UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Ingeniería del Software e Inteligencia Artificial

UN MARCO PARA LA DEFINICIÓN Y TRANSFORMACIÓN DE MODELOS EN LOS SISTEMAS MULTIAGENTES.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Iván García-Magariño García

Bajo la dirección de los doctores

Jorge J. Gómez Sanz
Rubén Fuentes Fernández

Madrid, 2010

- **ISBN: 978-84-693-3481-2**

Un marco para la definición y transformación de modelos en los sistemas multi-agentes

Tesis doctoral

Presentada por
Iván García-Magariño García
para optar al grado de Doctor en Informática

Dirigida por los Doctores:
Jorge J. Gómez Sanz
Rubén Fuentes Fernández

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid
Madrid, Julio de 2009

El hombre entusiasmado salió al bosque a talar. En un sólo día cortó dieciocho árboles. [...]

A la mañana siguiente, se levantó antes que nadie y se fue al bosque. A pesar de todo su empeño, no consiguió cortar más de quince árboles.

- Debo estar cansado- pensó. Y decidió acostarse con la puesta del sol.

Al amanecer, se levantó decidido a batir su marca de dieciocho árboles. Sin embargo, ese día no llegó ni a la mitad.

Al día siguiente fueron siete, luego cinco, y le último día estuvo toda la tarde tratando de cortar el segundo árbol.

Inquieto por lo que diría el capataz, el leñador fue a contarle lo que le estaba pasando [...]

El capataz le preguntó: "¿Cuándo afilaste tu hacha por última vez?"

Déjame que te cuente, Jorge Bucay

Agradecimientos

La tesis ha sido un largo camino que, por suerte, no he hecho sólo. Por tanto, quiero agradecer la colaboración, estímulo y compañía de muchas personas. En primer lugar, estoy agradecido a mis dos directores. Especialmente estoy agradecido al juicio auto-crítico que me ha transmitido Jorge, por el que empiezo a convertirme en mi mejor crítico. A Rubén, le estoy especialmente agradecido por haber aprendido con él a presentar las ideas de manera coherente, clara, y concisa.

También quiero agradecer esta tesis a Juan Pavón, quien también me ha guiado en este trabajo, y me ha buscado los medios económicos que han sido indispensables para su desarrollo. También hay que decir que este trabajo se ha desarrollado durante la beca y contrato de *Formación al Personal Investigador* (FPI) asociada al proyecto *Métodos y herramientas para modelado de sistemas multi-agente* con referencia TIN2005-08501-C03-01, financiada por el Ministerio de Ciencia e Innovación.

Quiero además recordar a todos los colaboradores con los que he trabajado en diferentes artículos. Con ellos, he compartido el trabajo duro, el estrés de los *deadlines*, la alegría de las aceptaciones, y el saber encajar los rechazos y aprender de ellos. Quiero mencionar especialmente a: Alma y Juan Carlos de la Universidad de Vigo; Sylvain de Toulouse; Massimo y Valeria de Palermo; y Jose Ramón, Celia, Ghislain y José Luis de mi departamento.

Quiero mencionar especialmente a mis compañeros de despacho, con los cuales he pasado largas jornadas laborales, desayunos, comidas, cafés por la tarde, y que me han transmitido su ilusión y apoyo. Entre ellos, menciono a Javi Arroyo, Guille, Marco, Raquel, Pablo Moreno, Miki, Virginia, Antonio, Federico, Juan Antonio, Pancho y Samer. También quiero mencionar a todas las demás personas del Departamento de Ingeniería del Software e Inteligencia Artificial de la Facultad de Informática en la Universidad Complutense de Madrid.

Esta tesis no hubiera sido posible sin mi familia. Mis padres, Adelaida y Eduardo me han dado su amor incondicional, apoyo y consejo para las decisiones importantes.

También quiero agradecer esta tesis a María por darme su cariño y apoyo, que han sido de gran ayuda para hacer el camino con ilusión y alegría. También estoy agradecido a su familia por haberme acogido muchos fines de semana.

Además, quiero mencionar a mis amigos, con los cuales he realizado diferentes actividades para descansar la mente, tales como voleibol, escalada y cenas. Su apoyo fue decisivo especialmente al principio de la tesis cuando carecía de financiación. Entre ellos, quiero mencionar especialmente a Dani, Gon, Alex, Javier Días, Pablo, Carlos, Crisna, Ali y Marta.

Finalmente, quiero agradecer esta tesis a Dios que me ha acompañado siempre, y a todos los medios que me han acercado a Él, especialmente a mi *Grupo Pequeño* de fe y a la *Comunidad de Grupos Católicos Loyola*.

Resumen

El Desarrollo del Software Dirigido por Modelos (DSDM) es un paradigma de desarrollo en el que los modelos son el producto principal, y a partir de ellos se generan los sistemas de forma automática, total o parcialmente. Del tratamiento de los modelos, surge la necesidad de definirlos y transformarlos, que se aborda respectivamente con la definición de metamodelos y transformaciones.

Uno de los principales beneficiarios del DSDM es la Ingeniería del Software Orientada a Agentes (ISOA). En ésta, se construyen Sistemas Multi-agente (SMAs), que son sistemas distribuidos compuestos por agentes autónomos que interactúan dando lugar a comportamientos complejos. Si bien algunas características del DSDM se han incorporado plenamente en la ISOA como prácticas habituales, todavía dos factores dificultan su implantación completa. En primer lugar, la definición de metamodelos depende de la experiencia del diseñador y no existen guías que faciliten esta labor. Por otro lado, las herramientas y lenguajes existentes no permiten definir transformaciones de modelos con un esfuerzo razonable. Por ello, esta tesis propone una guía para definir metamodelos y un procedimiento para generar transformaciones.

La guía incluye un armazón para estructurar los metamodelos, recomendaciones para las decisiones principales, y una secuencia de pasos para definir los metamodelos. El armazón se estructura en tres capas que contienen respectivamente la información del lenguaje de modelado, los aspectos de presentación de los modelos y la información específica de las herramientas. En la capa del lenguaje de modelado, se proponen ciertas representaciones de los elementos y se dan las pautas para asociar cada elemento con la representación más apropiada. Para esto se debe elegir entre una representación heterogénea, que minimiza el número de elementos de meta-modelado necesarios para representar los elementos del modelo, o una representación homogénea, con más elementos de meta-modelado pero más fácil de procesar automáticamente. También se debe elegir entre una representación redundante o no redundante de las referencias entre los elementos, dependiendo del nivel de navegabilidad que se desee, ya que dichas referencias son dirigidas. En los aspectos de presentación, se propone usar vistas que hagan referencia a diccionarios globales, facilitando el procesamiento de los modelos y evitando inconsistencias. En la tercera capa, se considera la información específica de las herramientas, que no se había considerado en aproximaciones anteriores. Como marco de experimentación, con esta guía se ha definido el metamodelo de la herramienta *INGENIAS Development Kit* (IDK) con el lenguaje ECore, permitiendo incorporar las facilidades tecnológicas de la comunidad Eclipse en dicha herramienta. Además, se ha definido un metamodelo para la generación de un editor de procesos de ISOA, basado en el *Software Process Engineering Metamodel* (SPEM) del *Object Management Group* (OMG).

En esta tesis, se considera la aproximación conocida como Generación de Transformaciones Basadas en Ejemplo (GTBE), donde se generan transformaciones a partir de parejas de modelos prototipo origen y destino. De esta forma, se evita que el diseñador tenga que conocer los lenguajes de transformación y tratar con detalles de bajo nivel de la especificación de los modelos, tales como las primitivas de meta-modelado involucradas en cada elemento del modelo. Las transformaciones generadas transforman los modelos que encajan en los modelos prototipo origen en los nuevos modelos que encajen en los modelos prototipo destino. Es habitual que exista un mecanismo de asociación de atributos para referenciar los modelos prototipo origen desde los modelos prototipo destino, indicando así la forma de transferir la información en las transformaciones generadas.

En esta línea, esta tesis presenta un nuevo algoritmo y herramienta para la GTBE, que mejora a los trabajos existentes en varios aspectos. En primer lugar, se permite realizar el mecanismo de asociación de atributos desde varios elementos de cada modelo prototipo origen, mientras que las aproximaciones existentes sólo permiten esta asociación desde un elemento de cada modelo prototipo. Este avance permite combinar información de los atributos de diferentes elementos. Además, el algoritmo permite trabajar con grafos no conexos en los modelos prototipo origen y destino, característica no presente en los trabajos anteriores. Como experimentación, se han generado transformaciones con utilidad práctica en ciertos desarrollos de SMAs. En los ejemplos propuestos, se observa como ciertas transformaciones no podrían haber sido generadas por otras herramientas de GTBE.

Abstract

The Model Driven Development (MDD) is a paradigm of development in which models are the main product and systems are totally or partially generated from these models. When processing models, it is necessary to define and transform them, which is accomplished with the definition of metamodels and transformations respectively.

One of the fields that is most benefitted from MDD is the Agent Oriented Software Engineering (AOSE). AOSE is concerned with the development of Multi-agent Systems (MASs), which are distributed systems made of autonomous agents that interact with each other producing emergent behaviors. Although some aspects of MDD are considered in AOSE, there are still two facts that prevent AOSE from fully incorporating MDD. First, the definition of metamodels depends on the expertise of designers, and there are not guidelines that facilitate this task. Second, existent tools and languages do not allow practitioners to define model transformations with a moderate effort. Thus, this thesis proposes a guideline for defining metamodels and a mechanism for generating transformations.

The guideline includes a framework for structuring metamodels, recommendations for taking the main decisions, and a sequence of activities for defining metamodels. The framework is structured in three layers that respectively contain the information of a modeling language, the aspects for presenting its models, and the tool-specific information. In the modeling language layer, several representations of elements are proposed, and the guideline indicates the manner in which each element can be associated with the most appropriate representations. In this guideline, practitioners decide between a heterogenous representation, which minimizes the number of meta-modeling elements required to represent the modeling elements, and a homogenous representation, which uses more meta-modeling elements but is easier to be automatically processed. The guideline also includes the choice between a redundant and a non-redundant representation, regarding the degree of navigability that is desired. It is worth noticing that references between elements are directed, so the effective processing of certain operations over relationships requires redundant inverse references. Regarding the aspects for presenting models, the guideline proposes to use views that make references to global dictionaries, facilitating the processing of models and avoiding inconsistencies. The third layer considers the tool-specific information, which is not considered in the existent approaches. For experimentation, the metamodel of the INGENIAS Development Kit (IDK) has been defined following the guideline using the ECore language, allowing the incorporation of the technological facilities of the Eclipse community in the IDK. Furthermore, a metamodel is defined for the generation of an editor of processes in AOSE,

based on the Software Process Engineering Metamodel (SPEM) of the Object Management Group (OMG).

The second issue considered in this thesis is the difficulties for a cost-effective definition of model transformations. For this problem, this research considers the approach known as Model Transformation By-Example (MTBE), which generates transformations from pairs of source and target model prototypes, avoiding that designers have to learn transformation languages and deal with low-level details of the metamodels of the involved modeling languages. The generated transformations transform models that match source prototype models into the new models that fit the target prototype models. These approaches usually include a mechanism for the mapping of attributes that refer to source prototype models from elements in the target prototype models, indicating the way of transferring information in the generated transformations.

In this line of research, this thesis presents a new algorithm and tool for MTBE, which overcome some limitations of the existent approaches. First, the presented approach provides a mechanism for mapping attributes from several elements of each source prototype model, while existent approaches only provide this mapping from one element of each source prototype model. This improvement allows one to combine information from the attributes of different elements. Moreover, the presented algorithm can process non-connected graphs in the source and target model prototypes, which is not possible in other approaches. For experimentation, the tool has generated transformations with practical utility in certain MAS developments. In the proposed examples, one can observe that certain transformations cannot be generated by other MTBE tools.

Índice

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción

Este capítulo describe brevemente el contexto de esta tesis, y establece sus objetivos. El capítulo finaliza describiendo la estructura y el contenido de este documento.

1.1 Contexto

En el punto de intersección de los sistemas computacionales distribuidos y la inteligencia artificial, se encuentran los *Sistemas Multi-Agente* (SMAs) (Ferber, 1999). En un SMA, se define el comportamiento individual y colectivo de ciertas entidades autónomas, llamadas agentes. Un sistema compuesto por estos agentes presenta algunos comportamientos emergentes y otros preestablecidos, que son consecuencia directa de los comportamientos individuales de los agentes. En su ideal, los SMAs son tolerantes a fallos dado que cualquier agente, al ser autónomo, busca cómo remediarlos. Además, la tecnología de SMAs sirve de envoltorio ingenieril para las técnicas básicas de inteligencia artificial, que deben ser incluidas en el comportamiento autónomo de los agentes.

En sus orígenes, los SMAs sólo estaban al alcance de un grupo pequeño de especialistas. Con el tiempo, han ido surgiendo armazones, metodologías y herramientas de desarrollo. Los armazones proporcionan un código de programación que es reusable en una variedad de SMAs, y comúnmente están compuestos de una plataforma y un código extensible para la definición de nuevos agentes. Estos armazones permiten crear un SMA y programar el comportamiento individual de los agentes autónomos con un esfuerzo reducido. Algunos ejemplos de plataformas son *JADE* (Bellifemine et al., 2001b) y *Repast* (Collier, 1999). Las metodologías existentes de SMAs proporcionan, en mayor o menor medida, pautas para el desarrollo de SMAs de forma ingenieril. Algunas de estas metodologías son: Gaia (Wooldridge et al., 2000), Prometheus (Padgham y Winikoff, 2002), INGENIAS (Pavón y Gómez-Sanz, 2003), ADELFE (Bernon et al., 2003), PASSI (Chella et al., 2006) y Tropos (Bresciani et al., 2004). También se han creado herramientas de desarrollo que dan soporte tecnológico para el seguimiento de algunas de las metodologías mencionadas. Este es el caso de las herramientas *INGENIAS Development Kit* (IDK) (Gómez-Sanz et al., 2008) y *Prometheus Design Tool* (PDT) (Padgham et al., 2008). Gracias a dichos armazones, metodologías y herramientas, el término de *Ingeniería del Software*

Orientada a Agentes (ISOA) cobra sentido como una rama de la ingeniería del software en la que la unidad básica es el agente.

Los grupos de investigación de algunas metodologías de ISOA, tales como INGENIAS (Pavón et al., 2006), Tropos (Amor et al., 2005), ADELFE (Rougemaille et al., 2009) y Prometheus (Padgham y Winikoff, 2002), apuestan por el *Desarrollo de Software Dirigido por Modelos* (DSDM) (Selic, 2003) para la creación de SMAs. El DSDM es una rama de la ingeniería del software que considera los modelos como ciudadanos de primera clase, y las *Transformaciones de Modelo* (TMs) (Sendall y Kozaczynski, 2003) como forma de evolución de los modelos hacia el sistema final. Los *metamodelos* definen los *Lenguajes de Modelado* (LMs). Las TMs se usan tanto para transformaciones modelo-a-modelo como para transformaciones modelo-a-texto; y están compuestas generalmente de reglas. Estas reglas pueden ser uno-a-uno, uno-a-muchos y muchos-a-muchos, dependiendo si se tienen uno o varios elementos en las entradas y salidas de las reglas.

La mayor parte de la experimentación de esta tesis se sustenta sobre la metodología de INGENIAS (Pavón y Gómez-Sanz, 2003) y su herramienta de soporte, el IDK (Gómez-Sanz et al., 2008), las cuales están fundamentadas en los principios del DSDM en la ISOA. Como se discute en el apartado 1.4, INGENIAS y el IDK presentan características que los hacen relevantes para experimentar las propuestas de esta tesis.

1.2 Motivación

La motivación de esta tesis es mejorar el DSDM, para lo cual se centra en el área de los SMAs como dominio de aplicación. Para la ISOA, el DSDM tiene las siguientes carencias:

- *Faltan guías de definición de metamodelos que se ajusten a las necesidades del problema.* No existen procesos formales que indiquen como tomar decisiones en la definición de metamodelos a partir de ciertos requisitos, tales como: relaciones n-arias, atributos en los cuerpos de las relaciones, atributos en los extremos de las relaciones y jerarquías para los elementos. Por otra parte, los metamodelos actuales no consideran la información específica de las herramientas de *Computer-aided Software Engineering* (CASE). Esto dificulta, por ejemplo, el intercambio de información entre herramientas que, empleando los mismos LMs, organizan sus espacios de trabajo de diferentes formas.

- *Faltan técnicas y herramientas que faciliten la definición de TMs a los expertos de SMAs.* Las TMs son uno de los pilares del DSDM; sin embargo, actualmente la creación de TMs presenta ciertas dificultades. Las TMs se definen habitualmente con texto, i.e. código. Por ejemplo, los diseñadores de SMAs suelen encontrar difícil la definición de TMs, dado que la mayoría no son especialistas en los LTMs existentes. Además no existen herramientas de desarrollo para las TMs con una funcionalidad similar a las de otros paradigmas de desarrollo convencionales. Por ejemplo, no hay herramientas de depuración para TMs. La *Generación de Transformaciones Basadas en Ejemplos* (GTBE) (Varro, 2006) se basa en la generación de TMs a partir de parejas de modelos fuente y destino. GTBE permite

a los diseñadores crear TMs sin conocer ningún LTM, al mismo tiempo que reduce los potenciales errores de diseño y codificación. Sin embargo, las herramientas actualmente existentes de GTBE (Varró y Balogh, 2007b; Wimmer et al., 2007) no satisfacen varias necesidades identificadas en los desarrollos de SMAs: no permiten reglas de transformación muchos-a-muchos con transferencia de información desde varios elementos de la entrada, ni reglas de transformación para grafos no conexos.

1.3 Objetivos

El objetivo general de esta tesis es mejorar las técnicas de DSDM en los desarrollos de SMAs. Este objetivo general se descompone en los siguientes:

- *Facilitar la definición de metamodelos a los expertos en SMAs.* Para ello, se debe proporcionar una guía para definir metamodelos. Por un lado, la guía debe considerar aspectos de las herramientas CASE para adaptar las mismas a las tecnologías existentes de la comunidad de DSDM. Buena parte de las herramientas existentes no están orientadas a estas tecnologías lo que dificulta su uso no sólo en contextos de DSDM, sino también en entornos heterogéneos donde se hace necesario el intercambio de información entre diferentes lenguajes y herramientas. Por otro lado, esta guía debe considerar los requisitos de la sintaxis abstracta de los LMs de los SMAs: relaciones n-arias, atributos en los cuerpos de las relaciones, atributos en los extremos de las relaciones y jerarquías para los elementos. La guía también debe considerar los requisitos de la sintaxis concreta de los LMs, como sus notaciones gráficas y la necesidad de trabajar con diferentes aspectos de los modelos en los diagramas. De esta manera, los ingenieros del área de SMAs pueden uniformizar los metamodelos de los LMs de los SMAs, o sencillamente ser conscientes de los requisitos de sus metamodelos, las alternativas que tienen, y las características de éstas.

- *Facilitar la creación de TMs a los expertos de SMAs.* Para ello, se ha desarrollado un algoritmo para la GTBE. El objetivo es que el algoritmo se ajuste a las necesidades identificadas en el desarrollo de SMAs: reglas de transformación muchos-a-muchos con un mecanismo de asociación de atributos entre grupos de elementos, y reglas que trabajen con grafos no conexos. El algoritmo se debe implementar para LTMs ampliamente extendidos, en particular para el *Atlas Transformation Language* (ATL) (Jouault y Kurtev, 2006b), que es el que cuenta actualmente con mayor soporte. Además, se espera que se proporcione una herramienta con una interfaz de usuario que facilite el uso a los expertos en SMAs.

1.4 Método de trabajo

Para realizar los objetivos mencionados anteriormente, se ha seguido el siguiente plan de trabajo:

- *Estudio del estado del arte.* Se han analizado los metamodelos existentes, especialmente los relacionados con SMAs. También se han revisado los LTMs existentes, así como las distintas aproximaciones de GTBE.

- *Definición de una guía de metamodelos para herramientas CASE.* Esta guía permite definir metamodelos para herramientas CASE organizando la siguiente información en capas diferentes: el LM, los aspectos de presentación, y la información específica de las herramientas. Esta estructura surge como una posible solución de las limitaciones detectadas en el anterior paso. Además, la guía contiene pautas detalladas para definir la sintaxis abstracta de los LMs teniendo en cuenta sus características estructurales.

- *Validación de la guía de metamodelos con la herramienta IDK y un editor de procesos orientado a agentes.* INGENIAS y el IDK presentan características relevantes para la validación de la guía: primero, el metamodelo de INGENIAS tiene una gran riqueza de elementos (260 elementos aproximadamente) y es heterogéneo, ofreciendo ejemplos de gran parte de los requisitos y problemas identificados en la definición de LMs; segundo, el IDK genera un esqueleto ejecutable completo a partir de una especificación de un SMA; finalmente, el IDK almacena información específica de la herramienta. Originalmente el metamodelo de INGENIAS fue definido con el lenguaje GOPRR (Kelly, 1997). Sin embargo, la mayoría de las herramientas disponibles en el DSDM trabajan bajo Eclipse y requieren que el metamodelo se describa con ECore, el lenguaje del *Eclipse Modeling Framework* (EMF) (Budinsky, 2003). La modernización del IDK para adoptar las nuevas tecnologías del DSDM ha seguido las pautas del *Architecture Driven Modernization* (ADM) (Newcomb, 2005). Por ello ha comenzado con la definición del metamodelo de INGENIAS en ECore, que ha servido como base de experimentación de la guía. Finalmente, la validación de la guía ha quedado completada con la definición de un metamodelo para generar un editor de procesos orientado a agentes, basado en el *Software Process Engineering Metamodel* (SPEM) (OMG, 2005b).

- *Definición de un algoritmo para la GTBE.* Este algoritmo tiene características que otras aproximaciones de GTBE no tienen, tales como la asociación de atributos entre grupos de elementos, y el tratamiento de grafos no-conexos.

- *Validación de la aplicación de la GTBE en un desarrollo de un SMA.* En concreto, esta validación ha perseguido proporcionar ejemplos de que las TMs producidas por el algoritmo de GTBE son suficientemente generales para aplicarse en diferentes desarrollos. Para ello se ha tomado como punto de partida el desarrollo en INGENIAS de un SMA para la evaluación de documentos con el proceso Delphi. En el proceso Delphi, un moderador guía a un grupo de expertos mediante cuestionarios para llegar a un acuerdo. A partir de él se han generado TMs que se han probado con el mismo proyecto y otros disponibles con INGENIAS. Estas TMs son ejemplos que pueden ayudar a otros ingenieros de SMAs, bien usándolas directamente en sus desarrollo o inspirándoles para que definan TMs con el algoritmo y/o herramienta de GTBE propuestos. De este modo se incide en una de las ventajas clave del DSDM, la reutilización de la experiencia y conocimiento de desarrollos previos bajo la forma de modelos y TMs.

1.5 Estructura en capítulos

Los detalles de esta investigación, así como los resultados obtenidos y la experimentación que conforman esta tesis se recogen en los siguientes capítulos, cuyo contenido se resume a continuación:

- **Capítulo 2: Estado del arte: definición y transformación de modelos.** Este capítulo se dedica a describir el contexto del campo de aplicación de este trabajo de investigación, esto es, el DSDM en la ISOA. Para ello, se centra principalmente en los metamodelos y las TMs. Además, describe las herramientas relacionadas.

- **Capítulo 3: Guía para la definición de metamodelos**. Este capítulo presenta una guía para la definición de metamodelos para las herramientas CASE de SMAs. La guía incluye la posibilidad de definir conjuntos de metamodelos para almacenar toda la información necesaria para las herramientas CASE. Los metamodelos de este conjunto contienen información acerca de: la sintaxis abstracta, la sintaxis concreta, las vistas o diagramas en forma de grafos, la posición espacial de los elementos en los ejes cartesianos, y los proyectos de desarrollo. Para la sintaxis abstracta de los LMs, la guía permite definir los metamodelos de una manera sistematizada teniendo en cuenta las características estructurales de los LMs de SMAs, que pueden ser: relaciones n-arias, relaciones con atributos, y relaciones con atributos en sus extremos. Además, existen diferentes alternativas de representación que permiten alcanzar el compromiso buscado entre el tamaño de los metamodelos resultantes (y por tanto de los modelos que definen), la facilidad de procesamiento y las extensiones futuras de los metamodelos. Atendiendo a los requisitos de su problema, los diseñadores deberán tomar las correspondientes decisiones en el diseño del metamodelo. En primer lugar, elegirán entre una representación homogénea o heterogénea de los elementos y, en segundo lugar, escogerán una representación redundante o no redundante de las referencias. A partir de estas decisiones, la guía proporciona reglas que asignan una representación a cada elemento.

- **Capítulo 4: Validación de la guía de definición de metamodelos en los sistemas multi-agente**. Como experimentación del capítulo anterior, se muestra el metamodelo necesario para guardar toda la información del la herramienta IDK, incluidos los aspectos esenciales del metamodelo de la sintaxis abstracta de INGENIAS. Además, se describe el metamodelo que se creó para generar automáticamente un editor de procesos orientados a agentes basados en SPEM.

- **Capítulo 5: Algoritmo para la generación de transformaciones**. Describe un algoritmo para la GTBE que cubre algunas necesidades del DSDM en la ISOA. Las innovaciones de este algoritmo frente a los existentes de GTBE son la generación de reglas muchos-a-muchos que permiten transferir los valores de los atributos desde varios elementos de la entrada, y el tratamiento de grafos no conexos. La generación de las reglas muchos-a-muchos se ve dificultada porque los LTMs más extendidos (como el lenguaje ATL (Jouault y Kurtev, 2006b)) no soportan reglas con más

de un elemento de entrada. Esta dificultad es salvada con las siguientes medidas en el algoritmo presentado: la activación de las reglas se condiciona con las restricciones apropiadas para las reglas con varios elementos de entrada; la asociación de atributos se hace por medio de expresiones que permiten acceder a varios elementos de la entrada de la regla. Además, en este capítulo se presenta una herramienta que implementa el algoritmo de GTBE para el lenguaje ATL, incluyendo una interfaz de usuario para su uso.

- **Capítulo 6: Experimentación con la generación de transformaciones en el desarrollo de sistemas multi-agentes**. Se presentan ejemplos de TMs que se hubieran podido generar para trabajar con especificaciones de SMAs existentes. De esta forma, se verifica si el algoritmo de GTBE es aplicable a las características que surgen en un desarrollo real. Las TMs presentadas se centran en un desarrollo con la metodología INGENIAS (Pavón y Gómez-Sanz, 2003). Estas TMs generan roles a partir de casos de uso, tareas a partir de un flujo de trabajo, unidades de interacción a partir del flujo de trabajo, y definiciones de despliegues y pruebas a partir de agentes. Estas TMs son aplicadas al desarrollo de un SMA basado en el proceso Delphi, en el cual un grupo de agentes, que representan expertos en evaluación de documentos alcanzan un acuerdo por medio de rondas de cuestionarios.

- **Capítulo 7: Conclusiones**. Presenta las conclusiones obtenidas en este trabajo y las líneas futuras de investigación a las que da lugar.

- **Capítulo 8: Conclusions (in English)**. De acuerdo con la normativa de *Doctorado Europeo*, se incluyen las conclusiones de esta tesis en inglés. Este capítulo es una traducción del capítulo anterior.

Capítulo 2

Estado del arte: definición y transformación de modelos

Este capítulo introduce la definición de metamodelos y las transformaciones de modelos, revisando la literatura existente e indicando los aspectos que necesitan ser mejorados para su aplicación en la ISOA. Para su completitud, el estudio también explora el alcance del DSDM más allá del dominio de los SMAs. Para ilustrar el resto de las discusiones en esta tesis con ejemplos concretos se ha seleccionado la metodología INGENIAS con su herramienta de soporte y desarrollos con ambas. Este capítulo también introduce brevemente tanto la metodología como la herramienta.

2.1 Introducción

Los agentes nacieron de la investigación en inteligencia artificial y más concretamente de la inteligencia artificial distribuida. Al principio, la inteligencia artificial se refería a los agentes como programas especiales cuya naturaleza y construcción no se llegaba a detallar. Se ha escrito mucho acerca de los agentes (Franklin y Graesser, 1997; Wooldridge y Jennings, 1998) y, aunque no hay un consenso definitivo, se suele hablar de características deseables: autonomía, proactividad, adaptabilidad y habilidad social.

La construcción de este tipo de software de forma disciplinada es responsabilidad de la ISOA, que evoluciona con la experiencia en desarrollos de SMAs y con la incorporación de resultados de la investigación en ingeniería del software. Una incorporación que ha tenido gran aceptación es el concepto de metamodelo. Así, es posible encontrar metodologías como Tropos (Bresciani et al., 2004), INGENIAS (Pavón y Gómez-Sanz, 2003), ADELFE (Bernon et al., 2003) y PASSI (Chella et al., 2006) que definen LMs con metamodelos para la especificación de SMAs. Esto conecta directamente con las líneas de investigación en el DSDM.

El DSDM se centra en la creación de modelos, que son abstracciones que describen elementos del sistema. El DSDM planea la generación automática de sistemas a partir de modelos por medio de TMs, que transforman los modelos hasta obtener el código de los sistemas. También simplifica el proceso de diseño, dado que partes del mismo se pueden obtener como fruto de aplicaciones de TMs. Además, la integración de sistemas diferentes se facilita al trabajar prin-

cipalmente con modelos en lugar de código de programación. Por todos estos motivos, se considera que el DSDM incrementa la productividad frente a otras aproximaciones anteriores de ingeniería del software. La especificación llamada *Model Driven Architecture (MDA)*(OMG, 2003b) es un ejemplo de puesta en práctica de los principios de DSDM, si bien pueden existir otros tipos de aplicaciones del DSDM.

El DSDM está fuertemente ligado con los metamodelos, como menciona Selic (2003). Los metamodelos son modelos que permiten definir LMs. En realidad, este concepto es extendido a una pirámide que consta de cuatro niveles, habitados respectivamente por: meta-metamodelos, metamodelos, modelos y datos. En esta pirámide, los modelos de cada nivel definen el lenguaje del siguiente nivel. Aunque existen definiciones de metamodelos, algunos de ellos aceptados como estándares (OMG, 2007a,b, 2008c), las guías de definición de metamodelos son apenas inexistentes.

El otro elemento esencial del DSDM son las TMs, ya que tal y como indican Sendall y Kozaczynski (2003) hacen posible que los modelos se transformen en sucesivos pasos hasta la generación del sistema informático. Sin embargo, el soporte de herramientas para definir TMs es todavía escaso.

En la definición de metamodelos y TMs en ISOA, esta tesis ha identificado dos problemas en los que se centrará la discusión:

- *Necesidad de guías precisas para la definición de metamodelos.* Existen definiciones con metamodelos de LMs de SMAs, pero no se observan patrones comunes ni se justifican las decisiones tomadas en su diseño. Esto hace de la definición de nuevos metamodelos de SMAs una tarea complicada para los diseñadores de SMAs. Estos problemas se deben, en parte, a que la comunidad de DSDM carece de este tipo de guías.

- *Necesidad de herramientas y mecanismos que permitan a los diseñadores de SMAs definir TMs de manera sencilla.* La mayoría de los LTMs existentes no satisfacen las necesidades propias de los SMAs, en particular la definición de las *reglas muchos-a-muchos*. En las TMs, las *reglas muchos-a-muchos* son aquellas reglas que tienen varios elementos en el patrón de entrada y varios elementos en el patrón de salida y pueden transferir información desde la entrada a la salida. Asimismo, las herramientas existentes para facilitar la creación de TMs, tales como las propuestas por Varro (2006), Lopes et al. (2005) y Wimmer et al. (2007), no permiten generar algunos tipos de reglas muchos-a-muchos, tales como las que transfieren atributos desde varios elementos de la entrada y las que tratan con grafos no conexos.

El resto del capítulo se estructura de la siguiente manera. El siguiente subapartado describe brevemente la metodología INGENIAS y su herramienta de soporte, el IDK, dado que muchos de los ejemplos de esta memoria están basados en dicha metodología y herramienta. El apartado 2.2 hace una descripción de las tendencias en la definición de metamodelos en la ingeniería del software. Dentro de dicho apartado, el apartado 2.2.1 define brevemente los metamodelos y la arquitectura de metadatos y el apartado 2.2.2 hace una breve descripción de los lenguajes de meta-modelado existentes. El apartado 2.2.3 hace una clasificación de los metamodelos existentes según la manera en que son definidos, con lo que se

muestra la gran variedad de alternativas existentes en la definición de metamodelos. El apartado 2.2.4 menciona las características estructurales del tipo más común de los LMs. El apartado 2.2.5 introduce las técnicas para la creación de las herramientas de modelado, cuyos principales beneficiarios son los lenguajes específicos de dominio. El apartado 2.2.6 indica algunas dificultades existentes en relación con la definición de metamodelos y el apartado 2.2.7 describe los metamodelos más relevantes en la comunidad de SMAs. El apartado 2.2.8 indica el estado del arte de la información específica de las herramientas CASE. Por otro lado, el apartado 2.3 describe las tendencias actuales en las TMs. Para ello, el apartado 2.3.1 define las TMs e indica sus aplicaciones principales; mientras que el apartado 2.3.2 clasifica las TMs según si los metamodelos de entrada y salida son el mismo y según la manera en que se definen. El apartado 2.3.3 presenta el estándar de definición de TMs más aceptado, *Query View Transformation* (QVT), y los LTMs principales. El apartado 2.3.4 presenta la GTBE, en la que se genera automáticamente TMs a partir de modelos prototipos. El apartado 2.3.5 describe ciertas problemáticas en las TMs, y el apartado 2.3.6 indica las TMs existentes en los SMAs. Finalmente, el apartado 2.4 presenta brevemente las conclusiones de este análisis sobre la definición y transformación de modelos.

2.1.1 INGENIAS y el IDK

La mayoría de los experimentos de esta tesis giran alrededor de la metodología INGENIAS (Pavón y Gómez-Sanz, 2003) y su herramienta de soporte, el *INGENIAS Development Kit* (IDK) (Pavón et al., 2006). Por ello, se ha incluido una breve descripción de ambos. INGENIAS (Pavón y Gómez-Sanz, 2003) es una metodología y un LM para SMAs que toma de punto de partida la metodología y el LM *Message* (Caire et al., 2002). La metodología Message se basa en los conceptos de agentes, organizaciones, roles, objetivos y tareas. Usa como notación una extensión del *Unified Modeling Language* (UML), adopta el *Rational Unified Process* (RUP) y define actividades para la identificación y especificación de componentes de SMAs en el análisis y parcialmente en el diseño. INGENIAS mejora Message en los siguientes aspectos:

- *Integración de las vistas de diseño del sistema.* INGENIAS conecta los conceptos de diferentes diagramas y permite referencias mutuas.
- *Integración de resultados de investigación.* Cada metamodelo ha sido elaborado de acuerdo los resultados de investigación en diferentes áreas como la coordinación, el razonamiento y la gestión del flujo de trabajo.
- *Integración del ciclo de vida del desarrollo del software.* La relación entre el RUP e INGENIAS es mayor. Además INGENIAS ha ido incluyendo otros procesos software en su evolución, tales como el proceso SCRUM (García-Magariño et al., 2009b).
- *El soporte tecnológico.* La herramienta de soporte de Message era un editor basado en la herramienta comercial MetaEdit+ (Tolvanen y Rossi, 2003), que actualmente está en desuso. El *INGENIAS Development Kit* (IDK) (Pavón et al., 2006) es una herramienta de código-abierto que permite la especificación y generación automática de código de los SMAs. Ha sido desarrollado con el propósito de constituir un framework completo de desarrollo de SMAs portable, extensible y configurable.

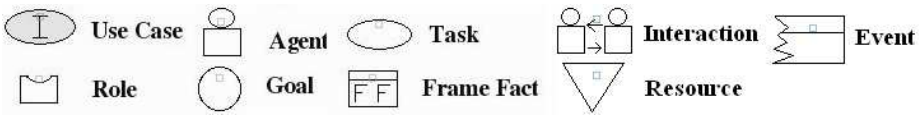


Figura 2.1: Algunos conceptos relevantes de la notación de *INGENIAS*

- *Implementación.* Message no contemplaba la implementación. INGENIAS plantea ciclos completos de DSDM hasta el sistema final. Estos ciclos completos se soportan con el IDK, el cual se describe más adelante en este apartado.

En cuanto al LM de INGENIAS, éste se define con un metamodelo, y permite especificar los SMAs bajo las siguientes vistas:

- la definición, control y gestión del estado mental de cada agente,
- las interacciones entre agentes,
- la organización de los SMAs,
- el entorno y
- las tareas y objetivos asignados a cada agente.

El DSDM en INGENIAS empleando este LM se basa en la herramienta IDK. El IDK proporciona una manera de desarrollar SMAs siguiendo los principios del DSDM (Pavón et al., 2006). El usuario define la especificación del SMA con el editor del IDK. Esta especificación representa el modelo que dirige el desarrollo de los SMAs. La Figura 2.1 muestra los elementos de la notación de *INGENIAS* necesarios para entender los ejemplos de esta tesis.

El IDK soporta transformaciones bidireccionales entre las especificaciones y el código generado a partir de ellas. El IDK usa las especificaciones para generar automáticamente código. El desarrollador puede después modificar este código y, si lo desea, usar el IDK para actualizar con él los modelos originales. Este proceso de generación y realimentación se logra mediante técnicas de DSDM basadas en plantillas. El núcleo del IDK es un editor que trabaja como anfitrión para plugins que generan documentación, verifican los modelos de especificación y generan código. El plugin central para la generación de código es el *INGENIAS Agent Framework* (IAF) que produce código para la plataforma *Java Agent DEvelopment framework* (JADE) (Bellifemine et al., 2001a). El código generado implementa agentes deliberativos y reactivos, y protocolos de comunicación entre agentes. El generador de código ha sido presentado en (Gómez-Sanz y Pavón, 2006; Gómez-Sanz et al., 2006).

La metodología INGENIAS y el IDK han sido aplicados con éxito en el desarrollo de SMAs en varios dominios tales como: la vigilancia (Pavón et al., 2007), sistemas basados en conocimiento (Soto et al., 2006), guías turísticas móviles (Pavón et al., 2004) y la simulación social (Pavón et al., 2008).

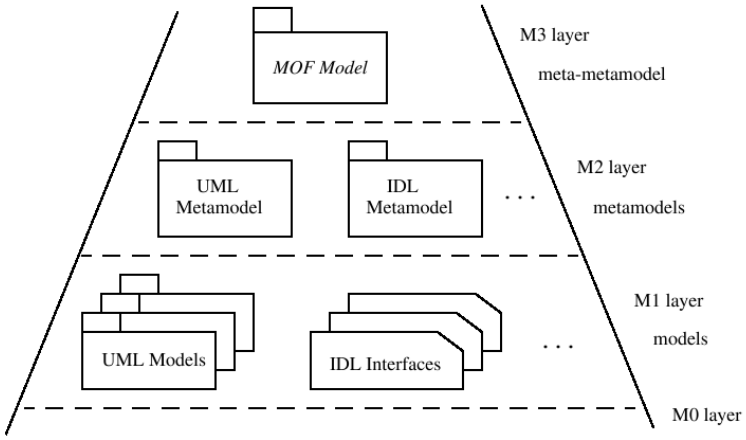


Figura 2.2: Arquitectura de Metadatos en MOF

2.2 Definición de metamodelos

Los metamodelos (Atkinson y Kuhne, 2003) son uno de los pilares del DSDM. Sin embargo, los expertos en SMAs suelen encontrar dificultades en la definición de los metamodelos por la gran variedad de representaciones posibles para los mismos y la ausencia de manuales que guíen su definición.

2.2.1 Qué son los metamodelos y la arquitectura de metadatos

Un modelo es una abstracción de la realidad, que la presenta de una manera simplificada. Un metamodelo es un modelo que describe un LM, con el que se describen otros modelos.

La noción de metamodelo se basa en la arquitectura de metadatos que se muestra en la Figura 2.2, adoptada por el consorcio *Object Management Group* (OMG) en la especificación del *Meta-Object Facility (MOF)* (OMG, 2006b). Esta figura muestra la diferencia entre información, modelos, metamodelos y meta-metamodelos. MOF nombra cada una de estas capas o niveles con los nombres M0, M1, M2 y M3 respectivamente, como se puede ver en la Figura 2.2. De ahora en adelante, se usarán los términos M0, M1, M2 y M3 para referirse a las capas de MOF. Es importante fijarse en que estos conceptos están en capas diferentes y poseen significados diferentes aunque usen la misma notación.

El nivel más alto es el nivel de los meta-metamodelos, M3. Algunos ejemplos de lenguajes en este nivel son MOF (OMG, 2006b) usado por OMG, ECore (Budinsky, 2003; Moore et al., 2004) usado por EMF, y GOPRR *(Grafo, Objeto, Propiedad, Relación, y Rol)*(Tolvanen, 2000; Kelly, 1997). En la sección 2.2.2, se habla de cada uno de ellos.

Las instancias de los lenguajes de M3 son los metamodelos, que definen LMs y corresponden al nivel M2. Un ejemplo bien conocido de metamodelo es la definición de UML (OMG, 2007a,b).

Luego están los modelos que representan, por ejemplo, diseños de aplicaciones. Éstos son instancias de los metamodelos y constituyen el nivel M1. Cual-

quier diseño concreto de una aplicación en UML podría servir como ejemplo. También cualquier modelo de un SMA en INGENIAS pertenecería a este nivel.

