figura 49 presenta el metamodelo del *framework* de <e-tutor> que rige cómo deben representarse los tutoriales para ser ejecutados en <e-Tutor>. Dicho *framework* posee más clases para su ejecución, pero estos componentes no son relevantes para la transformación que se desea estudiar (véase[76] para una descripción más completa). En este metamodelo, el elemento principal es la clase *ETTutorial*, que representa el tutorial en sí, al cual se añaden los demás componentes. Esta clase posee el método que inicia la ejecución del tutorial y configura la manera en la que se visualizarán los componentes. El tutorial comienza con la ejecución de un elemento inicial, derivado de la clase abstracta *ETTutorialElement,* establecido mediante el

método *setInitialTutorialElement* y se ejecuta mediante el método *run*. El orden de ejecución de los siguientes elementos se realiza mediante el atributo *next* de la clase *ETTutorialElement,* y por lo tanto de las clases que la especializan. La ejecución secuencial de algunos de estos elementos se puede realizar de manera temporizada, de tal forma que una vez acabado un tiempo de espera, se ejecuta el siguiente elemento de la cadena. La clase abstracta *ETTutorialTemporizedElement* implementa dicha funcionalidad, donde mediante el método *setDelay* se puede ajustar el tiempo de espera y el método *play* es el que inicia dicho intervalo de espera. El método *present* debe ser implementado por los elementos concretos, e implementa la forma en la que los elementos serán presentados dentro de la aplicación. De esta manera, el método *play* es invocado por el objeto tutorial sobre cada elemento generalizado de *ETTutorialElement* que se vaya alcanzando*,* mediante el flujo de elementos dado por el tutorial, pasando de un elemento a otro, ya sea por la finalización del tiempo de espera o por la interacción del usuario. Como elementos básicos sólo se tendrán en cuenta para el caso de estudio ETText, que representa un mensaje de texto de estilo configurable, y ETImage, que representa una imagen a mostrar.

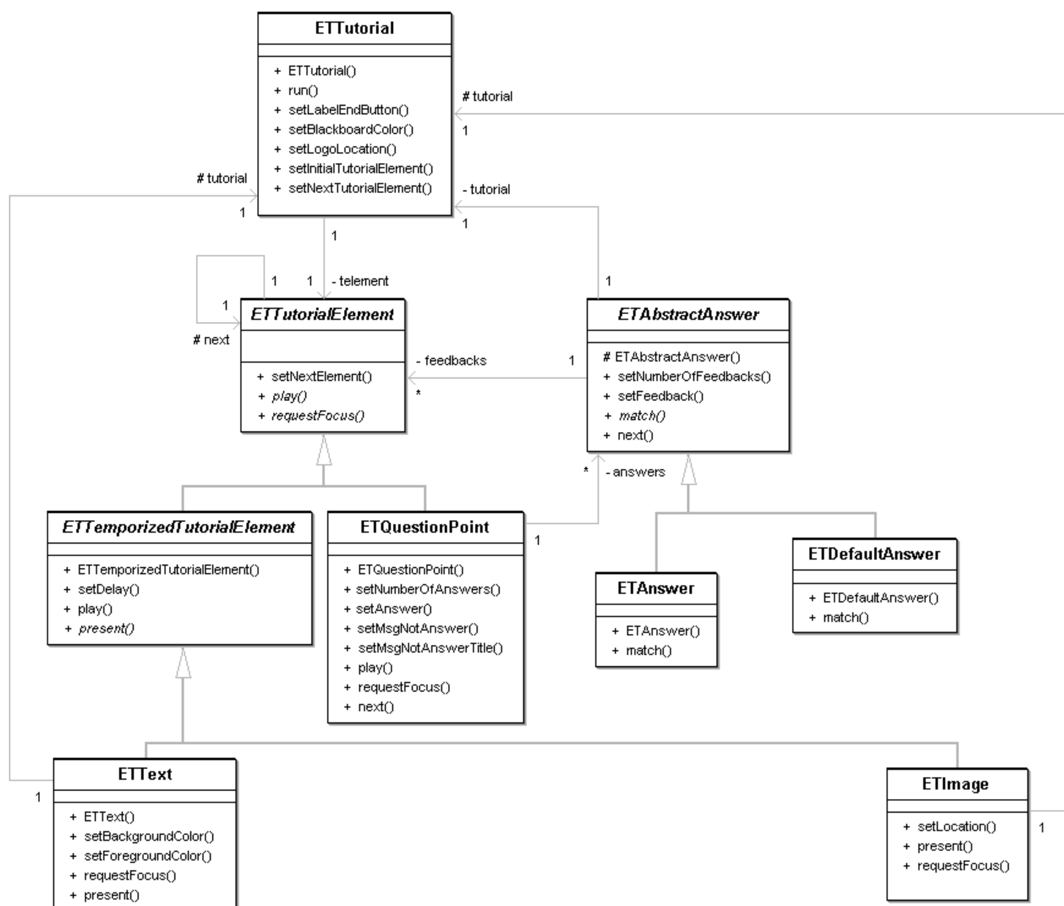

Figura 49. Metamodelo <e-tutor>

La interacción del usuario en el momento de responder una pregunta está representada por la clase *ETQuestionPoint,* la cual posee una lista de respuestas posibles, que pueden ser respuestas concretas *ETAnswer* o respuestas por defecto *ETDefaultAnswer*, ambas generalización de *ETAbstractAnswer*. El método *match* es el encargado de determinar si una respuesta coincide con la dada por el usuario. Si es así, se aumenta el contador de la respuesta con el método *next()*; dicho contador es el índice de la lista de *feedbacks, que* representa la realimentación mediante otro *ETTutorialElement,* que se mostrará posteriormente en la aplicación. Si la respuesta dada no coincide con ninguna respuesta se utiliza la respuesta por defecto, y si se acaban los *feedbacks* de una respuesta, el tutorial llega a su fin.

3.3 Un Metamodelo de Alto Nivel para la Representación de Tutoriales Interactivos

El metamodelo de <e-Tutor> presentado en la sección anterior es de demasiado bajo nivel, y está demasiado adherido a la *plataforma* <e-Tutor>. Por tanto, es interesante disponer de mecanismos descriptivos de más alto nivel para los tutoriales. La figura 50 propone un metamodelo para tales descripciones. Este metamodelo es útil para realizar una especificación de un tutorial totalmente independiente de la parte funcional de la aplicación.

En el metamodelo propuesto:

- La clase principal es *Tutorial*. Esta clase incluye las características básicas de presentación de un tutorial como el título, el logo, el texto que tiene el botón de finalización y color de fondo de la aplicación. *Tutorial* incluye también el atributo *init,* que indica el primer elemento que se mostrará en el tutorial. Dicho elemento ha de ser de tipo diálogo (*Speech)*.
- Los elementos de tipo *Speech* poseen, a su vez, una propiedad *content* que representa el contenido asociado con los mismos. Así mismo, se relacionan con el siguiente elemento en el tutorial por medio del campo *next.*
- Por otra parte, el metamodelo también incluye las preguntas (*Question*), las cuales también tienen asociado un contenido, además de otras características de presentación.
- Las preguntas también tienen una seria de realimentaciones (*Feedback*). Cada realimentación lleva al siguiente dialogo. El elemento *Feedback* posee como campos una posible respuesta (*answer*) y un índice (*count*) que indica la cantidad de intentos. Como *Feedbacks* por omisión, es decir, los que se utilizan cuando la respuesta dada no se ajusta a ninguna otra, se utiliza la lista indicada por *defaultSpeeches* de la clase *Question*; el orden en el que se ha de proporcionar las realimentaciones es el indicado por su posición en la lista.
- Finalmente, el contenido asociado con diálogos y con preguntas puede ser de tipo *Image* o de tipo *Text*. El contenido de tipo *Image* representa una imagen, y posee

como campo la ruta de dicha imagen. El contenido de tipo *Text* representa un mensaje textual, el cual, aparte del texto a mostrar, tiene como campos valores de presentación como el color de fondo, o el color y estilo de la fuente del texto.