Por último se encuentra el nivel M0, que sería la implementación de los diseños. Por decirlo en términos de programación Orientada a Objetos, en el nivel M0 se considerarían los objetos de las clases en ejecución y en el nivel M1 estarían las definiciones de las clases de objetos.

Notación

En este apartado, se establece la notación para los elementos de meta-modelado que se usa en esta memoria de tesis. Los modelos se encuentran en la capa M1. Los elementos de los modelos pueden ser entidades, relaciones, atributos u operaciones. Sin embargo, para no confundir estos elementos del modelo con los elementos del metamodelo, se usará el siguiente convenio:

- *Capa M1.* Contiene los modelos. Se distinguen:
  - *Entidad del modelo*: entidad o entidad de M1.
  - *Relación del modelo*: relación o relación de M1.
  - *Atributo del modelo*: atributo o atributo de M1.

  Para englobar todos los elementos anteriores se usará:
  - Elemento del modelo o elemento de M1.
- *Capa M2.* Contiene los metamodelos. Se distinguen:
  - *Entidad del metamodelo*: meta-entidad o entidad de M2.
  - *Relación del metamodelo*: meta-relación o relación de M2.
  - *Atributo del metamodelo*: meta-atributo o atributo de M2.

  Para englobar todos los elementos anteriores se usará:
  - Meta-elemento o elemento de M2.

Usando la notación anteriormente descrita, se pueden definir las siguientes relaciones entre las capas M1 y M2:

- Una entidad de M1 *es* una instancia de una meta-entidad o entidad de M2.
- Una relación de M1 *es* una instancia de una meta-relación o relación de M2.
- Un atributo de M1 *es* una instancia de un meta-atributo o atributo de M2.

2.2.2 Lenguajes de meta-modelado

Los metamodelos se definen mediante los *lenguajes de meta-modelado*. Los lenguajes de meta-modelado se definen con meta-metamodelos, por lo que la especificación de MOF (OMG, 2006b) los sitúa en el nivel M3 de la arquitectura de metadatos. Cada lenguaje de meta-modelado tiene sus propias primitivas. Este apartado describe los lenguajes de meta-modelado más relevantes, que son MOF, ECore y GOPRR, y hace una breve comparativa entre los mismos.

Lenguaje de Meta Object Facility (MOF)

Como se ha señalado anteriormente, MOF es el lenguaje de meta-modelado propuesto por el OMG, y que esta organización usa en la definición de sus estándares. La especificación de MOF (OMG, 2006b) proporciona, entre otras, las siguientes primitivas:

- *Clases.* Las clases especifican tipos de objetos. Las clases tienen unos atributos y unas operaciones. Los atributos pueden ser valores simples o referencias a otros objetos de otras clases. Las operaciones contienen un comportamiento. Además, las clases tienen la propiedad de la *herencia*. Pueden heredar unas de otras, adquiriendo sus atributos y operaciones. Por último, pueden existir *Clases Abstractas*, es decir, clases que no pueden ser instanciadas con objetos y pueden tener operaciones no definidas que esperan ser implementadas por sus descendientes.

- *Atributos.* Definen las propiedades de las clases. Pueden ser de tipos enteros o cadenas de caracteres.

- *Operaciones.* Definen las funcionalidades de las clases.

- *Asociaciones/Links.* Las *Asociaciones* expresan relaciones binarias bidireccionales entre dos clases, en un principio bidireccionales. Las instancias de las asociaciones reciben el nombre de *links*. Tanto las asociaciones como los *links* no tienen atributos ni operaciones. Cada asociación tiene dos extremos, en los que se puede definir una multiplicidad.

- *Tipos de Datos.* Son los tipos de datos primitivos (booleanos, enteros, etc.) y algunos más complejos (colecciones, listas, etc.)

- *Paquetes.* Los paquetes agrupan los elementos anteriormente descritos. En los paquetes se permiten la generalización de paquetes, el anidamiento de paquetes (unos contienen a otros) y la importación de paquetes. En la importación, un paquete hace visible sus elementos para el paquete que lo importa. Por último, dos paquetes pueden formar un *cluster*, en el que ambos paquetes tienen todos sus elementos visibles entre sí.

Los nombres de las primitivas de MOF y UML son similares. Sin embargo, es importante señalar que son lenguajes de diferente nivel. MOF pertenece al nivel M3, mientras que UML pertenece al nivel M2 según la notación descrita en el capítulo 2.2.1.

La especificación de MOF también menciona explícitamente los mecanismos de reflexión, con los cuales, por ejemplo, MOF se puede definir a sí mismo. Para la reflexión, se usan los siguientes elementos:

- *Elemento.* Es una instancia de una Clase, que define sus atributos y operaciones.

- *Factoría.* Una Factoría crea Elementos. En concreto está asociada con un Paquete y puede crear cualquier Elemento que sea instancia de las Clases de dicho Paquete.

- *Objetos.* Los *Objetos* engloban tanto a los Elementos como a los valores de los Atributos. Los Objetos pueden representar un valor de cualquier tipo.

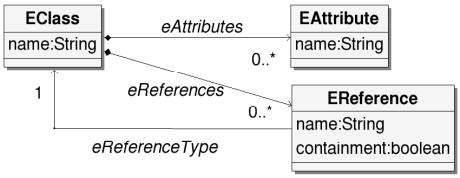


Figura 2.3: ECore, una pequeña parte del meta-metamodelo

Algunas herramientas que dan soporte tecnológico a MOF son: *NetBeans MDR* (Matula, 2003), *Enterprise Architecture* (Schekkerman, 2008) y *Rational Software Architect* (RSA) (Leroux et al., 2006). *NetBeans MDR* usa el formato de intercambio de datos establecido para MOF, y su última versión fue liberada en el 2003. Las otras dos herramientas son más actuales y usan otros formatos distintos.

EMF y su lenguaje ECore

El *Eclipse Modeling Framework* (EMF) (Budinsky, 2003) es un plugin de Eclipse que trabaja con metamodelos del lenguaje de meta-modelado *ECore*. Una versión del modelo *ECore* simplificada se puede ver en la Figura 2.3. El lenguaje ECore tiene los siguientes elementos:

- *EClass*. Son unidades con EAttributes, EReferences y EOperation. Las EClasses tienen un mecanismo de herencia con la propiedad *ESuperTypes*, que permite declarar qué clase se especializa.
- *EAttribute*. Contienen valores de unos tipos primitivos, tales como enteros o cadenas de caracteres.
- *EReference*. Son referencias que tienen o apuntan a objetos de una EClass. Cada EReference conecta dos EClasses tanto en M2 como en M1. Estas referencias tienen una multiplicidad definida entre unos límites. Dependiendo de estos límites, se puede tener una instancia de EReference *múltiple* en M2. Esta instancia de EReference múltiple se puede instanciar varias veces en M1. Además, las EReferences pueden ser de dos tipos:
  - *containment*. Son aquellas que crean un elemento al editar los modelos de M1.
  - *non-containment*. Son aquellas que referencian a un elemento de M1 creado previamente.
- *EOperation*. Definen un comportamiento.
- *EPackage*. Agrupan al resto de elementos. Cada paquete puede contener a otros paquetes. Además cada paquete tiene una *Uniform Resource Identifier* (URI), que le identifica en el espacio de nombres. Un paquete puede hacer referencia a elementos de otro paquete gracias a la URI. Se puede entender que este mecanismo es un mecanismo de importación.
- *EEnum*. Definen un tipo enumerado con una lista de EEnumLiteral.

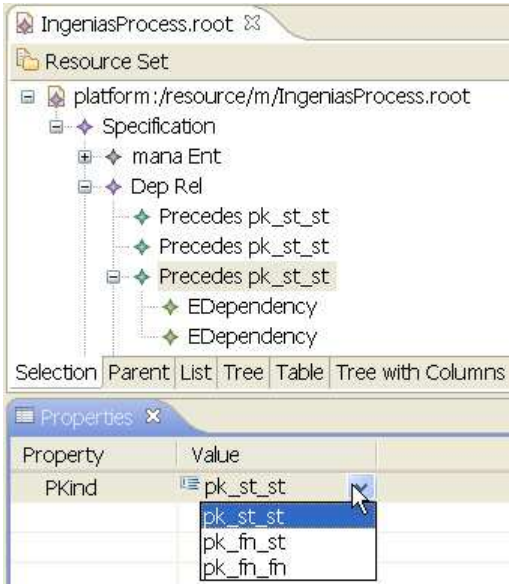


Figura 2.4: Ejemplo de editor generado automáticamente con EMF.

- *EEnumLiteral*. Definen valores de un tipo enumerado, a los que se hace alusión con nombres dados.
- *EAnnotation*. Permiten hacer una anotación sobre cualquiera de los otros elementos. Su cometido es documentar.

En cuanto al soporte técnico del lenguaje ECore, EMF genera automáticamente un editor básico para un LM a partir del metamodelo correspondiente. El editor generado se integra como plugin de Eclipse y representa la *sintaxis abstracta* de los modelos en forma arbórea. La sintaxis abstracta define el vocabulario y taxonomía del LM. La Figura 2.4 muestra un ejemplo de editor generado por EMF, que edita los modelos siguiendo una estructura de árbol marcada por las instancias de las *EReference containment*. La profundidad del árbol se indica por el espacio de sangría. Cada instancia de EClass es representada con un nodo del árbol, y los atributos de dichas EClasses se editan en la parte inferior del editor. En el editor, la creación de nodos se realiza con el menú contextual, mientras que los valores de los atributos se seleccionan de una lista o se escriben. Además, el *Graphical Modeling Framework* (GMF) (Eclipse, 2005) permite generar editores visuales a partir de metamodelos ECore y otros modelos adicionales que describen aspectos de representación de las entidades y comportamientos en el editor.

Respecto al proceso de instanciación, la Figura 2.5 muestra cómo se usan las capas de la arquitectura de metadatos en el contexto de EMF. ECore es el lenguaje de meta-modelado definido en M3. Algunos de los elementos de ECore son *EClass*, *EAttribute* y *EReference*. En M2, el metamodelo se construye usando las instancias de los elementos ECore. Por ejemplo, en la Figura 2.5, se muestra que *Agent* es una instancia de EClass, *State* es una instancia de EAttribute, y la conexión entre *Goal* y *Agent* es una instancia de EReference. Posteriormente, los modelos en M1 son definidos usando instancias de los elementos del metamodelo

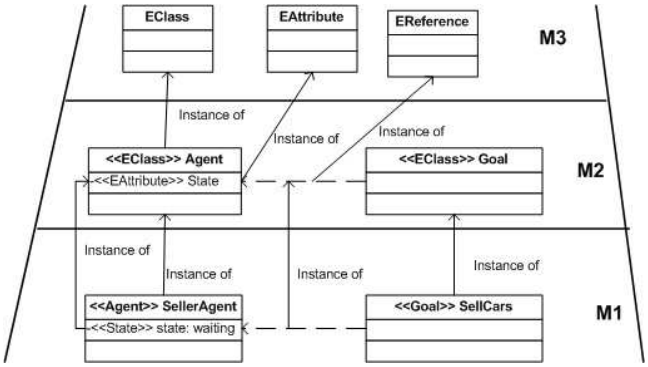


Figura 2.5: El proceso de instanciación de ECore en la arquitectura de metadatos.

de la capa M2. Por ejemplo en la Figura 2.5, *SellerAgent* es una instancia de Agent. El atributo *state* de *SellerAgent* es una instancia del meta-atributo *State* de *Agent*. Es importante destacar que el meta-atributo define su nombre y su tipo en M2, mientras que el atributo define su valor (i.e. *waiting* en este caso) en M1. La meta-referencia en M2 se instancia en M1 para definir una referencia entre *SellCars* y *SellerAgent*. Sin embargo, EMF no permite acceder a crear instancias del modelo, en el nivel M0, lo que se puede ver como una desventaja y ha implicado que cada diseñador escoja su propia solución para representar la instanciación de M1 a M0. Por ejemplo, Kolovos et al. (2006) usa bases de datos para representar el nivel M0. En esta aproximación, cada elemento de M1 se convierte en una tabla de una base datos en M0, y cada instancia en M0 se representa como una fila de una tabla de la base de datos.

Al serializarse, tanto los metamodelos ECore como sus instancias se representan con documentos XML. Cada uno de estos documentos tiene un elemento XML por cada elemento del metamodelo. Por ejemplo, un EPackage puede ser identificado por un elemento XML *ecore:EPackage* o un elemento *eSubpackages*. Una EClass se representa con un elemento *eClassifiers type="EClass"*. Algo parecido ocurre con el resto de los elementos (Budinsky, 2003). Entender las diferencias entre las EReferences *containment* y *non-containment* en la serialización del nivel M1 es necesario para entender el resto del documento. En las EReferences *containment* de M1, se dice que *se ha creado* un nuevo elemento (i.e. EClass de M1), y se refiere a que el elemento XML padre tiene los elementos XML en su interior. Por el contrario, las EReferences non-containment de M1 se representan con un atributo XML. El contenido de este atributo referencia a otro elemento XML, usando el lenguaje XPath (Clark et al., 1999a). Dicho elemento XML se corresponde a la EClass de M1 a la que apunta. La combinación de EReference *containment* y *non-containment* de M1 permite que una única EClass de M1 sea referenciada en más de un sitio sin tener que ser duplicada.

Notación del Lenguaje de ECore El lenguaje de M3 que más se usará en esta memoria es el lenguaje ECore y la notación usada se describe a continuación. Los elementos que tiene este lenguaje son EClass, EReference, EAttribute

y otros. Sin embargo, en esta memoria, la mayoría del tiempo se usarán las instancias de estos elementos, para definir ciertos meta-elementos.

Por no repetir constantemente la expresión *instancia de*, se asumirán ciertas abreviaturas implícitas que es importante tener en cuenta:

- Se usará *EClass* para referirse a las instancias de EClass.
- Se usará *EReference* para referirse a las instancias de EReference.
- Se usará *EAttribute* para determinar las instancias de EAttribute.
- Se usará *EOperation* para determinar las instancias de EOperation.
- Se usará *elementos ECore* para referirse a todas las instancias de los elementos del lenguaje ECore. Por tanto, engloba a todos los conceptos definidos anteriormente, instancias de EClass, instancias de EReference, instancias de EAttribute, e instancias de EOperation.

Estas abreviaturas generan confusión si se habla de ambas capas M2 y M1 a la vez, dado que no se sabe si el término EClass se está refiriendo propiamente a EClass o a una instancia de EClass. En estos casos, se indica explícitamente el nivel de la arquitectura de metadatos (i.e. M2 o M1).

En esta memoria, para representar los metamodelos y los modelos se utilizan dos posibles notaciones visuales:

- Notación del plugin de Eclipse de EMF. Se usa el plugin correspondiente para representar los metamodelos y modelos. Es una representación en forma de árbol, en la que el espacio de sangría indica la profundidad del árbol. Cuanto más espacio de sangría, mayor nivel de profundidad en el árbol. Cada elemento es representado con su nombre y una pequeña imagen a la izquierda que indica el tipo del elemento. En esta notación, los elementos de meta-modelado (véase Figura 2.6) son representados con las siguientes imágenes:
  - EPackage: Cuadrado dividido por una cruz.
  - EClass: Rectángulo dividido por dos líneas horizontales.
  - EAttribute: Rectángulo.
  - EReference: Rectángulo con flecha. Los límites de la cardinalidad aparecen justo debajo. Si no aparecen dichos límites, la cardinalidad es *uno*.
- Notación Gráfica. Es una notación bastante generalizada en muchos trabajos, como por ejemplo en (Leroux et al., 2006). Un ejemplo de esta notación se puede observar en la Figura 2.5. En los metamodelos (i.e. M2), se utilizan las siguientes representaciones gráficas:
  - EClass: Cuadrado
  - EReference: Flecha. Las EReference *containment* son continuas y las *non-containment* son discontinuas.
  - ESuperTypes: Símbolo de herencia de UML.

  La notación gráfica para los modelos (i.e. M1) es la siguiente:

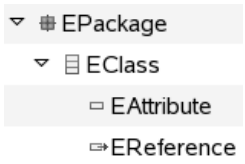


Figura 2.6: Notación del plugin de Eclipse de EMF para metamodelos.

- Instancia de EClass: Cuadrado.
- Instancia de EReference: Flecha. Las flechas son continuas o discontinuas dependiendo si son instancias de EReference *containment* o *non-containment* respectivamente.

Las entidades y relaciones son respectivamente instancias de meta-entidades y meta-relaciones; por tanto, las entidades y relaciones incluyen los nombres de dichas meta-entidades y meta-relaciones, entre los símbolos "<<" y ">>", para indicar explícitamente sus tipos.

GOPRR

GOPRR (Grafo Objeto Propiedad Relación Rol) (Tolvanen, 2000; Kelly, 1997) es un lenguaje de meta-modelado que permite la definición de metamodelos y modelos usando grafos en una notación visual. Los principales elementos de este lenguaje son:

- *Grafo*. Este elemento es el usado como raíz y contiene a todos los demás. Los grafos pueden conectarse con otros grafos a través de estructuras llamadas *Explosión*. Estos segundos grafos pueden ser usados para agrupar elementos. Los grafos tienen *Propiedades*.
- *Objeto*. Se usa para representar cualquier objeto genérico. Tiene varias *Propiedades*. Además los Objetos poseen herencia y pueden crear sus propias jerarquías.
- *Relación*. Sirve para relacionar cualquier número de Roles. Por tanto, representan relaciones n-arias. Las Relaciones también tienen *Propiedades*.
- *Rol*. Se usa para representar los extremos de las Relaciones. En M1, cada Rol conecta un Relación con un Objeto de una clase de objetos determinada. Los Roles también tienen *Propiedades*.
- *Propiedad*. Almacena un valor de tipo primitivo.

Tanto en (Tolvanen, 2000) como en (Kelly, 1997), se dan más detalles de cómo es GOPRR.

Otros Lenguajes de Meta-modelado

El lenguaje *Kernel Meta-MetaModel* (KM3) (Jouault y Bézivin, 2006) es un lenguaje específico de dominio que permite definir metamodelos y tiene asignada una sintaxis textual por defecto. Además, se han definido transformaciones

bidireccionales a los dos lenguajes de meta-modelado más aceptados: MOF y ECore. De esta manera, este lenguaje permite usar los recursos tecnológicos de estos dos lenguajes: Netbeans MDR y EMF.

Otra posibilidad es usar UML para representar metamodelos[1]. Es decir, se usa UML como si fuera un lenguaje de nivel M3. De hecho, la especificación de MOF (OMG, 2006b) tiene como objetivo futuro definir un lenguaje de meta-modelado universal llamado UDL (Date, 1980), que en realidad sería un refinamiento de UML. En concreto, Domokos y Varró (2002) proponen un entorno visual para crear modelos basados en metamodelos definidos con UML. Más tarde, se realiza un modelo, y se le añade cierta información gráfica. Con esto, el entorno es capaz de generar un documento SVG, visible por los navegadores y que representa gráficamente dicho modelo. Como ejemplo, Domokos y Varró (2002) proponen el metamodelo de las redes de Petri usando UML.

Comparación entre Lenguajes de Meta-modelado

La comparación se basa en los requisitos estructurales (Costagliola et al., 2002) más frecuentes en los lenguajes basados en conexiones, que son los empleados habitualmente en ingeniería del software. En una primera etapa de meta-modelado, es imprescindible saber que un metamodelo dado se puede representar en el lenguaje que se vaya a usar. Recurriendo a diferentes convenios, y usando un mismo elemento con finalidades diferentes, generalmente casi todos los metamodelos se pueden representar con todos los lenguajes mencionados en este capítulo (i.e. MOF, ECore, GOPRR, KM3 y UML).

En una segunda etapa, es conveniente ver qué elementos y características estructurales se quieren y en qué lenguaje de M3 es más fácil la representación del metamodelo. Entre otros, los elementos deseables (Costagliola et al., 2002) para satisfacer los requisitos estructurales de los lenguajes basados en conexión son:

- *Clases de datos*, que tengan atributos y operaciones.
- *Relaciones*, a ser posible n-arias, y que tengan sus propios atributos.
- *Extremos de las relaciones*, que conviene tengan sus propias propiedades definidas por el usuario.
- *Un mecanismo de agrupación*, como por ejemplo paquetes de elementos. También es deseable que tengan mecanismos de importación y anidamiento.

En función de estos elementos deseables en un metamodelo, se han comparado los tres lenguajes principales (MOF, EMF y GOPRR). Más adelante, se comparan otras características de los lenguajes que también pueden ser interesantes aunque en segundo plano.

A continuación, se muestran las equivalencias y diferencias entre MOF y el lenguaje ECore según los elementos deseables en un metamodelo:

- La clase de MOF se corresponde con la EClass de ECore, dado que ambas contienen atributos y operaciones.

[1] Esto se difiere de usar los *Profiles* de UML. Los perfiles de UML permiten definir particularizaciones de UML para LMs específicos pero es una aproximación diferente a la definición de metamodelos y, por tanto, se excluyen de la discusión de esta tesis

- Los atributos de MOF se corresponden con las EReference y EAttribute de las ECore. Sin embargo, las instancias de EReference tienen la posibilidad de establecer la multiplicidad con los atributos de M3 *lower-bound* y *upper-bound*, y en los atributos de MOF no se puede[2].

- Las asociaciones de MOF no tienen equivalente exacto en ECore. Estas asociaciones son bidireccionales y tienen multiplicidad en cada uno de los dos extremos. En ECore, se usan las EReference pero estás son unidireccionales y pertenecen a una EClass dada. La EReference sólo tiene multiplicidad en un extremo. Por tanto, es como si sólo tuviera un extremo en cierta forma.

- Los operaciones de MOF se corresponden con las EOperation de ECore. Sin embargo, sólo las operaciones de MOF consideran el lanzamiento de excepciones.

- Los paquetes de MOF se corresponden con los EPackage. Ambos tienen anidamiento e importación. La diferencia es que los paquetes de MOF tienen herencia.

- Las EReference de ECore son de dos tipos, *containment* y *non-containment*. Este concepto no tiene su equivalente en MOF. Esto constituye una diferencia sustancial.

Ahora, se muestran las equivalencias y diferencias de GOPRR con MOF y GOPRR con el lenguaje ECore, según los elementos deseables en un metamodelo:

- El Object de GOPRR se corresponde con la clase de MOF y a la EClass de ECore.

- Las relaciones de GOPRR son n-arias, mientras que las relaciones de MOF y las EReference de ECore sólo relacionan dos tipos de objetos. Además las relaciones de GOPRR tienen propiedades, siendo esto una ventaja frente a MOF y a ECore,

- Las propiedades de GOPRR son equivalentes a los atributos de MOF y a los EAttribute de ECore.

- Los roles de GOPRR no tienen equivalente en MOF ni en ECore, y permiten definir meta-atributos sobre los extremos de las relaciones. En MOF y ECore esto se puede hacer usando de nuevo algunos de los otros elementos como la clase de MOF o la EClass de ECore. Sin embargo, ni en MOF ni en ECore hay un elemento de M3 específico para representar los extremos de las relaciones, mientras que GOPRR sí lo tiene.

- Los grafos de GOPRR tienen sus similitudes con los paquetes de MOF y los EPackage de ECore, ya que sirven para agrupar al resto de elementos. Se difieren en que los grafos de GOPRR presentan agrupaciones de elementos interconectados con el objetivo de representar gráficamente una parte del metamodelo, mientras que los paquetes de MOF y los EPackage de ECore agrupan los elementos por su significado.

[2]MOF tiene atributos llamados *lower* and *upper* pero se refieren a otros conceptos

Categoría	Descripción	LMs
Basados en las Conexiones	Los LMs basados en conexiones tienen un conjunto de objetos interconectados, mientras que la posición espacial de los elementos no es relevante.	Unified Modeling Language (UML) (OMG, 2007a,b), Software Process Engineering Metamodel (SPEM) (OMG, 2008c), Business Motivation Model (BMM) (OMG, 2008b), CORBA Component Model Specification (OMG, 2006a), Knowledge Discovery Meta-Model (KDM) (OMG, 2008a), Common Warehouse Metamodel (CWMM) (OMG, 2003a)
Basados en la Geometría	En los LMs basados en la geometría, los elementos son situados espacialmente en el plano cartesiano. Ciertas reglas espaciales de composición son usadas para definir conceptos tales como la adyacencia, la concatenación o la inclusión.	Aufare-Portier (1995), Políticas de Autorización de Chang et al. (1997)

Tabla 2.1: LMs basados en conexión y LMs basados en geometría

- GOPRR no tiene nada equivalente a las operaciones de MOF ni a las EOperation de ECore.

MOF recoge los cuatro niveles de modelado, como ya se explicó en la Figura 2.2. Por el contrario EMF, sólo da soporte a dos niveles: metamodelo y modelo. En este aspecto, se puede decir que MOF es más completo.

En cuanto al soporte tecnológico, EMF es un plugin de Eclipse y proporciona una herramienta para el desarrollo de metamodelos. GOPRR también tiene una herramienta de desarrollo de metamodelos, llamada MetaEdit(Tolvanen y Rossi, 2003). Para MOF, hay varias: NetBeans MDR(Matula, 2003), *Enterprise Architecture* (Schekkerman, 2008) y RSA (Leroux et al., 2006).

MOF sólo permite exportar modelos para CORBA IDL. Por el contrario, EMF puede exportar a diferentes formatos: Clases Java y Modelos Rational. En este aspecto EMF es más versátil.

2.2.3 Clasificación de los metamodelos

Los metamodelos existentes definen un amplio rango de LMs. En particular, los LMs pueden ser clasificados en los *basados en las conexiones* y los *basados en la geometría*. La Tabla 2.1 proporciona una definición de cada tipo de LM e indica ejemplos de metamodelo para cada categoría. Como se resume en la Tabla 2.1, los LMs más populares, tales como UML (OMG, 2007a,b) y SPEM (OMG, 2008c), se basan en conexiones. Todos los metamodelos de LMs para SMAs están basados en conexiones y se describen en el apartado 2.2.7. Por el

contrario, los escasos LMs basados en geometría (Aufare-Portier, 1995; Chang et al., 1997) están obsoletos. Por esta razón, este trabajo de tesis se centra en los LMs basados en las conexiones.

Los metamodelos existentes se definen de manera muy diversa. Tomando como ejemplo el lenguaje ECore, la Tabla 2.2 muestra ejemplos de metamodelos clasificados por su representación. Incluso los LMs que tienen las mismas características estructurales se representan de maneras diferentes en los metamodelos. Por ejemplo, tanto el metamodelo de *Rational Software Architect* (RSA) (Leroux et al., 2006) como el de *Arquitectura para la Integración de Sistemas informáticos* (ARIS) (Kern y Kuhne, 2007) incluyen relaciones binarias sin atributos. Sin embargo, RSA representa las relaciones con EReferences, mientras que ARIS representa las relaciones con EClasses. Otro ejemplo es el *metamodelo de ingeniería de ontologías desarrolladas por modelos* (Pan et al., 2006), los *metamodelos de sistemas semánticos* (Kappel et al., 2007) y los *metamodelos para ontologías* (Kappel et al., 2006). Estos metamodelos tienen el mismo ámbito, pero usan tres representaciones diferentes de las relaciones: el primero EReferences, el segundo EClasses, y el tercero EClasses tanto para los cuerpos como para los extremos de las relaciones.

En cuanto a la agrupación de elementos de modelado en paquetes[3] (véase Tabla 2.2), predomina la agrupación por la semántica del dominio. Por ejemplo, SPEM (OMG, 2008c) es un metamodelo para procesos y los elementos están agrupados según los conceptos básicos de procesos, entre otros, en los paquetes: *núcleo, estructura, comportamiento, contenido de gestión* y *métodos* . Algo similar ocurre en el resto de especificaciones de OMG tales como UML (OMG, 2007a,b) y el *Knowledge Discovery Meta-Model* (KDM) (OMG, 2008a). Además, se observa un patrón recurrente en los metamodelos estudiados: existe un paquete, generalmente llamado *núcleo*, que contiene los elementos básicos de los cuales extienden los elementos del resto de los paquetes.

Los paquetes tienen además una variedad de usos alternativos en los metamodelos existentes (véase Tabla 2.2). Por ejemplo, Kappel et al. (2006) representan cada ontología de OWL con un EPackage. Mei et al. (2008) también representan cada ontología con un EPackage. Los EPackage también han sido usados para representar los diagramas en la integración de los diagramas de clase (Boronat et al., 2007) y en la integración entre los modelos de ARIS y EMF (Kern y Kuhne, 2007).

La conclusión del análisis presentado de las representaciones de los metamodelos es que existe una gran variedad en la representación de los metamodelos existentes. Principalmente, los puntos de divergencia son la representación de las relaciones y la agrupación en paquetes. Los diseñadores de metamodelos no pueden encontrar un punto de referencia fiable dado la diversidad de representaciones de modelos junto con la ausencia de guías. Por tanto, en el estado actual de las definiciones de metamodelos, esta tesis presenta una guía cuyo objetivo es proporcionar un marco adecuado para el aprendizaje y maduración en la definición de metamodelos.

[3]Los Paquetes de MOF están alineados (Mohamed et al., 2007) con los EPackage de ECore

Representación	**Metamodelos**
EReferences para las relaciones	Rational Software Architecture (RSA)(Leroux et al., 2006), Metamodelos de Quintero y Valderrama (Reina y Torres, 2007) para la orientación a aspectos, Metamodelos para crear guiones por introspección (Tombelle y Vanwormhoudt, 2006), Ingeniería de ontologías dirigidas por modelos (Pan et al., 2006)
EClasses para las relaciones	Arquitectura para la Integración de Sistemas Informáticos (ARIS) (Kern y Kuhne, 2007),
EClasses para los cuerpos de las relaciones y para los extremos de las relaciones	Metamodelo de una Herramienta para el Análisis de Seguridad (Suss et al., 2006), Análisis de la Arquitectura y Diseño del Lenguaje (AADL) (Bodeveix et al., 2005),
Dobles referencias para las relaciones	Los estándares internacionales ISO serie 19100 (Faucher y Lafaye, 2007), Arquitecturas orientadas por servicio dirigidas por procesos (Tran et al., 2007),
EClasses para las relaciones y dobles EReferences para las conexiones	Metamodelos para Sistemas Semánticos (Kappel et al., 2007),
EClasses para ambos cuerpos de relaciones y extremos de relaciones, y dobles EReferences para las conexiones	Metamodelos para Ontologías (Kappel et al., 2006),
EPackage para la agrupación de los elementos según la semántica del dominio	*Software Process Engineering Metamodel* (SPEM) (OMG, 2008c), *Unified Modeling Language* (UML) OMG (2007a,b), *Knowledge Discovery Meta-Model* (KDM) OMG (2008a)
EPackage para representación una ontología	*Lifting metamodels to ontologies* (Kappel et al., 2006), *Semantic Web Style* (Mei et al., 2008)
EPackage para representar un Diagrama	*Class Diagram Integration* (Boronat et al., 2007), *Model Interchange between ARIS and EMF* (Kern y Kuhne, 2007)

Tabla 2.2: Diversidad en la Definición de Metamodelos

2.2.4 Características estructurales de los lenguajes de modelado

Este apartado describe las principales características estructurales de los LMs *basados en conexiones*. Este estudio se centra los LMs *basados en conexiones* dado que la mayoría de metamodelos definen este tipo de LMs, como se justificó en el apartado 2.2.3. Estas características estructurales son las siguientes:

Característica Estructural 1 Entidades con atributos. *La información de los modelos suele representarse en entidades con atributos en casi todos los LMs.*

Característica Estructural 2 Varias relaciones de M2. *Esto significa que hay diferentes formas de relacionar las entidades. Por ejemplo, INGENIAS tiene* 89 *relaciones de M2. UML y SPEM también tienen varias relaciones de M2.*

Característica Estructural 3 Relaciones con atributos. *Esto es necesario cuando las relaciones necesitan almacenar cierta información. Por ejemplo, UML tiene la relación* aggregation *con el atributo* stereotype.

Característica Estructural 4 Extremos de las relaciones con atributos. *Estos atributos están asociado a cada extremo donde las relaciones se unen con las entidades. Por ejemplo, el atributo de* multiplicidad *es muy común en los diseños orientados a objetos, y está asociado a los extremos de ciertas relaciones. Como un ejemplo adicional, SPEM tiene los atributos* isNavigable *y* aggregationKind *en los extremos de la relación* association.

Característica Estructural 5 Relaciones n-arias. *Estas relaciones pueden enlazar más de dos entidades. Por ejemplo, INGENIAS y AUML*[4] *son ejemplos de LMs para SMAs que usan relaciones n-arias. Además, las relaciones n-arias se usan en LMs de otras disciplinas tales como en la web semántica (Russomanno, 2006) y la medicina (Noy et al., 2002).*

Característica Estructural 6 Jerarquías de entidades y Relaciones de M2. *Algunos LMs clasifican las entidades y las relaciones en jerarquías. Por ejemplo, este es el caso de INGENIAS y SPEM.*

2.2.5 Creación de herramientas de modelado

Una de las tendencias más importantes en el DSDM es la creación de herramientas de modelado. En esta rama del DSDM, el metamodelo dirige el desarrollo del software de las herramientas de modelado correspondientes. Existen herramientas de modelado de propósito general construidas con una aproximación DSDM, tales como el proyecto *Eclipse UML2*[5], que permite definir modelos UML. Sin embargo, los *Lenguajes de Modelado Específicos del Dominio* (LMEDs) (Chen et al., 2005; Magyari et al., 2003; Amyot et al., 2006) son los principales beneficiarios de esta aproximación, dado que se pueden construir herramientas para estos lenguajes, usados por minorías, con un coste aceptable.

Como aclaración, los LMEDs son LMs para dominios restringidos, y se contraponen a los LMs generales tales como UML. Algunos ejemplos de LMEDs son:

[4] http://www.auml.org/
[5] www.eclipse.org/uml2/

el *Chemical Markup Language (CML)* (Murray-Rust et al., 2001), un lenguaje de marcado para definir átomos y sus características; y el *GALEN Representation And Integration Language (GRAIL)* (Rector et al., 1997), un lenguaje para definir interfaces de usuario para clínicas y extender modelos reutilizable de la terminología médica. Para GRAIL, la herramienta *GRAIL Tool v1.3* (Darimont, 2002) tiene soporte para realizar modelos, publicarlos en la web, validar dichos modelos, y generar automáticamente documentación. Por otro lado el lenguaje de propósito general más relevante es UML, para el cual existen herramientas tales como *Eclipse UML2* y *Rational Software Architecture* (RSA) (Leroux et al., 2006).

Para dirigir el desarrollo de herramientas de modelado, se han desarrollado armazones que toman como entrada el metamodelo y, en algunos casos, otros modelos adicionales, y generan como salida una herramienta de modelado. La manera de definir los metamodelos influirá en la herramienta generada. Por tanto, se necesita tener presentes las características requeridas en el metamodelo y cómo plasmarlas. Formalizar y hacer disponible este conocimiento requiere guías de meta-modelado. Además, el diseñador necesita elegir y conocer bien el armazón para generar una herramienta de modelado. A continuación, se introduce brevemente dichos armazones.

XMF-Mosaic(Nytun et al., 2006) toma como entrada metamodelos definidos en MOF y restricciones expresadas en el *Object Constraint Language* (OCL); mientras que el *Generic Modeling Environment* (GME) (Ledeczi et al., 2001b) usa su propio lenguaje de M3. *Telelogic Tau G2*(IBM, 2008) usa los *Profiles* de UML 2.0 (OMG, 2007a,b), implementando el mecanismo de estereotipos. Además, tiene la peculiaridad de que puede tomar como entrada metamodelos que usen el mecanismo de extensión.

Eclipse, con EMF (Budinsky, 2003) y el Graphical Editing Framework (GEF) (Moore et al., 2004), proporciona generación de código y creación de editores arbóreos, a partir de metamodelos expresados con el lenguaje ECore. GEF permite definir un editor gráfico a partir de metamodelos construidos con EMF. En concreto, la última herramienta de Rational, RSA (Leroux et al., 2006), se ha montado sobre Eclipse con EMF y GEF. Por otro lado, GMF (Eclipse, 2005) permite generar un editor gráfico a partir de un metamodelo en ECore y ciertos modelos adiciones que indican las características del editor gráfico.

MetaEdit (Tolvanen y Rossi, 2003) también incluye generación de código a partir de metamodelos definidos con GOPRR.

En (Amyot et al., 2006), se pueden encontrar una evaluación y comparación más detallada de los siguientes armazones para la generación de herramientas de modelado: *XMF-Mosaic; GME; Telelogic Tau G2; Eclipse con EMF, GEF y GMF.*

Las técnicas presentadas en este apartado son útiles para la creación de herramientas de modelado para LMs de SMAs. Si bien, todavía la generación de herramientas es parcial, como en el IDK, el objetivo de esta tesis el proporcionar una guía de definición de metamodelos que, entre otras cosas, facilite la generación de herramientas de modelado para SMAs.

2.2.6 Dificultades en la definición de metamodelos

Las principales dificultades en la definición de metamodelos se describen a continuación:

- *Los lenguajes de meta-modelado (véase apartado 2.2.2) proporcionan elementos diferentes.* La definición del metamodelo varía según el LM (véase los lenguajes en apartado 2.3.3), las características estructurales del mismo (e.g. relaciones con atributos en el cuerpo y extremos, relaciones n-arias, jerarquía de entidades) y los objetivos del procesamiento del metamodelo (e.g. eficiencia, claridad y generación de herramientas de modelado). Estas variaciones implican que haya diferentes formas de definir metamodelos y se requiere un coste y esfuerzo en elegir la solución más apropiada.

- *Para la agrupación de elementos de meta-modelado en paquetes, se pueden tener en cuenta diferentes aspectos.* Se puede agrupar los elementos según la semántica del dominio, o se puede usar el elemento paquete para representar un concepto existente en el dominio (véase apartado 2.2.3 para más detalles). Por ello, se debe dedicar tiempo a elegir el criterio más apropiado, y no siempre es fácilmente comprensible por el resto de diseñadores.

- *Los elementos de meta-modelado se pueden considerar conceptos de una ontología.* En este caso, se debe tener en cuenta ciertas reglas de coherencia semántica, que se pueden formalizar con restricciones sobre los modelos. En esta línea, es relevante el trabajo de Opdahl y Henderson-Sellers (2001).

- *La terminología en la definición de metamodelos puede crear confusión.* Por ejemplo, en la definición de MOF (OMG, 2006b) se usa el término *class* para la clase de UML, para la clase MOF, y a su vez la clase de UML está definida por la clase de MOF. Esto es, hay que optar por ciertos convenios para evitar confusión, como usar nombres diferentes, usar prefijos *meta* o especificar el nivel de metadatos según MOF (M3, ..., M0). En esta memoria esta dificultad se ha resuelto con la notación del apartado 2.2.1.

2.2.7 Metamodelos en sistemas multi-agente

En los últimos años, se han definido algunos LMs de SMAs con metamodelos. Los siguientes trabajos describen dichos metamodelos: Tropos (Susi et al., 2005); PASSI y Agile PASSI (Cossentino et al., 2005); MaSE (DeLoach, 2006); ADELFE (Rougemaille et al., 2007) e INGENIAS (Pavón y Gómez-Sanz, 2003). Actualmente, el objetivo del metamodelo del *FAME Agent-oriented Modeling Language* (FAML) (Beydoun et al., 2006) es hacer un LM genérico que sirva para el modelado de cualquier SMA, independientemente de la metodología que se use.

Los requisitos estructurales de LMs de SMAs son: atributos en los cuerpos y extremos de las relaciones; relaciones n-arias; cada tipo de relación está restringido a conectarse con sólo ciertos tipos de entidades; jerarquías tanto para las entidades como para las relaciones. Además, según la definición del metamodelo, los elementos se agrupan en paquetes atendiendo a dos criterios posibles: agrupación de elementos según la semántica de los conceptos, y diferenciación del uso de los elementos de meta-modelado. Las propiedades estructurales de algunos metamodelos existentes de LMs de SMAs se recogen en la Tabla 2.3. De esta manera, se muestran las características estructurales que son necesarias para proporcionar una técnica para definir metamodelos para LMs de SMAs.

	Tropos	**PASSI**	**MaSE**	**ADEL-FE**	**INGE-NIAS**	**FAML**
Relaciones N-arias	√	√	√	√	√	√
Relaciones con Atributos	√	√	√	√	√	√
Relaciones con Atributos en los extremos	X	X	√	X	√	√
Extremos de Relaciones restringidos a Varios Tipos de Entidades	√	√	X	X	√	√
Jerarquía de Entidades	√	√	√	√	√	√
Jerarquía de Relaciones	X	X	X	X	√	√
Agrupación en Paquetes según semántica de los conceptos	√	√	X	√	√	√
Uso de los Paquetes para representación del metamodelo	X	X	X	X	√	X

Tabla 2.3: Propiedades estructurales de algunos metamodelos existentes de LMs de SMA

La Tabla 2.3 se complementa con otros aspectos relevantes relacionados con los metamodelos existentes de LMs de SMA:

- El metamodelo de *Tropos* (Susi et al., 2005) está definido con ECore y destaca por tener un metamodelo extendido. En un primer nivel, el metamodelo es definido para los conceptos básicos de SMAs; dicho metamodelo es extendido en un segundo nivel con conceptos relativos a la seguridad. Su entorno de modelado es *TAOM4e* y se sustenta en las librerías de EMF y GEF.

- Los metamodelos de *PASSI y Agile-PASSI* (Cossentino et al., 2005) son característicos por tener en cuenta en el metamodelo tres vistas ortogonales: el dominio del problema, el dominio de la solución, y el dominio relacionado con los agentes. Los elementos de cada una de estas vistas están agrupados en un paquete en el metamodelo.

- El metamodelo de *MaSE* (DeLoach, 2006) tiene todos los elementos del metamodelo agrupado en un único paquete. Los conceptos del metamodelo se centran en el conocimiento necesario para definir la organización, de tal manera que los agentes puedan formar equipos de trabajo de forma dinámica. DeLoach (2006) añade semántica sobre las instancias del metamodelo, con funciones numéricas, tales como la función *potencial* que, para cada tupla *(agente, rol, objetivo)*, devuelve un valor en el intervalo $[0, 1]$. Actualmente, es el único metamodelo de un LM de SMAs al que se le añade semántica con funciones numéricas.

- La metodología *ADELFE* (Rougemaille et al., 2007) sigue los principios del DSDM y se centra en los SMAs adaptativos. Los LMs relacionados con ADELFE se definen con dos metamodelos: el metamodelo para sistemas multi-agente adaptativos, llamado *metamodelo AMAS*, y el metamodelo que define el lenguaje de descripción de la micro-arquitectura, llamado metamodelo *μADL*. El metamodelo AMAS usa tres vistas para: el entorno, los agentes, y la cooperación. Por otro lado, el metamodelo *μADL* está más cercano a la implementación. Ambos metamodelos están definidos con el lenguaje ECore y se definen transformaciones modelo-a-modelo entre los dos metamodelos.

- El metamodelo de *INGENIAS* (Pavón y Gómez-Sanz, 2003) destaca por su gran cantidad de elementos, que suma aproximadamente unos ochenta tipos de entidades, unos ochenta tipos de relaciones y unos ochenta tipos de extremos de relaciones. En el 2003, este metamodelo incluyó todos los elementos de modelado existentes en las metodologías de ISOA de la época.

- El metamodelo *FAML* (Beydoun et al., 2006) proporciona un lenguaje genérico cuyo objetivo es abarcar todos los LMs existentes, y proporcionar un lenguaje en el que cualquier SMA pueda ser modelado. En el proceso de creación de este metamodelo se siguen los siguientes pasos iterativamente: primero, se decide un conjunto de conceptos generales; segundo, se eligen definiciones para los conceptos; tercero, se reconcilian las diferencias de significado de los diferentes conceptos dando definiciones híbridas; por último, se clasifican los conceptos en dos conjuntos, conceptos de ejecución y conceptos de diseño.

Tanto INGENIAS, en el 2003, como FAML, actualmente, buscan el mismo objetivo: definir un LM que permita modelar cualquier SMA, basándose en los LMs del momento. Sin embargo, siguen dos aproximaciones diferentes: en INGENIAS se hizo el conjunto de todos los elementos, mientras que, en FAML, se definen conceptos híbridos. Un análisis crítico muestra que el problema que se pretende resolver entraña gran dificultad, debido a la variedad de SMAs y enfoques para modelar. FAML cuenta con apoyo de los expertos de las metodologías de SMAs más importantes, y se trata por tanto de un trabajo bien documentado. Por otro lado, la definición de conceptos híbridos excluye ciertos matices en los conceptos y relaciones entre conceptos, que surgen para tipos de SMAs concretos, como los SMAs adaptativos que propone ADELFE. Por tanto, a pesar de la iniciativa de FAML, se prevé que seguirán existiendo y evolucionando diferentes LMs para SMAs, y por tanto seguirá existiendo la necesidad de definir nuevos metamodelos o cambiar los metamodelos existentes.

2.2.8 Información específica de las herramientas CASE de modelado

La información específica de las herramientas CASE de modelado se presenta en este apartado, dado que esta información se puede incluir también en los metamodelos acorde con la contribución de esta tesis. En general, las herramientas de modelado requieren un proyecto y un espacio de trabajo. Sin embargo, la forma de estructurar un proyecto y los atributos del espacio de trabajo son

variados. La Tabla 2.4 compara estas características sobre una herramienta de meta-modelado, *Eclipse junto con EMF*[6], varias herramientas CASE de modelado general, *BoUML*[7] y *StarUML*[8], y varias herramientas CASE de modelado específico de SMAs, *Prometheus Design Tool* (PDT) (Padgham et al., 2005), *ADELFE Toolkit*[9] y *IBM Rational Software Development (RSD)* para *PASSI* (Henderson-Sellers y Giorgini, 2005).

En concreto, la información específica de las herramientas CASE de modelado puede contener datos del proyecto, tales como su nombre, su descripción, y los nombres de sus autores. También puede contener cierta información temporal, como el último autor que hizo una modificación, y la versión del proyecto. Además, puede contener información referente a la herramienta con la que fue hecho el proyecto, siendo la información más útil la versión de la herramienta. La forma de estructurar el proyecto puede ser: según el sistema de archivos, con paquetes anidados, con vistas, o con una combinación de los anteriores.

Además, las herramientas pueden guardar ciertas preferencias de usuario, tales como: los módulos que se inician al abrir el proyecto, el formato de los diagramas en cuanto al tamaño (A3, A2, ...), o el tamaño de visualización de las letras. En cuanto a los plugins generados a partir de los modelos, las preferencias pueden ser el modo de ejecución del plugin o la ubicación en la que se ejecuta. También pueden incluirse preferencias que caractericen el LM, tales como su notación o su *perfil* (*profile*), siendo el perfil una particularización de un LM dado con estereotipos, inspirándose en la propuesta de OMG de perfiles para UML 2.0 (OMG, 2007a,b).

La información específica de las herramienta CASE de modelado también puede contener algún tipo de historial de cambios, tal como el historial de refactorización. Se entiende por refactorización un cambio que puede afectar a un grupo de elementos (e.g. un cambio del nombre de un elemento que altera a todos los elementos conectados con el mismo). El proyecto también puede tener referencias externas, tales como referencias a librerías o a otros proyectos. En algunas herramientas basadas en el DSDM, el proyecto puede tener una referencia al modelo principal que dirige el desarrollo. Finalmente, se pueden tener algunos datos referentes al sistema de archivos, tales como el nombre y ubicación del archivo donde el proyecto fue guardado la última vez.

Las características mencionadas han sido detectadas en nuestro estudio, pero existen más características en otras herramientas CASE de modelado. Además, con la aparición creciente de nuevas herramientas CASE de modelado, dichas herramientas pueden incorporar nuevos tipos de información que no hayan sido considerados hasta el momento.

A continuación se describe la información más relevante de cada una de las herramientas de modelado estudiadas, pero las características de cada herramienta se muestran con más detalle en la Tabla 2.4. Para empezar, *Eclipse* junto con EMF permite modelar metamodelos y generar plugins; y tiene una ruta local del lugar donde se ejecutan los plugins en desarrollo. Así mismo, Eclipse caracteriza el modo en que se debe ejecutar el plugin. Además, Eclipse estructura el proyecto con la misma estructura que las carpetas del sistema de archivos del espacio de trabajo. En cuanto a *BoUML*, la estructura se realiza

[6] http://www.eclipse.org/
[7] http://bouml.free.fr/
[8] http://staruml.sourceforge.net/en/
[9] http://www.irit.fr/ADELFE/Download.html

	Eclipse (con EMF)	Bo-UML	Star-UML	PDT	ADEL-FE Tool-kit	IBM RSD para PASSI
Nombre del proyecto	√	√	√	√	√	√
Descripción del proyecto	√	X	X	√	X	X
Autores del proyecto	X	X	X	√	X	X
Autor que hizo la última modificación	X	√	X	X	X	X
Versión del proyecto	X	√	X	√	X	X
Versión de la Herramienta con que se hizo el proyecto	X	X	X	√	X	X
Estructura del Proyecto igual que sistema de Archivos	√	X	X	X	X	X
Estructura del Proyecto con paquetes anidados	X	√	X	X	X	X
Estructura según vistas	X	√	√	√	√	√
Preferencias del Usuario de qué módulos se inician	X	X	X	X	√	X
Formato de los Diagramas (tamaño: A4, A3, ...)	X	√	X	X	X	X
Tamaño de fuente de letra	X	√	X	X	X	X
Modo de ejecución del Plugin	√	X	X	X	X	X
Ruta de Ejecución de Plugins	√	X	X	X	X	X
Notación del LM	√	√	X	X	X	√
Perfil (*profile*) del LM	X	X	√	X	X	X
Historial de refactorización	√	X	X	X	X	√
Librerías	√	X	X	X	X	X
Referencias a otros proyectos	√	X	X	X	X	√
Referencia al modelo que dirige el desarrollo	√	X	X	X	X	√
Nombre del archivo la última vez que se guardó	X	X	X	√	√	X
Ruta del archivo la última vez que se guardó	X	X	X	√	√	X

Tabla 2.4: Información específica de los proyectos de las herramientas CASE de modelado

con paquetes, que se incluyen unos dentro de otros, formando estructuras de árbol. Por el contrario, *StarUML* estructura el modelado del proyecto en cinco vistas: *los casos de uso, el análisis, el diseño, la implementación* y *el despliegue*. Dentro de cada uno de ellos, se pueden añadir diferentes conceptos y diagramas que incluyan dichos conceptos.

Respecto a las herramientas CASE de modelado específicas para SMAs, PDT tiene las siguientes propiedades de proyecto: nombre, descripción, autores, versión del proyecto, y versión de la herramienta. PDT estructura el proyecto en tres vistas: *especificación del sistema, diseño de la arquitectura*, y *diseño detallado*. Cada una de las vistas tiene varios diagramas. Todos los diagramas referencian a entidades que se encuentran en un diccionario común, lo que es similar a la aproximación de esta tesis. Sin embargo, PDT no permite tener una estructura de paquetes en árbol que contenga los diagramas. La herramienta *ADELFE Toolkit* permite diseñar SMAs adaptativos. Su información de proyecto contiene la ruta en donde fue guardado el proyecto la última vez. Además, se guardan las preferencias del usuario respecto a qué módulos quiere inicializar cada vez que se abra el proyecto; estos módulos son: *módulo descriptivo, módulo gestionador, módulo ejemplo, módulo de documentación* y *módulo de síntesis*. La herramienta IBM-RSD se usa en los desarrollos de la metodología de agentes PASSI, y está construida sobre Eclipse. Por este motivo, muchos de los atributos del proyecto coinciden con los de Eclipse, aunque tienen sus particularidades, tal como estructurar el proyecto en diferentes vistas: *de componentes*, *lógicas*, *de casos de uso*, y *de despliegue*.

En conclusión, existe una información específica de las herramientas CASE de modelado que es independiente del LM, tanto en herramientas de propósito general como en herramientas específicas para SMAs.

2.3 Transformaciones de modelos

Las TMs (Sendall y Kozaczynski, 2003) son parte esencial del DSDM. Sin embargo, en SMAs el uso de TMs tiene un ámbito reducido a las metodologías: Tropos y ADELFE.

2.3.1 Transformaciones de modelos y sus aplicaciones

Una TM está asociada a un metamodelo de entrada y a un metamodelo de salida. Una TM recibe como entrada un modelo conforme a un metamodelo de entrada y produce como salida otro modelo conforme al metamodelo de salida. Los metamodelos origen y destino pueden ser iguales o distintos. Las TMs (Sendall y Kozaczynski, 2003) son el motor en el DSDM y sus aplicaciones son las siguientes:

- *Transformaciones verticales* (Sendall y Kozaczynski, 2003), que transforman modelos de alto nivel de abstracción a modelos más cercanos a la implementación. Como ejemplo, MDA (OMG, 2003b) usa TMs para cambiar el nivel de abstracción en la cadena de modelos que une el modelo más abstracto y la implementación del sistema y que consta de: los *Modelos Independiente de Computación* (MICs), *Modelos Independientes de Plataforma* (MIPs) y *Modelos Específicos de Plataformas* (MEPs). Las TMs

verticales pueden ser de gran utilidad en el DSDM en SMAs, debido a la gran variedad de plataformas para SMAs. El enfoque de MDA se puede utilizar para definir modelos de SMAs que sean independiente de la plataforma y que luego pueda pasar por una cadena de TMs hasta llegar a la implementación en diferentes plataformas de SMAs (véase más detalles en la Sección 2.3.6).

- *Transformaciones horizontales* (Sendall y Kozaczynski, 2003), que transforman modelos de nivel de abstracción similar, tales como dos LMs específicos de plataforma. Se suelen aplicar en áreas en las que se quiere prescindir del lenguaje independiente de plataforma. Como ejemplo, Abd-Ali y El-Guemhioui (2005) definen TMs para transformar MEPs de la plataforma *Enterprise JavaBeans* (EJB) en MEPs de la plataforma ".Net". La razón de este tipo de TMs es la inexistencia de los correspondientes MIPs. Las TMs horizontales son inexistentes en el DSDM en SMAs, porque apenas están establecidos los lenguajes para MEPs y, por tanto, no ha surgido la necesidad. Aun así no se descarta que en el futuro surja esta necesidad en el DSDM en SMAs.

- *Refactorización o asistencia en el diseño de modelos* (Porres, 2003), que son transformaciones en las que el metamodelo de entrada y el metamodelo de salida son el mismo, y ayudan a refinar o mejorar los modelos de manera automática. Las TMs de refactorización podrían ser de utilidad en los DSDM en SMAs (véase más detalles en el apartado 2.3.6), pero se necesitarían reglas de transformación muchos-a-muchos, como se puede observar en los ejemplos mostrados en el capítulo 6; sin embargo, el soporte es escaso para la creación de este tipo de reglas (véase el apartado 2.3.4).

- *Traducción entre LMs similares* (Tratt, 2005), las cuales transforman modelos de un alto nivel de abstracción entre dos lenguajes del mismo dominio y que se pueden usar en la integración de herramientas de modelado que usan diferentes LMs. Por ejemplo, Tratt (2005) define TMs para la traducción entre LMs similares para intercambiar datos entre herramientas. Los modelos creados por ciertas herramientas son transformados en modelos legibles por otras herramientas. Este tipo de TMs se diferencian de las transformaciones horizontales porque este tipo de TMs obligatoriamente tienen que conservar el significado, mientras que las transformaciones horizontales pueden establecerse entre modelos con distintos significados. Las TMs de traducción serían de gran utilidad en el DSDM en SMAs, dado que existen diversas herramientas para el DSDM en SMAs y estas TMs podrían permitir el intercambio de modelos entre ellas.

2.3.2 Clasificación de transformaciones de modelo

Para la mejor comprensión de las TMs, las definiciones de éstas se clasifican según diversos criterios. Este apartado se centra en las clasificaciones de las transformaciones modelo-a-modelo, ya que este trabajo de tesis se centra en este tipo de transformaciones. Estas clasificaciones son una parte de la base necesaria para entender esta tesis. Para empezar, las TMs se pueden clasificar acorde a los LMs de origen y destino:

Categoría 1 *Las TMs* endógenas *tienen el mismo lenguaje para el origen y destino. Como ejemplo de transformaciones endógenas se pueden observar las transformaciones entre modelos UML (Stein et al., 2005; Varro et al., 2001; Kovse y Harder, 2002).*

Categoría 2 *Las TMs* exógenas *tienen los lenguajes origen y destino diferentes. Dentro de este tipo, se pueden distinguir entre:*

- Aquellas cuyos lenguajes origen y destino están definidos con metamodelos en el mismo lenguaje. *Generalmente estas transformaciones se pueden definir con LTMs. Por ejemplo el lenguaje ATL(INRIA, 1986) define transformaciones con metamodelos de entrada y salida definidos con el lenguaje ECore.*
- Aquellas cuyos metamodelos de entrada y salida están definidos con distintos lenguajes de meta-modelado. *Para estas TMs, se usan lenguajes de transformación generales, por ejemplo para XML o ficheros de texto planos. Estas TMs no son tan robustas como los otros tipos de TMs porque generalmente el soporte de los lenguajes de transformación generales no garantiza que los modelos de entrada y salida sean instancias de los metamodelos de entrada y salida. Para este tipo de TMs, el lenguaje más usado es el* Extensible Stylesheet Language Transformation *(XSLT) (Clark et al., 1999b). Una de las aplicaciones de este tipo de TMs es la traducción de metamodelos, en la cual las TMs están situadas en M3, un nivel por encima de lo habitual. Por ejemplo, en esta línea destaca las transformaciones entre metamodelos expresados respectivamente con MOF y EMF (Gerber y Raymond, 2003).*

Además, Duddy et al. (2003) clasifican las TMs según los elementos que las guíen. Según esta clasificación, las categorías son las siguientes:

- *Dirigidas por Origen.* Se centran en el origen de la transformación. Cada elemento del origen se transforma en uno o varios elementos del destino.
- *Dirigidas por Destino.* Conjuntos complejos de elementos del origen se transforma en único elemento del destino.
- *Dirigidas por Aspectos.* Conjuntos de elementos de entrada puede construir una parte (e.g. un atributo) del elemento destino o viceversa. Por tanto, no sólo se considera elementos, sino sus aspectos.

Czarnecki y Helsen (2003, 2006) proponen una clasificación de enfoques para definir TMs. Esta clasificación permite a los diseñadores esquematizar la manera en que les conviene definir las transformaciones. Estos enfoques de definición de transformaciones son:

Aproximación 1 Enfoque de manipulación directa. *Las transformaciones manipulan cada modelo para hacer cambios que no afectan al resto del modelo. Generalmente estas transformaciones usan lenguajes imperativos como Java.*

Aproximación 2 Enfoque declarativo. *Este enfoque se basa en reglas. Las reglas indican qué elementos de entrada son transformadas en qué elementos de salida. Además, estas reglas tienen asociadas restricciones, que deben ser satisfechas para que la regla se aplique. Un ejemplo del enfoque declarativo puede ser encontrado en el trabajo de Gerber et al. (2002).*

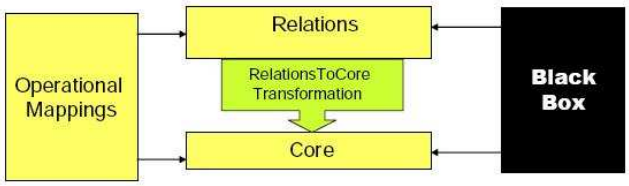


Figura 2.7: Relaciones entre metamodelos en QVT.

Aproximación 3 Transformaciones de grafos. *Estos lenguajes son aplicados a lenguajes de modelado con estructura de grafo. Por ejemplo, Andries et al. (1999) proporciona un trabajo sobre transformaciones de grafos.*

Aproximación 4 Enfoque dirigido por la estructura, también llamado enfoque imperativo (Jouault y Kurtev, 2006b). *Este enfoque consta de dos fases: primero, se crea una estructura jerárquica en el modelo destino; después, los elementos del origen son transformados para rellenar la estructura del modelo destino.*

Aproximación 5 Enfoques híbridos. *Los enfoques híbridos son combinaciones de los enfoques anteriores.*

2.3.3 Estándar y lenguajes de transformación de modelos

Query View Transformation (QVT) (OMG, 2005a) es una propuesta de estándar para las TMs. QVT tiene una parte declarativa y otra imperativa. La parte *declarativa* es una arquitectura de dos capas, como se puede ver en la Figura 2.7. La capa *relations* es amigable para el usuario, y permite definir transformaciones con reconocimiento de patrones y plantillas para la creación de nuevos objetos. La capa *núcleo* es una mínima extensión de EMOF (*Essential MOF*, un subconjunto de MOF) y OCL, y por tanto tiene construcciones muy básicas. Las capas de relaciones y núcleo son dos niveles de abstracción distintos. La parte *imperativa* se puede ver como secuencias de instrucciones.

Una transformación QVT se define como un conjunto de reglas, las cuales tienen un patrón de entrada y unos elementos MOF de salida. Además, dichas reglas pueden tener: unas condiciones, que determinan si se pueden activar las reglas; y una parte imperativa, que se ejecuta cuando se activan las reglas. Siguiendo parte de la especificación de QVT, el lenguaje más conocido es *Atlas Transformation Language (ATL)*(INRIA, 1986). Hay incluso lenguajes de transformación QVT *visuales*, como por ejemplo UMLX (Willink, 2003). Cabe recalcar que existe cierta controversia en la literatura acerca de si ATL es un lenguaje que sigue el estándar QVT, o si por el contrario ATL y QVT son dos lenguajes diferentes que se pueden alinear. El trabajo (Bézivin et al., 2003a) afirma lo primero, mientras que el trabajo (Jouault y Kurtev, 2006a) afirma lo segundo.

Como aclaración, el *Object Constraint Language* (OCL) (Richters y Gogolla, 2000) originalmente era un lenguaje de restricciones aplicable a UML, que acabó incluyéndose en la especificación de UML. Ahora se utiliza para definir restricciones de lenguajes de meta-modelado, como MOF y ECore. Actualmen-

	Restricciones					Acciones		Patrones de la regla	
	existe	atributo	agregación	referencia	elementos de un tipo	asignación	insertar / concatenar	múltiple origen	múltiple destino
QVT	√	√	√	√	√	√	√	X	√
ATL	√	√	√	√	√	√	√	X	√
XSLT	√	√	√	√	√	√	√	X	√
Tefkat	√	√	√	√	√	√	√	X	√
VIATRA2	√	√	√	√	√	√	√	√	√
MT	√	√	√	√	√	√	√	√	√
MOLA	√	√	√	√	√	√	√	√	√
RubyTL	√	√	√	√	√	√	√	√	√

Tabla 2.5: Propiedades de los Lenguajes de Transformación

te, OCL se está convirtiendo en la base de los lenguajes de trasformación que siguen el estándar QVT.

Los *Lenguajes de Transformación de Modelos* (LTMs) son lenguajes que expresan TMs. Entre otros, algunos ejemplos de LTMs son: ATL (Bézivin et al., 2003a; Jouault y Kurtev, 2006b), MT (Tratt, 2007), Tefkat (Lawley y Steel, 2006), VIATRA2(Varró y Balogh, 2007a), MOLA (Kalnins et al., 2006) y RubyTL(Cuadrado et al., 2006). A la hora de usar los LTMs, es necesario saber qué características tienen los LTMs. La Tabla 2.5 recoge las características comunes en la mayoría de los LTMs. Conocer estas características comunes es relevante para crear TMs que tengan una traducción directa en la mayoría de LTMs. Como breve introducción, la mayoría de LTMs están basados en reglas con un patrón de entrada y un patrón de salida, con unas restricciones que condicionan la activación de las mismas, y unas acciones básicas. Algunas de las construcciones básicas de las restricciones son: *existe*, *atributo*, *agregación*, *referencia* y *elementos de un tipo*. La restricción *existe* permite evaluar si un elemento del modelo existe. Las restricciones *atributo* evalúan si el atributo de un elemento tiene un valor dado. Las restricciones *agregación* y *referencia* permiten obtener el elemento apuntado por una agregación o referencia dada desde un elemento dado. La restricción *elementos de un tipo* permite acceder a una lista con todos los elementos de un tipo determinado. Ejemplos de acciones son la *asignación* de una variable de la TM y la *inserción* o *concatenación* de elementos en una variable. Por último, un aspecto relevante es si los patrones de entrada y salida de las reglas pueden ser múltiples o no (dependiendo si permiten incluir varios elementos o sólo uno). Además de las características mencionadas, se citan otras peculiaridades relevantes de cada LTM a continuación.

ATL (INRIA, 1986; Bézivin et al., 2003a; Jouault y Kurtev, 2006b) es usado por un plugin de Eclipse y permite la transformación entre modelos de EMF. Las transformaciones ATL permiten una definición inicial de variables y funciones, llamadas *helpers*. Cada regla tiene un elemento de entrada, que se transforma en unos elementos de salida. En las entradas, se permiten ciertas expresiones

booleanas del lenguaje de restricciones OCL. Además, cada regla tiene una parte imperativa que se ejecuta después de activarse la regla al producirse el encaje del modelo con la entrada de la regla. Por ello, ATL es un lenguaje híbrido entre una aproximación declarativa e imperativa, y sus reglas se pueden usar de muy diversas maneras y con muy diferentes objetivos. Jouault y Kurtev (2006b) explican que las transformaciones se pueden hacer de forma *declarativa* e *imperativa*. Esto depende de si la mayor parte de la transformación se hace con la parte declarativa de las reglas o con la parte imperativa de las reglas. Inicialmente, las transformaciones ATL fueron pensadas para el contexto de MDA (Bézivin et al., 2003; Bézivin et al., 2003b, 2004). Sin embargo, su ámbito de aplicación se ha ido ampliando. Por ejemplo, Bézivin et al. (2003a) muestran una transformación ATL de XSLT a XQuery, que son respectivamente un lenguaje de transformaciones y un lenguaje de consultas.

El lenguaje *MT* (Tratt, 2007) permite usar varios LMs en la entrada de las reglas de transformación. En este lenguaje, el orden de los elementos en los patrones importa, mejorando así esta carencia de QVT. Sin embargo, este lenguaje apenas tiene soporte. Varró y Balogh (2007a) presentan el LTM del armazón *VIATRA2*, el cual está basado en reglas y patrones para la manipulación de modelos de grafos. Este lenguaje combina las transformaciones de grafos y las máquinas de estado en un único paradigma. Kalnins et al. (2006) proponen una herramienta que implementa el lenguaje *MOLA*. El objetivo de ese trabajo es mejorar la eficiencia del encaje de patrones. Esta eficiencia se consigue usando una base de datos. Cuadrado et al. (2006) presentan el lenguaje de transformación *RubyTL*, cuya particularidad es que es extensible. En concreto, un mecanismo de plugin (Cuadrado y Molina, 2006) permite extender las características núcleo con nuevas características. *XSLT* (Clark et al., 1999b) es un lenguaje de transformaciones originalmente creado para transformar documentos XML. Sin embargo, dado que muchos LMs escogen una representación XML para su serialización, XSLT se ha convertido en una solución general para definir TMs. En particular, XSLT permite definir TMs que otros LTMs no pueden, como las traducciones entre metamodelos, un ejemplo de las cuales es la ya mencionada traducción EMF-a-MOF (Gerber y Raymond, 2003). *Tefkat* (Lawley y Steel, 2006) es un LTM y un motor de transformaciones de modelo. El lenguaje tiene un paradigma declarativo, y el motor de transformaciones está implementado como un plugin de Eclipse. Como ejemplo, Steel y Lawley (2004) usan Tefkat en un desarrollo del software dirigido por baterías de pruebas.

El desarrollo del soporte de herramientas para los LTMs es una tarea especialmente costosa. Por ello, a algunos LTMs tales como VIATRA2, se les da soporte por medio de una traducción a otro lenguaje de transformación. Cuando se escoge un LTM, se recomienda elegir uno que tenga un soporte continuado. Por ejemplo, ATL tiene el soporte de la comunidad Eclipse que continuamente desarrolla nuevas versiones del plugin que le da soporte. ATL tiene la carencia de que sólo permite un elemento en los patrones de entrada de las reglas, por lo que los elementos adicionales de entrada han de ser incluidos mediante restricciones extra. Sin embargo, por ejemplo, el lenguaje *MT* (Tratt, 2007) no tiene esta carencia pero su soporte es mucho menor. En este tipo de casos, se recomienda escoger el lenguaje con mayor soporte, para evitar tener que redefinir todas las transformaciones porque se quede obsoleto el lenguaje. Sería mejor definir las transformaciones de una forma que no dependiera del lenguaje, por ejemplo con modelos prototipos para la entrada y salida de las reglas; y que

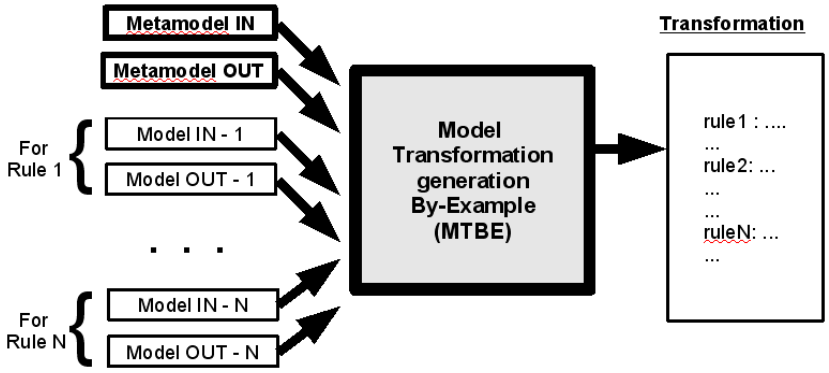


Figura 2.8: La entrada y la salida de la GTBE

luego pudieran ser generadas las TMs para diferentes LTMs. Sin embargo, esta línea de trabajo (véase el siguiente apartado) está poco avanzada, lo que es una de las motivaciones de esta tesis.

2.3.4 Generación de transformaciones basadas en ejemplos

El enfoque convencional de construcción de TMs requiere que el diseñador conozca la sintáis y la semántica de un LTM. Después habrá que desarrollar el "código" de la TM, en una forma muy similar a la programación tradicional, reduciendo así el nivel de abstracción del trabajo en el DSDM. Además, este desarrollo tiene el inconveniente de que no existen herramientas de desarrollo para la creación de TMs con una funcionalidad similar a la de sus equivalentes para lenguajes tradicionales. Por ejemplo, no hay herramientas de depuración para TMs. Para abordar estos obstáculos, surge la *Generación de Transformaciones Basadas en Ejemplos* (GTBE).

La GTBE se define como la generación de transformaciones a partir de parejas de modelos prototipo de origen y destino. Las transformaciones creadas deben, al menos, transformar los modelos de origen en los modelos destino de las parejas usadas en el proceso de generación. Es deseable que los modelos que tengan ciertas similitudes con los modelos prototipo de origen de las parejas se conviertan en modelos similares a los modelos prototipo de salida. Es también deseable que exista un mecanismo de asociación que describa como pasar la información de los modelos origen a los modelos destino. De esta forma, la generación automática de TMs a partir de ejemplos disminuye el esfuerzo en la definición de TMs.

Según Varro (2006), los pasos naturales en la GTBE son los siguientes:

1. *Definición de las parejas de modelos origen y destino.* El diseñador define prototipos de modelo que expresan tanto la entrada como la salida de las reglas de transformación.

2. *Generación automática de las reglas.* Una herramienta de GTBE (véase Figura 2.8) toma como entrada las parejas de modelos y sintetiza las reglas de salida. Las reglas generadas (véase Figura 2.9) deben, al menos,

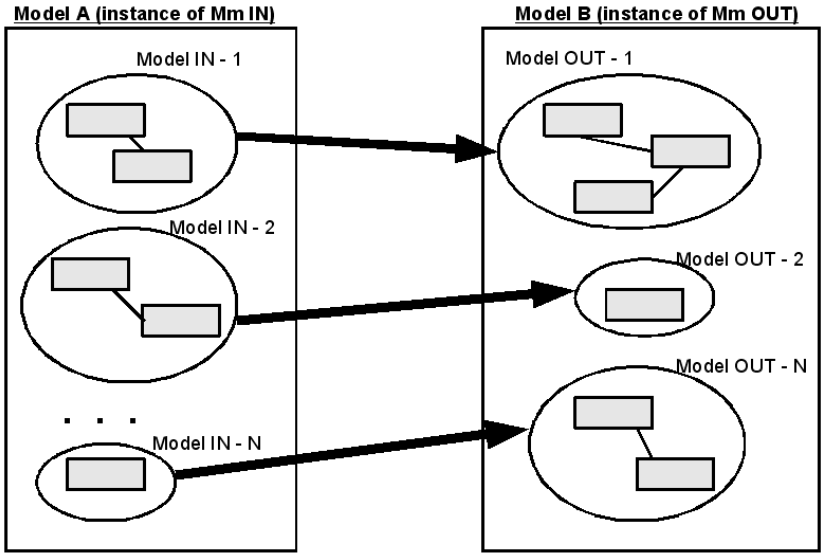


Figura 2.9: El comportamiento de la TM generada

transformar los modelos orígenes prototipo en sus equivalentes modelos destino. Se espera que otros modelos similares también se transformen con las reglas generadas.

3. *Refinamiento manual de las reglas.* El diseñador puede refinar las reglas generadas. Sin embargo, se recomienda que estos refinamientos se trasladen a los modelos prototipo, en la medida de lo posible, para que puedan volver a ser regeneradas las transformaciones sin que haya perdida de información en la sobrescritura.

4. *Ejecución automática de las reglas de transformación.* El diseñador prueba el comportamiento de la TM generada por medio de ciertos modelos de prueba. De esta manera, el diseñador puede comprobar si el comportamiento era el que esperaba.

Antes de que la GTBE se usara para producir TMs, la GTBE se usó con propósitos más generales. En concreto, Krishnamurthi et al. (2000) presentaron una *transformación basada en ejemplos* para XML. En ese trabajo se creó el sistema de transformación *XT3D* para trabajar con un lenguaje de transformaciones cuya sintaxis es muy cercana a la definición de modelos prototipo. Este trabajo fue uno de los principales precursores de lo que luego llegó a ser la GTBE para TMs.

Los trabajos clave en GTBE aplicada a TMs son: (Wimmer et al., 2007) y (Varro, 2006; Varró y Balogh, 2007b). La Tabla 2.6 indica las características más relevantes de estas dos líneas de trabajo.

El trabajo de Wimmer et al. (2007) permite la copia de los valores de atributos de entrada en los atributos de salida. Esta copia de atributos se caracteriza por la distinción de *atributos de ontología* y *atributos lingüísticos*. Los primeros tienen semántica para el usuario, mientras que los segundos complementan la definición de la sintaxis abstracta del LM. Por tanto, el usuario sólo describe

Características de la GTBE	Varró y Balogh (2007b)	Wimmer et al. (2007)
Asociación de Atributos	√	√
Propagación de Enlaces	√	√
Ejemplos Negativos	√	X
Generación de Restricciones	X	√
Emplazamiento explícito de los elementos destino	√	X
Análisis de Contexto	√	X
Mecanismo de Asociación de Atributos desde varios elementos de la entrada de cada regla	X	X
Reglas para transformar Grafos No Conexos	X	X

Tabla 2.6: Características de los principales trabajos en la GTBE para TMs.

qué atributos de ontología desea copiar. Hace transformaciones uno-a-uno, pero permite que un elemento, al ser transformado, conserve los enlaces con los correspondientes elementos transformados. Además, su trabajo tiene un mecanismo que genera ciertas restricciones básicas basadas en los atributos de los elementos de entrada. Este trabajo ilustra su técnica con el lenguaje ATL.

En la misma línea de GTBE para TMs, (Varro, 2006; Varró y Balogh, 2007b) usa programación de lógica inductiva para la generación de reglas. Las dos grandes innovaciones frente al resto de trabajos del área es el *análisis del contexto* y los *prototipos negativos*. Con el análisis del contexto, las reglas sólo se aplican si el modelo de entrada está situado en un contexto indicado por un diseñador. Con respecto a los prototipos negativos, el modelo origen se especifica con un prototipo positivo y varios negativos. Los prototipos negativos indican qué patrones no deben encajar con las reglas de transformación. Esto es útil para restringir la aplicación de ciertas reglas.

Como se puede ver en la Tabla 2.6, las técnicas y herramientas disponibles en GTBE no proporcionan un mecanismo de propagación de atributos desde varios elementos de la entrada de cada regla. Además, no permiten generar reglas que transformen grafos no conexos. En los SMAs, estos tipos de reglas son necesarias (como se verá en el apartado 2.3.6 y el capítulo 6) debido a que grupos de elementos tienen que transformarse en otros grupos de elementos, transfiriendo y combinando su información. Por tanto, las técnicas y herramientas disponibles en GTBE no satisfacen las necesidades básicas de los desarrollos de SMAs.

La GTBE para generar TMs es una técnica reciente. Si bien su uso no está implantado en todos los dominios de aplicación, ya son bastantes los autores que la usan. Por ejemplo, Strommer et al. (2007) aplican la GTBE sobre LMs de procesos de negocios. Además, Roser y Bauer (2006) aplican la GTBE entre ontologías. Uno de los objetivos de ese trabajo es generar las asociaciones de atributos automáticamente a partir de la semántica de los conceptos de la ontología. Por último, Milicev (2002) define TMs con modelos de extensiones de los diagramas de UML de objetos. En esta aplicación, los modelos orientados a objetos se benefician de GTBE.

Al igual que ciertos dominios de aplicación ya se están beneficiando de la GTBE, los SMAs también podría utilizarla. Por ejemplo, podría usarse para

producir las transformaciones horizontales en ADELFE, refinamientos de modelos INGENIAS, y TMs cualquiera en Tropos o Prometheus. Sin embargo, hasta este trabajo de tesis, ninguna de las metodologías de ISOA aplicaban esta técnica.

2.3.5 Problemática de la aplicación de transformaciones en un desarrollo

Algunos de los problemas en el uso de las TMs son los siguientes:

- *Las TMs no son siempre rentables.* Para saber si las TMs son rentables, hay que medir cuanto cuesta definirlas y entrenar al personal correspondiente en un desarrollo real. Medir el coste (Turner, 2003) es necesario para ver si realmente hay beneficio. Por ejemplo, Balsamo et al. (2004) describen el estado del arte de cómo predecir el rendimiento en el DSDM, en el cual las TMs juegan un papel importante. Para reducir los costes mencionados, esta tesis recomienda la GTBE.

- *Especificar las correspondencias sintácticas y semánticas entre lenguajes diferentes es complicado* (Murzek et al., 2006). Esto es necesario para la traducción de modelos entre LMs diferentes, lo cual es útil por ejemplo en el intercambio de modelos entre herramientas CASE. Las principales causas de la complejidad son que, primero, los lenguajes pueden no definir con suficiente detalle la semántica de sus conceptos y, segundo, la semántica de los conceptos de dos lenguajes se puede parecer y tener diferentes matices a la vez.

- *La transformación de LMs visuales implica el tratamiento de la información espacial.* Por ejemplo, las posiciones de nuevos elementos creados se pueden solapar con elementos existentes. El trabajo de Bardohl et al. (2004) intenta abordar este problema.

- *Dificultad en la interoperabilidad entre reglas* (Bézivin et al., 2003a). Las reglas no pueden llamar directamente a otras, sino que se deben comunicar a través de variables auxiliares o información de los elementos creados en TMs anteriores. Este problema se hace presente en las transformaciones modelo-a-modelo, modelo-a-texto y texto-a-texto. Esta tesis se centra en las transformaciones modelo-a-modelo.

- *Dificultad en la reutilización de transformaciones* (Bézivin et al., 2003a). Se pretende que las TMs se puedan reutilizar o, al menos, que algunas reglas de las TMs se puedan reutilizar. Esta reutilización es sólo posible cuando los LMs de entrada y salida de la TM, que se quiere obtener, son los mismos a los LMs de entrada y salida de la TM reutilizada.

Dos problemas de la aplicación de las TMs requieren una explicación más extensa. Se trata de los *múltiples encajes* y la *transitividad* en las reglas de transformación. Aclarar estos problemas es necesario para la correcta definición de TMs en este trabajo de tesis. Estos dos problemas surgen también en la GTBE, dado que según los modelos prototipo que se usen para la GTBE, las reglas generadas también pueden llegar a una situación de encaje múltiple o transitividad de reglas. Estos dos problemas se detallan en el siguiente subapartado.

Múltiples encajes y Transitividad en reglas de transformaciones

El *múltiple encaje* de patrones complica con frecuencia la definición de TMs. Una TM puede tener una o varias reglas. En cada lenguaje, el diseñador debe tener claro el comportamiento de su lenguaje, especialmente en el caso de que los patrones de entrada de varias reglas puedan encajar con los mismos elementos. Además, se debe ser consciente de si se puede producir el encaje transitivo. Como ejemplo, se describe el comportamiento para el lenguaje ATL y el plugin de Eclipse que le da soporte.

Para ilustrar el encaje múltiple, imagínese una transformación con las siguientes reglas.

1. A $\longrightarrow$ B
2. A $\longrightarrow$ C

En este ejemplo, se asume un modelo de entrada con varios elementos de tipo A. El diseñador debe ser consciente de qué reglas se aplican para los elementos de tipo A y dónde se ubican los elementos creados.

Para el lenguaje ATL y su soporte en Eclipse, se aplican por cada elemento A, las dos reglas. En el modelo destino, cada elemento que contiene el resultado de la transformación de A contiene sólo uno de los dos resultados, en concreto tiene el resultado de la última regla que se definió en el código de la TM. Los resultados de las reglas que no se colocan en el lugar de los elementos de entrada transformados se sitúan aparte del resto de elementos en el modelo destino. Es decir, todos los elementos de salida de las reglas (i.e. B y C) están en el modelo destino, pero sólo uno de ellos (i.e. C) está ubicado donde le corresponde a la transformación de A en el destino. El otro elemento (i.e. B) se sitúa al final del modelo destino, aparte del resto. Cabe mencionar que las dos anteriores reglas son equivalentes (i.e. se comportan igual) a la siguiente regla:

1. A $\longrightarrow$ B , C

En general, si se espera que se produzca un encaje múltiple (ya sea con varias reglas con la misma entrada o con reglas con varios elementos de salida), se deben usar variables globales para hacer que cada elemento esté contenido por el elemento que se desee. El emplazamiento por defecto (i.e. sin variables globales) de los elementos destino no se debe usar si se espera encaje múltiple.