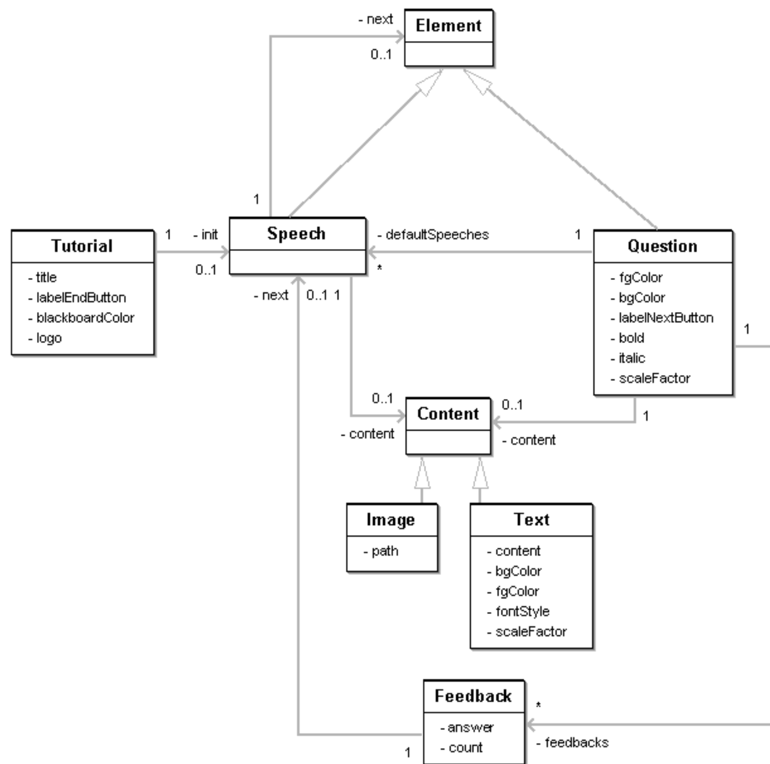

Figura 50. Metamodelo de alto nivel para la representación de tutoriales interactivos.

3.4 Transformación AGT

En esta sección se presentará la especificación AGT de la transformación de los modelos de tutoriales interactivos de alto nivel en los modelos fijados por el *framework* de <e-Tutor>. En las siguientes secciones se presentará el estilo con el cual se ha desarrollado la transformación (sección 3.4.1), así como la descripción de la misma mediante el lenguaje AGTL (sección 3.4.2). También se analizará la clase semántica utilizada para la transformación, la cual juega un importante papel en esta transformación debido a las restricciones de configuración que presenta el *framework* de <e-Tutor>.

3.4.1 Estilo de Transformación

La especificación de la transformación deseada resulta excesivamente compleja si se pretende realizar bajo un estilo completamente *funcional* (es decir, considerando que las funciones semánticas no producen efectos laterales). Esto es debido a la manera en la que son construidos los componentes del *framework*, que, en una especificación puramente funcional,

obligaría a mantener gran cantidad de atributos heredados y sintetizados, mediante sus respectivas ecuaciones, de tal manera que los valores calculados deberían propagarse sucesivamente a lo largo de todo el árbol de análisis para obtener el resultado esperado. Para simplificar la especificación, se ha planteado el uso de un estilo de especificación avanzada en gramáticas de atributos. Este estilo, descrito en [79], consiste en que las funciones semánticas se conciben como operaciones de un *tipo abstracto de datos* que mantiene explícitamente estado. De tal manera, el estado en sí se guarda como valores de campos en las instancias de la clase semántica. Por su parte, los atributos en la gramática representan *cambios* en dicho estado.

Para este caso de estudio, la clase semántica incluye explícitamente una referencia al elemento principal *ETTutorial* que, como indica el metamodelo del *framework* de <e-tutor>, es el elemento al cual se agregan los demás componentes. De esta forma, no es necesario propagar dicha referencia por todo el árbol de análisis.

Para resolver este caso de estudio son también necesarias algunas clases que mantengan temporalmente un conjunto de elementos para su posterior utilización en otras reglas. Tal es el caso de las respuestas y diálogos (*Speech*). El conjunto de respuestas de una pregunta debe ser procesado antes de ser agregado a un *QuestionPoint,* dada la restricción que posee el metamodelo objetivo. Efectivamente, el *framework* de <e-Tutor> obliga a que se deba conocer la cantidad de respuestas (método *setNumberOfAnswers*) antes de agregarlas una a una (método *setAnswer*). En cuanto a los diálogos, es evidente que se puede generar un ciclo en el momento de procesar un diálogo transformado previamente, de tal manera que es necesaria una ecuación de corte en el instante en que esto suceda. Se necesitará, por tanto, un *map* auxiliar que mantenga las transformaciones realizadas para los *Speechs*, y que permita recuperar el correspondiente elemento transformado previamente para su encadenamiento con los demás elementos.

En la especificación también se usará una función semántica *after*, que toma como parámetro dos expresiones semánticas, y devuelve siempre el valor de su segundo argumento. Dado que la evaluación de expresiones semánticas es puramente aplicativa, esta función será útil para introducir dependencias entre computaciones que cambian el estado de la clase semántica. Otra función semántica auxiliar será *getNull,* la cual simplemente retorna un objeto nulo.

3.4.2 Descripción de la transformación

La transformación, informalmente, se puede describir de la siguiente manera:

- Se creara un *ETTutorial* configurado a partir de los atributos de *Tutorial*.
- Los elementos *ETTemporizedTutorialElement* serán creados a partir del contenido (*Content*) de los diálogos (*Speech*) y preguntas (*Question*), donde cada referencia a una

instancia de *Image* producirá una instancia de *ETImage* y cada referencia a una instancia de *Text* producirá una instancia de *ETText*.

- Las apariciones de instancias de *Question* supondrá crear, además, un *ETQuestionPoint,* que estará referenciado como el siguiente elemento del *ETTemporizedTutorialElement* también creado durante la transformación de *Question.*
- Cada conjunto de *Feedbacks* de una *Question* con el mismo valor en *answer* será transformado en una única *ETAnswer*, referenciada por la *ETQuestionPoint* resultante, de la transformación de *Questión*. Por su parte, el resultado de tran*sformar el Speech referido* por el atributo *next* de cada uno de estos *Feedbacks* debe ser agregado co*mo* un ítem de la lista de *feedbacks* de *ETAnswer,* res*ultante* de la transformación del mismo *Feedback*, según el *count* indicado en el metamodelo *orig*en.
- La lista defaultSpeeches, campo de *Question*, se transforma en una *ETDefaultAnswer,* que tendrá como lista de *feedbacks* referencias a las respectivas transformaciones de los contenidos de *Speech* en *ETTemporizedTutorialElement.* La *ETDefaultAnswer* estará referenciada por la *ETQuestionPoint* resultante de la transformación de la instancia de *Question* a la que pertenece la lista *defaultSpeeches*.
- El atributo *next* de *Speech* será transformado en el atributo *next* del correspondiente *ETTemporizedTutorialElement* y referenciará al correspondiente *ETTemporizedTutorialElement* creado a partir del elemento siguiente (*Speech* o *Question*). Si el atributo *next* de Speech es nulo, entonces el *ETTemporizedTutorialElement* también será nulo.

En la figura 51 se da una posible especificación en AGTL para realizar la transformación deseada. Esta especificación caracteriza sintácticamente el metamodelo origen como sigue:

- El objeto raíz ha de ser de tipo *Tutorial*, y exhibir un campo *init*, que, a su vez, ha de referir a una *Speech* (regla 1).
- Los elementos del tutorial pueden ser de tipo *Question* (regla 2), *Speech* (regla 3), o bien una referencia nula (regla 4). En el caso de elementos de tipo *Question*, estos han de incluir un campo *content* refiriendo al contenido, un campo *feedbacks*, refiriendo a las realimentaciones, y un campo *defaultSpeeches* refiriendo a las realimentaciones por defecto (regla 2).
- Por su parte, como indica la regla 5, una *Speech* ha de exhibir un campo *content*, refiriendo al contenido, y un campo *next*, refiriendo al siguiente elemento del tutorial.
- Las reglas 6 y 7 caracterizan la lista de *feedbacks* de las *Questions*.
- La regla 8 caracteriza cada *Feedback*.
- Las reglas 9 y 10 caracterizan las listas de las realimentaciones por defecto.
- Por último, las reglas 11 y 12 caracterizan los posibles contenidos (*Image* y *Text*).

En lo que se refiere a la parte semántica de la especificación, como se mencionó anteriormente, la clase semántica mantiene explícitamente el contexto en forma de estado.

Por lo tanto, en este caso no es necesario utilizar atributos heredados para representar dicho contexto, haciendo más sencilla la especificación. Efectivamente:

```
     nt(<Tutorial>,[],[tutorial]).
     nt(<Element>,[],[elem]).
     nt(<Speech>,[],[elem]).
     nt(<DefaultFeedbacks>,[],[defaultFeedbacks]).
     nt(<DefaultFeedback>,[],[defaultFeedback]).
     nt(<Feedbacks>,[],[answers]).
     nt(<Feedback>,[],[feedback]).
     nt(<Content>,[],[elem]).


1.  <Tutorial> ::= Tutorial{init: <Speech>}
               {<Tutorial>.tutorial = after(makeTutorial(),
                    configureTutorial(<Speech>.elem,@title,@labelEndButton,
                    @blackboardColor,@logo))}.

2.  <Element> ::= Question{content:<Content>,feedback:<Feedbacks>,
                defaultSpeeches:<DefaultFeedbacks>}
                {<Element>.elem = after(processQuestion(<Content>.elem,
                    <Feedbacks>.answers, <DefaultFeedbacks>.defaultFeedbacks,@fgColor,
                    @bgColor, @labelNextButton,@bold, @italic, @scaleFactor),
                    <Content>.elem)}.

3.  <Element> ::= Speech:<Speech>{<Element>.elem = <Speech>.elem}.

4.  <Element> ::= null{<Element>.elem = getNull()}.

5.  <Speech> ::= Speech {content:<Content>,next:<Element>}
                {<Speech>.elem = after(processSpeech(@this,
                    <Content>.elem,<Element>.elem),<Content>.elem)}
                {<Speech>.elem = getSpeech(@this)}.

6.  <Feedbacks> ::= com.agt.core.Array {first: <Feedback>,butfirst: <Feedbacks>}
                {<Feedbacks>(0).answers =
                     addFeedback(<Feedback>.feedback,<Feedbacks>(1).answers)}.

7.  <Feedbacks> ::= null {<Feedbacks>.answers = newFeedbacks()}.

8.  <Feedback> ::= Feedback{next: <Speech>}
                {<Feedback>.feedback = makeFeedback(@answer,@count,<Speech>.elem)}.

9.  <DefaultFeedbacks> ::= com.agt.core.Array {first:<Speech>,
                butfirst: <DefaultFeedbacks>}
                {<DefaultFeedbacks>(0).defaultFeedbacks =
                      addDefaultFeedback(<Speech>.elem,
                      <DefaultFeedbacks>(1).defaultFeedbacks)}.

10. <DefaultFeedbacks> ::= null
                {<DefaultFeedbacks>.defaultFeedbacks = newDefaultFeedbacks()}.


11. <Content> ::= Image{}
                {<Content>.elem =newImage(@path)}.

12. <Content> ::= Text{}
                    {<Content>.elem =
                    newText(@content,@bgColor,@fgColor,@fontStyle,@scaleFactor)}.
```

Figura 51. Especificación en AGTL de la transformación del caso de estudio.

- En la regla 1 se establece que el *tutorial* se *configura después de crearlo* (función semántica *after*). La creación se realiza mediante la función semántica *makeTutorial*,

que crea un ETTutorial vacío que se mantendrá referenciado en la clase semántica mientras se realiza la transformación, y al cuál se agregarán los otros componentes del tutorial. La configuración se lleva a cabo invocando la función semantica *configureTutorial*, que configurará el tutorial con los valores de los campos de presentación en la instancia de *Tutorial*, y agregará a dicho tutorial el primer elemento, que es la transformación del primer *Speech* (atributo <Speech>.*elem* en la ecuación semántica).

- La ecuación semántica asociada a la regla 2 permite computar, como valor del atributo sintetizado *elem*, el elemento de tutorial que resulta de transformar el contenido referido desde la *Question*, *después de haber procesador dicha* Question. El procesamiento se lleva a cabo mediante la función semántica *processQuestion*, que lleva a cabo el procesamiento a partir del contenido, la información de las realimentaciones y de las realimentaciones por defecto, y los valores de los campos de presentación.
- La regla 3 es una mera regla puente, y la ecuación semántica asociada una ecuación de copia, que establece que el resultado de transformar un elemento que resulta ser una *Speech* es el resultado de transformar dicha *Speech*.
- La ecuación semántica en la regla 4 establece que los elementos referidos que son nulos se transforman en referencias nulas en el correspondiente modelo <e-Tutor>.
- La primera ecuación semántica en la regla 5 establece que el resultado de transformar una *Speech* es el resultado de transformar su contenido, *una vez que se ha procesado dicha* Speech. Dicho procesamiento se lleva a cabo mediante la función semántica *processSpeech*, que realiza el procesamiento a partir de la transformación del contenido, así como del elemento del tutorial <e-Tutor> que sigue a dicha transformación. Por su parte, esta regla introduce también una ecuación de corte: en caso de que la *Speech* se analice de nuevo en el mismo camino, se recupera el resultado previo de la transformación mediante la función semántica *getSpeech*. Dicha información habrá sido almacenada previamente en un *map* en la instancia de la clase semántica por la función semántica *processSpeech*.
- Las ecuaciones en las reglas 6 y 7 sintetizan información intermedia relativa a las respuesta, como valor del atributo semántico *answers*. Los valores de dichos atributos agrupará la información de los posibles *feedbacks* para las distintas respuestas. Dicha agrupación es necesaria debido a las peculiaridades del *framework* <e-Tutor>, que obliga a conocer el número de *feedbacks* de cada respuesta antes de registras los correspondientes elementos, así como el número de *respuestas* de una pregunta antes de registrar las mismas en dicha pregunta. De esta forma, la función semántica *addFeeback* se encarga de actualizar la citada clasificación con el nuevo *feedback* examinado, mientras que la función semántica *newFeedbacks* se encarga de producir una clasificación inicial de *feedbacks* por respuestas.

- La ecuación de la regla 8 recopila información respecto al *feedback* a partir de la transformación de la siguiente *Speech* referida, así como de los correspondientes atributos de presentación. La función semántica *makeFeedback* se encarga de dicho aspecto.
- Las ecuaciones en las reglas 9 y 10 recolectan los *feedbacks* por defecto por medio de los atributos *default*Feedbacks. Los valores de dichos atributos son listas de *feedbacks*. La función semántica *addDefaultFeedback* añade un nuevo *feedback* a la lista, mientras que la función *newDefaultFeedback* crea una lista vacía.
- Por último, las ecuaciones semánticas de las reglas 11 y 12 se encargan de transformar los contenidos en elementos de <e-Tutor> adecuados, aplicando, para ello, las funciones semánticas *newImage* y *newText*.