El segundo punto a aclarar en los LTMs es si es posible la aplicación *transitiva* de las reglas de transformación. Para ilustrar este problema, asúmase una transformación con las siguientes reglas y un modelo de entrada con un elemento de tipo A.

1. A $\longrightarrow$ B
2. B $\longrightarrow$ C

Algunos diseñadores podrían pensar que se puede aplicar primero la primera regla y obtener un elemento de tipo B y que luego se aplique la segunda regla y tome como entrada el elemento recién creado de tipo B. Sin embargo, este comportamiento, que se llama *transitivo*, sólo tendría sentido en transformaciones endógenas, y no es habitual en los LTMs. La única excepción identificada hasta

el momento es el lenguaje Maude (Clavel et al., 2003), cuyos autores lo describen como un lenguaje de reescritura. Por tanto, el comportamiento habitual es que sólo se aplique la primera regla, y que la regla B $\longrightarrow$ C nunca se llega a aplicar. Aunque después de aplicar la primera regla haya elementos B, estos no encajan como entrada del resto de las reglas. Este comportamiento difiere de los motores de sistemas de reglas de producción como CLIPS.

Por el contrario, si las reglas se sitúan en diferentes TMs y se aplican secuencialmente, entonces el resultado de una TM puede ser transformado por otra TM. Es decir, si se quiere obtener el efecto transitivo que no se puede normalmente en los LTMs existentes, lo que se puede hacer es separar las reglas implicadas en TMs diferentes y aplicar dichas TMs en el orden deseado.

Dadas las diferentes maneras de encajar las reglas y TMs, es importante decidir si incluir todas las reglas de transformación en una TM, o si por el contrario, dividirlas en grupos en diferentes TMs. En caso de dividirlas en varias TMs, el orden en que se aplican las TMs es relevante. Incluso, en una misma TM, el orden en que se pongan las reglas puede variar el comportamiento, sobre todo si se usan variables para almacenar ciertos elementos.

En conclusión, este apartado ha expuesto los aspectos más relevantes del encaje de reglas de las TMs, que son necesarios para entender esta tesis y que se resumen a continuación:

- Si el encaje de reglas es múltiple en un LTM (i.e. un elemento encaja con varias reglas de la misma TM), entonces un elemento puede hacer que varias reglas diferentes se disparen.
- En la mayoría de LTMs (i.e. todos menos Maude) el encaje de reglas es no-transitivo; es decir, los elementos producidos por una regla no sirven como entrada de otra regla de la misma TM.

2.3.6 Transformaciones de modelos en sistemas multi-agente

Si bien el DSDM empieza a ser común en SMAs y el uso de transformaciones modelo-a-texto se aplica con éxito, el uso de transformaciones modelo-a-modelo en SMAs todavía es escaso, como se justifica más adelante, y quizás se deba a la falta de soporte técnico para los desarrolladores de SMAs. Algunos de los trabajos pioneros de transformaciones modelo-a-modelo en SMAs se discuten a continuación.

Las transformaciones modelo-a-modelo se han aplicado en la metodología Tropos (Bresciani et al., 2004). Tropos está organizado en cinco fases principales o disciplinas: *requisitos iniciales, requisitos detallados, diseño arquitectónico, diseño detallado* e *implementación*. El modelado en Tropos es concebido como un proceso incremental, en el cual un modelo inicial es refinado mediante la inserción de nuevos elementos resultantes del objetivo y planes de cada actor.

En la fase de requisitos iniciales, Bresciani et al. (2002) aplican TMs para refinar los requisitos iniciales, de tal forma que se asiste la transición entre las fases de requisitos iniciales y requisitos detallados. Estas TMs contienen reglas muchos-a-muchos que transfieren información desde varios elementos de la entrada.

En la fase de diseño arquitectónico de Tropos, Perini y Susi (2006) aplican las transformaciones modelo-a-modelo para la *síntesis* de elementos en el modelo de diseño detallado. El diseño detallado de Tropos se expresa con los diagramas

	Meta-modelo	Tipo de Aplicación de las Transformaciones de Modelo-a-modelo			
		horizontales	verticales	refactorización o asistencia	traducción
Tropos	√	X	√	√	X
ADELFE	√	X	√	X	X
PASSI y Agile PASSI	√	X	X	X	X
Prometheus	√	X	X	X	X
MaSE	√	X	X	X	X
INGENIAS	√	X	X	X	X
Vowels	X	X	X	X	X
MAS-CommonKADS	X	X	X	X	X
GAIA	X	X	X	X	X
AALAADIN	X	X	X	X	X

Tabla 2.7: Uso de transformaciones de modelo-a-modelo en metodologías de SMAs.

de actividad y los diagramas de secuencia de UML hechos con herramientas de modelado generales. Con ello se trata de aprovechar el metamodelo de UML 2.0 para el diseño. Los autores de dicho trabajo usan el formalismo propuesto por QVT para definir la TM Tropos-a-UML. Por ejemplo, se transforma un *plan* en los siguientes elementos: una *acción*, varios *nodos* y varios *flujos de trabajo*. Los elementos destino de las TMs pueden variar según ciertas restricciones. Estas TMs son expresadas y ejecutadas con el lenguaje y motor de transformaciones Tefkat[10]. El objetivo de usar las TMs en este trabajo es mantener la sincronización entre modelos, así como la trazabilidad entre sus elementos.

También Amor et al. (2005) aplican los principios de MDA a la metodología de Tropos. En concreto, los MIPs son expresados con el metamodelo Tropos y, a partir de estos, se crean los MEPs expresados con el armazón de *Malaca*. En cuanto a las TMs aplicadas sobre Tropos no se han encontrado detalles técnicos de cómo fueron definidas. Esta falta de detalles técnicos dificulta que otros expertos puedan tomar como ejemplo dichas TMs.

En esta misma línea de investigación, la metodología ADELFE (Bernon et al., 2005) usa transformaciones modelo-a-modelo para transformar MIPs en MEPs. En concreto, AMAS-ML es el lenguaje de alto nivel de abstracción, mientras que *µADL* (*micro-Architecture Description Language*) (Rougemaille et al., 2007) es el lenguaje específico de plataforma. El lenguaje µADL expresa aspectos operativos de bajo nivel de los agentes. El comportamiento cooperativo de los agentes se expresa en los modelos del lenguaje *AMAS-ML*. De esta manera, se mantienen separados ambos aspectos. Las TMs relacionadas con ADELFE se han definido con ATL.

A pesar de los ejemplos mencionados, la aplicación de TMs es limitada en ISOA. Para mostrar este hecho, se ha realizado un breve estudio sobre las metodologías de SMAs y las aplicaciones de transformaciones de modelo-a-modelo

[10]Tefkat es parte del proyecto Pegamento de DSTC en la Universidad de Queensland http://www.dstc.edu.au/Research/Projects/Pegamento/ tefkat/index.html

para cada metodología según la categorización del apartado 2.3.1. Como dato adicional, se considera para qué metodologías se han definido metamodelos.

Como resultado del estudio (véase Tabla 2.7), se observa que, para las metodologías de ISOA que no se ha definido un metamodelo, tampoco se han definido transformaciones modelo-a-modelo. Este es el caso de las metodologías: Vowels, MAS-CommonKADS, GAIA y AALAADIN. En la mayoría de las metodologías en las que se ha definido un metamodelo, tampoco se han definido transformaciones modelo-a-modelo. Ejemplos de este caso son: Prometheus, MaSE, PASSI y Agile PASSI. Las pocas metodologías de ISOA que usan transformaciones modelo-a-modelo sólo usan algunas de las aplicaciones posibles. Existen cuatro tipos de aplicaciones de transformación (véase el apartado 2.3.1): horizontales, verticales, refactorización y traducciones. Sin embargo, ADELFE sólo usa las transformaciones verticales; y Tropos sólo usa las transformaciones verticales y las transformaciones para la refactorización. Si bien Tropos usa TMs para la refactorización, no se ha detallado ningún proceso de desarrollo entero asistido por dichas TMs.

Como resumen de este estudio, se puede observar que de las diez metodologías de SMAs consideradas, sólo dos de ellas usan las transformaciones modelo-a-modelo y tan sólo las usan para una pequeña parte de las aplicaciones posibles.

2.4 Conclusiones

Este capítulo ha descrito del estado actual en el DSDM en SMAs. Principalmente se ha centrado en dos grandes bloques: la definición y uso de metamodelos; y la definición y aplicaciones de las TMs. Además, el capítulo indica la notación para la definición de metamodelos que es usada a lo largo de este documento. Las conclusiones de este análisis del estado del arte de la definición de metamodelos se mencionan a continuación:

- *Los metamodelos existentes son definidos sin seguir un proceso riguroso.* Incluso metamodelos con las mismas características estructurales, se definen de forma diferente en la literatura. Los mayores puntos de divergencia son la forma de definir las *relaciones* y la manera de agrupar los elementos en *paquetes*. Esta diversidad en la definición sugiere la necesidad de tener guías para que los diseñadores puedan razonar explícitamente sobre la definición. Cabe destacar que, actualmente, la mayoría de los LMs existentes son los *basados-en-conexiones*. Luego parece natural que las primeras guías se tengan que centrar en este tipo de lenguajes.

- *Los metamodelos se están incorporando en la ISOA.* Se han definido metamodelos para varios LMs de SMAs, tales como: Tropos (Susi et al., 2005), PASSI y Agile PASSI (Chella et al., 2006), Prometheus (Padgham y Winikoff, 2002), MaSE (DeLoach et al., 2001; DeLoach, 2005), e INGENIAS (Pavón y Gómez-Sanz, 2003). Las características estructurales de estos metamodelos varían, pero se pueden observar ciertas características estructurales recurrentes, tales como: relaciones n-arias, relaciones con atributos en los cuerpos y los extremos, extremos de las relaciones restringidos a ciertos tipos de entidades, y jerarquías de entidades y relaciones.

Con respecto a las TMs, las conclusiones del estado del arte se citan a continuación:

- *Las técnicas y herramientas de GTBE no poseen soporte para la mayoría de reglas muchos-a-muchos.* La GTBE podría facilitar la definición de TMs en la ISOA, dado que las TMs son generadas a partir de ejemplos en los LMs origen y destino, por lo que los diseñadores no necesitan conocer ningún LTM. Además, la GTBE reduce la necesidad de los diseñadores de un conocimiento extenso de los metamodelos. Sin embargo, las técnicas y herramientas de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b) existentes no satisfacen las necesidades básicas del DSDM de SMAs en lo referente a las reglas muchos-a-muchos: no se permite la asociación de atributos desde varios elementos de una entrada de una regla, y las reglas generadas no pueden tratar grafos no conexos.

- *El uso de las TMs es todavía escaso en la ISOA.* Las principales aplicaciones de las TMs en el DSDM son: transformaciones verticales, transformaciones horizontales, refactorización o asistencia en el diseño de modelos, y traducción entre LMs similares. Sin embargo, las TMs identificadas en SMAs son: transformaciones verticales para aplicar los principios de MDA respectivamente a Tropos (Amor et al., 2005) y ADELFE (Bernon et al., 2005); y transformaciones para la asistencia del diseño de modelos en Tropos (Bresciani et al., 2004). Por tanto, la existencia de transformaciones modelo-a-modelo todavía es escasa en la ISOA.

Capítulo 3

Guía para la definición de metamodelos

Este capítulo presenta una guía para la definición de metamodelos. Los metamodelos se estructuran en tres capas: la capa núcleo, que incluye la sintaxis abstracta y concreta del lenguaje de modelado; la capa de información visual, que considera la división en vistas del modelo global y la distribución espacial en los diagramas; y la capa de información específica de las herramientas CASE. Además, el capítulo presenta una guía más detallada para la parte de la sintaxis abstracta de los lenguajes de modelado, incluyendo varios tipos de representaciones, recomendaciones para las decisiones importantes, y una serie de actividades.

3.1 Introducción

En el DSDM (Selic, 2003), los modelos son el producto principal. Los LMs de dichos modelos se definen con metamodelos, y su uso está soportado por herramientas CASE (Sommerville, 2004) de modelado, tales como *Eclipse con EMF, BoUML, StarUML, Prometheus Design Tool, ADELFE Toolkit* e *IBM-RSD*.

Las representaciones de metamodelos son muy diversas (véase el apartado 2.2.3) y dada la carencia de decisiones explícitas acerca de su diseño en la literatura, no hay procesos consensuados para su definición. Por este motivo, esta tesis propone una guía organizada alrededor de una serie de decisiones necesarias para crear metamodelos que permitan definir tanto los LMs como todas sus características asociadas.

Parte de la hipótesis de este trabajo es que se puede incluir en estos metamodelos información de las herramientas CASE de modelado para facilitar su integración. Para ello se incluye en un metamodelo la información específica de las herramientas CASE de modelado anteriormente citadas, tal como: el espacio de trabajo, estructura del proyecto, o versión de la herramienta (véase una lista más detallada en el apartado 2.2.8). Si bien existen trabajos (Ledeczi et al., 2001a) sobre el uso de metamodelos para modelar la información requerida por las herramientas CASE en general, todavía no existe una solución consensuada.

Los LMs no tienen por objetivo el capturar información específica de la herramienta CASE de modelado que los implementa, pero la realidad es que esto supone un obstáculo a la hora de colaborar en desarrollos donde el coste de la licencia y otras restricciones hacen que cada individuo elija una solución diferente. Una misma especificación, al ser cargada y modificada por cada herramienta concreta, encuentra problemas por dos motivos: la información específica de las herramientas CASE que no es modelada, y las representaciones diferentes del mismo LM.

La guía presentada en este trabajo permite definir metamodelos tanto para los LMs aislados como para los LMs con la información específica de las herramientas CASE de modelado. Sin embargo, en cualquier caso, la información específica de las herramientas se mantiene aislada del resto, para permitir reutilizar la información del LM. En concreto, la guía propone un armazón con cinco metamodelos, que se pueden combinar según los requisitos del LM y/o herramienta. Estos metamodelos son los siguientes:

- Metamodelo *Objetos*. Define la sintaxis abstracta del LM. La sintaxis abstracta define los conceptos de un LM, definiendo así su vocabulario y taxonomía, sin considerar su representación visual. Por tanto, este metamodelo identifica los conceptos de modelado (e.g. entidades y relaciones) sin tener en cuenta su visualización.

- Metamodelo *Concreto*. Este metamodelo describe la sintaxis concreta del LM. En general, la sintaxis concreta de un LM indica su representación visual. Este metamodelo asocia una visualización gráfica (e.g. icono) a cada concepto de modelado.

- Metamodelo *Vistas*. Contiene diferentes agrupaciones de elementos del metamodelo *Objetos*. Esta selección de elementos se basa en la idea de *vistas* de un modelo. Un modelo, que es instancia del metamodelo *Objetos*, es considerado como un *todo* que no es descompuesto en partes ni diagramas. Con el objetivo de facilitar el manejo de la información y su comprensión, los diseñadores definen vistas lógicas que consideran parte y no todo el modelo. Estas vistas son representadas con modelos que son instancias del metamodelo *Vistas*. Estos modelos son *grafos*, que están compuestos de nodos conectados a través de aristas, porque tienen un formato especialmente apropiado para los LMs *basados en conexiones*. Como se justificó en el apartado 2.2.3, los LMs *basados en conexiones* son la mayoría de los LMs actuales.

- Metamodelo *Layouts*. El objetivo de este metamodelo es la representación espacial de los elementos del lenguaje. Incluye posiciones y dimensiones de los elementos de las vistas proporcionadas por el metamodelo anterior.

- Metamodelo *Proyecto*. Considera el concepto del proyecto de la herramienta, que es específico de cada herramienta CASE de modelado.

Los metamodelos *Objetos, Concreto*, y *Layouts* son similares a otros que pueden ser encontrados en la literatura (Fondement y Baar, 2005; OMG, Abril 2006) sobre meta-modelado, aunque no se encuentren todos juntos en una aproximación. En concreto, Fondement y Baar (2005) separa la sintaxis abstracta de la sintaxis concreta, en un modo similar a la separación de información entre los

metamodelos *Objetos* y *Concreto*. La especificación de *UML Diagram Interchange* (UML-DI) (OMG, Abril 2006) define la distribución visual de los elementos de los modelos UML, similar a la información contenida en el metamodelo *Layouts*. Sin embargo, los metamodelos *Vistas* y *Proyecto* son específicos de este trabajo. El metamodelo *Vistas* está basado en la idea de *vistas*, explicada anteriormente y que es diferente a la tendencia actual en el meta-modelado usando UML-DI (OMG, Abril 2006). En concreto, UML-DI usa el concepto de *recortes*, por el cual un diagrama muestra un recorte de otro diagrama, y contiene un atributo de *visibilidad* que permite ocultar elementos en un diagrama. Las diferencias entre UML-DI y nuestra aproximación se discuten con más detalle en el apéndice A. El metamodelo *Proyecto* contiene información específica para la herramienta CASE de modelado, como la presentada en el apartado 2.2.8. Las otras aproximaciones no consideran el meta-modelado de esta información, haciendo que cada herramienta almacene esta información de forma ad-hoc.

Si bien la representación de la información considerada en estos metamodelos depende del lenguaje de meta-modelado específico, la estructura del armazón se puede definir con varios lenguajes de meta-modelado. Entre las alternativas tales como ECore (Budinsky, 2003), MOF (Meta Object Facility) (OMG, 2006b) y GOPRR (Graph, Object, Property, Relationship, and Role) (Kelly, 1997), este trabajo usa ECore para ilustrar esta aproximación dado que tiene un amplio soporte. Sin embargo, se proporciona también una breve discusión con MOF (OMG, 2006b) dada su relevancia como estándar promocionado por el consorcio OMG.

La representación de la sintaxis abstracta de los LMs es muy variada (véase el apartado 2.2.3) y hay una carencia tanto de guías como de decisiones de diseño explícitas para la sintaxis abstracta. Por este motivo, este trabajo proporciona dentro de la guía general una guía específica para representar el metamodelo *Objetos*, que es el que describe la sintaxis abstracta de los LMs. La guía trata las entidades y las relaciones de los LMs basados en conexiones, que son casi la totalidad de los LMs actuales como se justificó en el apartado 2.2.3. La guía proporciona una representación para las entidades y varias representaciones posibles para las relaciones. Las representaciones de las relaciones se deciden atendiendo a ciertos requisitos estructurales de los LMs, tales como la existencia de relaciones n-arias o atributos en las relaciones, y a otros requisitos no-estructurales, tales como la eficiencia de procesamiento de los modelos o la navegabilidad en los mismos. En función de los requisitos estructurales del LM, puede demostrarse que existe un número mínimo de elementos de meta-modelado para representar cada entidad y relación, si se usa ECore como lenguaje de meta-modelado. Esta guía para la definición de la sintaxis abstracta sigue los siguientes pasos:

- *Determinar las características estructurales del LM.* Conocer dichas características es relevante para las decisiones de diseño del metamodelo y la definición del mismo.

- *Elegir entre representación homogénea y heterogénea.* El diseñador debe elegir si usar para todas las relaciones la misma representación, o usar la representación más apropiada para cada relación en particular. Esta decisión se debe tomar llegando a un compromiso entre diversos factores tales como: la minimización del número de elementos, la usabilidad de las herramientas generadas y la facilidad de procesamiento.

- *Elegir entre una representación no-redundante o redundante.* La representación *no-redundante* usa referencias unidireccionales para conectar los elementos de meta-modelado, mientras que la representación *redundante* usa parejas de referencias en direcciones opuestas para conectar los elementos de meta-modelado. El diseñador tomará la decisión de usar una u otra representación dependiendo del grado de *navegabilidad* que se desee en el metamodelo y en sus instancias. La navegabilidad determina la facilidad con la que se puede acceder desde unos elementos a otros a través de referencias. Se dice, por tanto, que los modelos tienen su máxima navegabilidad si se puede acceder a todos los elementos desde todos los elementos del modelo.

- *Definir el metamodelo.* Se define el metamodelo de acuerdo con las decisiones previamente tomadas, con una serie de actividades presentadas en un diagrama de actividades de UML.

Los apartados restantes en este capítulo se organizan como sigue. En primer lugar, en el apartado 3.2, se describe un armazón con cinco metamodelos. En el apartado 3.2.1, se define la estructura del armazón y se dan ciertas pautas para representar cada uno de sus metamodelos. El apartado 3.2.2 indica cómo se representa el armazón con MOF, un lenguaje diferente a ECore que es el usado normalmente para la explicación en este trabajo. El apartado 3.2.3 indica las aplicaciones de ciertos subconjuntos de los metamodelos del armazón, que se denominan *configuraciones* del armazón. Por otra lado, el apartado 3.3 proporciona una guía detallada para definir el metamodelo *Objetos*, que contiene la sintaxis abstracta del LM. Esta guía a su vez propone una estructura en paquetes, indicada en el apartado 3.3.1, y un catálogo de diversos tipos de representaciones de las entidades y relaciones del LM, descrito en el apartado 3.3.2. El apartado 3.3.3 prueba formalmente que el número de elementos usados es mínimo para algunos tipos de representaciones de entidades y relaciones con el lenguaje ECore, y el apartado 3.3.4 resume los pasos de la guía de definición de metamodelos para la sintaxis abstracta. Finalmente, el apartado 3.4 indica las conclusiones de este capítulo.

3.2 Armazón de metamodelos

Este trabajo permite definir tanto los LMs como los lenguajes de las herramientas CASE de modelado usando un armazón de metamodelos. En caso de definir el lenguaje de una herramienta CASE de modelado, se define una instancia del armazón para la herramienta que incluye información tanto del LM correspondiente como de la propia herramienta.

3.2.1 Estructura del armazón

El armazón está compuesto de cinco metamodelos. La Tabla 3.1 resume estos metamodelos y la información considerada en cada uno de ellos. La Figura 3.1 muestra la organización global de estos metamodelos y sus relaciones de dependencia (representadas con flechas). El objetivo de esta estructura es obtener una alta modularidad en el armazón, donde los elementos de un mismo metamodelo tienen una fuerte cohesión (i.e. muchas referencias entre sí) y entre diferentes

Metamodelo	Información
Objetos	Sintaxis abstracta del LM. La estructura interna de este metamodelo depende del LM.
Concreto	Sintaxis concreta del LM. Este metamodelo define la notación visual de los elementos de la sintaxis abstracta.
Vistas	Los diagramas representados como grafos. Este metamodelo es genérico para varios LMs y herramientas CASE de modelado.
Layout	Las posiciones y dimensiones de los elementos de los grafos. Este metamodelo es genérico para varios LMs y herramientas CASE de modelado.
Proyecto	Información acerca del proyecto de modelado. Este metamodelo depende de la herramienta CASE de modelado.

Tabla 3.1: Metamodelos en el armazón y su información.

metamodelos del armazón hay un bajo acoplamiento (i.e. pocas referencias entre distintos metamodelos). La Figura 3.1 también muestra la organización lógica de estos metamodelos en las siguientes capas:

- *Núcleo.* Contiene la definición del LM mediante su sintaxis abstracta (con el metamodelo *Objetos*) y su sintaxis concreta (con el metamodelo *Concreto*).

- *Información Visual.* Esta capa proporciona un mecanismo para estructurar los modelos en diferentes grupos de elementos llamados *vistas*. Las vistas son similares a la organización de los modelos en diagramas, aunque es una estructura lógica. Las vistas se representan como grafos, que están compuestos de nodos conectados a través de aristas. El metamodelo *Vistas* define estos grafos y el metamodelo *Layouts* proporciona la distribución de los elementos de los grafos en el espacio. Estos metamodelos son genéricos para varios LMs y varias herramientas.

- *Información de Herramientas CASE de modelado.* Describe la información requerida exclusivamente por la herramienta CASE de modelado. El metamodelo *Proyecto* contiene toda esta información.

Para un LM y una herramienta CASE de modelado, estos metamodelos son instanciados en varios modelos, con excepción del metamodelo *Concreto* que no necesita instanciarse. El metamodelo *Concreto* define la notación visual del lenguaje con valores constantes de los atributos fijados en M2, que no se pueden cambiar en el nivel M1. Por tanto, este metamodelo es útil para indicar la sintaxis concreta a las herramientas CASE de modelado; pero no es útil para almacenar información en los modelos de M1.

Los siguientes sub-apartados explican los diferentes metamodelos de la Tabla 3.1 con mayor detalle. El lenguaje para esta discusión es ECore y los ejemplos han sido tomados del proyecto de modernización de la herramienta IDK. Una versión *Beta* de la herramienta IDK modernizada con el armazón presentado está disponible en la web de Grasia (Grasia, 2009c).

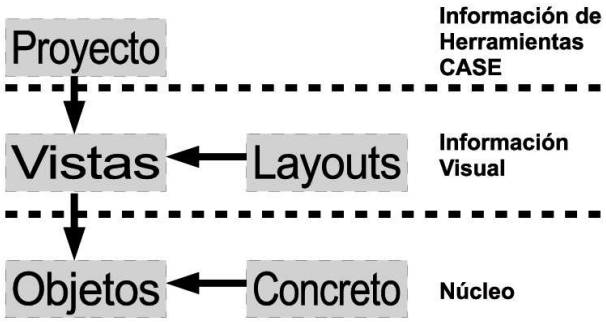


Figura 3.1: Estructura del Armazón.

Metamodelo *Objetos*

El metamodelo *Objetos* define la sintaxis abstracta (Fondement y Baar, 2005) del LM, que es su vocabulario y taxonomía. Debido a la gran variedad de representaciones de metamodelos para la sintaxis abstracta (véase el apartado 2.2.3), es necesario establecer una guía que permita definir el metamodelo según las características del LM y de otros requisitos. Dado que la práctica totalidad de los LMs actuales están basados en conexiones (véase el apartado 2.2.3), este trabajo incluye una guía con una aproximación *Entidad-Relación* (ER). En esta aproximación ER, los elementos del LM se clasifican en entidades y relaciones. A las entidades se les asigna un tipo de representación, mientras que las relaciones pueden ser representadas con diferentes tipos de representación dependiendo de sus características estructurales. La guía toma como entrada la descripción de un LM. Considerando sus características estructurales y otros requisitos, tales como la navegabilidad o la eficiencia de procesamiento, la guía determina unos pasos para definir el metamodelo. Esta guía se presenta con detalle en el apartado 3.3.

Metamodelo *Concreto*

El metamodelo *Concreto* define la sintaxis concreta del LM. La sintaxis concreta (Fondement y Baar, 2005) proporciona la notación visual de los elementos de la sintaxis abstracta. Por tanto, en este armazón, el metamodelo *Concreto* proporciona la notación de los elementos del metamodelo *Objetos*. El metamodelo *Concreto* contiene dos paquetes, uno para la sintaxis concreta de las entidades y otro para la sintaxis concreta de las relaciones. Estos paquetes se llaman *cEntities* y *cRelations* respectivamente. Para cada entidad y relación llamada <*nombre*>, se define una EClass de M2 llamada *C*<*nombre*> que extiende la EClass de M2 <*nombre*>. Por tanto, los elementos de la sintaxis concreta extienden los elementos de la sintaxis abstracta. Además de los atributos heredados, los elementos de la sintaxis concreta tienen un atributo que es *no-modificable* (atributo *changeable* con valor *false*) en M1. Cada atributo de este tipo contiene una ruta de una imagen, que es la notación visual de cada concepto del LM. Es importante saber que la ruta de la imagen se establece en M2. Para distinguir fácilmente estos atributos de otros, se recomienda que su nombre sea *p*<*nombre*>, siendo <*nombre*> el nombre del elemento de la sintaxis abstracta.

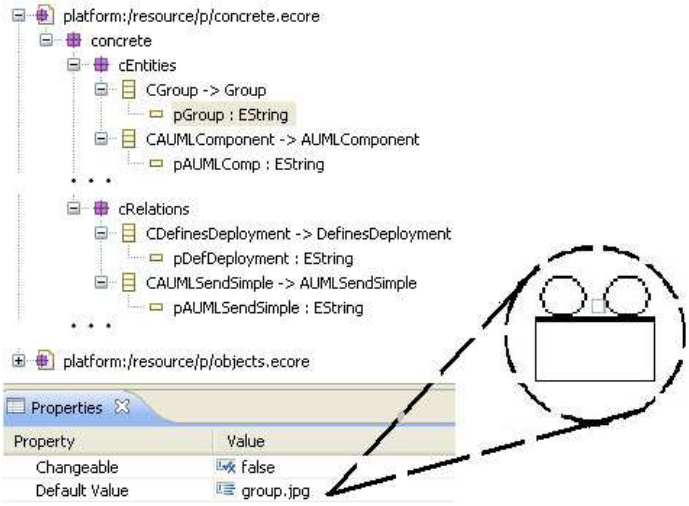


Figura 3.2: Metamodelo *Concreto* para el lenguaje de INGENIAS.

La Figura 3.2 muestra un ejemplo de metamodelo *Concreto* para el lenguaje INGENIAS siguiendo las anteriores indicaciones. Contiene una entidad llamada *CGroup* con su atributo *pGroup*, cuyo valor es una ruta de la imagen que se puede ver dentro de un círculo discontinuo en la misma Figura 3.2.

Se debe hacer una aclaración final sobre el metamodelo *Concreto*. La sintaxis concreta es necesaria en la definición de un LM visual. En concreto, las herramientas CASE de modelado necesitan la sintaxis concreta para mostrar correctamente los modelos. Sin embargo, todos los atributos en el metamodelo *Concreto* son no-modificables. En M1, las instancias de este metamodelo no pueden definir información nueva o modificarla porque la notación visual se define con valores constantes de atributos de M2. Por este motivo, las instancias del armazón propuesto no incluyen instancias en M1 específicas del metamodelo *Concreto*.

Metamodelo *Vistas*

Los modelos de sistemas complejos contienen un número grande de entidades y relaciones. Entender tales modelos es una tarea complicada para los diseñadores. Una manera de simplificar esta tarea es permitir una exploración de partes del modelo que contienen entidades y relaciones lógicamente agrupadas. Estas partes definen vistas de un modelo centrado en ciertos aspectos de interés para el desarrollador. Este tipo de aproximación es habitual en herramientas CASE de modelado, tales como *BoUML*[1] y *StarUML*[2], y en herramientas CASE de modelado específicas de SMAs, tales como PDT (Padgham et al., 2005), *ADELFE Toolkit* [3] e IBM-RSD para *PASSI* (Henderson-Sellers y Giorgini, 2005) (véase el apartado 2.2.8), donde un modelo es dividido en partes más pequeñas.

[1] http://bouml.free.fr/
[2] http://staruml.sourceforge.net/en/
[3] http://www.irit.fr/ADELFE/Download.html

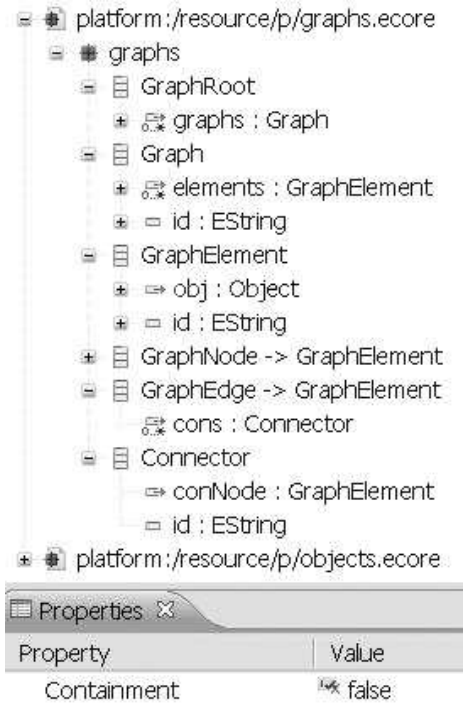


Figura 3.3: Metamodelo *Vistas* sin restricciones en los diagramas.

Sin embargo, estas herramientas reflejan esta división lógica del modelo en su estructura también.

Este armazón considera la idea de *vistas* de un modelo, mostrando partes del modelo sin dividirlo. El modelo del sistema es considerado como un *todo* que no se divide, dado que todos los elementos tienen conexiones entre ellos. Sin embargo, las vistas pueden ser definidas para el análisis del modelo. Una vista es un subconjunto de entidades del modelo total y las relaciones entre ellas. Por tanto, los grafos son una representación natural de las vistas al tener nodos conectados a través de aristas de una forma similar a los LMs *basados en conexiones*, y el metamodelo *Vistas* representa estos grafos. El armazón considera dos representaciones alternativas para este metamodelo. Ejemplos de éstas aparecen en las Figuras 3.3 y 3.4.

La primera alternativa para el metamodelo *Vistas* (véase la Figura 3.3) tiene una raíz (EClass de M2 *GraphRoot*) que contiene un conjunto de grafos (EClass de M2 *Graph*), donde cada uno de ellos incluye elementos de grafos (EClass de M2 *GraphElement*). Un elemento del grafo contiene una *EReference non-containment* de M2, llamada *obj*, a la clase *Object* del metamodelo *Objetos* (véase apartado 3.2.1). Con esta referencia, cada elemento de M1 del grafo apunta a un objeto (i.e. entidad o relación) del modelo *Objetos*. El hecho de que la EReference de M2 *obj* es *non-containment* permite que haya un único elemento en M1 en el modelo *Objetos* aunque varios elementos en M1 del modelo *Vistas* lo apunten. Esta estructura precisamente consigue la noción de vista introducida anteriormente. Los elementos del grafo se diferencian en nodos (EClass de M1 *GraphNode*) y aristas (EClass de M1 *GraphEdge*). Las aristas están conec-

tadas con otros elementos del grafo a través de los conectores (EClass de M1 *Connector*) que sólo pueden enlazar elementos del mismo grafo (i.e. diagrama). Esta alternativa tiene la limitación de que no puede definir diferentes tipos de diagramas (i.e. grafos).

La Figura 3.4 ilustra la segunda alternativa para la estructura del metamodelo *Vistas*. Ésta permite definir varios tipos de diagramas (i.e. vistas del modelo) que sólo se asocian con tipos específicos de objetos (i.e. entidades y relaciones). Esta alternativa extiende los elementos *Graph* y *GraphElement* definidos en la primera alternativa para cada diagrama del metamodelo. El armazón recomienda usar la siguiente notación:

- G<*nombre-diagrama*> para extender la EClass de M2 Graph (*GAgentDiagram* y *GTaskAndGoalsDiagram* en el ejemplo de la Figura 3.4). G<*nombre-diagrama*> contiene una referencia múltiple a GE<*nombre-diagrama*>.
- GE<*nombre-diagrama*> para extender la EClass de M2 GraphElement (*GEAgentDiagram* y *GETaskAndGoalsDiagram* en el ejemplo de la Figura 3.4). Esta EClass de M2 define los elementos de grafo de un diagrama específico. En la capa M2, cada elemento de grafo contiene referencias a los tipos de entidad y relaciones permitidos en cada vista. Sin embargo, en la capa M1, cada elemento de grafo apunta exactamente a un elemento del modelo Objetos, que es una instancia del metamodelo Objetos. La razón es que cada elemento del grafo de M1 es la vista de una sola entidad o relación del modelo total.

Para facilitar esta representación, el elemento *GraphElement* tiene un atributo adicional llamado *geType*. Este atributo toma su valor de una enumeración que contiene los valores *GraphNode* y *GraphEdge*, que denotan respectivamente los nodos y aristas. De esta manera, las EClass de M2 GraphNode y GraphEdge no se necesitan para cada tipo de diagrama, porque el atributo *geType* distingue entre nodos y aristas.

Como ejemplo, el metamodelo *Vistas* del lenguaje INGENIAS define nueve tipos diferentes de diagramas con una media de unos veinte tipos de elementos en cada uno. Un tipo de diagrama de este ejemplo es el *Agent Diagram* (véase una pequeña parte del metamodelo *Vistas* de INGENIAS en la Figura 3.4). Este tipo de diagrama puede contener varios tipos de entidades, como *agent* y *role*, entre otras muchas. El diagrama puede contener varios tipos de relaciones como *WFPlays* y *WFPursue*, entre otras muchas. Por tanto, se definen las EClasses de M2 *GAgentDiagram* y *GEAgentDiagram*. La EClass de M2 *GEAgentDiagram* tiene referencias a las mencionadas entidades y relaciones, tales como *geadAgent* y *geadRole*. Los ocho tipos restantes de diagrama son especificados de la misma manera.

Como este ejemplo ilustra, la ventaja de la segunda alternativa es que permite restringir los tipos de entidades y relaciones que aparecen en cada tipo de vista (i.e. diagrama). Sin embargo, la primera alternativa usa menos elementos de meta-modelado en su definición que la segunda alternativa, dado que la primera alternativa no usa elementos para especificar cada tipo de diagrama.

Metamodelo *Layouts*

El metamodelo *Layouts* proporciona la información espacial para los elementos del metamodelo *Vistas*. El metamodelo *Layouts* tiene la EClass de M2 *La-*

Figura 3.4: Una pequeña parte de un metamodelo *Vistas* que incluye restricciones.

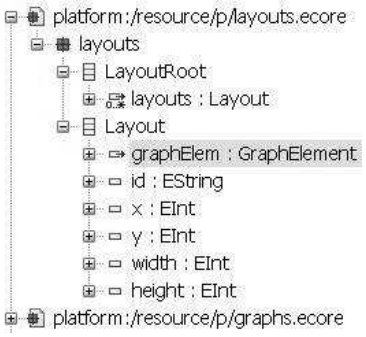


Figura 3.5: Metamodelo *Layouts*.

youtRoot como su elemento raíz, que contiene todos los elementos *Layout*. Un elemento *Layout* representa la descripción espacial de un nodo de un grafo y contiene atributos para indicar la posición (e.g. atributos x e y) y sus dimensiones con, por ejemplo, atributos para anchura y altura (i.e. atributos *width* y *height*). Cada elemento *Layout* del metamodelo *Layouts* está enlazado con un elemento del metamodelo *Vistas* a través de una EReference *non-containment* de M2 (llamada *graphElem*). De esta forma, la información espacial y las agrupaciones lógicas en vistas se mantienen separadas en diferentes metamodelos. Por tanto, se puede cambiar la distribución espacial (en los modelos *Layouts*) sin alterar las agrupaciones lógicas (en los modelos *Vistas*).

El metamodelo Layout es especialmente útil para LMs *basados en geometría* (Costagliola et al., 2002) (véase el apartado 2.2.3), que necesita describir la disposición espacial de los elementos en el espacio cartesiano. El ejemplo concreto en la Figura 3.5 (con atributos para la posición, anchura y altura) es apropiado para los lenguajes de *Caja*, que son una sub-categoría de los lenguajes basados en conexión. Sin embargo, los atributos de la EClass de M2 *Layout* pueden ser cambiados para adaptar este metamodelo a otros tipos de lenguajes basados en geometría, cómo los lenguajes *Icónicos*, que tratan con puntos del espacio en el eje cartesiano, llamados iconos, los cuáles se pueden solapar o estar próximos unos de otros. Además, se pueden buscar otras representaciones para distribuciones espaciales diferentes, como cualquier tipo de polígonos. Este metamodelo se puede adaptar también para representar más dimensiones en el espacio.

Metamodelo *Proyecto*

Las herramientas CASE de modelado necesitan almacenar información del proyecto que no pertenece a los LMs (véase el apartado 2.2.8). Esta información puede ser, por ejemplo, el nombre del proyecto, la jerarquía de diagramas, la versión del proyecto o las preferencias de usuario. Normalmente, cada herramienta CASE de modelado elige un formato propietario para esta información sin definir un metamodelo formal. Esta realidad hace difícil la importación de esta información desde otra herramienta CASE de modelado. El metamodelo para herramientas CASE de modelado de este trabajo incluye el metamodelo *Proyecto* para definir esta información. De esta manera, el armazón proporciona facilidades para importar o compartir la información de proyecto entre

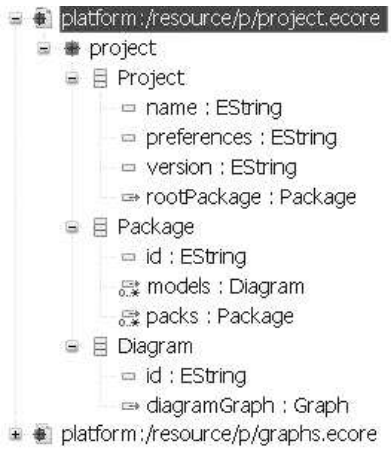


Figura 3.6: Metamodelo *Proyecto*.

herramientas CASE de modelado, y mantiene esta información separada de la información directamente relacionada con el LM.

La información específica de las herramientas CASE de modelado es muy variada e independiente del LM. El apartado 2.2.8 proporciona un estudio de la misma. Todos los tipos de información que se mencionan en dicho apartado se pueden incluir en el metamodelo *Proyecto*.

En general, se recomienda definir una EClass para almacenar los atributos relevantes y que además incluya una referencia a la estructura de paquetes del proyecto. También se pueden usar más EClasses y EReferences en caso de representar algún tipo de información que lo requiera, tal como el historial de cambios.