3.4.3 La Clase semántica

En la figura 52 se presenta un fragmento de la clase semántica utilizada para la transformación. En ésta se pueden observar los campos que guardan el estado, *tutorial* y *speechElements*. También se puede ver cómo se definen las funciones semánticas *after*, *getNull* y *makeTutorial*, las cuales fueron presentadas en secciones anteriores.

Así mismo, también se incluye la definición de una función más compleja: *processQuestion*. En esta definición puede observarse la construcción de una *QuestionPoint,* además de cómo son construidas y agregadas las instacias de *ETDefault*Answer, y cómo también son agregadas las *ETAnswer* que están almacenadas en un objeto *AnswersInfo,* que es una estructura que almacena las *ETAnswer* construidas como resultado del procesamiento de los *Feedbacks,* aspecto que se omite en este fragmento. Finalmente se puede ver cómo se enlaza el elemento previo con este nuevo *QuestionPoint* construido al *ETTemporizedTutorialElement* que, dada la especificación, es la transformación del contenido de una instacia de *Question*.

Es interesante observar también la implementación de la función semántica *processSpeech*, que lleva a cabo el enlace de un *ETTemporizedTutorialElement* con la instacia de un *ETTutorialElement* que representa el siguiente componente del tutor. Notese que en este método se realiza el registro del objeto *Speech* con su transformación correspondiente, de tal manera que pueda recuperarse la instacia de la transformacón ya realizada mediante la función *getSpeech*, que tambien aparece en el fragmento.

Finalmente tambien se puede observar en el fragmento cómo se construye un componente *ETText* a partir de los atributos de la clase *Text,* mediante la función semática *newText.* En la implementación de esta función se puede observar cómo el campo *tutorial* de la clase semantica se utiliza para dicha construcción, evitando que en la especificación sea necesario la propagación de un atributo a través del árbol de análisis. El codigo completo de la clase semantica utilizada para la transformación del caso de estudio puede encontrarse en el apendice A.

```java
public class ETutorSemanticClass {
	private ETTutorial tutorial;
	private Map<Object, ETTemporizedTutorialElement> speechElements;

	public Object after(Object o1, Object o2) {
		return o2;
	}

	public Object getNull() {
		return null;
	}

	public void makeTutorial() {
		tutorial = new ETTutorial();
	}

	public ETTutorial configureTutorial(ETTutorialElement e, String title,
			String labelEndButton, Color blackboardColor, String logoPath) {
		tutorial.setInitialTutorialElement(e);
		tutorial.setTitle(title);
		tutorial.setLabelEndButton(labelEndButton);
		tutorial.setBlackboardColor(blackboardColor);
		tutorial.setLogoLocation(logoPath);
		return tutorial;
	}

	public void processQuestion(ETTemporizedTutorialElement questionElement,
			AnswersInfo answers,
			List<ETTemporizedTutorialElement> defaultFeedbacks,
			String fcolor,
			String bcolor, String lnb, String isBold, String isItalic,
			String sfactor) {
		ETQuestionPoint qp = new ETQuestionPoint();
		ETInput input = new ETSimpleInput();
		qp.setInput(input);
		qp.setNumberOfAnswers(answers.getNumber() + 1);
		Iterator<String> answerTexts = answers.getAnswerTexts();
		int i = 0;
		while (answerTexts.hasNext())
			qp.setAnswer(i++, answers.getAnswer(answerTexts.next()));
		ETDefaultAnswer da = new ETDefaultAnswer();
		da.setNumberOfFeedbacks(defaultFeedbacks.size());
		int j = 0;
		for (ETTemporizedTutorialElement defaultFeedback : defaultFeedbacks) {
			da.setFeedback(j++, defaultFeedback);
		}
		qp.setAnswer(i, da);
		questionElement.setNextElement(qp);
	}

	public void processSpeech(Object speechID,
			ETTemporizedTutorialElement speechElement,
			ETTutorialElement nextElement) {
		speechElement.setNextElement(nextElement);
		speechElements.put(speechID, speechElement);
	}


	public ETTemporizedTutorialElement getSpeech(Object speechID) {
		return speechElements.get(speechID);
	}

	public ETText newText(String content, Color bgColor, Color fgColor,
			int fontStyle, double scaleFactor) {
		ETText eTText = new ETText();
		eTText.setText(content);
		eTText.setFontScaleFactor(scaleFactor);
		eTText.setBackgroundColor(bgColor);
```

```
                eTText.setForegroundColor(fgColor);
                eTText.setFontStyle(fontStyle);
                eTText.setTutorial(tutorial);
                return eTText;
        }
...
}
```

Figura 52. Fragmento de la clase semántica para la transformación del caso de estudio.

3.4.4 Aplicación de la transformación

A fin de ejemplificar el efecto de la transformación, considérese el modelo origen descrito directamente en Java en la figura 53 (por simplicidad, se omiten detalles de presentación), que a su vez representa el fragmento del tutorial de la figura 48. La aplicación de la transformación sobre dicho modelo permite obtener una representación <e-Tutor> que, a su vez, puede ejecutarse dentro de sistema para obtener la ejecución del tutorial mostrada en la figura 54.

```
Tutorial tutorial = new Tutorial();
Speech speech = new Speech();
Text bienvenida = new Text();
bienvenida.setContent("Bienvenido!");
speech.setContent(bienvenida);
tutorial.setInit(speech);

Speech speech1 = new Speech();
Speech speech2 = new Speech();
Speech speech3 = new Speech();
Speech speech4 = new Speech();
Speech speech5 = new Speech();

Text text1 = new Text();
Text text2 = new Text();
Text text3 = new Text();
Text text4 = new Text();
Text text5 = new Text();

text1.setContent("No! Ahí es donde se realizan los carnavales");
text2.setContent("No! Piensa en el nombre del país ...");
text3.setContent("Correcto! Vamos con otra pregunta...");
text4.setContent("No! La respuesta correcta es Brasilia.");
text5.setContent("Intenta con otra pregunta...");

speech1.setContent(text1);
speech2.setContent(text2);
speech3.setContent(text3);
speech4.setContent(text4);
speech5.setContent(text5);

Question question = new Question();
speech.setNext(question);
Text pregunta = new Text();
pregunta.setContent("Cual es la capital de Brasil?");

Feedback feedback1 = new Feedback();
Feedback feedback2 = new Feedback();
Feedback feedback3 = new Feedback();

feedback1.setAnswer("Brasilia");
feedback1.setCount(1);
feedback1.setNext(speech3);

feedback2.setAnswer("Rio de Janeiro");
feedback2.setCount(1);
```

```
feedback2.setNext(speech1);

feedback3.setAnswer("Rio de Janeiro");
feedback3.setCount(2);
feedback3.setNext(speech4);

question.addFeedbacks(feedback1);
question.addFeedbacks(feedback2);
question.addFeedbacks(feedback3);

question.addDefaultSpeechs(speech2);
question.addDefaultSpeechs(speech4);

speech4.setNext(speech5);
```

Figura 53. Especificación de un modelo de alto nivel para <e-tutor>

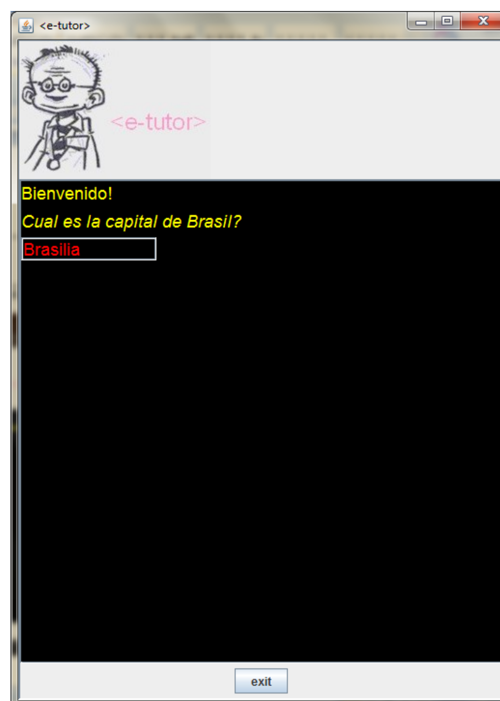

Figura 54. Captura del modelo <e-tutor> en ejecución.