Como ejemplo, la Figura 3.6 muestra una simplificación del metamodelo *Proyecto* del IDK. El elemento raíz del metamodelo *Proyecto* es la EClass de M2 *Project*. Este elemento contiene atributos con la información sobre el proyecto, como el nombre del proyecto, las preferencias de usuario, y la versión del proyecto. La herramienta IDK también necesita estructurar los diagramas de la capa M1 en paquetes. Estos paquetes para la herramienta IDK son diferentes de los EPackages del lenguaje ECore, dado que los primeros pertenecen a la capa M1 y los EPackages pertenecen a M2. Para representar los paquetes de la herramienta, la EClass de M2 *Project* contiene una EReference de M2 llamada *rootPackage* que apunta a la EClass de M2 *Package*, que representa los paquetes. Los paquetes contienen recursivamente a otros paquetes, y contienen diagramas (EClass de M2 Diagram) que apuntan a los vistas del metamodelo *Vistas*. Por tanto, los proyectos tienen estructuras de tipo arbóreo usando el concepto de *paquete*. Cada diagrama tiene un diagrama y apunta a un grafo del metamodelo *Vistas*.

3.2.2 Descripción del armazón con Meta-Object Facility

El apartado 3.2.1 presentó el metamodelo del armazón con el lenguaje ECore (Budinsky, 2003). Sin embargo, otros lenguajes de meta-modelado pueden usarse para especificar el armazón. Este apartado introduce brevemente unas pautas

ECore	**MOF**
EPackage	*Package*
EClass	*Class*
EAttribute	*Property*
Containment EReference	*Association* con la propiedad *isComposite* a cierto
Non-containment EReference	*Association* con la propiedad *isComposite* a falso
EEnum	*Enumeration*

Tabla 3.2: Asociación de los elementos de ECore con los elementos de MOF para la descripción del armazón.

para representar este metamodelo usando el lenguaje MOF (OMG, 2006b). La Tabla 3.2 resume las correspondencias asumidas entre los conceptos de ECore y MOF que permiten traducir los metamodelos previos a MOF. Como norma, cada metamodelo del armazón es situado en un archivo diferente para ambos lenguajes de meta-modelado.

3.2.3 Configuraciones posibles

La introducción a la estructura del armazón de meta-modelado para herramientas CASE de modelado en el apartado 3.2.1 menciona que algunos de los metamodelos del armazón referencian a elementos de otros metamodelos del armazón en su definición. Las flechas en la Figura 3.1 indican cuando un metamodelo tiene una referencia a otro metamodelo. Siempre y cuando un conjunto de metamodelos del armazón no referencia a un metamodelo fuera del conjunto, el conjunto tiene sentido por sí mismo. Estos grupos válidos de metamodelos se llaman *configuraciones* del armazón, que son variantes del armazón general. Cada configuración es aplicable tanto al nivel de meta-modelado como al de modelado. Las instancias de las configuraciones son modelos válidos, ya que las configuraciones son metamodelos válidos. El diseñador debe elegir la configuración más apropiada para definir el LM y/o la herramienta CASE de modelado.

La Tabla 3.3 contiene una lista de las posibles configuraciones con los metamodelos del armazón y su alcance, haciendo alusión a los lenguajes que pueden describir. Esta clasificación hace referencia en parte a la clasificación de lenguajes visuales del apartado 2.2.3. Las configuraciones 1-5 no incluyen la sintaxis concreta de los lenguajes. Consecuentemente, todas ellas pueden ser completadas con el metamodelo *Concreto*, generando las configuraciones 6-10 respectivamente, las cuales se omiten en la Tabla 3.3 por brevedad. Dado que las instancias del armazón en M1 nunca instancian el metamodelo *Concreto* (véase la explicación en el apartado 3.2.1), las configuraciones para las instancias de M1 no cambian con la inclusión del metamodelo *Concreto*.

El apéndice A compara el armazón presentado con el metamodelo propuesto por el consorcio OMG para el intercambio de diagramas en UML, i.e. UML-DI (OMG, Abril 2006).

Configuración	**Metamodelos**	**Alcance**
1	Objetos	Esta configuración representa la sintaxis abstracta. No hay información adicional de cómo mostrar los modelos. Sin embargo, esta configuración puede ser usada por editores con forma de árbol, como los generados automáticamente por EMF.
2	Objetos, Vistas	Este conjunto proporciona información sobre la sintaxis abstracta y las *vistas*. Esta configuración no tiene información espacial y es especialmente adecuada para los lenguajes basados en conexión.
3	Objetos, Vistas, Layouts	Esta configuración considera los lenguajes de modelado y los diagramas visuales. Contiene las *vistas* e información de la distribución de sus elementos en el espacio, lo que permite representar los diagramas. Es especialmente útil para los lenguajes híbridos entre los lenguajes basados en conexiones y los lenguajes basados en geometría. Esta configuración se usa también para los lenguajes basados en geometría; en este caso, el metamodelo Vistas tendría nodos, pero no aristas.
4	Objetos, Vistas, Layouts, Proyecto	Esta configuración es apropiada para un proyecto completo para una herramienta CASE de modelado. Sin embargo, esta configuración no tiene información sobre la sintaxis concreta del lenguaje de modelado.
5	Objetos, Vistas, Proyecto	Esta configuración es similar a la Configuración 2, pero incorpora la información del proyecto y, por tanto, es apropiada para herramientas CASE de modelado. Sin embargo, esta herramienta no puede incorporar información de la distribución en el espacio. Esta configuración es útil para las herramientas CASE de modelado orientadas a los lenguajes basados en conexión.

Tabla 3.3: Configuraciones del armazón.

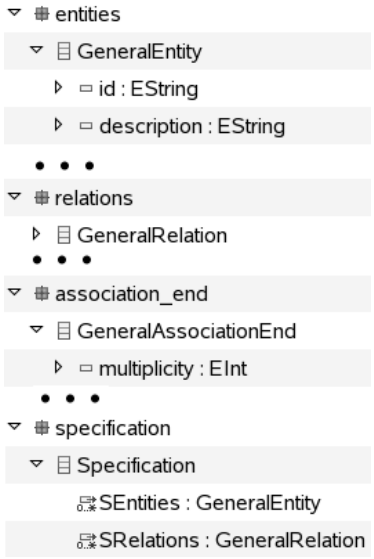


Figura 3.7: Estructura Básica del Metamodelo para la sintaxis abstracta.

3.3 Guía para definir el metamodelo *Objetos*

Esta guía proporciona una estructura para organizar los contenidos del metamodelo *Objetos* (véase apartado 3.2.1), un catálogo de representaciones para las entidades y las relaciones, y unas pautas para tomar decisiones explícitas de diseño. La mayoría de los LMs actuales están basados en conexiones (véase el apartado 2.2.3), es decir, los elementos del lenguaje se clasifican en entidades y relaciones. Por ello, esta guía sigue una aproximación *Entidad-Relación* (ER) (García-Magariño et al., 2009). Mientras que la guía propone una única representación para las entidades, hay diferentes representaciones para las relaciones cuyos objetivos son satisfacer distintos requisitos. Los siguientes sub-apartados discuten más detalladamente estos aspectos.

3.3.1 Estructura básica de un metamodelo para la sintaxis abstracta

Esta guía propone una estructura básica para el metamodelo *Objetos* independientemente de las características del LM, tales como la existencia de relaciones n-arias o existencia de atributos en los cuerpos o extremos de las relaciones. La Figura 3.7 muestra la estructura básica en un editor EMF. La estructura contiene cuatro *EPackages* de M2 que corresponden a los diferentes elementos de esta perspectiva ER de los lenguajes *basados en conexiones*. Estos paquetes definen jerarquías de herencia de *EClasses* de M2 con una única raíz en cada paquete. De acuerdo con la semántica de ECore (Budinsky, 2003), las *EClasses* de M2 heredan los atributos en estas jerarquías. Por tanto, la definición de atributos en la raíz de una de estas jerarquías es un mecanismo para definir atributos para todos los elementos de un paquete dado. Los cuatro *EPackages* de M2 son:

- *entities*. Contiene todas las entidades de un LM organizadas en una jerarquía, cuya raíz se llama *GeneralEntity*.

- *relations*. Este paquete contiene los cuerpos de las relaciones en un LM si se necesitan. Este es el caso, por ejemplo, de los LMs que incluyen relaciones n-arias o relaciones con atributos. La raíz de esta jerarquía es la EClass de M2 *GeneralRelation*.

- *association_ end*. Incluye todos los extremos de relaciones de los LMs si se necesitan. Por ejemplo, los LMs con atributos en los extremos de las relaciones requieren estos elementos. Este paquete es una jerarquía cuya raíz es el extremo de relación abstracto *GeneralAssociationEnd*.

- *specification*. Este paquete contiene la EClass de M2 *Specification*, y ésta contiene dos *EReferences containment* múltiples de M2, llamadas respectivamente *SEntities* y *SRelations*. Cada EClass *Specification* de M1 representa el elemento raíz de un modelo dado, y este elemento raíz contiene todas las entidades y relaciones del modelo en el nivel M1, respectivamente con las *EReferences* de M1 que son instancias de las dos EReferences de M2 mencionadas.

3.3.2 Catálogo de representaciones de entidades y relaciones

En el catálogo de representaciones, se proponen una categoría de representación para las entidades y varias categorías de representación para las relaciones. Los nombres de las categorías para las relaciones se denominan: *No-EClass*, *EClass-cuerpo* y *EClass-cuerpo-extremo*; y estos nombres hacen referencia a las EClasses de M2 que se usan por cada relación. Además, para cada categoría de representación de las relaciones, existe una variante *no-redundante* y otra variante *redundante*, según si se usan EReferences o parejas de EReferences en M2 y M1. Los diferentes tipos de representaciones de entidades y relaciones se describen en los siguientes sub-apartados.

Representación de Entidades

Este armazón representa una entidad como una *EClass* de M2 con *EAttributes*. Como se menciona anteriormente (véase apartado 3.3.1), el *EPackage* de M2 *entities* contiene estas *EClasses* de M2.

La representación No-*EClass* para las relaciones

La representación No-*EClass* describe las relaciones con una única *EReference non-containment* de M2. La *EClass* de M2 que corresponde con la entidad origen de la relación contiene esta *EReference* de M2 que apunta a la *EClass* de M2 que representa la entidad destino de la relación.

De acuerdo con la especificación de ECore (Budinsky, 2003), esta decisión tiene varias consecuencias. Primero, múltiples relaciones pueden enlazar con la misma entidad en M1 al usarse *EReferences non-containment* en M1. Segundo, las *EReferences* en M1 pueden conectar exactamente dos *EClasses* de M1 y

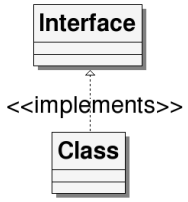
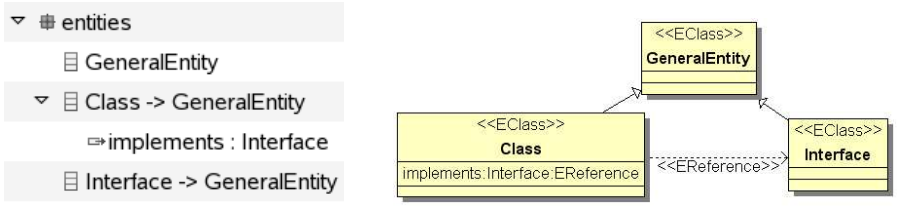


Figura 3.8: Un extracto del metamodelo UML con la relación *Implements* entre *Class* y *Interface*.



(a) Notación EMF (b) Notación Visual

Figura 3.9: Representación No-*EClass* para la relación *Implements* de la Figura 3.8.

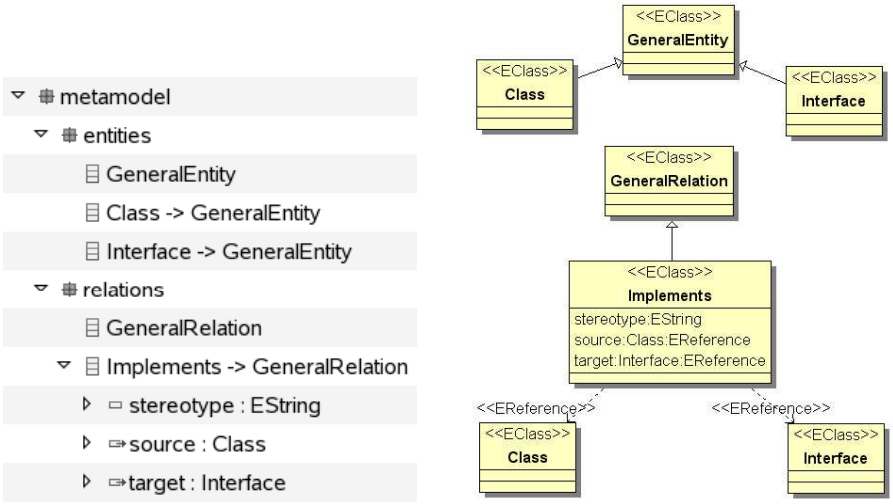
no tienen atributos, luego esta representación es sólo adecuada para relaciones binarias sin atributos.

Como ejemplo de esta representación, la Figura 3.8 muestra parte del metamodelo UML con la relación *Implements* entre las entidades *Class* y *Interface*. La Figura 3.9 describe la representación No-*EClass* de esta relación. La *EReference* de M2 *implements* aparece en la *EClass* de M2 llamada *Class* y señala a la *EClass* de M2 llamada *Interface*.

Representación *EClass-cuerpo* para las relaciones

En la representación *EClass-cuerpo*, cada relación se representa con una *EClass* de M2, y una *EReference* de M2 por cada entidad que conecta la relación. La *EClass* de M2 representa el cuerpo de la relación. Dicha *EClass* de M2 contiene los atributos de la relación, como *EAttributes*, y la relación se conecta con varias entidades con las *EReferences* de M2. Como se indica en el apartado 3.3.1, el *EPackage* de M2 llamado *relations* del metamodelo contiene estas *EClasses* de M2. De la misma forma que en la representación No-*EClass* (véase el apartado anterior), estas *EReferences* de M2 son *non-containment* para permitir que varias relaciones enlacen con las mismas entidades. Además, hay *EReferences* de M2 para cada posible tipo de entidad que cada tipo de relación pueda enlazar, lo que restringe los tipos de entidades que se pueden enlazar con una relación dada en M1. Con esta implementación, la representación *EClass-cuerpo* es adecuada para representar relaciones n-arias con atributos.

La Figura 3.10 presenta un ejemplo de representación *EClass*-cuerpo para la relación *Implements* de UML (véase la Figura 3.8). En esta representación, la *EClass* de M2 *Implements* representa la relación *Implements*, que permite que el atributo *Stereotype* sea incluido en la relación. Las *EReferences* de M2 llamadas *source* y *target* conectan la *EClass* de M2 del cuerpo de la relación



(a) Notación EMF (b) Notación Visual

Figura 3.10: Representación *EClass-cuerpo* para la relación *Implements* de la Figura 3.8.

con las *EClasses* que representan el origen y destino enlazadas por la relación respectivamente.

La representación *EClass-cuerpo-extremo* para las relaciones

La representación *EClass-cuerpo-extremo* usa los siguientes elementos para cada relación:

- Una *EClass* de M2 para el cuerpo de la relación. Esta *EClass* de M2 está situada en el *EPackage* llamado *relations*. Los *EAttributes* de M2 de esta *EClass* representan los atributos de la relación.

- Una *EClass* de M2 para cada extremo de la relación. Una relación n-aria puede tener más de dos extremos. Las *EClasses* de M2 para estos extremos de las relaciones se sitúan en el *EPackage association_ends*. Los *EAttributes* de M2 de estas *EClasses* representan los atributos de los extremos de las relaciones. Ejemplos de estos atributos son las indicaciones sobre la navegabilidad y la cardinalidad de las relaciones.

- Una *EReference containment* de M2 para enlazar el cuerpo de la relación con cada extremo de la relación.

- Una *EReference non-containment* de M2 para conectar cada extremo de relación con cada meta-entidad posible que puede enlazar. Esto consigue que cada relación sólo pueda conectar ciertos tipos de entidades. En M1, el extremo de relación sólo puede conectar con una entidad del modelo. Por tanto, en cada extremo de relación, sólo una *EReference* de M1 puede señalar a una entidad y todas las otras *EReferences* de M1 no usadas deben ser establecidas a nulo.

De acuerdo con esta estructura, la representación *EClass-cuerpo-extremo* puede describir relaciones n-arias con atributos tanto en los cuerpos como en sus extremos. Los dos casos de estudio presentados más adelante usan esta representación.

Representaciones Redundantes de las relaciones

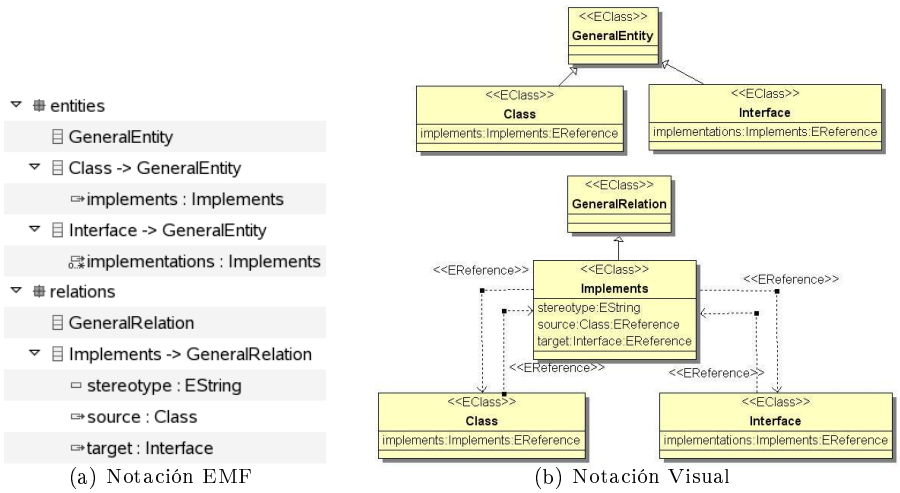
Las representaciones de las relaciones en los sub-apartados anteriores se llaman en este catálogo *no-redundantes*. Estas son usadas para conectar cada pareja de *EClasses* de M2 con una sola *EReference* de M2, lo que minimiza el número de estos elementos en el metamodelo. Sin embargo, dado que una *EReference* de M1 describe un enlace binario unidireccional, esta representación incrementa el tiempo requerido para ciertos procesos sobre los modelos debido a su baja *navegabilidad*; es decir, no se puede *navegar* (i.e. ir de unos elementos a otros por medio de referencias) en las direcciones opuestas de las *EReferences* de M1. Por ejemplo, para saber las relaciones en las que participa una entidad de M1, se requiere el análisis de todas las relaciones de M1 con las representaciones *EClass-cuerpo* y *EClass-cuerpo-extremo*, y de todas las entidades de M1 que tengan relaciones con la representación *No-EClass*.

Para configuraciones en las que la eficiencia de procesamiento es especialmente relevante, el armazón propone alternativas *redundantes* para las representaciones previas. Las versiones redundantes de las representaciones de las relaciones usan parejas de *EReferences* de M2 y M1 donde las versiones no-redundantes usan sólo una *EReference* de M2 y M1. Estas parejas de *EReferences* de M2 y M1 enlazan las EClasses de M2 y M1 en sentidos opuestos. De esta manera, las representaciones redundantes soportan navegaciones bidireccionales de las relaciones de M1, lo que reduce el tiempo de procesamiento de los modelos. Sin embargo, debe aclararse que estas representaciones redundantes requieren mecanismos adicionales para mantener la consistencia de las parejas de *EReferences* de M1. La Tabla 3.4 resume la estructura de las representaciones redundantes. Todas las representaciones redundantes necesitan conocer la cardinalidad de los extremos de las relaciones de M2; es decir, el número de relaciones de M1 de un tipo que pueden unirse con cada tipo de entidad en M1. Teniendo en cuenta estas cardinalidades, se establecen las cardinalidades de las EReferences de M2.

La Figura 3.11 muestra un ejemplo de representación *EClass-cuerpo redundante* para la relación *Implements* de UML (véase la Figura 3.8). Siguiendo la representación de las entidades del armazón (véase el apartado 3.3.2), una *EClass* de M2 representa cada entidad *Class* e *Interface*. La representación de la relación *Implements* usa una *EClass* de M2 llamada *Implements* y dos parejas de *EReferences* de M2. La primera pareja incluye las *EReferences* de M2 llamadas *implements* y *source* que conectan las *EClasses* de M2 *Implements* y *Class*. La segunda pareja está formada por las *EReferences* de M2 llamadas *implementations* y *target* para conectar las *EClasses* de M2 *Interface* e *Implements*. La cardinalidad de los extremos de la relación *Implements* es *1:N* para los extremos de *Class* e *Interface* respectivamente. Es decir, cada *Class* sólo se relaciona con un *Interface*, pero cada *Interface* puede relacionarse con varias *Class*. Por tanto, las EReferences de M2 *implements* e *implementations*, situadas respectivamente en los tipos de entidades *Class* e *Interface* en M2, tienen las mismas cardinalidades que los extremos de la relación de M2 *Implements* para dichos tipos de entidades, es decir 1 y N respectivamente.

Representación	**Estructura**
No-*EClass* Redundante	Dos *EReferences non-containment* de M2, cada una de ellas situadas en cada entidad de M2 involucrada en la relación.
EClass-cuerpo Redundante	Una *EClass* de M2 (véase un ejemplo en la Figura 3.11) para representar el cuerpo de la relación. Este cuerpo de relación conecta con *EClasses* de M2 que representan entidades. Estas conexiones se proporcionan con parejas de *EReferences non-containment* de M2. Para cada pareja, una de las *EReferences* de M2 es situada en la *EClass* de M2 que representa el cuerpo de la relación, y la otra es situada en la *EClass* de M2 de la entidad enlazada.
EClass-cuerpo-extremo Redundante	Una *EClass* de M2 (véase un ejemplo en la Figura 3.12) para representar el cuerpo de la relación y una *EClass* de M2 para cada extremo de la relación. La *EClass* de M2 del cuerpo de la relación conecta con un par de *EReferences* de M2 cada *EClass* de M2 que representa un extremo de la relación. La *EReference* de M2 situada en la EClass del cuerpo de la relación es *containment*, mientras que la *EReference* de M2 situada en la *EClass* de M2 del extremo de la relación es *non-containment*. Cada *EClass* de M2 de un extremo de la relación conecta con la *EClass* de M2 de la entidad correspondiente con una pareja de *EReferences non-containment* de M2. En estas parejas, una *EReference* de M2 se sitúa en la *EClass* de M2 del extremo de la relación y otra en la *EClass* de M2 de la entidad.

Tabla 3.4: Estructura de las representaciones *redundantes* de las relaciones.



(a) Notación EMF (b) Notación Visual

Figura 3.11: Representación *EClass-cuerpo redundante* para la relación *Implements* de la Figura 3.8.

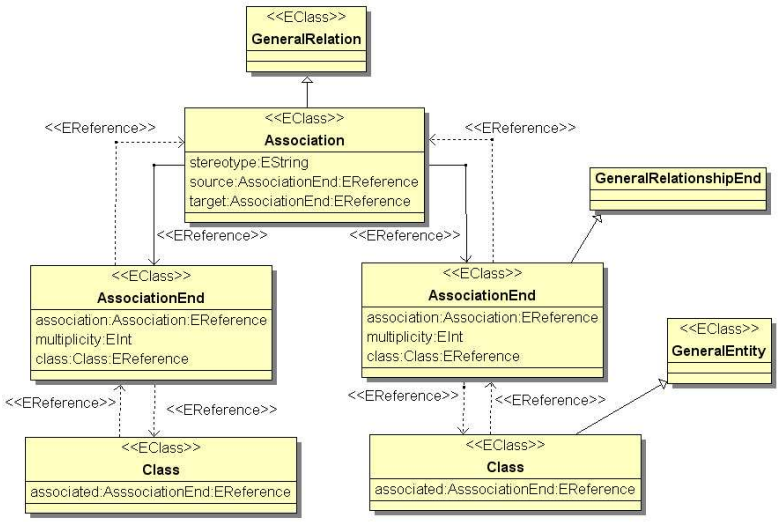


Figura 3.12: La representación *EClass-cuerpo-extremo redundante* para la relación *Association* de UML.

La Figura 3.12 muestra un ejemplo de representación *EClass-cuerpo-extremo redundante* para la relación *Association* de UML. En este ejemplo, el cuerpo de la relación se llama *Association*, el extremo de la relación se llama *AssociationEnd*, y la entidad usada se llama *Class*. Las *EClasses* de M2 se conectan con parejas de *EReferences* de M2 de la forma que se explica en la Tabla 3.4.

Las representaciones homogéneas y heterogéneas de las relaciones

Las relaciones de un LM pueden tener diferentes requisitos estructurales. Por ejemplo, un LM puede tener relaciones binarias y n-arias, con y sin atributos tanto en los cuerpos como en los extremos. Para tal LM, el diseñador de metamodelos debe elegir entre proporcionar una representación específica para cada relación (i.e. una *representación heterogénea*, véase la Definición 1) o usar la misma representación para todas las relaciones (i.e. una *representación homogénea*, véase la Definición 2).

Definición 1 *Una **representación heterogénea** contiene diferentes categorías de representación en el metamodelo para las relaciones de un LM.*

Definición 2 *Una **representación homogénea** contiene la misma categoría de representación en el metamodelo para todas las relaciones de un LM.*

La representación homogénea describe todas las relaciones en el LM con la representación usada para la relación más compleja, siendo la representación No-*EClass* la más simple, y la representación *EClass-cuerpo-extremo* la más compleja. Esta uniformidad tiene ventajas para el diseño y uso del metamodelo. La elección de la representación homogénea ahorra tiempo de diseño porque no existe la necesidad de decidir la representación apropiada para cada relación. Las herramientas que procesan los metamodelos son también más fáciles de construir que con una representación heterogénea, porque procesan todas las relaciones de

la misma manera. Finalmente, el entendimiento de un metamodelo homogéneo demanda menos esfuerzo por parte de los lectores dado que, entendiendo una estructura, se entiende el resto, al tratarse de réplicas de la misma solución.

La representación heterogénea proporciona una representación específica para cada relación. Por tanto, esta representación requiere frecuentemente menos elementos para representar las relaciones en M2 y M1. Por ejemplo, un LM puede contener dos tipos de relaciones: relaciones binarias sin atributos, y relaciones n-arias con atributos en el cuerpo y en los extremos. Mientras que el primer tipo de relación puede ser representado con una *EReference* en M2 y M1, el segundo tipo necesita la representación *EClass-cuerpo-extremo* que incluye varias *EReferences*, *EClasses*, y *EAttributes* tanto en M2 como en M1. La representación heterogénea usaría sólo los elementos necesarios en M2 y M1; sin embargo, la representación homogénea usaría la representación con más elementos (i.e. la representación *EClass-cuerpo-extremo*) para ambos tipos de relaciones, usando más elementos que los estrictamente necesarios en M2 y M1.

3.3.3 Representaciones con el mínimo número de elementos

Este apartado demuestra que, en caso de usarse una representación *no-redundante* y *heterogénea* (véase el apartado 3.3.2), siguiendo la guía se obtiene una representación con el mínimo número de elementos requeridos para especificar entidades y relaciones de un LM con ciertas características estructurales, salvo para el caso particular de las relaciones con atributos en sus extremos y sin atributos en sus cuerpos. El motivo de esta excepción es mantener la coherencia y claridad entre las diferentes representaciones de las relaciones. La métrica de meta-modelado usada en este apartado es el número de elementos empleados en un metamodelo. En esta demostración, se asume el uso del lenguaje ECore. Para demostrar esta propiedad, los siguientes sub-apartados también demuestran teoremas y proposiciones adicionales sobre el número de elementos involucrados en las representaciones del catálogo.

Demostración para la representación de las entidades

Teorema 1 *Las entidades con atributos necesitan ser representadas por al menos una* EClass *de M2.*

Dado que el único elemento ECore que tiene atributos es la *EClass* de M2 (véase el apartado 2.2.2), una entidad con atributos debe ser representada al menos con una *EClass* de M2. Por tanto, el Teorema 1 está demostrado.

Considerando el Teorema 1, la representación de las entidades por este armazón (véase el apartado 3.3.2) usa el mínimo número de elementos, porque se asume que las entidades de los LMs tienen atributos. Este número mínimo de elementos es exactamente una *EClass* de M2 por entidad.

Demostración para la representación No-*EClass* de las relaciones

Teorema 2 *Las relaciones necesitan al menos una* EReference *de M2 para ser representadas.*

Las relaciones conectan entidades, y las entidades deben ser representadas con *EClasses*, como se deriva del Teorema 1. Dado que el único elemento ECore que puede conectar *EClasses* de M2 es el elemento *EReference* de M2, se hace obligatorio usar al menos una *EReference* de M2 para representar una relación, quedando demostrado el Teorema 2. En sí misma, una relación puede representarse como una *EReference* de M2 en el caso más trivial, o requerir más elementos como en los ejemplos del apartado 3.3.2.

De acuerdo con el Teorema 2, la representación No-*EClass* usa el mínimo número de elementos ECore para las relaciones, dado que usa una única *EReference* de M2. Como se indica en el apartado 3.3.2, esta representación es sólo adecuada para relaciones binarias sin atributos.

Demostración para la representación *EClass-cuerpo* de las relaciones

Proposición 1 *Una relación con atributos necesita ser representada por al menos una* EClass *de M2, y una* EReference *de M2 para cada tipo de entidad que conecta.*

Debido a que el único elemento del lenguaje ECore que tiene atributos es la *EClass* de M2 (véase el apartado 2.2.2), una relación con atributos debe ser representada con al menos una *EClass* de M2. Una *EReference* de M2 se necesita para conectar esta *EClass* de M2 con cada *EClass* de M2 que representa cada entidad conectada por la relación. Por tanto, queda confirmada la Proposición 1.

Una estructura alternativa a esta representación establecería un orden de entidades enlazadas *e0*, *e1*,..., *en*, y conectaría cada entidad *e(i)* con la entidad *e(i+1)* usando una *EReference* de M2. Entonces, usaría otra *EReference* de M2 para conectar la *EClass* de M2 del cuerpo de la relación con una de las *EClasses* para las entidades. De cualquier forma, esta representación usaría el mismo número de elementos ECore que la anterior, pero ésta sería menos clara estructuralmente. Otra opción sería usar esta última representación pero quitando la *EClass* de M2 para el cuerpo de las relaciones y poniendo los atributos de las relaciones en una de las entidades conectadas. Pero esto no es posible dado que no se podrían distinguir los atributos de la entidad de los atributos de la relación.

De acuerdo con la Proposición 1, la representación *EClass-cuerpo* usa el mínimo número de elementos ECore para las relaciones con atributos, dado que usa una *EClass* de M2, y una *EReference* de M2 por cada entidad enlazada. Como se establece en el apartado 3.3.2, esta representación se recomienda para las relaciones con atributos.

Proposición 2 *Una relación n-aria debe ser representada por al menos una* EClass *de M2 y, por cada entidad que conecta, una* EReference *de M2.*

Una *EReference* de M2 puede conectar sólo dos *EClasses* de M2. Dado que las relaciones n-arias no binarias conectan más de dos *EClasses* de M2, se necesita más de una *EReference* de M2. El único elemento ECore que puede contener varias *EReferences* de M2 es la *EClass* de M2. Por tanto, se necesita al menos una *EClass* de M2 para contener las *EReferences* de M2 de una relación n-aria no binaria, quedando confirmada la Proposición 2. La discusión para la Proposición 1 ha indicado una posible alternativa para esta *EClass* de M2, la

representación que directamente conecta las entidades con referencias. Sin embargo, esta representación no es válida para las relaciones n-arias, dado que no podría distinguirse entre una relación n-aria y varias relaciones binarias del mismo tipo. Para ser más precisos, en el caso general con dicha representación alternativa, no podría distinguirse entre una relación n-aria y varias relaciones n-arias (o binarias) del mismo tipo de menor número de extremos. Podría pensarse en añadir atributos adicionales para indicar las referencias que corresponden a cada relación n-aria no binaria. Sin embargo, esta opción no es posible dado que ECore no permite asociar los *EAttributes* de M2 con las *EReferences* de M2 (véase el apartado 2.2.2).

Por otro lado, se necesita una *EReference* de M2 para conectar la *EClass* de M2, que representa el cuerpo de la relación, con cada *EClass* de M2 que representa una entidad enlazada con la relación.

Por tanto, ambos tipos de relaciones, las que tienen atributos y las que son n-arias, necesitan el mismo número de elementos ECore. Este número de elementos ECore es exactamente el número de elementos usados por la representación *EClass-cuerpo* recomendada en el apartado 3.3.2 para estos tipos de relaciones.

Demostración para la representación *EClass-cuerpo-extremo* para las relaciones

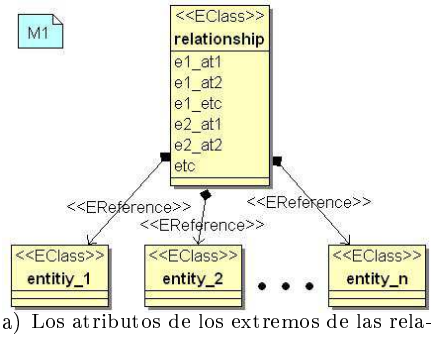
Para demostrar que la representación *EClass-cuerpo-extremo* usa el mínimo número de elementos ECore para representar una relación con atributos en el cuerpo y sus extremos, es necesario demostrar primero otros teoremas.

Teorema 3 *Si una relación tiene atributos en sus extremos, entonces cada uno de sus extremos debe ser representado con al menos una* EClass *de M2.*

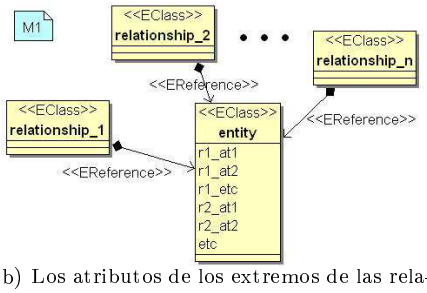
Debido a que el único elemento ECore que tiene atributos es la *EClass* de M2 (véase el apartado 2.2.2), cada extremo de la relación con atributos debe ser representado con al menos una *EClass* de M2. Puede ser discutido que los atributos pueden ser situados en otra *EClass* de M2, o bien en la que representa el cuerpo de la relación (véase la Figura 3.13(a)), o bien en aquellas que representan las entidades conectadas a la relación (véase la Figura 3.13(b)). Sin embargo, ambas alternativas se enfrentan al mismo problema. ECore no proporciona medios para almacenar en un *EAttribute* referencias a otra *EClass* de M2 diferente de la contenedora del *EAttribute* (véase el apartado 2.2.2). Por tanto, los atributos localizados en las *EClasses* alternativas no pueden ser asociados con un extremo de la relación en particular. Por tanto, los atributos de los extremos no deben estar dentro de las *EClasses* de M2 que representa el cuerpo de la relación o las entidades enlazadas. En conclusión, cada extremo de la relación debe estar representado con al menos una *EClass* de M2, lo que demuestra el Teorema 3.

Teorema 4 Si *para una relación, su cuerpo, sus extremos y las entidades que enlaza, están representados cada uno con una* EClass *de M2,* entonces *debe haber al menos dos* EReferences *de M2 para cada extremo de relación.*

La premisa del Teorema 4 considera que el cuerpo de la relación, sus extremos y las entidades que enlaza son representadas cada uno de ellos por una *EClass* de M2. Por tanto, para cada extremo, existen tres *EClasses* de M2 que deben



(a) Los atributos de los extremos de las relaciones no pueden ser situados en la *EClass* de M2 que representa el cuerpo de la relación.



(b) Los atributos de los extremos de las relaciones no puede ser situado en la *EClass* de M2 que representa la entidad.

Figura 3.13: Alternativas para representar los atributos de los extremos de las relaciones.

ser conectadas. Al menos, debe haber dos *EReferences* de M2 para conectarlas dado que cada *EReference* de M2 sólo puede conectar dos *EClasses* de M2 (véase el apartado 2.2.2). De esto, queda demostrado el Teorema 4. Estructuras de conexión alternativas a la representación *EClass-cuerpo-extremo* son posibles, como aquella que conecta cada extremo de relación con otro extremo y la entidad enlazada, y uno de los extremos es enlazado con el cuerpo de la relación. Sin embargo, estas alternativas necesitan el mismo número de *EReferences* de M2 y son menos claras estructuralmente.

Considerando la representación de relaciones con atributos en su cuerpo y sus extremos y los teoremas previos, las siguientes consecuencias aparecen. De acuerdo con el Teorema 3, cada extremo de relación necesita al menos una *EClass* de M2 para ser representada. De acuerdo con la Proposición 1, dado que la relación contiene atributos en su cuerpo, el cuerpo de la relación debe representarse con una *EClass* de M2. Las entidades deben ser representadas con *EClasses* como se demostró en el apartado 3.3.3. Las premisas del Teorema 4 son ciertas y, por tanto, la relación necesita al menos dos *EReferences* de M2 para cada extremo de la relación. Estos elementos ECore son exactamente los utilizados en la representación *EClass-cuerpo-extremo*. La guía recomienda la representación *EClass-cuerpo-extremo* para las relaciones con atributos tanto en el cuerpo como sus extremos. Por tanto, la guía propone una representación con el mínimo número de elemento ECore para las relaciones con atributos tanto en el cuerpo como en los extremos.

Sin embargo, la guía también recomienda la representación *EClass-cuerpo-extremo* para relaciones con atributos en los extremos pero sin atributos en el cuerpo. En este caso particular, la guía no recomienda una representación con el mínimo número de elementos. De hecho, con dichos requisitos estructurales, las *EClasses* que representan los extremos de las relaciones pueden ser directamente conectados con *EReferences*. Esta guía no recomienda esta última representación por dos motivos: esta representación no es tan clara estructuralmente; y el uso de esta representación sería confuso cuando se combinara con las otras representaciones de las relaciones que usan *EClasses* para su cuerpo.

En conclusión, en caso de usarse una representación *no-redundante* y *heterogénea*, la guía recomienda la representación con el mínimo número de elementos

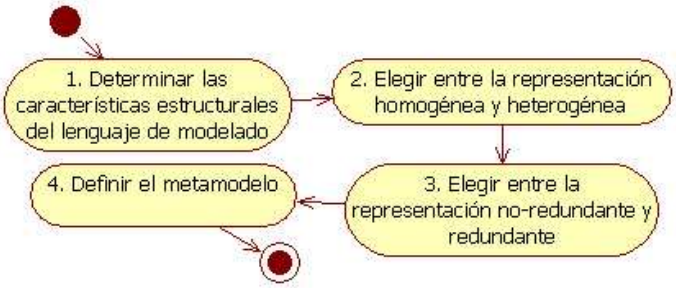


Figura 3.14: Pasos de la guía para diseñar un metamodelo de la sintaxis abstracta de un LM.

necesarios para las relaciones, salvo para el caso particular de las relaciones con atributos en los extremos pero sin atributos en el cuerpo. Para dicho caso, se usa una representación que no es mínima para garantizar la claridad y la coherencia con el resto de representaciones.

3.3.4 Pasos de la guía para el metamodelo *Objetos*

El objetivo de la guía introducida en este apartado es asistir al diseñador en la definición del metamodelo Objetos. El proceso propuesto se puede observar en la Figura 3.14. La guía empieza determinando las características principales del lenguaje con un conjunto de preguntas preliminares en la actividad 1, y toma algunas decisiones iniciales dependiendo del contexto de aplicación del metamodelo en las actividades 2 y 3. La guía continúa proponiendo en la actividad 4 un metamodelo para especificar el lenguaje de acuerdo con los requisitos previos del diseñador. Este metamodelo usa el armazón descrito en el apartado 3.3.2. Los siguientes sub-apartados explican las actividades del proceso de forma más detallada.

Determinar las características estructurales del lenguaje de modelado

La primera actividad en la guía (véase la Actividad 1 en la Figura 3.14) es especificar las características estructurales que conforman el LM. Las características que la guía considera provienen de la aproximación ER adoptada en esta guía y fueron introducidas en el apartado 2.2.4. La Tabla 3.5 muestra más ejemplos de las características estructurales usadas en esta guía.

En un LM dado, cada relación puede tener diferentes características estructurales. El diseñador debe saber aproximadamente cuantas relaciones tiene cada una de las características estructurales. Estas estimaciones deben considerar el estado actual del LM, pero también su posible evolución. De esta manera, la guía aconsejará una representación que facilite el futuro mantenimiento del metamodelo.

Característica Estructural	**Ejemplos**
Relaciones con atributos	UML 2.2 (OMG, 2007a,b) tiene el atributo *estereotipo* para las relaciones de agregación.
Extremos de relaciones con atributos	UML 2.2 (OMG, 2007a,b) el atributo *multiplicidad* en los extremos de las relaciones para indicar cuántas instancias de una *Clase* son enlazadas con cuantas instancias de otra *Clase*. SPEM (OMG, 2008c) usa los atributos *isNavigable* y *aggregationKind* en el extremo de la relación de *asociación*. Éstos indican, respectivamente, si un extremo es navegable, y el tipo del extremo.
Relaciones n-arias	OWL (W3C, 2004), INGENIAS (Pavón et al., 2005) y AUML (Bauer y Odell, 2005) contienen relaciones que enlazan más de dos entidades.

Tabla 3.5: Características estructurales consideradas en la guía.

Elegir entre las representaciones homogéneas y heterogéneas

La segunda actividad en la guía (véase la Actividad 2 en la Figura 3.14) elige entre las representaciones *homogéneas* y *heterogéneas* (véanse respectivamente las Definiciones 1 y 2, al final del apartado 3.3.2) para las relaciones. Esta decisión implica llegar a un compromiso entre minimizar el número de elementos de meta-modelado usados y, por otro lado, agilizar el futuro mantenimiento, facilitar el procesamiento de los modelos, y mejorar la usabilidad de los editores generados.