3.5 A modo de conclusión

Como se ha podido comprobar durante el desarrollo del caso de estudio, AGT provee suficiente expresividad y alcance para poder expresar transformaciones complejas. La clase semántica otorga la flexibilidad suficiente para realizar operaciones complejas, dado que se dispone de toda la flexibilidad ofrecida por el lenguaje Java a efectos de proporcionar las operaciones locales requeridas por cada transformación. Así mismo, en este ejemplo se ha visto cómo es posible simplificar la especificación de la gramática mediante el manejo explícito de estado en la clase semántica.

El caso de estudio presenta varios desafíos, dada la manera en que se construyen sus componentes. Sin embargo AGT puede ajustarse para enfrentarlos, pero es evidente que la transformación, en especial la clase semántica se vuelve compleja, añadiendo estructuras auxiliares, que vuelven el proceso difícil de seguir y especificar. En particular, en el caso de estudio esbozado en este capítulo, es notorio observar que el código de la clase semántica excede en tamaño a la especificación de la transformación en AGTL. No obstante, en este caso, dicho efecto es lógico, debido a la peculiaridad de los mecanismos de configuración introducidos por el metamodelo del *framework* <e-Tutor>. Obsérvese que, sea cual fuere el modelo de transformación elegido, sería necesario abordar los aspectos especificados declarativamente en AGTL. De esta forma, mediante AGTL es posible especificar declarativamente la manera de procesar el modelo origen, mientras que en Java es posible proporcionar las operaciones locales que se precisan para llevar a cabo dicho procesamiento. En cualquier caso, una manera de simplificar la complejidad de la clase semántica sería *encadenar* dos transformaciones AGTL: una del metamodelo origen a una representación intermedia que contenga la información de configuración requerida, tal y como se hace en la especificación de la transformación introducida, y otra de dicho modelo intermedio de *configuración* al modelo <e-Tutor>. No obstante, resultando la misma obvia, se omitirá el desarrollo de esta alternativa, dado que no aporta ningún aspecto nuevo sobre el uso de AGT, objetivo principal de este capítulo.

CONCLUSIONES Y TRABAJO FUTURO

En este proyecto de investigación se ha explorado la factibilidad de basar la transformación directa de modelos en el formalismo de las gramáticas de atributos. Para ello, se ha diseñado e implementado un marco de transformación, AGT, basado en estos conceptos. AGT incluye un lenguaje de especificación, AGTL, que permite especificar transformaciones entre modelos siguiendo un estilo similar al que se adopta cuando se especifica un traductor para un lenguaje informático convencional:

- Por una parte, en AGTL se especifica la *sintaxis* del metamodelo origen. Para ello, AGTL permite describir cómo superponer un árbol de análisis que *recubre* cada modelo procesado. Dicho árbol de análisis es análogo al árbol de análisis sintáctico que una gramática libre del contexto convencional puede imponer sobre una frase.

- Por otra parte, AGTL permite expresar el procesamiento siguiendo un estilo dirigido por dicha sintaxis, añadiendo atributos semánticos y ecuaciones semánticas para computar los valores de dichos atributos en los nodos del árbol de análisis.

Con AGTL los usuarios pueden concentrarse en especificar las reglas que serán utilizadas en la evaluación de los atributos, dejando de lado la necesidad de indicar el orden de ejecución de dichas reglas. En cuanto al motor de transformación, el árbol sintáctico en AGT no se crea, manteniéndose sólo el grafo de dependencias entre atributos, lo cual reduce el gasto de recursos. Por otra parte, AGTL se integra totalmente con Java, a través de la clase semántica, que implementa las funciones semánticas utilizadas en las ecuaciones. La clase semántica aporta una gran flexibilidad, y añade todo el potencial del lenguaje Java, una de las ventajas de propuestas como MT (véase el capítulo **Error! Reference source not found.**), el cual está implementado como un lenguaje especifico de dominio (DSL) embebido. También se ha podido comprobar en este proyecto que la inclusión explícita de estado en la misma contribuye a reducir considerablemente el tamaño de la especificación. La clase semántica es, además, un punto de extensión del que puede tomar ventaja el usuario, por ejemplo para agregar comportamiento de trazabilidad personalizado.

Más allá de estos detalles técnicos, la propuesta realizada aplica un modelo clásico dirigido por la sintaxis utilizado en el procesamiento de lenguajes informáticos al escenario de la transformación *modelo a modelo*. De esta forma, las reglas semánticas se encuentran especificadas de manera declarativa y ligadas a la estructura sintáctica, permitiendo que las

transformaciones sean expresadas de forma concisa y clara, y garantizando que éstas operan sobre los modelos sobre los que tienen que operar La modularidad de una especificación también se garantiza por la estructura en bloques de las gramáticas de atributos, derivada de las reglas de producción, así como por la no necesidad de expresar el orden en el que tienen que aplicarse las ecuaciones semánticas; por el contrario, dicho orden se deriva de las dependencias entre atributos Esto hace posible añadir nuevas reglas sintácticas y semánticas, de forma tal que la maquinaria de la transformación se reorganiza automáticamente.

La principal diferencia entre el modelo de procesamiento introducido por AGT y los modelos imperantes en los lenguajes de transformación analizados en el capítulo **Error! Reference source not found.** radica en el estilo dirigido por la sintaxis del primero, frente al estilo *transformacional* de los segundos. Efectivamente, para realizar una transformación, AGT primeramente *analiza* la estructura sintáctica del modelo, y, seguidamente, *sintetiza* el resultado evaluando valores de atributos semánticos en dicha estructura. Por su parte, los lenguajes analizados funcionan manteniendo una especie de *memoria de trabajo* con el modelo actual, *seleccionando* fragmentos de dicho modelo, y *transformando* dichos fragmentos, hasta alcanzar el resultado deseado. Desde este punto de vista, en este tipo de lenguajes es muy difícil, sino imposible, verificar estáticamente que la entrada aceptada por una transformación se ajusta al metamodelo origen, o que la salida generada por la misma se ajusta al metamodelo destino. Efectivamente, estructura y procesamiento se encuentran desligados, desacoplados, al contrario de lo que propugna el paradigma basado en gramáticas de atributos. Estas consideraciones son, así mismo, aplicables a los enfoques basados en gramáticas de grafos, ya que, como se ha indicado en el capítulo **Error! Reference source not found.**, dichas gramáticas se utilizan, en realidad, como vehículos de transformar grafos en grafos, en lugar de cómo soporte a un modelo clásico de *análisis – síntesis* para el procesamiento de lenguaje.

La propuesta realizada en este trabajo de investigación y encarnada en AGT es similar, de esta forma, a la propuesta de May Dehayni y Louis Féraud [1]. No obstante, la principal diferencia es, como ya se ha comentado repetidas veces en este documento, que AGT evita el uso de sintaxis textuales intermedias para codificar los modelos, contrastando, de esta forma, el uso de un formalismo basado en gramáticas de atributos para especificar *directamente* transformaciones *modelo a modelo*.