Esta guía recomienda usar la representación homogénea para lenguajes en evolución. Esta representación permite que las relaciones tengan más requisitos estructurales en el futuro, debido a que selecciona la representación más flexible (i.e. siendo la más flexible del catálogo la *EClass*-cuerpo-extremo y la menos la No-*EClass*) de las representaciones requeridas para todas las relaciones del lenguaje. En la representación homogénea, los nuevos requisitos estructurales como mucho añaden atributos a las EClasses de M2 existentes, y estos cambios en el metamodelo no hacen que los modelos existentes dejen de ser instancias del nuevo metamodelo si se usa la representación más flexible. Por el contrario, en la representación heterogénea, cuando se incluyen nuevos requisitos estructurales, algunas de las representaciones de las relaciones cambian (e.g. de ser representadas con una EReference de M2 a ser representadas con una EClass de M2). Esto hace que todos los modelos que eran instancia del metamodelo antiguo no sean instancias del metamodelo nuevo, por la manera de serializarse en XMI. Las EClasses se serializan con elementos XML, y las representaciones de las relaciones antiguas no tienen todos los elementos XML que se necesitan para representar instancias del metamodelo nuevo. Las explicaciones sobre el formato de XMI usado por EMF se omiten en esta memoria por ser breves, pero se encuentran en (Budinsky, 2003).

También, la representación homogénea suele incrementar la usabilidad de los editores generados. La razón es que los usuarios pueden trabajar con todos los elementos de una forma similar.

En contraste, la representación heterogénea es apropiada para los LMs estables donde cada relación tiene diferentes características estructurales. Por ejemplo, la mayoría de las especificaciones de OMG, como SPEM (OMG, 2008c) y UML (OMG, 2007a,b), usan la representación heterogénea para las relaciones. La ventaja de esta representación es que reduce el número de elementos del metamodelo.

Elegir entre las representaciones no-redundantes y redundantes

La tercera actividad (véase la Actividad 3 en la Figura 3.14) consiste en decidir entre las representaciones no-redundante y redundante. Como se explica en el apartado 3.3.2, la representación no-redundante usa una *EReference* de M2 para conectar dos *EClasses* de M2, mientras que la representación redundante usa parejas de *EReferences* de M2 para el mismo propósito. Esto no significa que la representación de una relación sólo use *EReferences*. Dependiendo de las características estructurales de una relación (véase el apartado 3.3.4), su representación puede contener varias *EClasses*, *EReferences*, y *EAttributes* en M2.

Considerando los compromisos vistos en el apartado 3.3.2, el diseñador debe elegir la representación redundante si se necesita un acceso rápido a las relaciones conectadas a una entidad. La representación no-redundante es una mejor alternativa cuando es importante reducir el número de elementos de modelado usados en el metamodelo, y lo hace más accesible a una edición manual. Esta última representación es también aconsejable con algunas herramientas que generan automáticamente editores para LMs desde sus metamodelos, como EMF (Budinsky, 2003). Si los editores generados carecen de mecanismos que garanticen la consistencia de las parejas de *EReferences* de M1 en la representación redundante, el tratamiento de la consistencia se convierte en tarea del usuario, incrementando considerablemente la carga de trabajo.

Definir el metamodelo

La última actividad del proceso (véase la Actividad 4 en la Figura 3.14) construye el metamodelo para el lenguaje considerando las decisiones del diseñador en las actividades anteriores. Este proceso aparece en la Figura 3.15. El metamodelo es construido con el catálogo de representaciones introducido en el apartado 3.3.2. Este proceso propone usar la única representación disponible para las entidades (Actividad 4.1) y elegir la representación más apropiada para cada relación (desde la Decisión 4.2 a la Actividad 4.13) de acuerdo con las características estructurales. La Decisión 4.2 establece un bucle por el cual todas las relaciones de las sintaxis abstracta de un LM son tratadas, seleccionándose de una en una en la Actividad 4.3. Cada relación se asocia con un tipo de representación según las características estructurales de dicha relación (desde la Decisión 4.4 a la Actividad 4.9). En concreto, la Decisión 4.4 determina si la relación tiene atributos en su extremos y, si es así, la Actividad 4.5 asigna la representación *EClass-cuerpo-extremo* para dicha relación. En caso de que no se asigne dicha representación a la relación, la Decisión 4.6 determina si la relación

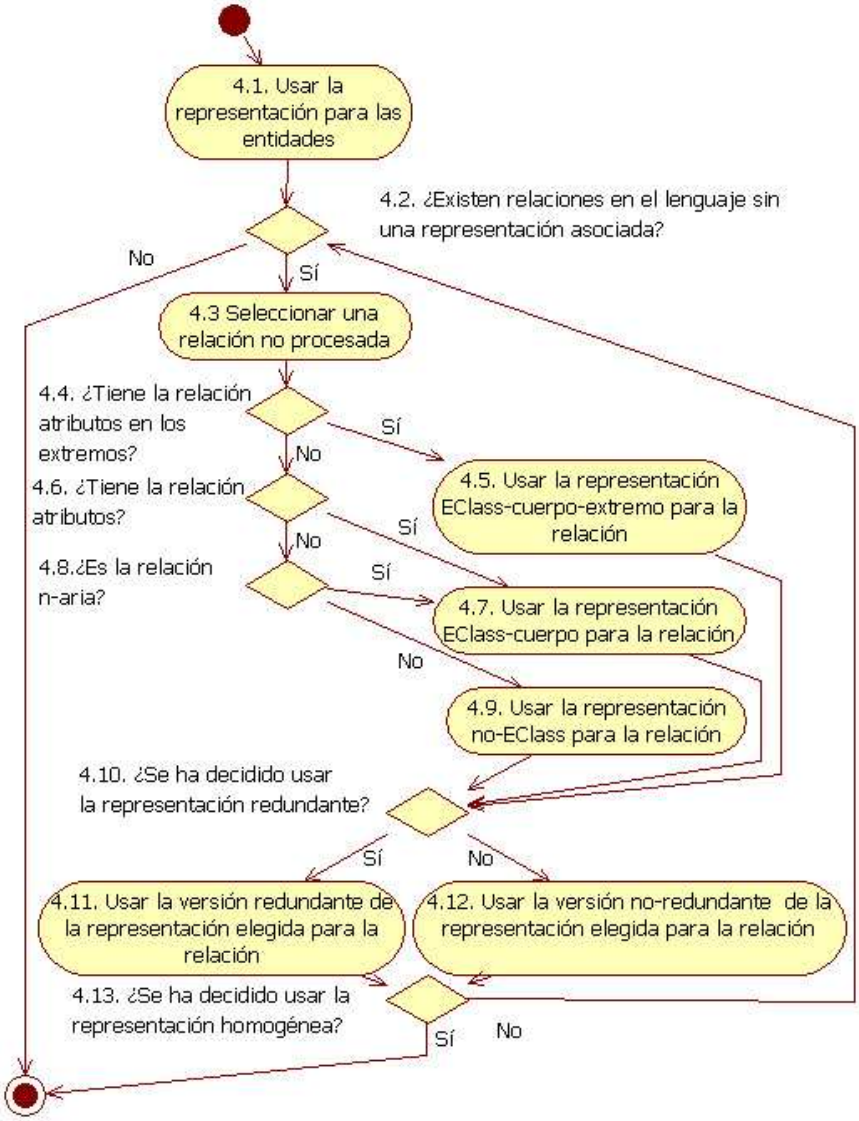


Figura 3.15: Actividad 4 de la guía detallada: elección de las representaciones del catálogo de acuerdo con los requisitos estructurales del metamodelo.

tiene atributos y la Decisión 4.8 determina si la relación es n-aria. En caso de que el resultado de alguna de las dos decisiones anteriores sea cierta, entonces la Actividad 4.7 asocia la representación *EClass-cuerpo* a la relación. En caso de que el resultado sea negativo en las Decisiones 4.4, 4.6 y 4.8, se le asocia la representación *No-EClass* a la relación, en la Actividad 4.9. De acuerdo con la decisión de diseño anteriormente tomada, la Decisión 4.10 indica si se le asocia la versión *redundante* (en la Actividad 4.11) o la versión *no-redundante* (en la Actividad 4.12) del tipo de representación asociado.

La Decisión 4.13 determina si se debe usar una representación homogénea o heterogénea, de acuerdo con la decisión de diseño tomada anteriormente. Si la representación es heterogénea, cada relación se representa con el tipo de representación asignado. En el caso de que la representación sea homogénea, todas las relaciones adoptan la misma representación. Esta representación es la más flexible requerida para todas las relaciones del lenguaje, considerando la representación *EClass-cuerpo-extremo* la más flexible y la representación *No-EClass* la menos flexible. En este caso, si alguna relación necesita la representación más flexible, la representación *EClass-cuerpo-extremo*, entonces no es necesario analizar las propiedades del resto de las relaciones y el bucle puede terminar (en la Decisión 4.13). La razón es que, en este caso particular, todas las relaciones usarán la representación más flexible independientemente de sus características estructurales.

3.4 Conclusiones

Este trabajo surge de la falta de guías para la definición de los metamodelos y de la gran diversidad de representaciones en los metamodelos existentes (véase el capítulo 2). Por tanto, se define una guía cuyo objetivo es ayudar a los diseñadores en la definición de metamodelos, permitiendo meta-modelar tanto los LMs como las herramientas CASE de modelado. Para ello, esta guía proporciona un armazón con varios metamodelos, permitiendo desde configuraciones tan sencillas como el metamodelo sólo con la sintaxis abstracta de un LM hasta configuraciones tan complejas que incluyen ciertos aspectos necesarios para las herramientas CASE de modelado. Dentro de esta guía, se incluye otra guía para los metamodelos Objetos, que incluyen las sintaxis abstractas de los LMs. Esta otra guía sigue una aproximación ER, la cual es apropiada para los LMs *basados en conexiones*, que son casi la práctica totalidad de los LMs existentes actualmente.

Las ventajas de la guía general presentada en este capítulo sobre las aproximaciones existentes son las siguientes:

- *Se proporciona un armazón con varios metamodelos para los LMs y/o herramientas CASE de modelado.* De esta forma, se mantienen por separado los siguientes aspectos: sintaxis abstracta del LM, sintaxis concreta del LM, agrupaciones lógicas del modelo, distribución visual en el espacio, e información específica de la herramienta CASE de modelado. Además, existen diferentes configuraciones de esta estructura que se adaptan a los diferentes tipos de LMs y herramientas CASE de modelado.

- *Se proporciona una guía para la construcción del metamodelo de sintaxis abstracta (i.e. metamodelo Objetos).* Esta guía toma como entrada los

requisitos estructurales del LM. También considera ciertas decisiones de diseño llegando a un compromiso entre la navegabilidad, la simplicidad de representación y la usabilidad de las herramientas generadas a partir de los metamodelos. La guía define una secuencia de actividades que facilita la definición del metamodelo Objetos, según las necesidades del problema.

La guía presentada en este capítulo ha permitido definir varios metamodelos de especial relevancia en la comunidad de SMA. En primer lugar, se ha modernizado el IDK con un metamodelo con EMF (García-Magariño y Gómez-Sanz, 2008) definido con la guía; dicho metamodelo usa la configuración con todos los metamodelos del armazón. Este metamodelo ha permitido realizar la generación de transformaciones basadas en ejemplos para el IDK, presentada más adelante en esta tesis. En segundo lugar, se ha definido un metamodelo de procesos software (basado en SPEM (OMG, 2008c)) para la definición de procesos en la ISOA (García-Magariño et al., 2007, 2009b,a,b). Ambos metamodelos, el del IDK con EMF y el del editor de procesos, han servido como marco de experimentación de esta guía y son presentados en el siguiente capítulo.

Capítulo 4

Validación de la guía de definición de metamodelos en los sistemas multi-agente

La guía presentada en el anterior capítulo se ha aplicado para instanciar el armazón en dos casos. En primer lugar, se ha instanciado el armazón con toda la información necesaria en la herramienta IDK. En este caso, se incluye la capa de núcleo, la capa de información visual y la capa específica de la herramienta CASE. La herramienta IDK se ha modernizado para exportar instancias del metamodelo resultante, mostrando así que el armazón se ha aplicado a una herramienta CASE existente actual en la ISOA. Por otro lado, el armazón se ha instanciado para definir un metamodelo Objetos para la definición de procesos software para la creación de SMAs, y a partir de este metamodelo se ha generado un editor de procesos que considera más conceptos de modelado que otras herramientas existentes.

4.1 Introducción

Este capítulo de tesis recoge experimentación resultante de aplicar la guía de definición de metamodelos propuesta en el capítulo anterior. Con esta experimentación, se valida la utilidad de la guía al obtenerse dos instancias del armazón con utilidad práctica en la ISOA.

La primera instancia del armazón de metamodelos permite modelar el LM de INGENIAS con toda la información necesaria para su herramienta de soporte, el IDK, y usa la Configuración 4, que está compuesta por cuatro metamodelos (i.e. *Objects, Vistas, Layouts* y *Proyecto*). La segunda instancia del armazón define un LM para definir procesos en la ISOA y está basado en SPEM (OMG, 2008c). Este segundo armazón está compuesto únicamente por el metamodelo de la sintaxis abstracta, i.e Configuración 1, y se usa en la generación automática de un editor de procesos de ISOA (García-Magariño et al., 2007, 2009b,a,b) con EMF. Ambas instancias del armazón han servido como experimentación de la guía de definición de metamodelos, propuesta en el anterior capítulo, y los aspectos más relevantes de dichos armazones se describen en este capítulo.

Además, los dos casos de estudio presentados son también dos ejemplos de aplicación de la guía para la sintaxis abstracta (i.e. metamodelo Objetos), descrita en apartado 3.3, en la ISOA. En ellos se pueden observar varios factores comunes. En concreto, debido a la continua evolución de los LMs en la ISOA, se ha escogido en ambos casos una representación homogénea *EClass*-cuerpo-extremo, que es la representación que más fácilmente se adapta a futuros cambios en los requisitos estructurales de las relaciones.

También se observa que ambos casos de estudio escogen una representación no-redundante. En el primer caso, el del IDK, no son necesarias las estructuras redundantes, debido a que el IDK usa estructuras de datos internas. Por otro lado, en el segundo caso, el del editor de SPEM, los editores generados por EMF no garantizan el mantenimiento de la consistencia de las estructuras redundantes. Por tanto, el mecanismo de consistencia se debería programar manualmente subiendo los costes de creación y mantenimiento del editor, o la información redundante la debería introducir el usuario manualmente incrementando considerablemente la carga de trabajo. Por tanto, el denominador común es la falta de mecanismos automáticos que garanticen el mantenimiento de la consistencia de las estructuras redundantes en las instancias de los metamodelos, lo que implica que se suela elegir la representación no-redundante en los metamodelos de la ISOA.

El apartado 4.2 describe la instancia del armazón para la herramienta IDK y el apartado 4.3 presenta el metamodelo Objetos definido para el editor de procesos de ISOA. Finalmente, el apartado 4.4 indica las conclusiones obtenidas de la experiencia de aplicación de la guía para la definición de metamodelos en el contexto de la ISOA.

4.2 Aplicación del Armazón para el IDK

El primer caso de estudio surge de una limitación del IDK: el uso de un formato XML propietario para almacenar los modelos de INGENIAS. Esta limitación dificulta el posible procesamiento de sus modelos con herramientas externas, tales como las herramientas para TMs. Por ejemplo, ATL requiere que los metamodelos de entrada y salida estén expresados con ECore. Este aspecto fue el origen del proceso de modernización del IDK. Teniendo en cuenta las recomendaciones de ADM (Newcomb, 2005), la primera fase de la modernización (i.e. mejora del IDK para aplicar nuevas tecnologías) fue crear un metamodelo para el LM de INGENIAS. Las instancias de este metamodelo son las especificaciones de INGENIAS. Este metamodelo ha sido definido con la guía propuesta en el anterior capítulo. Además, este trabajo también presenta una versión beta del IDK que almacena los modelos como instancias del metamodelo presentado. Tanto la versión del metamodelo que sigue la guía propuesta en esta tesis como la versión del IDK que trabaja con él se encuentran disponibles en la web de *Grasia* (Grasia, 2009c).

El primer paso para proporcionar un metamodelo para el IDK fue elegir la configuración apropiada entre las descritas en el apartado 3.2.3. Dado el tipo de información a considerar en los proyectos IDK (sintaxis abstracta, vistas, disposición espacial, e información específica de la herramienta), se eligió la Configuración 4. De acuerdo con esta configuración, el modelo para representar

el proyecto completo está dividido en cuatro modelos que son instancias de los metamodelos Objetos, Vistas, Layouts, y Proyecto.

Los siguientes sub-apartados describen cada uno de los metamodelos usados en el armazón para modelar el IDK.

4.2.1 Metamodelo *Objetos* para INGENIAS

El diseño del metamodelo siguió el proceso del apartado 3.3.4. La instanciación de las actividades de dicho proceso fue como sigue:

- *Actividad 1, características estructurales.* El lenguaje INGENIAS tiene todas las características estructurales mencionadas en el apartado 3.3.4. Primero, hay relaciones con atributos. Este es el caso de la relación *WF-Consumes* entre una *Tarea* y un *Hecho* con el atributo *condición*, que indica si la ejecución de una *Tarea* consume un *Hecho* del estado mental de un agente. Segundo, también tiene extremos de relaciones con atributos. Por ejemplo, la Figura 4.1 muestra el atributo de *multiplicidad* relacionado con los extremos. Finalmente, INGENIAS también incluye relaciones n-arias, como la relación *UIInitiates*. Aunque actualmente no todas las relaciones de INGENIAS tengan todas estas características estructurales, se espera que algunas de ellas incluyan nuevas características estructurales en el futuro.

- *Actividad 2, representación homogénea o heterogénea.* El LM presenta varias de las propiedades para las cuales el apartado 3.3.4 recomienda una representación homogénea. Primero, el lenguaje INGENIAS no es todavía estable y nuevas relaciones o características estructurales pueden ser añadidas en el futuro; por tanto, el metamodelo debe poder adaptarse a las nuevas necesidades. Segundo, se realizarán varios tipos de procesamientos sobre los modelos de IDK, tales como las TMs presentadas en el siguiente capítulo. Estos dos objetivos son más fáciles de conseguir con la opción homogénea.

- *Actividad 3, representación redundante o no-redundante.* El IDK usa sus propias estructuras internamente para acceder a los datos. Por tanto, no se necesita facilitar la navegación de modelos con estructuras adicionales en el metamodelo. Además, con la representación no-redundante, las herramientas no requieren mecanismos específicos para mantener la consistencia en M1. Por estos dos motivos, la representación no-redundante es más conveniente para el IDK.

La anterior información se toma como entrada de la Actividad 4, que es el proceso de definición del metamodelo. La Figura 3.15 detalló esta actividad. De acuerdo con los anteriores datos, el proceso de definición se puede resumir como sigue:

- *Actividad 4.1.* La guía propone la única representación para las entidades.

- *Decisión 4.2.* Ninguna de las relaciones tiene una representación asignada.

- *Actividad 4.3.* Los diseñadores eligen una relación compleja. De acuerdo con la información de la Actividad 1, se usan relaciones n-arias con atributos tanto en el cuerpo como en los extremos, como por ejemplo la relación *UIInitiates*.

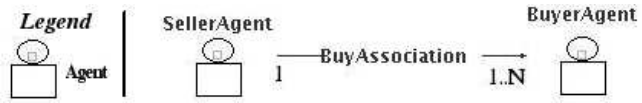


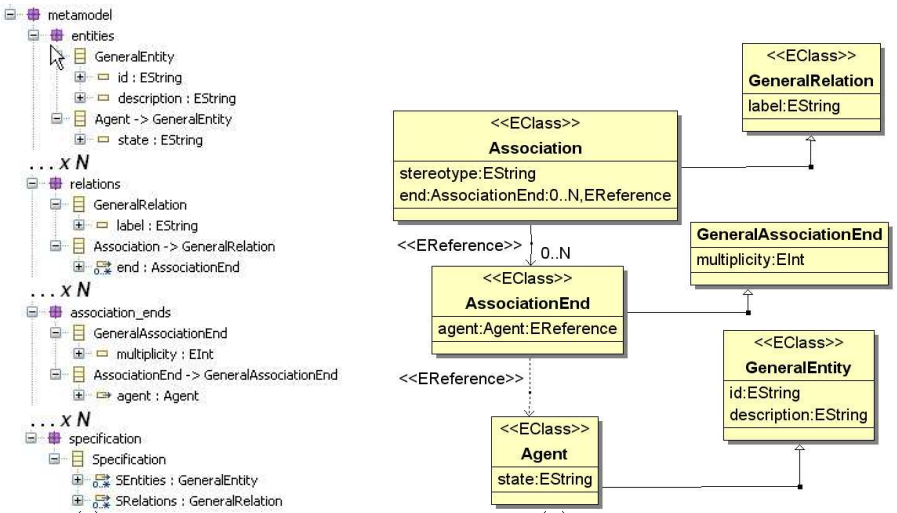
Figura 4.1: Modelo INGENIAS. Relación N-aria entre agentes.

- *Decisión 4.4.* La relación tiene atributos en los extremos y, por tanto, el proceso se dirige a la Actividad 4.5.
- *Actividad 4.5.* La guía recomienda la representación *EClass*-cuerpo-extremo para la relación, porque se considera la más apropiadas para las relaciones con atributos.
- *Decisión 4.10.* De acuerdo con la decisión de la Actividad 3, se adopta una representación no-redundante.
- *Actividad 4.12.* La relación usa la representación *EClass*-cuerpo-extremo no-redundante.
- *Decisión 4.13.* De acuerdo con la decisión de la Actividad 2, los diseñadores adoptan una representación homogénea. Como se ha encontrado una relación que usa la representación más compleja, i.e. *EClass*-cuerpo-extremo, entonces se para el bucle del proceso de definición, y todas las relaciones del metamodelo usan una representación *EClass*-cuerpo-extremo no-redundante.

La Figura 4.1 y la Figura 4.2 ilustran una relación en el metamodelo resultante. La Figura 4.1 contiene un extracto de un modelo en INGENIAS (Pavón et al., 2005) con una relación *BuyAssociation* entre agentes que puede conectar más de dos agentes. El extracto del metamodelo INGENIAS en la Figura 4.2, que es un ejemplo de la representación *EClass*-cuerpo-extremo, puede definir este modelo. Este metamodelo incluye una entidad llamada *Agent* y una relación llamada *Association*. Los extremos de la relación *Association* se llaman *AssociationEnd*. Las entidades, relaciones y los extremos de las relaciones pueden tener atributos. Por ejemplo, la entidad *Agent* tiene el atributo *state* y el extremo *AssociationEnd* tiene el atributo *multiplicity* heredado del elemento *GeneralAssociationEnd*.

La Figura 4.3 describe la instanciación de este ejemplo en la arquitectura de meta-datos de MOF (véase el apartado 2.2.1). Los elementos de meta-modelado de la capa M2 (véase la Figura 4.2) son instancias de los elementos ECore de la capa M3. Posteriormente, los elementos de modelado en la capa M1 (véase la Figura 4.1) son instancias de los elementos de meta-modelado de la capa M2. Por ejemplo, el elemento *SellerAgent* es una instancia de la entidad *Agent*; los extremos *AE1* y *AE2* de la asociación son las *EClass* de M1 que son instancias de la *EClass AssociationEnd* de M2; el cuerpo de la relación *BuyAssociation* es una instancia de la relación *Association*; finalmente, el atributo *state* de *SellerAgent* es una instancia del meta-atributo *state* de la *EClass* de M2 *Agent*. Los tres elementos *AE1*, *BuyAssociation* y *AE2* juntos representan una relación *Association* como se puede observar en la Figura 4.3.

Para ejemplificar la instanciación de este metamodelo del armazón y de los siguientes metamodelos, se toma un ejemplo de especificación del IDK; en con-



(a) Notación EMF (b) Notación Visual

Figura 4.2: Representación *EClass*-cuerpo-extremo para la relación n-aria de la Figura 4.1

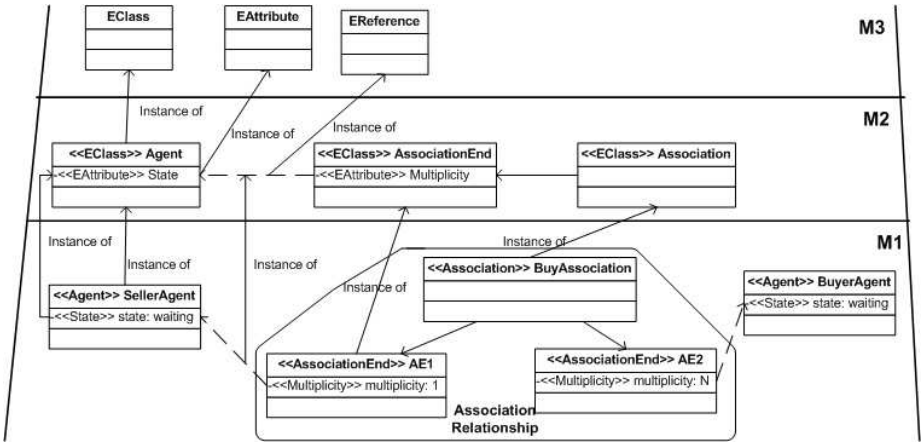


Figura 4.3: Instanciación de una relación entre agentes, en las capas de la arquitectura de meta-datos de MOF.

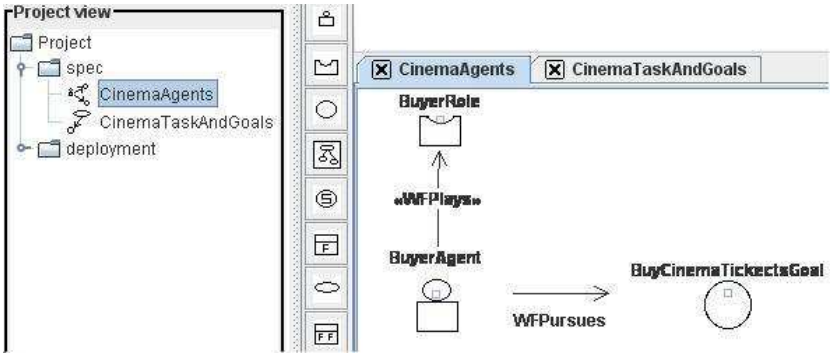


Figura 4.4: Ejemplo del Cine. Diagrama *CinemaAgents*

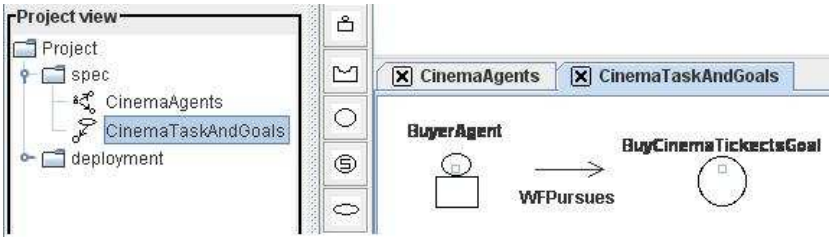


Figura 4.5: Ejemplo del Cine. Diagrama *CinemaTasksAndGoals*

creto, se escoge el ejemplo del *Cine*, que forma parte de la distribución del IDK 2.7 y está disponible en la web de *Grasia* (Grasia, 2009b). Este proyecto especifica un SMA que ayuda al usuario a comprar entradas de cine. Las Figuras 4.4 y 4.5 presentan capturas de pantalla de la herramienta IDK para este proyecto. Este proyecto tiene dos paquetes, llamados *spec* y *deployment* (véase la Figura 4.4). El paquete *spec* contiene los diagramas *CinemaAgents* (véase la Figura 4.4) y *CinemaTaskAndGoals* (véase la Figura 4.5). Estos diagramas incluyen varias entidades (el agente *BuyerAgent*, el rol *BuyerRole*, y el objetivo *BuyCinemaTicketsGoal*) y las relaciones que las conectan. Las entidades *BuyerAgent* y *BuyCinemaTicketsGoal*, y la relación *WFPursues-buyer* aparecen en ambos diagramas. Estos elementos son únicos, en el sentido de que sólo hay una instancia de cada uno, pero son mostrados en diferentes diagramas.

En cuanto a la instanciación del metamodelo *Objetos* para el IDK, la Figura 4.6 muestra el modelo Objetos que contiene las entidades y relaciones del modelo. Las entidades del modelo (en la capa M1) son *BuyerAgent, BuyerRole* y *BuyCinemaTicketsGoal*. Estas entidades del modelo son respectivamente instancias de las entidades *Agent, Role* y *Goal* del metamodelo (en la capa M2). De una manera similar, las relaciones *WFPlays-buyer* y *WFPursues-buyer* del modelo son respectivamente instancias de las relaciones *WFPlays* y *WFPursue* del metamodelo. Como un ejemplo de la representación de las relaciones en el modelo Objetos, la relación *WFPlays-buyer* contiene dos extremos (i.e. *WFPlayssource* y *WFPlaystarget*) que se conectan respectivamente a las entidades *BuyerAgent* y *BuyerRole*. La relación se conecta a los extremos por medio de *EReferences containment* de M1, y estas conexiones se simbolizan usando una estructura de árbol. Cada extremo está conectado a la correspondiente entidad

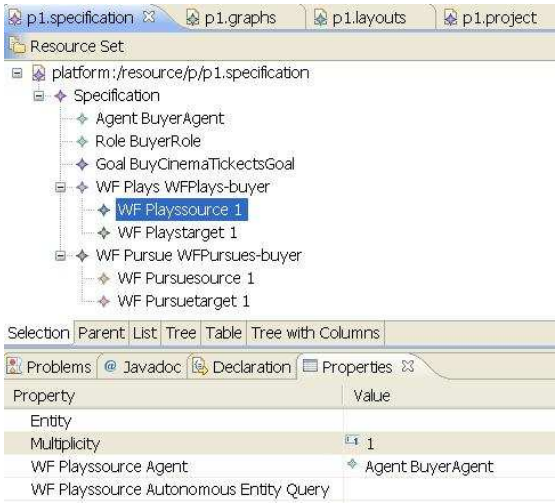


Figura 4.6: Instanciación del metamodelo *Objetos*

con una *EReference non-containment* de M1. Como ejemplo, la parte inferior de la Figura 4.6 muestra esta conexión para el extremo *WFPlayssource*.

El metamodelo original de INGENIAS fue especificado con GOPRR (Grafo, Objeto, Propiedad, Relación, y Rol) (Kelly, 1997). La migración al nuevo metamodelo expresado con ECore usó una transformación XSLT (Clark et al., 1999c) para automatizar la migración dado el alto número de entidades y relaciones en el LM, tal y como se detalla en (García-Magariño y Fuentes-Fernández, 2009). XSLT había sido usado anteriormente para traducciones de metamodelos. Por ejemplo, Gerber y Raymond (2003) propusieron XSLT para migrar modelos entre MOF (OMG, 2006b) y EMF (Budinsky, 2003).

4.2.2 Metamodelo *Vistas* del IDK

Dado que el apartado 3.2.1 presentó como ejemplos el metamodelo *Vistas* y los siguientes metamodelos del armazón para el IDK, en el apartado actual y los siguientes apartados se insistirá más en la parte de instanciación de M1. En concreto, para el metamodelo *Vistas*, se presentaron dos alternativas, y este apartado usa la alternativa con tipos de diagrama. El metamodelo correspondiente del IDK aparecía en la Figura 3.4.

La Figura 4.7 corresponde al modelo *Vistas*, instancia del metamodelo *Vistas*, que contiene diferentes vistas del modelo Objetos, representadas como grafos. En este ejemplo, hay dos grafos, uno para cada diagrama. El nombre del primer diagrama (en la capa M1) es *CinemaAgents-G* y es una instancia del tipo de diagrama *GAgentDiagram* (en la capa de M2). De manera similar, el diagrama *CinemaTaskAndGoals-G* es una instancia del tipo de diagrama *GTaskAndGoals*. Como se indicó en el apartado 3.2.1, el modelo Vistas contiene elementos de grafos apuntando a los elementos del modelo Objetos. Por ejemplo, las propiedades del elemento de grafo *BuyerRoleAGE*, que se muestran en la parte inferior de la Figura 4.7, indican que contiene una EReference de M1 *GeadRole* que apunta la entidad *BuyerRole*. Además, hay una propiedad *GeTy-*

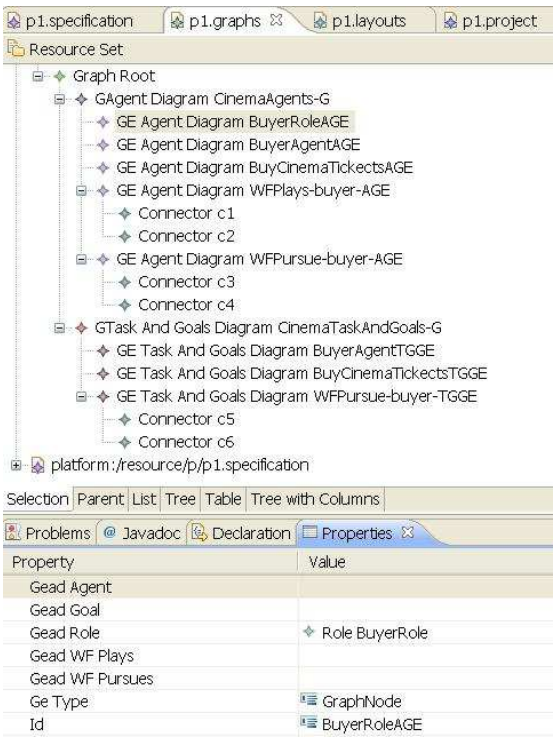


Figura 4.7: Instanciación del metamodelo *Vistas*.

pe para indicar el tipo del elemento de grafo, bien sea un *nodo* o una *arista*. Esta representación permite que varios elementos en grafos diferentes apunten al mismo objeto del modelo Objetos. Por ejemplo, los elementos *BuyerAgentAGE* y *BuyerAgentTGGE* pertenecen a diferentes grafos en M1, pero apuntan a la misma entidad *BuyerAgent* de M1.

4.2.3 Metamodelo *Layouts* del IDK

La representación espacial del modelo de las *vistas*, se almacena en el modelo *Layouts*, instancia del metamodelo *Layouts*. En el anterior capítulo, la Figura 3.5 mostró el metamodelo *Layouts* para el IDK; en este apartado, la Figura 4.8 muestra un ejemplo de instancia de dicho metamodelo para un modelo del IDK. Un ejemplo de las propiedades de los elementos del modelo Layout se puede ver en la parte inferior de la Figura 4.8, para el elemento de distribución *L1*. Este ejemplo de elemento para la distribución espacial esta enlazado con el elemento de grafo *BuyerRoleAGE* a través de la EReference *LGraphElem* en M1. Las propiedades de la distribución espacial proporcionan la posición (atributos *Lx* y *Ly*) y la dimensión (atributos *Lheight* y *Lwidth*) para el elemento de la vista.

4.2.4 Metamodelo *Proyecto* del IDK

Finalmente, el modelo *Proyecto* en la Figura 4.9 contiene información para la herramienta CASE. El elemento *Project* nombrado *Project-example* tiene varios

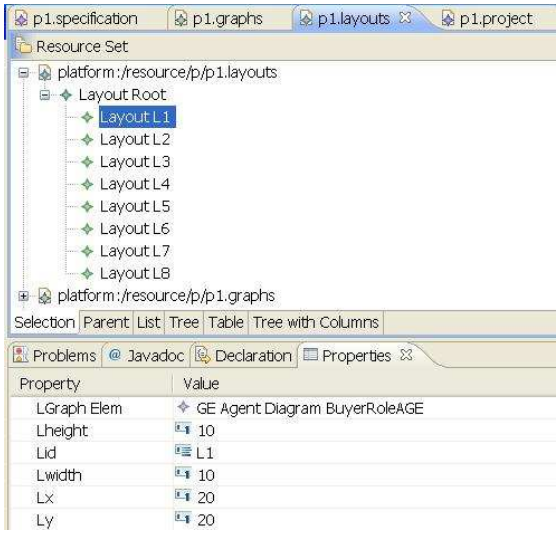


Figura 4.8: Instanciación del metamodelo *Layouts*.

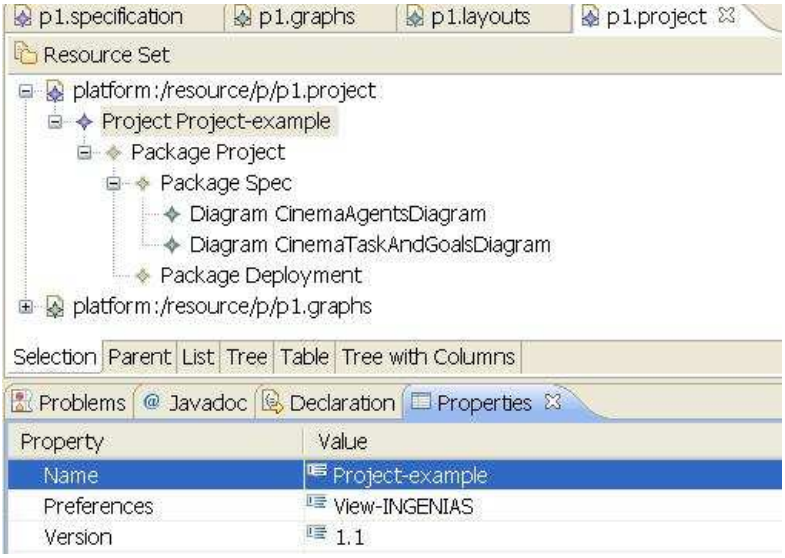


Figura 4.9: Instanciación del metamodelo *Proyecto*.

atributos, correspondiente a la información necesaria para el IDK, los cuales se presentan en la columna de la izquierda de la Tabla 4.1, y son los siguientes: su nombre, las preferencias del usuario para la notación (INGENIAS o UML), la versión del proyecto, y las rutas de varias carpetas necesarias para un proyecto del IDK (véase la Tabla 4.1 para más detalles). Algunos de estos atributos se pueden ver en la parte inferior de la Figura 4.9. La Tabla 4.1 presenta además de todos los atributos que el IDK contiene, los atributos que se planean incluir en el IDK por encontrarse información similar en otras herramientas.

El IDK usa el nombre del proyecto para identificarlo frente a otros. En cuanto a la notación, si bien nuestro grupo de investigación ha divulgado la notación específica de INGENIAS que se ha usado en los diagramas anteriores, la notación de UML 2.0 está mucho más extendida. Por ello, INGENIAS proporciona una notación basada en UML 2.0 que usa clases *estereotipadas* para denotar los diferentes elementos de modelado de su lenguaje. Cada usuario puede tener unas preferencias determinadas, incluida la notación a emplear que se puede conmutar en el IDK, y dichas preferencias se guardan en los modelos específicos de la herramienta CASE. También se incluye un número de versión del proyecto, que empieza por 1.1.

En cuanto a las rutas de carpetas, el IDK dedica generalmente una carpeta dentro de su espacio de trabajo para cada proyecto de un SMA. Para dar cuenta de esta carpeta, se almacena un atributo en el modelo llamado *ruta principal de todo el proyecto*. Como se indicó en el apartado 2.1.1, el IDK sigue una aproximación DSDM donde una parte del código se genera automáticamente a partir de los modelos y otra parte del código se crea manualmente por el programador. Además también se genera documentación HTML. Por tanto, dentro de la carpeta principal existe: una carpeta donde se genera el código de los agentes y se sobrescribe en cada generación; una carpeta donde se genera el código de ciertas aplicaciones internas/externas, eventos y otros recursos, que sólo se escribe en la primera generación dado que se espera que el programador cambie el código de dichos elementos manualmente; una carpeta de código fuente donde el programador puede añadir todo el código extra que necesite; y una carpeta donde se genera la documentación de HTML. Por defecto, estas carpetas se llaman respectivamente: *gensrc, permsrc, src* y *html*. Sin embargo, el usuario puede cambiar dichos nombres por defecto, y las nuevas rutas se almacenan en diferentes atributos del proyecto (véase la Tabla 4.1).

Además, el modelo *Proyecto* contiene la estructura de paquetes de las especificaciones. En este ejemplo de instanciación (recuérdese la Figura 4.4 y véase la Figura 4.9), el paquete *project* contiene dos paquetes: *spec* y *deployment*. Por ejemplo, el paquete *spec* contiene dos diagramas, *CinemaAgentsDiagram* y *CinemaTaskAndGoalsDiagram*, cada uno de los cuales tiene una referencia respectivamente a los grafos *CinemaAgents-G* y *CinemaTaskAndGoals-G* del modelo Vistas.

Finalmente, observando la información que incluyen otras herramientas CASE (presentada en el apartado 2.2.8), este trabajo sugiere que se pueden incluir nuevas propiedades en los proyectos del IDK, las cuales se recogen en la columna derecha de la Tabla 4.1. Entre otras propiedades, está la descripción del SMA del proyecto, que resulta útil para que otros diseñadores entiendan el modelo. Además, almacenar los nombres de los autores del proyecto sirve para que otros diseñadores conozcan de quién es el trabajo, e indicar el autor que hizo la última modificación facilitaría el desarrollo del SMA, al identificar a la persona con un

Propiedades que se incluyen actualmente	**Propiedades que se plantean incluirse**
- Nombre del proyecto - Notación que prefiere el usuario (INGENIAS / UML) - Versión del proyecto - Ruta donde el IDK encuentra los módulos - Ruta donde se generan el código de los agentes JADE - Ruta de la carpeta con el código de programación externo - Ruta principal de todo el proyecto - Ruta con el código de los elementos que sólo se generan una vez - Ruta donde se genera la documentación HTML - Estructura del proyecto con paquetes anidados	- Descripción del SMA - Autor que hizo la última modificación en el proyecto - Autores del proyecto - Versión del IDK con la que se hizo el proyecto - Tamaño de la letra en los diagramas - Tamaño de página para presentar los diagramas - Historial de cambios

Tabla 4.1: Propiedades del metamodelo *Proyecto* del IDK.

conocimiento más reciente del modelo. Además, también es interesante conocer la versión del IDK con la que se hizo el proyecto para que, en la medida de lo posible, se use con la misma versión que fue hecha o, en caso de utilizar un IDK de una versión superior, los diseñadores estén prevenidos ante problemas debidos al cambio de versión de la herramienta. También se podría incluir cierta información de presentación, tal como el tamaño de letra y el tamaño de página para los diagramas. Por último, también sería útil incluir un historial de cambios, para permitir deshacer los cambios no deseados.

4.3 Metamodelo para procesos en la Ingeniería del Software Orientada a Agentes

La *Foundation for Intelligent Physical Agents* (FIPA) (FIPA, 2000) es una organización para el desarrollo de estándares relacionados con SMAs (Weiss, 1999). Su *comité tecnológico para metodologías* (Cossentino et al., 2005) recomienda la definición de modelos de procesos en la ISOA. Con el objetivo de crear estos modelos, FIPA recomienda el uso de SPEM (OMG, 2008c).

Este trabajo de tesis incluye un editor de procesos de ISOA (García-Magariño et al., 2009b), que se ha realizado en colaboración con la Universidad de Vigo. La necesidad de este editor surgió de la falta de elementos de modelado en las herramientas existentes para SPEM, tales como *Eclipse Process Framework* (EPF) (EPF, 2009) y *A Process Engineering Software* (APES2) (IPSquad, 2009). Por ejemplo, EPF carece de los elementos de modelado *componente de un proceso* y *paso*, que se necesitan para definir el proceso SCRUM en INGENIAS, descrito

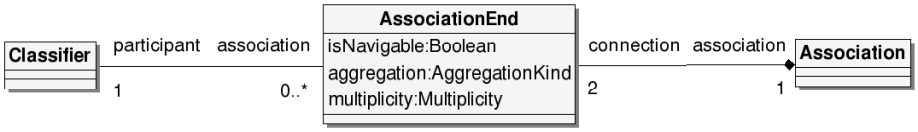


Figura 4.10: Una parte del metamodelo de SPEM con MOF. La relación *Association* de la especificación de SPEM.

en (García-Magariño et al., 2009b), y el proceso de *Agile PASSI* (Chella et al., 2006). En APES2 faltaban elementos tales como *ciclo de vida, fase* y *guía*, que se necesitan en la definición de la mayoría de procesos. Además, en la ISOA surgen nuevos procesos con nuevas necesidades de modelado, debido a la intensa actividad de investigación en este campo. Por tanto, se decidió que el editor tuviera todos los conceptos de SPEM y que el metamodelo de su LM fuera fácilmente modificable, siendo menos importante la parte gráfica. Este editor se creó con una aproximación del DSDM, en la que se generó automáticamente el editor a partir del metamodelo. Dicho metamodelo se definió con la guía presentada en el capítulo anterior.

El primer paso fue la elección de la configuración apropiada del armazón. Dado que el principal objetivo del editor era tener una sintaxis abstracta apropiada del LM (i.e. particularizable para ciertos aspectos de la ISOA), en vez de tener una interfaz visual, se eligió una configuración sencilla, i.e. Configuración 1, la cual sólo tiene el metamodelo *Objetos*.

SPEM (OMG, 2008c) es una especificación de OMG, por lo que su metamodelo está descrito con MOF (OMG, 2006b). La Figura 4.10 muestra parte de este metamodelo. Dado que nuestro trabajo usa el soporte de EMF (Budinsky, 2003) para generar el editor de procesos de ISOA, se requirió redefinir SPEM en términos del lenguaje ECore. Los diseñadores determinaron que el metamodelo de SPEM debía tener las siguientes propiedades según la guía:

- *Actividad 1, características estructurales.* La especificación de SPEM (OMG, 2008c) describe que las relaciones heredan del elemento general *ModelElement*, que tiene atributos. Además, los extremos de las relaciones incluyen, entre otros, atributos para la *navegabilidad* y la *multiplicidad* (véase la Figura 4.10). Por tanto, el lenguaje de SPEM tiene relaciones con atributos tanto en el cuerpo de la relación como en los extremos. Respecto al número de extremos de las relaciones, SPEM sólo contiene relaciones binarias.

- *Actividad 2, representación homogénea o heterogénea.* Dado que OMG especifica el metamodelo de SPEM por medio de estándares, su elaboración implica largos periodos de tiempo y los cambios son lentos. Sin embargo, SPEM no es un estándar cerrado y el objetivo de la versión ECore de este metamodelo es su uso en herramientas. El apartado 3.3.4 aconseja la representación homogénea para metamodelos con estas características.

- *Actividad 3, representación redundante o no-redundante.* EMF, que fue usado para generar automáticamente un editor a partir del metamodelo, no proporciona un mecanismo para garantizar la consistencia de las representaciones redundantes en M1. Por tanto, de acuerdo con las recomendaciones de la guía, se elige la representación no-redundante.

Los diseñadores usan esta información para definir el metamodelo en la Actividad 4 (véase la Figura 3.15 para los pasos detallados), tal y como se indica a continuación:

- *Actividad 4.1.* Hay una representación única para las entidades.
- *Decisión 4.2.* Ninguna de las relaciones tiene una representación asignada.
- *Actividad 4.3.* Los diseñadores escogen una relación compleja, i.e. la relación *Association*. De acuerdo con la información de la Actividad 1, SPEM sólo tiene relaciones binarias pero éstas pueden tener atributos tanto en el cuerpo como en los extremos, como la relación escogida (véase la Figura 4.10).
- *Decisión 4.4.* Los extremos de las relaciones tienen atributos y esto lleva el proceso a la Actividad 4.5.
- *Actividad 4.5.* La relación requiere la representación *EClass*-cuerpo-extremo.
- *Decisión 4.10.* La Actividad 3 eligió la representación no-redundante.
- *Actividad 4.12.* La relación usa la representación *EClass*-cuerpo-extremo no-redundante.
- *Decisión 4.13.* La Actividad 2 eligió una representación homogénea. Por tanto, dado que se ha elegido la representación más compleja (i.e. *EClass*-cuerpo-extremo) para una relación (i.e. la relación *Association*), los diseñadores adoptan esta representación para todas las relaciones del metamodelo.

En este caso, la limitación de EMF de no gestionar automáticamente la información redundante implicó elegir una representación no-redundante para evitar un esfuerzo excesivo de desarrollo. La representación redundante se hubiera elegido si fuera indispensable un procesamiento de modelos que necesitara navegar por sus elementos en los dos sentidos de las conexiones; sin embargo, este no es el caso.

El metamodelo definido para SPEM en este trabajo está disponible en la web de *Grasia* (Grasia, 2009a). La Figuras 4.10, 4.11 y 4.12 ilustran este metamodelo con una relación con atributos tanto en su cuerpo como en sus extremos. La Figura 4.10 muestra la especificación original de la relación *Association* en la especificación de SPEM (OMG, 2008c) con MOF (OMG, 2006b). La especificación de la relación incluye su cuerpo, llamado *Association*, y sus extremos, llamados *AssociationEnd*. Las Figuras 4.11 y 4.12 muestran la representación *EClass*-cuerpo-extremo de esta relación en el nuevo metamodelo expresado con ECore. La *EClass* de M2 *Association* representa el cuerpo de la relación y la *EClass* de M2 *AssociationEnd* representa los extremos de la relación. Las *EReferences* de M2 *sourceConnection* y *targetConnection* enlazan el cuerpo de la relación con ambos extremos. Estas *EReferences* de M2 se sitúan en la *EClass* de M2 *Association*. La *EReference* de M2 *participant* permite enlazar los extremos de la relación con las entidades correspondientes.

Usando EMF, un editor de SPEM fue generado desde el metamodelo, sin ningún esfuerzo de desarrollo adicional. Se debe aclarar que este editor es un editor estándar de EMF en forma de árbol. Sin embargo, este editor contempla

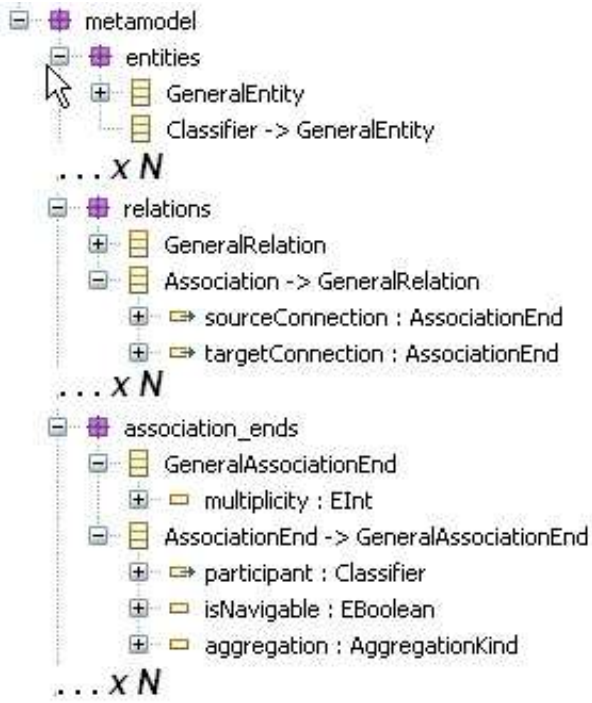


Figura 4.11: La relación *Association* con notación EMF para ECore

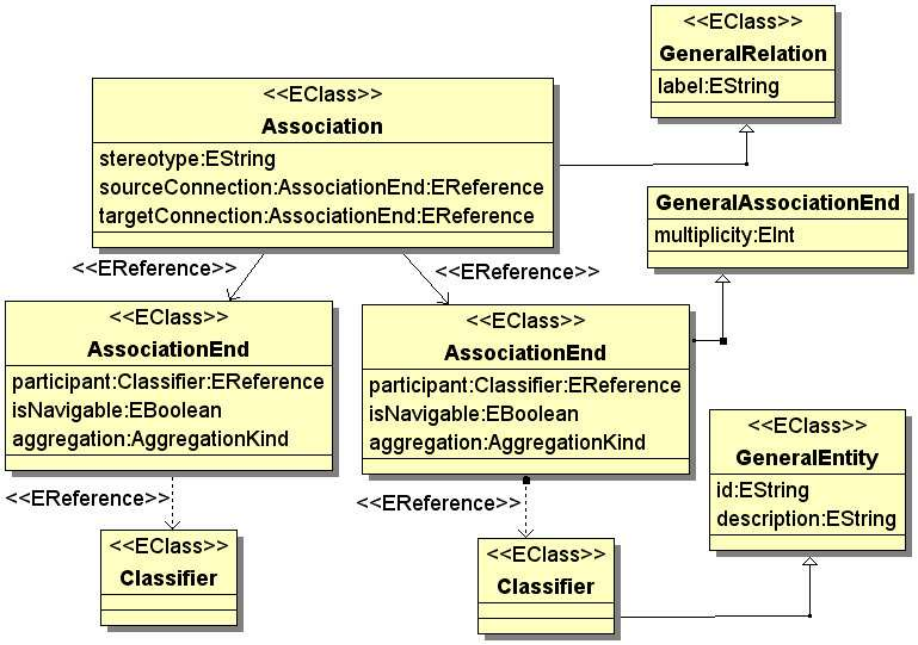


Figura 4.12: La relación *Association* con notación visual para ECore

más conceptos que el resto de editores para SPEM, siendo algunos de estos conceptos esenciales para el modelado de procesos en la ISOA tal y como se indicó en (García-Magariño et al., 2009b). Por ejemplo, APES2 (IPSquad, 2009) no permite definir *disciplinas* ni *guías* en los procesos software, lo cual se considera clave en la definición de procesos de ISOA, tal y como se justifica en (García-Magariño et al., 2009b). Por otro lado, si bien EPF (EPF, 2009) cuenta con una gran comunidad de soporte, tampoco incluye ciertos elementos de modelado que nuestro editor sí incluye, tales como: *objetivos, precondiciones, componentes de un proceso*, y los elementos referentes a la *transición de estados*.

4.4 Conclusiones

Como se ha podido observar en este capítulo, la guía de definición de metamodelos del Capítulo 3 se ha podido aplicar para definir metamodelos con objetivos variados dentro de la ISOA. En primer lugar se ha modernizado el IDK, de tal forma que puede almacenar sus modelos como instancias de metamodelos expresados con ECore. Esto permite, por ejemplo, que se les puedan aplicar transformaciones expresadas en lenguajes específicos de TMs, tales como ATL. En segundo lugar, la guía ha permitido definir un metamodelo para la generación de una herramienta de modelado. Este es el caso de la definición de SPEM con ECore, con el que se ha generado automáticamente un editor de procesos de ISOA. Además, el armazón presentado en esta tesis se ha instanciado con varias configuraciones diferentes, es decir, diferentes conjuntos de metamodelos: una configuración compleja (i.e. para el IDK) y otra sencilla (i.e. para el editor de SPEM).

Como resultado de la experimentación de este capítulo, se observan los siguientes beneficios y conclusiones:

- La guía proporciona una serie de actividades para la definición de metamodelos, indicando pautas para que los diseñadores realicen cada actividad. Además, se presenta un flujo de actividades que guía al diseñador en la definición de los metamodelos, proporcionando recomendaciones para tomar las decisiones apropiadas. En concreto, la guía ha permitido definir dos metamodelos útiles en la ISOA: el metamodelo del IDK y el metamodelo del editor de procesos de ISOA.

- El armazón de la guía tiene diferentes configuraciones que permiten definir metamodelos para diferentes necesidades: modelado de una herramienta CASE completa, y un editor de modelado sólo para la sintaxis abstracta de un LM.

- En cuanto a la sintaxis abstracta, la representación homogénea usando la representación más flexible para todas las relaciones (i.e. la representación *EClass*-cuerpo-extremo) ha resultado ser la más adecuada para los metamodelos de ISOA contemplados, dado que esta representación es la que más fácilmente se adapta a la inclusión de nuevos requisitos estructurales en LMs en continua evolución. Además, la representación homogénea en esta guía complementa la manera usual de definir metamodelos del consorcio OMG, el cual usa una representación heterogénea para las relaciones de los LMs.

- La representación *no-redundante* se ha escogido en ambos metamodelos contemplados en la ISOA. El motivo principal es la ausencia de mecanismos para garantizar la consistencia en representaciones redundantes con implementaciones sencillas. Además, nuestros casos no tenían las necesidades de rendimiento en el procesamiento de los modelos que justificasen la inversión en el desarrollo. Por tanto, por el momento, la representación *no-redundante* ha resultado ser la más apropiada en ISOA, aunque no se descarta la posibilidad de que se use la representación *redundante* en el futuro cuando existan los mecanismos apropiados para garantizar la consistencia entre parejas de referencias en M1.

Capítulo 5

Algoritmo para la generación de transformaciones basadas en ejemplos

Este capítulo presenta un algoritmo para la generación de transformaciones basadas en ejemplos. Este algoritmo permite generar reglas de transformación muchos-a-muchos con transferencia de la información de los atributos. Además, permite tratar modelos prototipo con grafos no conexos. El algoritmo se define con un pseudo lenguaje con características comunes a gran parte de los lenguajes de transformación existentes. Como prueba de concepto, el algoritmo se ha implementado para ATL y se ha incluido en una herramienta disponible públicamente.

5.1 Introducción

El DSDM es una aproximación para el desarrollo del software en el que los modelos son el producto principal del desarrollo. Las TMs son un medio para automatizar el ciclo de vida de los artefactos en estos desarrollos. El DSDM considera transformaciones de modelo-a-modelo y modelo-a-texto. Sin embargo, este trabajo se centra en las transformaciones modelo-a-modelo, en las que una TM recibe como entrada un modelo, que es instancia de un metamodelo de entrada, y genera otro modelo, que es instancia de un metamodelo de salida.

La propuesta de desarrollo de transformaciones de este capítulo se enmarca en la *Generación de Transformación Basadas en Ejemplos* (GTBE) (Varro, 2006). En esta aproximación, las reglas de transformación son generadas automáticamente a partir de parejas de modelos prototipo. Los pasos habituales en la aproximación de GTBE fueron descritos en el apartado 2.3.4.

Este trabajo de tesis contribuye a la aproximación GTBE con un algoritmo que resuelve las siguientes limitaciones de las aproximaciones de GTBE existentes (véase el apartado 2.3.4): las asociaciones de atributos no permiten tomar información desde varios elementos para una misma regla; no trabajan con grafos no conexos para una misma regla; están ligadas a un LTM concreto.

La generación de reglas de transformación muchos-a-muchos del algoritmo presentado está basada en la creación de restricciones para la simulación de patrones de entrada con varios grafos. Este algoritmo también trabaja con asociaciones embebidas en los modelos para propagar la información desde varios elementos de la entrada a varios elementos de la salida de cada regla de transformación. Esto significa que son los mismos modelos prototipo los que indican cómo la información debe ser transferida entre los modelos, y que se supera la limitación de sólo tomar información de un único elemento de la entrada de cada regla en los mecanismos de asociación de atributos, propia de los trabajos de GTBE existentes.

Dado que este algoritmo de GTBE usa restricciones para los patrones de entrada, necesita que el lenguaje de transformación elegido para su implementación de soporte a esta característica. Los lenguajes de transformación comunes, tales como ATL (Jouault y Kurtev, 2006b), QVT (OMG, 2005a), Tefkat (Lawley y Steel, 2006), VIATRA2 (Varró y Balogh, 2007a) y XSLT (Clark et al., 1999b), satisfacen este requisito. Adviértase que los diseñadores no necesitan ser conscientes de cómo se escribe el código de las transformaciones. Ellos especifican transformaciones con parejas de modelos prototipo y la herramienta que implementa el algoritmo usa restricciones para generar automáticamente la transformación requerida.

Como prueba de concepto, esta investigación ha desarrollado una herramienta que implementa este algoritmo para ATL (Jouault y Kurtev, 2006b). Esta herramienta se llama *MTGenerator* y está disponible en la web de *Grasia* (Grasia, 2009d). Debido a los requisitos de ATL, esta herramienta usa el lenguaje ECore (Budinsky, 2003) para los metamodelos, y el lenguaje OCL (Warmer y Kleppe, 2003) para las restricciones. Las principales razones para la elección de ATL son su uso extendido y el soporte técnico que se ofrece para el mismo.

El resto del capítulo se organiza de la siguiente forma. El apartado 5.2 introduce el algoritmo para GTBE, mientras que el apartado 5.3 introduce su implementación particular con ATL. Finalmente, el apartado 5.4 discute las principales aportaciones del algoritmo.

5.2 El algoritmo de generación de transformaciones basadas en ejemplos

El algoritmo de GTBE presentado produce TMs que contienen reglas capaces de tratar transformaciones muchos-a-muchos entre grafos de elementos de modelado. La presentación de este algoritmo está dividida en varios sub-apartados. El apartado 5.2.1 proporciona algunas definiciones formales preliminares que los siguientes sub-apartados usan. El apartado 5.2.2 describe el algoritmo, que usa las funciones detalladas en los apartados 5.2.3, 5.2.4, 5.2.5 y 5.2.6.

5.2.1 Definiciones formales para el algoritmo

Este apartado introduce las definiciones sobre las nociones de metamodelo, modelo y elemento consideradas en el algoritmo de GTBE.

Definición 3 *Un **modelo*** M *es una tupla* $M := < E, G, R, A, O_R, O >$ *con los siguientes componentes:*

- **E**, *llamado* ***elementos***, *es un conjunto de tuplas* $\{< n, t > | n \in \mathcal{I}, t \in \mathcal{T}\}$, *donde* $\mathcal{I}$ *es el conjunto de identificadores válidos y* $\mathcal{T}$ *el conjunto de tipos válidos.*

- **G** *es un conjunto de tuplas* $\{< e_1, e_2, n, t > | e_1, e_2 \in \mathbf{E}, n \in \mathcal{I}, t \in \mathcal{T}\}$ *llamadas* ***relaciones de agregación***. *La relación de agregación (*agregación *si se abrevia) con identificador n y tipo t indica que el elemento* e_2 *es parte del elemento* e_1. *En este conjunto, no hay secuencias* $e_1 e_2 \ldots e_{n-1} e_n$ *tal que* $e_1 = e_n$ *y* $< e_i, e_{i+1}, n, t > \in \mathbf{G}$, *(i.e., no hay ciclos formados por relaciones de agregación).*

- **R** *es un conjunto de tuplas* $\{< e_1, e_2, n, t > | e_1, e_2 \in \mathbf{E}, n \in \mathcal{I}, t \in \mathcal{T}\}$ *llamadas* ***referencias***. *Denota una relación arbitraria desde* e_1 *a* e_2.

- ***A*** *es un conjunto de tuplas* $\{< e, v, n, t > | e \in \mathbf{E}, v \in t, n \in \mathcal{I}, t \in \mathcal{T}_\mathcal{P}, \}$ *llamadas* ***atributos***, *donde* e *es un elemento,* v *es un valor de un tipo primitivo,* n *denota el nombre del atributo, y* t *denota su tipo. El conjunto* $\mathcal{T}_\mathcal{P}$ *contiene todos los tipos primitivos como los números enteros, las cadenas de caracteres, los booleanos y otros.*

- $\mathbf{O_R} \in \mathbf{E}$ *es la* raíz del modelo *forzada por la serialización de los modelos que son expresados en XML para algunos de los LTMs vistos. Estos modelos tienen una única raíz, que aparece como una entidad distinguida en el modelo. Sin embargo, la raíz del modelo es opcional y depende del lenguaje seleccionado para la implementación. Por ejemplo, QVT usa modelos* basados en grafos, *que no tienen una raíz forzada por el modelo.*

- **O** *es un conjunto de elementos* $e \in E$ *tal que* $((\exists O_R \wedge \exists < O_R, e, n, t > \in G) \bigvee (\nexists O_R \wedge \nexists < e_2, e, n, t > \in G))$. *Estos elementos se denotan como* raíces lógicas.

Definición 4 *Un* ***metamodelo*** *declara cómo se construye un modelo. Dados los modelos* $M :=< E, G, R, A, O_R, O >$, *un metamodelo declara el conjunto de tipos válidos que pueden ser usados en* **E**. *También declara los conjuntos de los tipos válidos de relaciones de agregación y referencias,* **G** *y* **R**, *entre el conjunto de tipos válidos de entidades,* **E**. *Finalmente, declara los atributos de cada entidad en* **A**. *Un modelo instancia un metamodelo mediante la creación de elementos de los tipos declarados por el metamodelo y conectándolos por las relaciones permitidas por el metamodelo.*

Dadas las definiciones de metamodelo y modelo mencionadas, este trabajo considera las siguientes notaciones para trabajar con ellas.

Notación 1 *Dado un* ***modelo*** $M :=< E, G, R, A, O_R, O >$, *M.E, M.G, M.R, M.A,* $M.O_R$ *y M.O denotan respectivamente E, G, R, A,* O_R *y O en el modelo M. Cuando se aplica a un metamodelo MM, MM.E, MM.G, MM.R, MM.A indican respectivamente los tipos declarados en el metamodelo MM de E, G, R y A para su instanciación.*

Notación 2 *Dado un modelo* $M :=< E, G, R, A, O_R, O >$, *un* ***elemento*** $e \in \mathbf{E}$, *una* ***relación de agregación*** $g \in \mathbf{G}$, *y una* ***referencia*** $r \in \mathbf{R}$,

- *e.n y e.t, donde* $e =< n, t >\in E$*, denotan respectivamente el nombre y el tipo del elemento*
- *g.n y g.t, donde* $g =< e_1, e_2, n, t >\in G$*, denotan respectivamente el nombre y el tipo de la relación de agregación*
- *r.n y r.t, donde* $r =< e_1, e_2, n, t >\in R$*, denotan respectivamente el nombre y el tipo de la referencia*

Notación 3 *Dado un modelo* $M :=< E, G, R, A, O_R, O >$*, un* ***elemento*** $e \in E$*, una* ***relación de agregación*** $g \in G$*, y una* ***referencia*** $r \in R$*,*

- $e.G$ *denota* $\{< e_1, e_2, n, t >\in G : e = e_1\}$
- $e.R$ *denota* $\{< e_1, e_2, n, t >\in R : e = e_1\}$
- $e.A$ *denota* $\{< e_x, v_x, n, t >\in A : e = e_x\}$
- $e.g.e_2$ *denota* $e_2 : g =< e, e_2, n, t >\in G$
- $e.r.e_2$ *denota* $e_2 : r =< e, e_2, n, t >\in R$

Además de los modelos y los metamodelos, este trabajo necesita también razonar sobre los LTMs. Debido a que cada LTM tiene su propia notación, este trabajo adopta un pseudo LTM que captura las características comunes más relevantes de varios LTMs. La tabla 2.5 del capítulo 2 presentó las características de los LTMs siguientes: ATL (Bézivin et al., 2003a; Jouault y Kurtev, 2006b), MT (Tratt, 2007), Tefkat (Lawley y Steel, 2006), VIATRA2 (Varró y Balogh, 2007a), MOLA (Kalnins et al., 2006) y RubyTL (Cuadrado et al., 2006). Como se podrá observar, los LTMs estudiados tienen disponibles todas las características presentes en esta gramática para las TMs. Algunos de estos LTMs no incluían la posibilidad de múltiples entradas y, por tanto, la gramática general sólo permite un único elemento de entrada en cada regla. Sin embargo, el algoritmo presentado permite tener múltiple elementos de entrada mediante el uso de restricciones.

La gramática que define este pseudo LTM se describe usando EBNF (ISO/IEC JTC1, 1996) donde las palabras resaltadas en negrita representan los símbolos terminales, las palabras no resaltadas representan los símbolos no-terminales, y ε representa la cadena vacía. Una TM se define con el símbolo MT como se indica a continuación:

1: MT::= **header** DefVariables **begin** Rules **end**
2: DefVariables::= DefVariable DefVariables
DefVariables::= ε
DefVariable::=**Define** Name : Type
3: Type ::= List
4: Rules ::= Rule Rules
Rules ::= Rule
5: Rule::= **rule** InputElement (Constraint) **to** OutputElements
imperative Actions **endrule**
6: InputElement::= ElementType
7: Constraint::= Constraint **and** Constraint
Constraint::= Constraint **or** Constraint

Constraint::= **exist**(Element)
Constraint::= **attributeConstraint**(Attribute)
Constraint::= **aggregationConstraint**(Aggregation)
Constraint::= **referenceConstraint**(Reference)
Constraint::= **allElementsOfType**(ElementType)
Constraint::= **BasicRootConstraint**
Constraint::= Expression
8: OutputElements::= ElementType (Actions) , OutputElements
OutputElements::= ElementType (Actions)
9: Expression::= Element→Attribute→Expression
Expression::= Element→Aggregation→Expression
Expression::= Element→Reference→Expression
Expression::= Element
Expression::= Value
10: Actions ::= Actions Action
Actions ::= Action
11: Action ::= **assign**(Aggregation, Element)
Action ::= **assign**(Attribute, Value)
Action ::= **assign**(Aggregation, Expression)
Action ::= **assign**(Attribute, Expression)
Action ::= **assign**(Variable, Expression)
Action ::= **assign**(Variable, Variable)
Action ::= **insert**(Variable, Variable)
Action ::= **insert**(Variable, Element)

Los símbolos *Element, ElementType, Attribute, Aggregation* y *Reference* se refieren a los elementos, tipos de elementos, atributos, relaciones de agregación y referencias descritos en la definición 3. El símbolo *Value* es un valor específico de un tipo. A una variable (representada con símbolo *Variable*) se le puede asignar cualquier valor de su tipo. Una lista (representada con el símbolo *List*) es un tipo de variable, que representa un lista ordenada de valores.

El símbolo *allElementsOfType* es una restricción del LTM que indica una selección de todos los elementos de un tipo determinado en un modelo. Por tanto, devuelve una lista de elementos de un mismo tipo, que se puede filtrar añadiéndole más condiciones. El símbolo *BasicRootConstraint* se refiere al elemento que es una raíz en el patrón de entrada de una regla y, por tanto, es el punto de origen para navegar a través de las relaciones y atributos a otros elementos del patrón de entrada. El operador lógico *and* tiene preferencia en el orden de evaluación frente al operador lógico *or*. Las restricciones de *atributo*, *agregación* y *referencia* (representados con los símbolos terminales *attributeConstraint, aggregationConstraint* y *referenceConstraint*) indican patrones que ciertos elementos deben satisfacer. Por ejemplo, se puede especificar que un elemento tiene un atributo con cierto valor o que está enlazado mediante una agregación con una entidad de un tipo dado. Las expresiones (representadas con el símbolo no-terminal *Expression*) corresponden a los caminos de navegación expresados en la notación 3.

La acción de asignación (con la palabra reservada *assign*) tiene dos campos. El primer campo es una variable, una relación de agregación, una referencia o un atributo; mientras que el segundo campo es una expresión (indicada en la gra-

mática con *Expression*) o una variable. La acción de asignación asigna el valor del segundo campo al primer campo. Obsérvese que el primer campo puede ser una relación de agregación, referencia y un atributo que, como se han definido anteriormente, son tuplas. En estos casos, sólo se altera un componente de la tupla. En el caso de la agregación y la referencia, sólo se altera el componente e_2 de la tupla según la definición 3. En el caso de los atributos, sólo se altera el componente v de la tupla según la definición 3. En el caso de que el primer campo sea una variable, entonces se le asigna el valor del segundo campo completo, incluso si el segundo campo es una tupla (e.g. un elemento). Como aclaración, algunos LTMs pueden asignar directamente a un elemento el valor de otro elemento, lo que en realidad sería una operación de reescritura. Sin embargo este tipo de asignación no es posible en varios LTMs, tales como ATL o XSLT; por tanto, no se incluye en este pseudo LTM y no se utiliza en nuestro algoritmo. La acción de inserción (con la palabra reservada *insert*) sólo se puede aplicar sobre variables que sean listas, y su semántica es que otra variable o elemento es insertado en la variable de la lista.

Finalmente, la descripción del algoritmo usa un lenguaje de pseudo código. La mayoría de sus primitivas son auto-explicativas. No obstante, algunas de ellas requieren una cierta explicación:

- La función **replace**(str,s,t) devuelve una cadena de caracteres con el mismo valor que *str* pero donde la cadena t reemplaza las ocurrencias de la cadena s.
- La notación <**no-terminal**> indica un tipo de cadena de caracteres derivable del no-terminal de la gramática anteriormente mencionada para TMs.
- Los nodos **terminales** de la gramática de TMs se resaltan en negro.
- El operador $\oplus$ concatena las palabras clave de una TM.

5.2.2 El algoritmo

El algoritmo presentado produce TMs desde parejas de modelos prototipo. Primero, genera la cabecera de la transformación; luego, genera una regla de transformación por cada pareja de modelos prototipo, respectivamente procesando el modelo prototipo origen y el modelo prototipo destino; por último, se genera una última regla que ubica los elementos generados por las otras reglas.

El algoritmo genera el código necesario para la ubicación explícita de los elementos de salida de las reglas. La ubicación se consigue creando unas variables globales en la transformación que almacenan todos los elementos de las salidas de las reglas, y los elementos de estas variables globales se añaden al modelo de salida con otra regla de transformación. Esto es necesario porque el algoritmo genera reglas de transformación muchos-a-muchos y este tipo de reglas necesita esta ubicación explícita en la mayoría de los LTMs dado que dichas reglas tienen varios elementos de salida. Este problema de *encaje múltiple* se explicó en el apartado 2.3.5.

Además, el algoritmo genera restricciones para manejar patrones de entrada múltiple. Aunque algunos LTMs dan soporte directamente a estos patrones, no es la norma y la mayoría de los LTMs usan restricciones para este propósito

(véase la Tabla 2.5). Tales restricciones están presentes en la definición de la gramática para TMs en el apartado 5.2.1.

El algoritmo también incluye un mecanismo de asociación de atributos. Para cada pareja de modelos prototipo, el modelo destino puede incluir enlaces al modelo prototipo origen, por medio de los identificadores de los elementos del modelo origen. Dado que el algoritmo asume que el LTM puede permitir sólo reglas con un elemento de entrada, el algoritmo genera expresiones que referencian a los elementos del modelo prototipo origen desde el único elemento de entrada de cada regla si es posible. Para el resto de los elementos origen, las expresiones usan los tipos de los elementos y otras restricciones. Luego, estas expresiones se utilizan en la salida de la regla generada cuando se necesita transferir información (i.e. valores de atributos) desde la entrada. Esta transferencia de información es generada de acuerdo con las asociaciones de atributos entre los modelos prototipo.

function mtbe(mm_i, mm_o : **metamodel**; pairs: $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$):MT

El algoritmo recibe como entrada dos metamodelos, mm_i y mm_o, y parejas de instancias de esos metamodelos. Como precondición, se espera que los metamodelos representen LMs *basados en conexiones*, y deben estar expresados en el lenguaje de meta-modelado requerido por el LTM en que se implemente. Las parejas son un subconjunto de $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$, que es el conjunto de conjuntos cuyos elementos son parejas de modelos instanciados de ambos metamodelos (i.e. $\{< m_i, m_o > | m_i$ es instancia de mm_i, m_o es instancia de $mm_o\}$). La salida del algoritmo es una TM que satisface la gramática presentada en el apartado anterior.

```
1: begin
2: dictionary := ∅
3: vars := GenerateVariables(mm_o)
4: rules := ∅
5: for each < m_i, m_o > ∈ pairs do
6:     main := SelectMainElement(m_i.E)
7:     UpdateDictionary(dictionary, main, BasicRootConstraint)
8:     inCons := true
9:     GenerateInputConstraints(main, inCons)
```

La transferencia de información desde los modelos de origen a los modelos destino se consigue por medio de un diccionario. En este caso, un **diccionario** es un conjunto de tuplas $< e_x, c_x >$, donde $e_x \in E$ es un elemento y c_x una expresión de restricción del LTM.

El algoritmo empieza con un diccionario vacío (en Línea 2) y una lista de variables (en Línea 3). Para cada pareja de modelos prototipo, se selecciona una raíz lógica (en Línea 6) del modelo de entrada, denotada como el elemento *principal*. Si hay varias posibles raíces, entonces cualquiera pueda ser elegida. Este elemento raíz es explorado hasta llegar a todos sus elementos conectados por medio de relaciones de agregación, referencia o atributo. Por un lado, la función *UpdateDictionary* (en línea 7) añade al diccionario una expresión para cada elemento del modelo prototipo de entrada, que indique el camino desde el elemento principal a dicho elemento. Estas expresiones se usarán más adelante

para la transferencia de información en las reglas. Por otro lado, la función *GenerateInputConstraints* (en línea 9) añade restricciones para restringir el encaje de una regla dada a sólo modelos similares al modelo prototipo origen. Para conseguir este objetivo, añade una restricción parcial por cada elemento procesado del modelo prototipo origen. Las restricciones se almacenan en la variable *inCons*, que se inicia al valor *cierto* (i.e. *true*) en la Línea 8.

```
 4: ...
 5: for each < m_i, m_o > ∈ pairs do
 6:     ... Continuando el bucle principal...;
10:     for each logicRoot ∈ (m_i.O − {main})
11:         logicRootConstraint = allElementsOfType ( logicRoot.t)
12:         UpdateDictionary(dictionary, logicRoot, logicRootConstraint)
13:         GenerateInputConstraints(logicRoot, inCons)
14:     end for
```

Después de que un punto de referencia es elegido para la regla, i.e. el elemento *principal*, otras posibles raíces lógicas se analizan y se incluyen en la regla (en Líneas 10-14), permitiendo así analizar grafos no conexos. Para analizar estas otras raíces lógicas (almacenadas en la variable *logicRoot*), es necesario usar la primitiva *allElementsOfType*. De esta manera, se seleccionan todos los elementos del tipo de una raíz lógica dada (i.e. *logicRoot.t*), que luego se filtran con más condiciones acerca de conexiones entre elementos y valores de atributos. Dichas condiciones son generadas por el procedimiento *GenerateInputConstraints*, que también fue usado para el elemento *principal*. De esta forma, este tipo de restricciones encajan con elementos muy similares a las raíces lógicas tratadas. Las nuevas expresiones son añadidas al diccionario y a las restricciones de entrada. Hasta aquí, todas los elementos de entrada han sido tenidos en cuenta en *inCons*. La variable *inCons* se construye siguiendo la gramática para TMs (véase la regla 7 de la gramática para TMs). Después de esto, es el turno de considerar los elementos de salida.

```
 4: ...
 5: for each < m_i, m_o > ∈ pairs do
 6:     ... Continuando el bucle principal ...;
15:     aggregatedToRootElements = ∅
16:     outElems = ∅
17:     for each e ∈ m_o.O do
18:         outElems := outElems ⊕ GenerateOutputElement(dictionary, e)
19:         aggregatedToRootElements:= aggregatedToRootElements ∪ {e}
20:     end for
21:     actions = GenerateActions (aggregatedToRootElements)
22:     rule = rule main.t ( ⊕ inCons ⊕ ) ⊕ to ⊕ outElems ⊕
23:             imperative ⊕ actions ⊕ endrule
24:     rules = rules ⊕ rule
25: end for
```

Esta otra parte del algoritmo visita todos los elementos que son raíces en el modelo de salida (i.e. los elementos de $m_o.O$). Para cada uno de estos elementos,

se define una *acción* (véase la Línea 21) para el emplazamiento de los elementos de salida de acuerdo con la regla 10 de la gramática del pseudo LTM del apartado anterior. Después, una regla de transformación (véase la Línea 22) se compone usando la regla 5 del pseudo LTM. La información relevante es almacenada en las variables *outElems*, *aggregatedToRootElements*, *actions* y *rule*. Las reglas son encadenadas unas con otras para formar la TM final, siguiendo la regla 4 del pseudo LTM.

```
25: ...
26: rootRule = GenerateRootRule(m_i, m_o)
27: rules = rules ⊕ rootRule
28: mt = header ⊕ vars ⊕
29:         begin ⊕ rules ⊕ end
30: return mt
31: end
```

La transformación final *mt* se construye usando las variables inicialmente extraídas, que están disponibles en *vars* y *rules* obtenidas para cada pareja de modelos. La combinación de partes de una TM sigue la regla 1 de la gramática del pseudo LTM.

5.2.3 Emplazamiento de los elementos destino

La simulación de las reglas de transformación muchos-a-muchos implica que el número de elementos de entrada y el número de elementos de salida pueden ser diferentes. En la mayoría de LTMs, el emplazamiento de los elementos destino debe ser controlado explícitamente al usar este tipo de reglas de transformación.

Este algoritmo controla el emplazamiento de los elementos destino definiendo una variable auxiliar para cada tipo de elemento del metamodelo destino. El algoritmo crea estas variables auxiliares en la Línea 3 usando la función *GenerateVariables* (véase la Línea 32). La función construye una lista de variables llamadas exactamente como los tipos de entidades mencionados en el metamodelo.

```
32: function GenerateVariables(MMB: metamodel): DefVariables
33: vars := ∅
34: for each e ∈ MMB.E do
35:     vars := vars ⊕ {e.t ⊕ : List}
36: end for
37: return vars
```

Más adelante, las reglas de transformación generadas por las parejas prototipo almacenan los elementos destino en estas variables por medio de acciones. La función *GenerateActions* (véase la Línea 38) produce una acción *insertar* (véase la Línea 41) para cada elemento destino (i.e., *entry.n*) en las variables anteriores (i.e. *entry.t*). Obsérvese que *entry.t* encaja con el nombre t asignado a las variables en la función *GenerateVariables* (véase la Línea 32).

```
38: function GenerateActions(elements:E): Actions
39: actions = ∅
40: for each entry ∈ elements do
```

41: actions = actions ⊕ **insert(** ⊕ entry.t ⊕,⊕ entry.n ⊕ **)**
42: **end for**
43: **return** actions

Finalmente, el algoritmo usa la función *GenerateRootRule* (véase la Línea 44) para crear una regla final que transforma la raíz origen en la raíz destino, y añade todos los elementos destino en la raíz destino a través de las correspondientes acciones en la parte imperativa de la regla. Adviértase otra vez que $g.e_2.t$ denota el nombre de la variable generada por la función *GenerateVariables* (véase la Línea 32).