Como se ha podido comprobar a lo largo de este documento, la propuesta realizada en este proyecto de investigación, en esencia, realiza una transformación de redes de objetos en otras redes de objetos, en línea con la caracterización del problema de la transformación *modelo a modelo* en el contexto del desarrollo de software dirigido por modelos. De esta forma, la propuesta puede ajustarse a otros estilos de modelado diferentes de los imperantes en las propuestas sujetas a modelado basado en lenguajes tipo UML. De hecho, AGT considera, en realidad, metamodelos representados como clases Java y modelos representados como

instancias de dichas clases. De esta forma, AGT puede ser aplicable en otros contextos que puedan mapearse en este convenio de representación tan laxo: por ejemplo, en la transformación de programas o estructuras de datos en ejecución. Además el desarrollo de todo el marco de transformación, incluido el evaluador, el cual se ha llevado a cabo en Java, puede permitir una integración más natural con otras herramientas, especialmente dentro de MDA, con marcos como *Eclipse Modeling Framework,* que también ha sido desarrollado en Java.

AGT aún se encuentra en una etapa temprana de desarrollo. De esta forma, surgen multitud de líneas de trabajo futuro, entre las cuáles destacamos las siguientes:

- Actualmente AGTL únicamente considera campos de tipos atómicos o de tipos *array*. Dado que los campos son los elementos básicos utilizados en AGTL para representar relaciones, puede ser interesante ampliar este repertorio, introduciendo, por ejemplo, soporte para otros tipos de datos, como las *tablas asociativas* (*maps*). Para este propósito puede seguirse una estrategia similar a la seguida en la incorporación de *arrays* al lenguaje, mediante el uso de nuevas clases *envoltorio*, del estilo de *com.agt.core.Arrray*.

- Otro aspecto a abordar en relación con AGTL es el relativo a la modularidad. Aunque, como se ha indicado anteriormente, las gramáticas de atributos son naturalmente modulares, actualmente AGTL únicamente soporta especificaciones monolíticas, contenidas en un único archivo. Como trabajo futuro se plantea, de esta forma, introducir mecanismos que permitan modularizar las especificaciones, así como separar las mismas físicamente en múltiples archivos.

- Tal y como se ha evidenciado durante el desarrollo del caso de estudio, cuando se abordan transformaciones complejas puede ser conveniente separar las mismas en varias fases, que se encadenan originando varios modelos intermedios hasta generar el modelo objetivo. A este respecto, se plantea también ampliar AGTL para poder especificar este tipo de cadenas transformacionales.

- Otra característica que puede contribuir a mejorar substancialmente AGT es permitir tipar los atributos en AGTL, y realizar comprobaciones estáticas de tipos. Así mismo, utilizando reflexión puede ser posible comprobar el correcto tipado de las expresiones semánticas en las ecuaciones.

- Una línea de trabajo importante es integrar AGT con propuestas estándar de metamodelado (p.ej., MOF o ECore). Para ello puede dotarse al marco con una arquitectura de conectores que facilite la integración de dichas propuestas.

- Otra línea de trabajo consiste en desarrollar un entorno integrado de desarrollo para AGTL. A fin de facilitar esta tarea, dicho entorno puede basarse en plataformas tipo Eclipse.

- Es necesario, así mismo, analizar cuantitativamente la eficiencia del motor de transformación de AGTL, el cual ha de ser evaluado y comparado, desde el punto de vista de la eficiencia, con otras propuestas.

- Por último, se estima de suma importancia poder observar la usabilidad del lenguaje por parte de usuarios finales y su curva de aprendizaje, e igualmente compararlo, desde aspectos tales como la expresividad o la facilidad de uso, con otros lenguajes de transformación.

REFERENCIAS

[1] May Dehayni y Louis Féraud, "An Approach of Model Transformation Based on Attribute Grammars," in *Object-Oriented Information Systems*.: Springer Berlin / Heidelberg, 2003.

[2] Charles W. Krueger, "Software Reuse," in *ACM Comput. Surv. 24 (2)*, 1992, pp. 131-183.

[3] J. Neighbors, "The Draco approach to constructing software from reusable components," in *Readings in artificial intelligence and software engineering*.: Morgan-Kauffmann, 1986, pp. 525-535.

[4] Sinan Si Alhir, *Learning UML*.: O'Reilly Media, Inc., 2003.

[5] Kunwoo Lee, *Principles of CAD/CAM/CAE*.: Prentice Hall, 1999.

[6] Anneke Kleppe, *Software Language Engineering*. Estados Unidos de América: Addison-Wesley, 2008.

[7] Thomas Stahl et ál, *Model-Driven Software Development*. Inglaterra: John Wiley & Sons, Ltda., 2006.

[8] Paul Klint, Joost Visser Arie van Deursen, *Domain-Specific Languages: An Annotated Bibliography*.: SIGPLAN Notices 35(6), 2000.

[9] Jan Heering, Anthony M. Sloane Marjan Mernik, "When and how to develop domain-specific languages.," *ACM Comput. Surv. 37*, vol. 4, pp. 316 - 344, 2005.

[10] Cristina Videira Lopes, Bedir Tekinerdogan, Gregor Kiczales Kim Mens, "Aspect-Oriented Programming Workshop Report," *ECOOP Workshops*, pp. 483-496, 1997.

[11] David S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley, Ed., 2003.

[12] S. W. Ambler. Agile Modeling (AM) Home Page: Effective Practices for Modeling and Documentation. [Online]. http://www.agilemodeling.com/

[13] Ethan K. Jackson. The Model Integrated Computing Approach to Software Architecture. [Online]. http://research.microsoft.com/en-us/um/people/ejackson/publications/asm07.pdf

[14] D. A. Thomas and B. M. Barry, "Model-Driven Development: the Case for Domain-Oriented Programming," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA*, New York, 2003, pp. 2-7.

[15] K. Short, S. Cook y S. Kent J. Greenfield, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*.: John Wiley and Sons, Inc., 2004.

[16] W3C. (2008, Noviembre) Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online]. http://www.w3.org/TR/xml/

[17] W3C. Document Object Model (DOM). [Online]. http://www.w3.org/DOM/

[18] Object Management Group, Inc. (OMG). (2003, Junio) MDA Guide Version 1.0.1. [Online]. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf

[19] Object Management Group, Inc. (OMG). (2009, Febrero) OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. [Online]. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/

[20] Object Management Group, Inc. (OMG). (2009, Febrero) OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2. [Online]. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[21] Object Management Group, Inc. (OMG). (2003, Marzo) Common Warehouse Metamodel (CWM) Specification. [Online]. http://www.omg.org/cgi-bin/doc?formal/03-03-02.pdf

[22] Pattie Maes, "Concepts and experiments in computational reflection," in *Conference on Object Oriented Programming Systems Languages and Applications OOPSLA'87*, Orlando, Florida, Estados Unidos de America, 1987.

[23] Object Management Group, Inc. (OMG). (2006, Mayo) Object Constraint Language, OMG Available Specification, Version 2.0. [Online]. http://www.omg.org/spec/OCL/2.0/PDF/

[24] The Eclipse Foundation. Eclipse Modeling Framework Project (EMF). [Online]. http://www.eclipse.org/modeling/emf/?project=emf

[25] Object Management Group, Inc. (OMG). (2006, Enero) Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0. [Online]. http://www.omg.org/spec/MOF/2.0/PDF/

[26] Object Management Group, Inc. (OMG). (2007, Diciembre) MOF 2.0/XMI Mapping, Version 2.1.1. [Online]. http://www.omg.org/cgi-bin/doc?formal/2007-12-02

[27] W3C. (1999, Noviembre) XSL Transformations (XSLT), Version 1.0. [Online]. http://www.w3.org/TR/xslt

[28] Object Management Group, Inc. (OMG). (2008, Abril) Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. [Online]. http://www.omg.org/spec/QVT/1.0/PDF

[29] ATL Project. [Online]. http://www.eclipse.org/m2m/atl/

[30] Laurence Tratt. The MT model transformation language. [Online]. http://tratt.net/laurie/research/publications/papers/tratt__the_mt_model_transformation_language_sac.pdf

[31] MOFScript Home page. [Online]. http://www.eclipse.org/gmt/mofscript/

[32] Edward D. Willink. UMLX : A Graphical Transformation Language. [Online]. http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/UMLX/doc/MDAFA2003-4/MDAFA2003-4.pdf

[33] Natalia Correa and Roxana Giandini. Lenguajes de Transformación de Modelos. Un análisis comparativo. [Online]. http://www.cacic2007.unne.edu.ar/papers/037.pdf

[34] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. [Online]. http://www4.informatik.tu-muenchen.de/publ/papers/marschall_braun-mdafa03.pdf

[35] Epsilon. [Online]. http://www.eclipse.org/gmt/epsilon/

[36] A., Kalmar, Z., Karsai, G., Shi, F., Vizhanyo, A. Agrawal. (2003, Noviembre) GReAT User Manual. [Online]. http://www.escherinstitute.org/Plone/tools/suites/mic/great/GReAT%20User%20Manual.pdf

[37] Sun Developer Network: Java Metadata Interface (JMI). [Online]. http://java.sun.com/products/jmi/

[38] D.H. Akehurst, W.G. Howells, and McDonald-Maier K.D. Kent Model Transformation Language. [Online]. http://docs.google.com/viewer?a=v&q=cache:plQHzCr7xUMJ:citeseerx.ist.psu.edu/viewdoc/download%3Fdoi%3D10.1.1.96.6822%26rep%3Drep1%26type%3Dpdf+Kent+Model+Transformation+Akehurst&hl=es&gl=es&pid=bl&srcid=ADGEESglXga2Vln-z0GTjtWXZly47SAsPwgdTIUhpczGixW4dFEd

[39] Kermeta - Breathe life into your metamodels. [Online]. http://www.kermeta.org/

[40] Model to Text (M2T). [Online]. http://wiki.eclipse.org/M2T

[41] Mod-Transf. [Online]. http://modelware.inria.fr/rubrique15.html

[42] MOLA Project. [Online]. http://mola.mii.lu.lv/

[43] AlphaWorks: Model Transformation Framework. [Online]. http://www.alphaworks.ibm.com/tech/mtf

[44] J. Bézivin, and G. Guillaume. M. Peltier. MTRANS: A general framework, based on XSLT, for model transformations. [Online]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.3854&rep=rep1&type=pdf

[45] MTL. [Online]. http://modelware.inria.fr/rubrique.php3?id_rubrique=8

[46] QVT - Wikipedia. [Online]. http://en.wikipedia.org/wiki/QVT

[47] Jesús Sánchez Cuadrado, Jesús J. García Molina, and Marcos Menárguez Tortosa, "RUBYTL: UN LENGUAJE DE TRANSFORMACIÓN DE MODELOS," in *XV Jornadas de Ingeniería del Software y Bases de Datos*, Barcelona, 2006.

[48] Stratego/XT. [Online]. http://strategoxt.org/

[49] Tefka- The EMF Transformation Engine. [Online]. http://tefkat.sourceforge.net/

[50] Project Overview. [Online]. http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html

[51] Executable UML. [Online]. http://en.wikipedia.org/wiki/Executable_UML

[52] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. [Online]. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ATLandQVT-PRELIMINARY%20VERSION.pdf

[53] INRIA. (2005, Marzo) ATL TRANSFORMATION EXAMPLE - Class to Relational. [Online]. http://www.eclipse.org/m2m/atl/atlTransformations/Class2Relational/ExampleClass2Relational[v00.01].pdf

[54] Fowler and Martin. MF Bliki : Domain Specific Language. [Online]. http://www.martinfowler.com/bliki/DomainSpecificLanguage.html

[55] Tata Consultancy Services. (2003, Marzo) Initial submission for MOF 2.0 Query / Views / Transformations RFP : QVT Partners. [Online]. http://www.tratt.net/laurie/research/publications/papers/qvtpartners1.0.pdf

[56] Laurence Tratt, "The QVT-Partners model transformations approach - Issues with the approach," in *The MT model transformation language*.: Department of Computer Science, King's College London, 2005, ch. 3.4, pp. 10-11.

[57] J. Oldevik. (2005, Marzo) MOFScript User Guide. [Online]. http://www.uio.no/studier/emner/matnat/ifi/INF5120/v05/undervisningsmateriale/MOFScript-User-Guide.pdf

[58] Edward D. Willink. (2003, Octubre) A concrete UML-based graphical transformation syntax : The UML to RDBMS example in UMLX. [Online]. www.eclipse.org/gmt/umlx/doc/M4M03/M4M03.pdf

[59] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman, *Compilers: principles, techniques and tools* , 2nd ed.: Addison-Wesley, 2007.

[60] Noam Chomsky. (1956) Three models for the description of language. [Online]. http://www.chomsky.info/articles/195609--.pdf

[61] D. Knuth, "Semantics of context free languages," in *Mathematical Systems theory*., 1968.

[62] Jukka Paakki, "Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation," *ACM Computing Surveys*, vol. 27, no. 2, pp. 196-255, 1995.

[63] Eva Magnusson and Görel Hedin, "Circular reference attributed grammars - their evaluation and applications.," *Sci. Comput. Program.*, vol. 68(1), pp. 21-37, 2007.

[64] Grzegorz Rozenberg, Ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations.*: World Scientific Publishing Co., 1997.

[65] Juan De Lara, Hans Vangheluwe, and Manuel Alfonseca. (2003, Junio) Meta-modelling and graph grammars formulti-paradigm modelling in AToM. [Online]. http://www.cs.mcgill.ca/~hv/publications/03.SoSyM.AToM3.pdf

[66] Andy Schürr. (1995) Specification of Graph Translators with Triple Graph Grammars. [Online]. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.249

[67] Alexander Königs. (2005, Septiembre) Model Transformation with Triple Graph Grammars. [Online].

http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/konigs__model_transformation_with_triple_graph_grammars.pdf

[68] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. [Online]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.5342&rep=rep1&type=pdf

[69] T. Reps and T. Teitelbaum, "The Synthetizer Generator," in *ACM SIGSOFT Software Engineering Notes 9(3)*., 1984, pp. 42 - 48.

[70] U. Kasterns, "Ordered Attributed Grammars," *Arca Informatica*, vol. 13, pp. 229-256, 1980.

[71] Object Management Group, Inc. (OMG). (2004, Agosto) Human-Usable Textual Notation. Version 1.0. [Online]. http://www.omg.org/cgi-bin/doc?formal/2004-08-01

[72] Esther Guerra and Juan de Lara. (2006, Octubre) Attributed Typed Triple Graph Transformation with Inheritance in the Double Pushout Approach. [Online]. http://arantxa.ii.uam.es/~jlara/investigacion/techRep_UC3M.pdf

[73] Gerwin Klein. JFlex - The Fast Scanner Generator for Java. [Online]. http://jflex.de/

[74] Scott Hudson, Frank Flannery, and C. Scott Ananian. CUP. [Online]. http://www2.cs.tum.edu/projects/cup/

[75] E. Schmidt M. E. Lesk. Lex - A Lexical Analyzer Generator. [Online]. http://dinosaur.compilertools.net/lex/

[76] J.L. Sierra, A. Fernández-Valmayor, and B Fernández-Manjón, "From Documents to Applications Using Markup Languages," *IEEE Software* , vol. 25, no. 2, pp. 68-76, 2008.

[77] A. Bork, *Personal Computers for Education*.: Harper & Rows, 1985.

[78] A. Sarasa, J.L. Sierra, and A. Fernández-Valmayor, "Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de e-Learning.," *IEEE-RITA*, vol. 4, no. 3, pp. 175-183, 2009.

[79] U. Kastens, "Attribute Grammars as an Specification Method," in *Attribute Grammars, Application and Systems*, Springer, Ed., 1991.

[80] Jim Hugunin, "Python and Java: The Best of Both Worlds," in *The 6th International Python Conference* , 1997.

[81] Tcl community. About Tcl/Tk, Tcl Developer Xchange. [Online]. http://www.tcl.tk/about/index.html

[82] Frédéric Fondement et ál. Metamodel-Aware Textual Concrete Syntax. [Online]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.9659&rep=rep1&type=pdf

[83] Oracle Corporation. (2010) JavaServer Pages Technology. [Online]. http://java.sun.com/products/jsp/

[84] Henk Alblas, "Attribute Evaluations Methods," in *Attribute Grammars, Applications and Systems*. Praga, Checoslovaquia: Springer-Verlag, 1991, ch. 6, pp. 103-109.

APÉNDICE A. CLASE SEMÁNTICA PARA EL CASO DE ESTUDIO

```java
package es.ucm.fdi.etutor.highLevel;

import java.awt.Color;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

import es.ucm.fdi.eucm.etutor.framework.ETAbstractAnswer;
import es.ucm.fdi.eucm.etutor.framework.ETAnswer;
import es.ucm.fdi.eucm.etutor.framework.ETImage;
import es.ucm.fdi.eucm.etutor.framework.ETInput;
import es.ucm.fdi.eucm.etutor.framework.ETQuestionPoint;
import es.ucm.fdi.eucm.etutor.framework.ETSimpleInput;
import es.ucm.fdi.eucm.etutor.framework.ETTemporizedTutorialElement;
import es.ucm.fdi.eucm.etutor.framework.ETText;
import es.ucm.fdi.eucm.etutor.framework.ETTutorial;
import es.ucm.fdi.eucm.etutor.framework.ETTutorialElement;

public class ETutorSemanticClass {
	private ETTutorial tutorial;
	private Map<Object, ETTemporizedTutorialElement> speechElements;

	public Object after(Object o1, Object o2) {
		return o2;
	}

	public Object getNull() {
		return null;
	}

	public void makeTutorial() {
		tutorial = new ETTutorial();
	}

	public ETTutorial configureTutorial(ETTutorialElement e, String title,
			String labelEndButton, Color blackboardColor, String logoPath) {
		tutorial.setInitialTutorialElement(e);
		tutorial.setTitle(title);
		tutorial.setLabelEndButton(labelEndButton);
		tutorial.setBlackboardColor(blackboardColor);
		tutorial.setLogoLocation(logoPath);
		return tutorial;
	}

	public FeedbackInfo makeFeedback(String answer, int count,
			ETTemporizedTutorialElement nextSpeech) {
		return new FeedbackInfo(answer, count, nextSpeech);
	}

	public AnswersInfo addFeedback(FeedbackInfo feedback, AnswersInfo answers) {
		if (answers.contains(feedback.getAnswer()))
			answers.addFeedback(feedback);
		else
			answers.createAnswer(feedback);
		return answers;
	}
```

```java
public void processQuestion(ETTemporizedTutorialElement questionElement,
                AnswersInfo answers,
                List<ETTemporizedTutorialElement> defaultFeedbacks,
                String fcolor,
                String bcolor, String lnb, String isBold, String isItalic,
                String sfactor) {
        ETQuestionPoint qp = new ETQuestionPoint();
        ETInput input = new ETSimpleInput();
        qp.setInput(input);
        qp.setNumberOfAnswers(answers.getNumber() + 1);
        Iterator<String> answerTexts = answers.getAnswerTexts();
        int i = 0;
        while (answerTexts.hasNext())
                qp.setAnswer(i++, answers.getAnswer(answerTexts.next()));
        ETDefaultAnswer da = new ETDefaultAnswer();
        da.setNumberOfFeedbacks(defaultFeedbacks.size());
        int j = 0;
        for (ETTemporizedTutorialElement defaultFeedback : defaultFeedbacks) {
                da.setFeedback(j++, defaultFeedback);
        }
        qp.setAnswer(i, da);
        questionElement.setNextElement(qp);
}


public void processSpeech(Object speechID,
                ETTemporizedTutorialElement speechElement,
                ETTutorialElement nextElement) {
        speechElement.setNextElement(nextElement);
        speechElements.put(speechID, speechElement);
}

public List<ETTemporizedTutorialElement> newDefaultFeedbacks() {
        return new ArrayList<ETTemporizedTutorialElement>();
}

public List<ETTemporizedTutorialElement> addDefaultFeedback(
                ETTemporizedTutorialElement defaultFeedback,
                List<ETTemporizedTutorialElement> defaultFeedbacks) {
        defaultFeedbacks.add(0, defaultFeedback);
        return defaultFeedbacks;
}

public ETTemporizedTutorialElement getSpeech(Object speechID) {
        return speechElements.get(speechID);
}

public ETText newText(String content, Color bgColor, Color fgColor,
                int fontStyle, double scaleFactor) {
        ETText eTText = new ETText();
        eTText.setText(content);
        eTText.setFontScaleFactor(scaleFactor);
        eTText.setBackgroundColor(bgColor);
        eTText.setForegroundColor(fgColor);
        eTText.setFontStyle(fontStyle);
        eTText.setTutorial(tutorial);
        return eTText;
}

public ETImage newImage(String path) {
        ETImage eTImage = new ETImage();
        eTImage.setLocation(path);
        eTImage.setTutorial(tutorial);
        return eTImage;
}

public AnswersInfo newAnswers() {
        return new AnswersInfo();
```

```
}

private class FeedbackInfo {
	private String answer;
	private int count;
	private ETTemporizedTutorialElement nextSpeech;

	public FeedbackInfo(String answer, int count,
			ETTemporizedTutorialElement nextSpeech) {
		this.answer = answer;
		this.count = count;
		this.nextSpeech = nextSpeech;
	}

	public String getAnswer() {
		return answer;
	}

	public int getCount() {
		return count;
	}

	public ETTemporizedTutorialElement getNextSpeech() {
		return nextSpeech;
	}

}

private class AnswersInfo {
	private Map<String, AnswerInfo> answers;

	public AnswersInfo() {
		answers = new TreeMap<String, AnswerInfo>();
	}

	public int getNumber() {
		return answers.keySet().size();
	}

	public Iterator<String> getAnswerTexts() {
		return answers.keySet().iterator();
	}

	public ETAnswer getAnswer(String text) {
		AnswerInfo ainfo = answers.get(text);
		ETAnswer a = new ETAnswer();
		List<ETTemporizedTutorialElement> feedbacks =
			 ainfo.getFeedbacks();
		a.setNumberOfFeedbacks(feedbacks.size());
		for (int i = 0; i < feedbacks.size(); i++)
			a.setFeedback(i, feedbacks.get(i));
		a.setTutorial(tutorial);
		return a;
	}

	public boolean contains(String answer) {
		return answers.containsKey(answer);
	}

	public void addFeedback(FeedbackInfo feedback) {
		AnswerInfo ainfo = answers.get(feedback.getAnswer());
		for (int i = answers.size(); i <= feedback.getCount(); i++)
			ainfo.setFeedback(i, feedback.getNextSpeech());
		ainfo.setFeedback(feedback.getCount(), feedback.getNextSpeech());
	}

	public void createAnswer(FeedbackInfo feedback) {
		answers.put(feedback.getAnswer(), new AnswerInfo());
		addFeedback(feedback);
```

```
		}
	}

	private class AnswerInfo {
		private List<ETTemporizedTutorialElement> feedbacks;

		public AnswerInfo() {
			feedbacks = new ArrayList<ETTemporizedTutorialElement>();
		}

		public List<ETTemporizedTutorialElement> getFeedbacks() {
			return feedbacks;
		}

		public void setFeedback(int i, ETTemporizedTutorialElement e) {
			feedbacks.set(i, e);
		}
	}

}
```