44: **function** GenerateRootRule (ma: mm_i, mb: mm_o): *Rule*
45: rule:= **rule begin** ⊕ $ma.O_R.t$ ⊕ **()** **to** $mb.O_R.t$ ⊕ **imperative**
46: **for each** g ∈ $mb.O_R.G$ **do**
47: rule := rule ⊕ **assign(**$mb.O_R.g$ **,** $g.e_2.t$ **)**
48: **end for**
49: rule:= rule ⊕ **endrule**
50: **return** rule

5.2.4 Simulación de la parte de entrada de las reglas por medio de restricciones

Algunos de los MTLs existentes no pueden definir directamente reglas que reciban entrada desde varios grafos de elementos no conectados entre sí. Por esta razón, este algoritmo simula la parte de entrada de las reglas por medio de las restricciones. Uno de los elementos del modelo de entrada es seleccionado como el elemento *principal*, que es el único elemento de entrada de la correspondiente regla. Este elemento proporciona acceso al grafo al que pertenece. La regla incluye elementos de entrada a otros elementos a través de restricciones que dependen del elemento *principal*.

La función *GenerateInputConstraints* contiene un algoritmo recursivo que genera estas restricciones. Esto reproduce la estructura de un grafo en el modelo de entrada por medio de expresiones *attributeConstraint*, *referenceConstraint* y *aggregationConstraint* (véase la regla 7 de la gramática del LTM). El parámetro con el elemento (i.e. parámetro *element*) es el punto de entrada a ese grafo. La función es la siguiente:

51: **function** GenerateInputConstraints (e: element; **var** inCons: *Constraint*)
52: **for each** a ∈ e.A **do**
53: **if** $a.v \neq wildcard$ **then**
54: inCons := inCons ⊕ **and attributeConstraint(** ⊕ a ⊕**)**
55: **end if**
56: **end for**
57: **for each** r ∈ e.R **do**
58: inCons := inCons ⊕ **and referenceConstraint(** ⊕ r ⊕ **)**
59: GenerateInputConstraints($e.r.e_2$, inCons)
60: **end for**
61: **for each** g ∈ e.G **do**
62: inCons := inCons ⊕ **and aggregationConstraint(** ⊕ g ⊕ **)**
63: GenerateInputConstraints($e.g.e_2$, inCons)

64: **end for**

La función incluye los siguientes pasos:

1. Para cada atributo del elemento pasado como parámetro (véase la Línea 52), una nueva restricción es creada para comprobar su valor. En el modelo de entrada, los atributos pueden tener cadenas de caracteres o números enteros como valores. El algoritmo interpreta un valor específico como la necesidad de tener ese valor exactamente en el modelo de entrada. Si el atributo tiene valores vacíos, tales como la cadena vacía o MAXINT[1] en el caso de ser un número entero, se asume que el atributo puede tener cualquier valor. En tal caso, el valor del atributo es denotado como *comodín* (i.e. *wildcard* en la Línea 53). Los comodines no añaden ninguna restricción nueva como se puede observar en la misma Línea 53 del algoritmo.

2. La función crea otra restricción para cada agregación y referencia que salga del elemento pasado por parámetro (véase las Líneas 58 y 62). Esta nueva restricción captura las dependencias entre los elementos de entrada, de tal forma que la regla generada pueda encajar con estructuras similares. Esta restricción describe el enlace desde el elemento pasado por parámetro al elemento actual considerando si es una agregación o referencia.

3. Para cada agregación o referencia, la función genera recursivamente nuevas restricciones para sus elementos destino (véase las Líneas 59 y 63).

Dado que el algoritmo trabaja desde sólo un elemento *principal* del modelo de entrada, todas las restricciones que se puedan deben ser enlazadas desde ese elemento. La Línea 9 en el algoritmo crea la restricción inicial para el elemento *principal* en *inCons*. Todo el resto de restricciones son añadidas a esta restricción inicial. El modelo de entrada puede incluir varios grafos de elementos. Estos grafos pueden ser accedidos desde el resto de las raíces lógicas (véase la Definición 3). Por tanto, el algoritmo invoca la función *GenerateInputConstraints* con las raíces lógicas para conseguir las restricciones del modelo de entrada completo. Estas llamadas a dicha función aparecen en las Líneas 10-14. Cada una de estas llamadas añade sus restricciones a la anteriormente creada, generando una sola restricción final para el modelo de entrada.

Este proceso permite considerar diferentes tipos de configuraciones en los prototipos de los modelos de entrada. En el caso más simple, todos los elementos del modelo origen pueden ser alcanzados desde el elemento *principal* de la regla, que es también la única raíz lógica del modelo. Esta es la configuración habitual que las propuestas de GTBE existentes consideran.

Sin embargo, este algoritmo también puede acceder a otros grafos no-conexos del modelo. Esto se consigue añadiendo restricciones para todas las raíces lógicas diferentes a la *principal* (almacenadas en la variable *logicRoot*) en cada iteración del bucle de la Línea 10. Para ello, se seleccionan todo los elementos del tipo de una raíz lógica dada con la primitiva *allElementsOfType* del pseudo LTM (en la Línea 11 del algoritmo), y luego se le añaden restricciones con la función *GenerateInputConstraints* explicada en este apartado. El tratamiento de grafos no-conexos constituye una mejora clave sobre las otras aproximaciones GTBE.

[1] Máximo valor del tipo entero

Una aclaración final es que la anterior función debe también evitar bucles recursivos infinitos, lo que se puede conseguir recordando los elementos que son procesados, por medio de una pila de elementos procesados. Este mecanismo se omite en el anterior pseudo código por brevedad.

5.2.5 Generación de la parte de salida de las reglas

La parte de salida de las reglas en la mayoría de los LTMs (véase la Tabla 2.5) puede tener varios elementos de modelado. Por tanto, no hay limitaciones obvias para la complejidad del modelo de salida. Sin embargo, el modelo de salida no preserva explícitamente las interconexiones entre los elementos, por lo que el algoritmo debe hacerlo.

El proceso de generación se aplica para cada una de las raíces lógicas del modelo de salida (véase la Línea 18 en el algoritmo), y está basado en la función recursiva *GenerateOutputElement* (véase la Línea 65).

```
65: function GenerateOutputElement( dictionary, e: element): OutputElements
66: outElems = ∅
67: for each g ∈ e.G do
68:             outElems = outElems ⊕ GenerateOutputElement(dictionary, g.e2)
69: end for
70: out = e.t ⊕ (
71: for each a ∈ e.A do
72:             value = a.v
73:             for each entry ∈ dictionary do
74:                         value = replace(value, entry.e.n, entry.c)
75:             end for
76:             out = out ⊕ assign( ⊕ a ⊕ , ⊕ value ⊕ )
77: end for
78: for each g ∈ e.G do
79:             out = out ⊕ assign( ⊕ g ⊕ , ⊕ e.g.e2 ⊕ )
80: end for
81: for each r ∈ e.R do
82:             out = out ⊕ assign( ⊕ r ⊕ , ⊕ e.r.e2 ⊕ )
83: end for
84: out = out ⊕ )
85: outElems = outElems ⊕ , ⊕ out
86: return outElems
```

Los pasos del proceso de generación son los siguientes:

1. El código de la TM para los elementos contenidos por el elemento pasado se generan con una llamada recursiva a la misma función (véase las Líneas 67-69), y el código generado se almacena en una variable del algoritmo (i.e. *outElems*).

2. Se crea el principio del código para el elemento pasado por parámetro en la Línea 70, en la variable *out* del algoritmo.

3. Para los atributos del elemento pasado como parámetro, se crea el código necesario para establecer el valor de dichos atributos en la parte de salida

de la regla de transformación. Como se observa en las Líneas 71-77, se usa un diccionario para realizar la asociación de información entre el modelo de entrada y el modelo de salida. Obsérvese que los registros del diccionario son parejas compuestas de un elemento (i.e., *entry.e*) y una restricción (i.e., *entry.c*), y se usan para reemplazar (en la Línea 74) los identificadores (i.e. *entry.e.n*) del modelo de entrada por restricciones (i.e. *entry.c*) que referencian los elementos de entrada correspondientes desde la parte de salida de la regla. Si no se reemplaza ningún identificador, entonces el valor se asume constante y se asigna al atributo correspondiente como valor constante en la salida de la regla. Todo el código generado se añade a la variable *out*, que contiene el código generado para el elemento pasado por parámetro

4. El algoritmo genera el código para vincular el elemento pasado por parámetro (véase las Líneas 78-80) con el código de los elementos generados en la llamada recursiva (en la Línea 68). Dado que este paso se hace después de la generación de los elementos señalados por el elemento pasado por parámetro, la salida de la regla generada conserva las conexiones entre los elementos destino. La recursión termina porque las relaciones de agregación no pueden tener ciclos (véase la Definición 3).

5. El código generado para el elemento pasado (almacenado en variable *out*) se completa añadiendo el símbolo que cierra el elemento (i.e. un paréntesis) en la Línea 84, y se añade al código previamente generado (almacenado en la variable *outElems*) para el resto de elementos de salida en la Línea 85.

Cuando el anterior proceso se aplica sobre un raíz lógica del modelo destino (en la Línea 18), se genera el código de la salida de una regla que puede reproducir todo el árbol de dicha raíz lógica.

5.2.6 Mecanismo de asociación de atributos entre los modelos de entrada y salida

El algoritmo asume que los atributos en el modelo de salida pueden contener como valores expresiones que se refieran a valores encontrados en elementos del modelo de entrada (véase el apartado 5.2.5). En el modelo prototipo de salida, el diseñador de TMs se refiere a la información del modelo prototipo de entrada usando los identificadores de sus elementos. Dado un identificador de un elemento del modelo prototipo de entrada, se puede generar código de la salida de una regla que pueda tomar información de cualquier atributo del elemento identificado en el prototipo de entrada. Durante la generación de la salida de la regla, los atributos son asignados con la expresión correcta obtenida desde el correspondiente elemento del modelo de entrada. Esta expresión es usualmente el camino al correspondiente elemento desde el elemento *principal* de la parte de entrada de la regla. La construcción de la expresión sigue la regla 9 del pseudo LTM.

En primer lugar, durante la generación de la parte de entrada de la regla (véase la Línea 7 y la Línea 12), parejas de, primero, elementos del prototipo de entrada y, segundo, la restricción necesaria para acceder a éstos son insertadas en el diccionario en la Línea 90 del procedimiento *UpdateDictionary*. Cabe destacar que la función *UpdateDictionary* (véase la Línea 87) no sólo añade al diccionario

el elemento pasado por parámetro, sino que también añade todos los elementos que se pueden acceder desde él. Para ello, hace una llamada recursiva en la Línea 93, añadiendo al parámetro c un segmento de la ruta (i.e. <restricción> → <nombre agregación o referencia>). Se necesita también comprobar que la llamada recursiva no entra en bucles infinitos, por lo que se debe comprobar si ya se ha pasado por el mismo elemento. Esto es fácil dado que el diccionario lleva cuenta de por qué elementos se ha pasado; aun así, el código correspondiente se omite para ser breves.

Como se hace primero la llamada sobre el elemento *principal* (en Línea 7) y luego sobre el resto de raíces lógicas (en Línea 12), ocurre lo siguiente. Si es posible, la expresión de restricción referencia el correspondiente elemento usando la ruta con el elemento *principal* como origen. En caso de que no sea posible, la expresión de restricción se construye filtrando la lista de elementos de un tipo particular, que es similar a las restricciones usadas para el resto de raíces lógicas, explicadas en el apartado 5.2.4.

87: **procedure** UpdateDictionary (**var** dictionary, e, c)
88: **where** e: element;
89: c: constraint expression of the element from *main* element
90: $dictionary = dictionary \bigcup \{< e, c >\}$
91: **for each** r ∈ {e.R ∪ e.G} **do**
92: newConstraint = c ⊕ → ⊕ r.n
93: UpdateDictionary(dictionary, $e.r.e_2$, newConstraint)
94: **end for**

En segundo lugar, en la generación de la salida, que se vio anteriormente en el apartado 5.2.5, los valores de atributos en el modelo prototipo de salida que contengan identificadores de los elementos de entrada son reemplazados (en la Línea 74 del algoritmo) con las expresiones de restricción previamente almacenadas en el diccionario. Dichas restricciones indican cómo transferir la información de la entrada a la salida de las reglas de transformación. Obsérvese que se pueden reemplazar expresiones con información de varios atributos de elementos diferentes de la entrada. Esto permite combinar la información de varios atributos de elementos diferentes (e.g. concatenando cadenas de caracteres), lo que no era posible en aproximaciones anteriores de GTBE (véase el apartado 2.3.4).

5.3 Implementación para el Atlas Transformation Language

El algoritmo propuesto puede ser implementado para producir TMs escritas de acuerdo con la mayoría de LTMs, dado que usa primitivas que ya aparecen en éstos (véase la Tabla 2.5). Como prueba de concepto, una implementación en Java para producir TMs en ATL está disponible en la web de Grasia (Grasia, 2009d).

El pseudo lenguaje de transformación usado en este trabajo (véase el apartado 5.2.1) es asociado con expresiones ATL para producir reglas en este lenguaje. La Tabla 5.1 muestra las asociaciones más relevantes. Más detalles sintácticos

Pseudo language	**ATL**
restricción existe	*metamodel*!*entity_type*.allInstances()−> select(e| *InnerConditions*).notEmpty()
restricción atributo	*entity.attribute*= *value*
restricción agregación	*entity.aggregation* −> select(t| InnerConditions).notEmpty()
restricción referencia	*entity.reference* −> select(t| *InnerConditions*).notEmpty()
allElementsOfType	*metamodel*!*entity_type*.allInstances()
asignación	*variable* <- *value*
inserción	*list*<-*list*.append(*element*)
múltiple salida	*output_1*,...,*output_N*

Tabla 5.1: Correspondencia entre el pseudo lenguaje y ATL

sobre ATL se pueden encontrar en su documentación proporcionada en la web de ATL[2].

Los siguientes sub-apartados aclaran con más detalle esta implementación, indicando sólo los aspectos relevantes para entenderla. Estos sub-apartados usan como ejemplos extractos de una TM generada por nuestra aproximación, y dicha TM completa se encuentra en el apéndice B.

5.3.1 Selección de los patrones de entrada

En ATL, la selección de elementos para los patrones de entrada de las reglas difiere entre los casos no-múltiples y múltiples. Es necesario considerar esto para entender la manera de generar la parte de entrada de las reglas en ATL según la parte del algoritmo que se describió en 5.2.4. En concreto, esta selección se realiza cuando se generan las restricciones de referencias y relaciones de agregación, respectivamente en las Líneas 58 y 62 del algoritmo. Los casos no-múltiples son aquellos en los que la relación de agregación o referencia tiene cardinalidad uno; mientras que los casos múltiples son aquellos en los que dicha cardinalidad es mayor que uno. En los casos no-múltiples, la regla puede directamente referirse a la clase. Por ejemplo, dado un elemento de entrada `cin` para una regla con una referencia `iniSource` de cardinalidad uno, un elemento de salida puede acceder de la siguiente forma:

```
cin.iniSource
```

Sin embargo, las referencias múltiples apuntan a listas de elementos. Por tanto, las referencias múltiples deben ser filtradas con condiciones internas para obtener los elementos deseados. Obsérvese que, en este caso, el resultado de la selección es un conjunto que puede contener cero o más elementos. Esto también se aplica en las Líneas 58 y 62 del algoritmo. Un ejemplo de estos tipos de restricciones para una referencia llamada `iniTarget` se muestra a continuación:

```
cin.iniTarget->select(t|<restricciones-internas>)
```

Los ejemplos anteriores empiezan su navegación en un element *cin* dado. Sin embargo, éste no es siempre el caso. Por ejemplo, dada una entrada para una

[2] http://www.eclipse.org/m2m/atl/

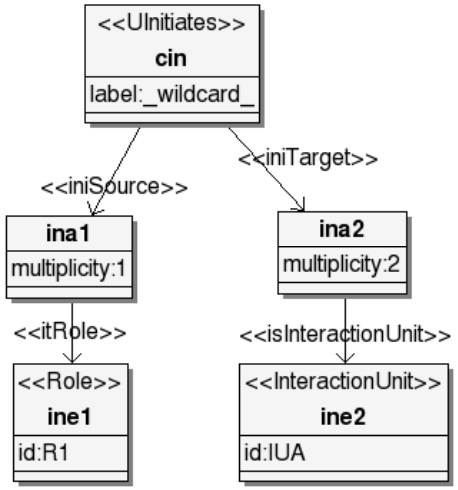


Figura 5.1: Un ejemplo de modelo origen de una pareja de modelos prototipo para la GTBE

TM con varios grafos no conexos, el elemento principal de la TM sólo puede proporcionar acceso a uno de esos grafos, pero las reglas necesitan conseguir acceso a los otros grafos, como se explicó en el apartado 5.2.4. Cuando, hay elementos aislados en un patrón de entrada de una regla ATL (i.e., no están conectados con el elemento de la entrada de la regla), se usan restricciones expresadas con OCL equivalentes a la primitiva *allElementsOfType* del pseudo LTM, que extraen todas las entidades de ciertos tipos y las filtran de la manera requerida. Esto se aplica en la Línea 11 del algoritmo. Un ejemplo de este tipo de restricción OCL es el siguiente:

```
MMA!Role.allInstances()->select(e|<restricciones-internas>).notEmpty()
```

Esta restricción de ejemplo selecciona todas las entidades del tipo *Role*, y filtra las entidades que satisfacen algunas `restricciones-internas`.

Un ejemplo más completo, que puede facilitar el entendimiento de la implementación descrita, corresponde al modelo origen mostrado en la Figura 5.1. En el algoritmo, éste es el resultado almacenado en la variable *inCons*, después de haber sido inicializada en la Línea 8 del algoritmo y haber sido modificada por sucesivas llamadas a la función *GenerateInputConstraints* en las Líneas 9 y 13 del algoritmo. El patrón de entrada de la regla generada es el siguiente:

```
cin:MMA!UInitiates( true  and
    cin.iniSource.multiplicity=1 and
    cin.iniSource.itRole.id='R1' and
    cin.iniTarget->select(t|
        t.multiplicity=2 and
        t.isInteractionUnit.id='IUA').notEmpty() )
```

5.3.2 Generación de las salidas de la regla

ATL (Jouault y Kurtev, 2006b) proporciona directamente soporte para las reglas de transformación con varios elementos de modelado en el patrón de salida. Para

cada elemento del modelo prototipo destino, un elemento de salida es creado en la regla, tal y como se describió en el apartado 5.2.5. En las reglas ATL, para cada elemento de salida se incluyen en la Línea 70 del algoritmo, por este orden, el nombre del elemento, una separación de dos puntos, y el principio de la definición del elemento de salida que contiene el tipo del elemento. Obsérvese que lo de añadir un nombre antes es característico de ATL y por tanto no se incluye en la definición genérica del algoritmo.

En el ejemplo presentado a continuación, los nombres de los elementos de salida son *outa1, outa2* y *outr1*. Esta implementación del algoritmo usa el identificador de cada elemento del modelo prototipo destino para nombrar cada elemento de salida de la regla. De esta forma, se garantiza que dicho nombre es unívoco, que es lo que se necesita. Los atributos del modelo de salida son copiados al elemento de salida en la regla reemplazando ciertos valores del *diccionario*, como se indica en las Líneas 71-80 del algoritmo. La regla generada también establece las conexiones (i.e. relaciones de agregación y referencias) entre los elementos de salida, en las Líneas 79 y 82 del algoritmo. Un extracto de la salida generada para una regla se muestra a continuación:

```
...
outa1:MMB!InteractionSource(
        multiplicity<-1,
        isInteractionUnit<-oute1),
outa2:MMB!InteractionTarget(
        multiplicity<-1,
        itRole<-oute3),
outr1:MMB!UColaborates(
        label<-”,
        colSource<-outa1,
        colTarget<-outa2),
...
```

En el ejemplo presentado, el valor es *1* para el atributo *multiplicity*, y *colSource* y *colTarget* son ejemplos de referencias para la conexión entre los elementos de salida. La conexión de estas referencias usa las variables *outa1* y *outa2*, cuyos nombres se toman de los identificadores de los modelos prototipo de destino.

La regla inserta todos los elementos en variables locales en la Línea 41 del algoritmo. También se añaden estos elementos a las variables *helper* anteriormente creadas (véase la Línea 3 en el algoritmo del apartado 5.2.2). De esta manera, los elementos creados pueden ser ubicados en el modelo destino si se requiere. En el siguiente ejemplo, la raíz física tiene dos relaciones de agregación llamadas *newEntities* y *newRelations*, de tipos *GeneralEntity* y *GeneralRelation*. Por eso, se añaden los elementos de salida a las variables *helper* llamadas *newEntities* y *newRelations*. En la regla ATL, el siguiente código imperativo consigue esta tarea:

```
do {
    thisModule.newEntities
    <-thisModule.newEntities.append(oute1);
    thisModule.newRelations
```

```
        <-thisModule.newRelations.append(outr1);
...
        }
}
```

Más adelante, se genera la regla que convierte la raíz física y añade todos los elementos almacenados en las variables helper, con la función *GenerateRootRule* definida en la Línea 44 del algoritmo. Un ejemplo de esta regla se incluye en el apéndice B.3 para unos modelos cuyas raíces físicas se llaman *Specification*.

5.3.3 Mecanismo de asociación de atributos entre los modelos de entrada y salida

Algunas TMs necesitan establecer valores de atributos en los modelos de salida, usando los valores de los atributos de los modelos de entrada. Como se indica en el apartado 5.2.6, en el modelo de salida, una alusión al identificador de un modelo de entrada permite obtener las propiedades del correspondiente elemento.

ATL proporciona soporte para esta funcionalidad mediante las variables de elementos. Sin embargo, la salida de una regla ATL solo puede hacer alusión al único elemento de modelado de la entrada de la regla (i.e. el elemento *principal*). En la implementación del algoritmo presentado, un identificador se sustituye con la expresión OCL que obtiene el elemento de modelado apropiado, en la Línea 74 del algoritmo. Los siguientes pasos consiguen dicha substitución. Primero, en la generación de la entrada de la regla, se asocia el identificador de cada elemento de entrada con la expresión OCL requerida, usando la función *UpdateDictionary* presentada en el apartado 5.3.3. Si se puede, la expresión OCL hace referencia al elemento desde el elemento *principal*. En otro caso, se usa la expresión OCL para los elementos aislados descrita en el apartado 5.3.1. Segundo, en la generación de la salida, los identificadores se substituyen con las expresiones OCL asociadas en la Línea 74 del algoritmo.

La Figura 5.2 muestra un ejemplo de aplicación del mecanismo de asociación de atributos entre los modelos de entrada y de salida. En este ejemplo, el valor del atributo *id* se copia para las *InteractionUnits*. El atributo `id` del elemento *oute1* contiene el valor `ine2.id`. El *ine2* es el identificador de la *InteractionUnit* del modelo de entrada. El siguiente extracto es generado en la salida de la regla ATL:

```
...
oute1:MMB!InteractionUnit(
      id<-cin.iniTarget.isInteractionUnit.id),
...
```

En este ejemplo, el identificador *ine2* en la salida de la regla se sustituye con la siguiente expresión:

```
cin.iniTarget.isInteractionUnit
```

Esta expresión indica el elemento del modelo de entrada al que se deseaba referirse. La ruta empieza con el elemento de entrada *cin*, que en realidad es el único elemento de entrada de la regla ATL.

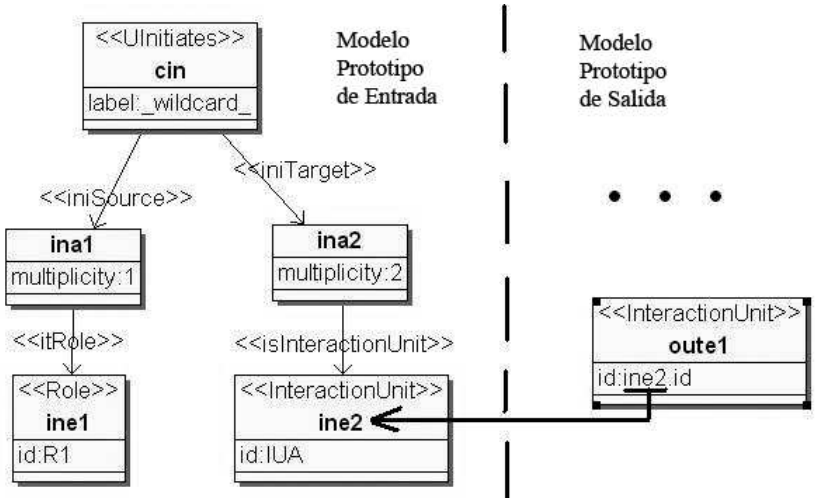


Figura 5.2: Asociación entre los valores de los atributos entre los modelos prototipo de entrada y de salida

El modelo de salida puede contener valores constantes para los atributos. El algoritmo distingue estas constantes de otros valores. Para esta implementación del algoritmo, las cadenas de caracteres de valores constantes se representan con comillas simples en los modelos prototipo, como por ejemplo *'IUA'*. Por el contrario, los valores constantes de los enteros se representan con sus literales, como por ejemplo *2*. Los valores constantes se copian directamente en la generación de transformaciones, que es lo que ocurre cuando después de la copia del valor del atributo en la Línea 72 del algoritmo no se produce ningún cambio del valor en la Línea 74.

5.3.4 Descripción de la herramienta para generación de transformaciones basadas en ejemplos

Este apartado presenta una herramienta para la GTBE que contiene la implementación del algoritmo presentado para ATL. La herramienta se llama *MTGenerator* y está disponible en la web de *Grasia* (Grasia, 2009d) junto con su breve manual de usuario y algunos ejemplos. Su interfaz permite al usuario seleccionar los metamodelos de entrada y salida de la transformación, y las parejas de modelos prototipo que la definen (véase la Figura 5.3).

Dado que la herramienta usa ATL (Jouault y Kurtev, 2006b), la herramienta trabaja con LMs cuyos metamodelos están definidos con el lenguaje ECore (Budinsky, 2003). Como precondición, se espera que estos metamodelos representen LMs *basados en conexiones*. El usuario indica estos metamodelos por medio de la selección de sus rutas en el sistema de archivos, en la parte superior izquierda de la interfaz de la herramienta.

Posteriormente, el usuario proporciona las parejas prototipo con modelos de entrada y salida. Los modelos de cada pareja deben ser instancias de los metamodelos de entrada y salida. El usuario selecciona estos modelos especificando las correspondientes rutas de archivo en la parte superior derecha de la interfaz

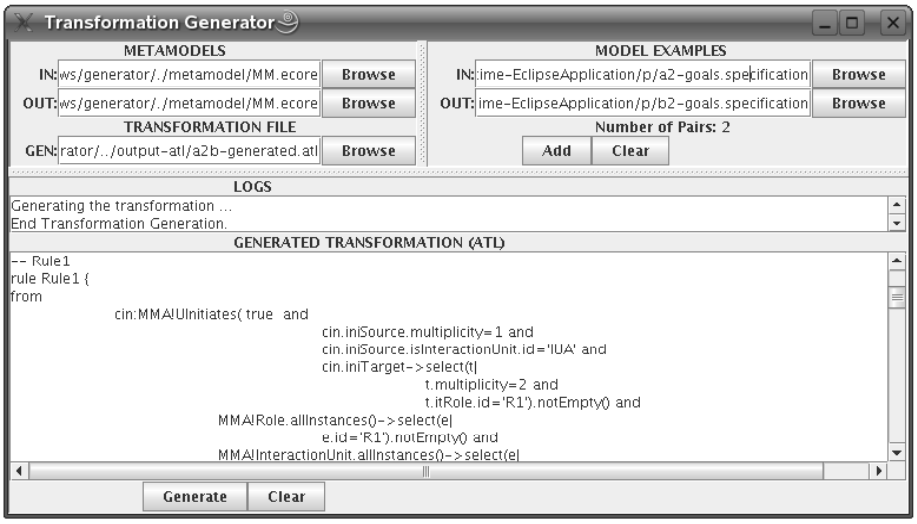


Figura 5.3: La herramienta *MTGenerator*.

y pulsando el botón *Add* para añadir cada nueva pareja de modelos. La herramienta hace ciertas operaciones básicas de comprobación sobre estas parejas, tales como verificar que realmente son instancias de los metamodelos elegidos y que no hay parejas repetidas.

Finalmente, el usuario elige la ruta de archivo en donde la TM generada será guardada. En la interfaz, el área para esta elección se indica con la etiqueta *Transformation File*. Después, el usuario puede presionar el botón *Generate*, y la herramienta genera la transformación ATL siguiendo el algoritmo presentado y la guarda en la ubicación elegida. La herramienta muestra ciertos mensajes sobre el estado del proceso de generación en el área de texto debajo de la etiqueta *Logs*. El area de texto en la parte inferior de la interfaz muestra la transformación resultante, de tal forma que el usuario puede examinarla.

5.4 Conclusiones

Este trabajo presenta un algoritmo para la GTBE por el cual se puede generar una TM a partir de un conjunto de parejas de modelos prototipo de entrada y salida. Por cada una de estas parejas prototipo, se genera una regla dentro de la TM. Si bien la aproximación de GTBE existía anteriormente a este trabajo, este algoritmo ha mejorado varios aspectos, facilitando así su futura integración en la práctica habitual de la ISOA. Las características principales de este algoritmo de GTBE son las siguientes:

- *El algoritmo se ha definido basándose en un pseudo LTM.* Este pseudo lenguaje sólo tiene propiedades comunes a la mayoría de LTMs existentes, tales como: el uso de reglas que transforman un patrón de entrada en un patrón de salida; restricciones sobre la existencia de elementos, atributos, relaciones de agregación, referencias y entidades de un tipo; y las acciones de asignación e inserción. Esto permite que el algoritmo sea compatible

con diferentes lenguajes de transformación tales como: ATL, XSLT, QVT, Tefkat y VIATRA2.

- *Los metamodelos de entrada y salida deben representar LMs basados en conexiones.* Como precondición, se espera que el algoritmo reciba como entrada metamodelos que definan LMs *basados en conexiones*. Por este motivo se pone tanta atención en transformaciones de grupos de elementos interconectados y grupos de grafos no conexos, porque esto es propio de este tipo de LMs.

- *El algoritmo genera reglas que reciben como entrada múltiples grafos.* Dado que varios lenguajes existentes no permiten más que un único elemento de entrada en cada regla, los patrones de entrada se simulan con restricciones. En concreto, nuestro algoritmo genera restricciones, de manera que las reglas reciben como entrada varios grafos no conexos. Este tipo de reglas no puede ser generado por las otras aproximaciones existentes de GTBE.

- *El mecanismo de asociación de atributos considera las reglas muchos-a-muchos.* Si bien las aproximaciones de GTBE existentes consideran mecanismos de asociación de atributos, este algoritmo es el primero que proporciona un mecanismo de asociación que toma como entrada atributos de varios elementos de entrada de una regla muchos-a-muchos. Con las anteriores aproximaciones, no se podía combinar atributos de elementos diferentes (e.g. concatenar cadenas de caracteres procedentes de dichos elementos), mientras que con la aproximación presentada esto es posible. Más aún, este mecanismo de asociación de atributos también se aplica a reglas que toman como entrada múltiples grafos no conexos.

Como prueba de concepto, se ha implementado el algoritmo para el lenguaje ATL, que es uno de los más extendidos y con mayor soporte tecnológico. Dicha implementación del algoritmo se ha incluido en la herramienta llamada *MTGenerator*, que está disponible públicamente en nuestra web (Grasia, 2009d). El siguiente capítulo presenta algunos experimentos realizados con el algoritmo y la herramienta presentados en este capítulo.

Capítulo 6

Experimentación con la generación de transformaciones basadas en ejemplo en el desarrollo de sistemas multi-agentes

El algoritmo y herramienta presentados en el anterior capítulo se han utilizado para generar transformaciones de modelos en el desarrollo de un sistema multi-agente. Esta experimentación muestra ejemplos de generación para diferentes tipos de reglas, y permite evaluar las ventajas y limitaciones de la aproximación de esta tesis.

6.1 Introducción

Este capítulo ilustra la experimentación del algoritmo y la herramienta (i.e. *MT-Generator*) de GTBE presentados en el capítulo anterior a través de la creación de TMs que son aplicables en un ejemplo real de SMA. Como se ha visto en el capítulo anterior, este algoritmo representa los patrones de entrada de las reglas muchos-a-muchos con restricciones. La definición de los elementos de los patrones de salida usa las capacidades comunes de la mayoría de los LTMs existentes. El algoritmo enlaza recursivamente estos elementos durante su creación. Además, puede crear las correspondencias apropiadas entre atributos, de tal forma que la TM generada propague la información desde el modelo origen al modelo destino. La herramienta *MTGenerator* implementa este algoritmo para ATL, y puede aplicarse sobre los modelos de INGENIAS cuando se exportan con el formato del EMF.

Este capítulo usa como marco de experimentación un SMA para evaluar documentos siguiendo el método Delphi (García-Magariño et al., 2008). Este SMA fue desarrollado con la manera tradicional propuesta por la metodología INGENIAS, que no incluye transformaciones modelo-a-modelo. Sin embargo,

esta tesis propone ciertas TMs genéricas que se pueden aplicar en el desarrollo de este SMA. Estas TMs se han creado con las técnicas propuestas por esta tesis, y se han aplicado sobre la especificación del SMA mencionado. Para comprobar las TMs, se ha comparado el resultado generado por las TMs con la especificación anteriormente definida con INGENIAS. Como ejemplos de TMs aplicables a ejemplos reales de SMAs, se presentan TMs cuyos objetivos son: la definición de roles a partir de los casos de uso, la definición de las tareas del flujo de trabajo, la creación de las unidades de interacción, y la generación de despliegues y tests.

Además de la experimentación presentada en este capítulo, el apéndice C presenta: la reutilización de algunas de las TMs en un SMA sobre *gestión de crisis*; propuestas de otras TMs creadas por GTBE para aplicarlas en INGENIAS; y un ejemplo práctico de GTBE en la metodología ADELFE de ISOA.

La experimentación de GTBE de esta tesis persigue dar soporte a las siguientes afirmaciones:

- *La aproximación de GTBE se ha aplicado a ejemplos reales de SMAs.* En el SMA de Delphi, se han definido TMs que se consideran útiles para el desarrollo, y se han aplicado para evaluar en qué medida las TMs reproducen una parte de la especificación existente para este SMA. Además, se ha validado estas TMs aplicándolas a otro SMA existente, sobre *gestión de crisis*, y cuya experimentación se presenta en el apéndice C.1.

- *El algoritmo de GTBE genera reglas de diferentes tipos: uno-a-uno, uno-a-muchos y muchos-a-muchos, permitiendo grafos no conexos.* Las reglas uno-a-uno, uno-a-muchos y muchos-a-muchos se distinguen por el número de elementos de entrada y el número de elementos de salida, independientemente si todos los elementos de la entrada o la salida pertenecen a un mismo grafo conexo. Esta tesis presenta ejemplos de estos tipos de reglas y su aplicación en ejemplos de SMAs existentes.

- *El algoritmo de GTBE da soporte a tipos de TMs que otras aproximaciones de GTBE no dan soporte.* En esta línea, las principales aportaciones del algoritmo GTBE son las reglas de transformación muchos-a-muchos con encaje de atributos provenientes de más de un elemento de entrada, y las reglas de transformación con grafos no conexos. En esta experimentación, se proporcionan ejemplos prácticos de la ISOA donde se utilizan dichas aportaciones, y que hace que la aproximación de GTBE de esta tesis sea más apropiada para la ISOA que el resto de aproximaciones de GTBE (Varró y Balogh, 2007b; Wimmer et al., 2007).

- *Las TMs obtenidas por GTBE se pueden reutilizar.* Si bien lo normal es que las TMs se puedan aplicar a varios modelos, en GTBE es necesario comprobarlo explícitamente dado que se generan a partir de modelos prototipo concretos y existe la posibilidad de que sólo sean aplicables sobre estos. En particular, la experimentación de esta tesis incluye TMs que se pueden aplicar varias veces en el mismo desarrollo (como se verá para las TMs relacionadas con el *flujo de trabajo*), y en varios desarrollos (i.e. en los SMAs sobre *Delphi* y *gestión de crisis*). Esto resalta la utilidad del mecanismo de *asociación de atributos* (véase el apartado 5.2.6) y el mecanismo de los *comodines* (véase el apartado 5.2.4) en la aproximación de GTBE de esta tesis. Estos mecanismos permiten generar TMs con patrones de

entrada lo suficientemente generales como para poder ser aplicados en varios casos, pero lo suficientemente específicos como para ser aplicados sólo en un cierto grupo de casos de interés. Principalmente, las TMs creadas están orientadas a participar en los procesos software de la metodología INGENIAS.

El resto del capítulo se organiza como se indica a continuación. El siguiente apartado describe el marco de experimentación, en el que los diseñadores definen los modelos prototipo en el IDK, y a bajo nivel se trabaja con modelos EMF y transformaciones ATL. El apartado 6.3 describe el SMA sobre el que se realizan los experimentos de GTBE y que usa el proceso Delphi para determinar la relevancia de documentos. El apartado 6.4 proporciona ejemplos de TMs generadas a partir de modelos prototipo y la aplicación de dichas TMs en el SMA; además, introduce brevemente la aplicación de dichas TMs en otro SMA, sobre gestión de situaciones de crisis. El apartado 6.5 hace un resumen de todas las TMs descritas en este capítulo y otras TMs, descritas en el apéndice C, para resaltar las ventajas de la aproximación de GTBE presentada frente a otras aproximaciones existentes. Finalmente, el apartado 6.6 debate las conclusiones sobre la aproximación de GTBE presentada y su aplicación en la ISOA.

6.2 El marco de experimentación

Para contextualizar la aplicación práctica de GTBE sobre el IDK, es necesario describir el marco usado para los experimentos, que se puede ver en la Figura 6.1. El diseñador tiene que definir los modelos prototipo con el IDK y, por tanto, con la notación de INGENIAS o UML a la que se asume que está acostumbrado. Los pares de modelos prototipo del IDK definen una TM (i.e. TM de alto nivel). Sin embargo, esta TM de alto nivel se implementa con una TM de bajo nivel expresada con ATL.

En la GTBE de este capítulo, los modelos prototipo de INGENIAS se exportan al formato de modelos EMF como instanciación de metamodelos ECore, tal y como se explicó anteriormente en el apartado 4.2. Dicha implementación para la exportación está disponible en (Grasia, 2009c). Después de la exportación, los modelos origen y destino se pueden usar como entrada de la herramienta *MT-Generator* que genera una transformación ATL, como se explicó en el capítulo anterior.

Para la aplicación de una TM (véase la Figura 6.1), es necesario que primero se exporte el modelo IDK a un modelo EMF, segundo se transforme el modelo con la trasformación ATL previamente generada y tercero se importe el modelo EMF creado en el IDK.

6.3 Caso de estudio: un sistema multi-agente con el proceso Delphi

Este capítulo usa como base de experimentación un SMA que incorpora el proceso *Delphi* para la evaluación de la relevancia de documentos. Este SMA fue desarrollado con las técnicas convencionales propuestas por la metodología en INGENIAS tal y como se describe en (García-Magariño et al., 2008). Este SMA

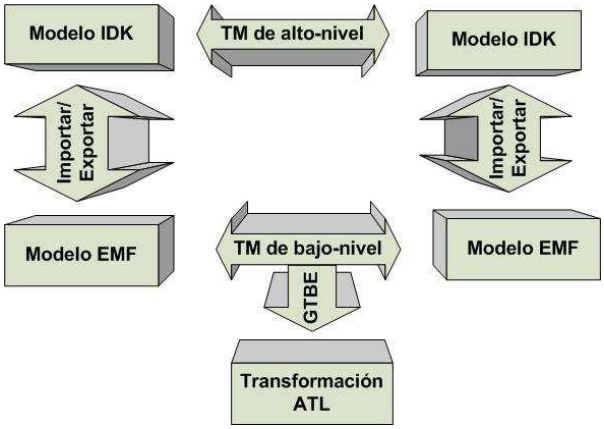


Figura 6.1: Marco de experimentación de GTBE con el IDK.

busca mecanismos alternativos de coordinación entre agentes, y explora la adaptación del protocolo Delphi a los SMAs. El protocolo Delphi se puede aplicar cuando se requiere que una comunidad de expertos proporcione una respuesta consensuada. El problema de llegar a un acuerdo se considera en la literatura (Ren y Beard, 2005), pero su completa automatización es todavía un reto. Intuitivamente, los expertos deben dialogar, intercambiar ideas, y cambiar de opinión según la discusión avanza. El SMA elegido como base de experimentación automatiza la discusión entre agentes expertos y muestra como pueden ser conducidos hasta una conclusión de la discusión por medio del proceso Delphi. El SMA elegido contiene 15 diagramas de M1, en los cuales aparecen 119 entidades de M1 y 86 relaciones de M1.

6.4 Ejemplos de generación de transformaciones en INGENIAS

Este apartado presenta TMs que son aplicables al SMA de Delphi, que ha sido presentado en el apartado 6.3. Para esto, se indican las parejas de modelos prototipo que fueron usadas como entrada del algoritmo de GTBE. Luego, se aplica las TMs generadas y se compara el resultado con los elementos que existen en la especificación. De esta manera se evalúa si las TMs se pueden aplicar a ejemplos reales y si los resultados son los esperados.

En el DSDM, las TMs se deberían poder reutilizar en varios desarrollos. En el caso de la GTBE es importante comprobar explícitamente que esto ocurre, dado que las TMs se crean a partir de ejemplos muy específicos.

Para ilustrar estos aspectos, el resto de la sección se estructura en dos partes. Los primeros sub-apartados describen TMs que son útiles respectivamente para: generar la definición de los roles a partir de los casos de uso, crear las tareas de un flujo de trabajo, crear las unidades de interacción, y generar los despliegues y los tests del SMA. Después, el último sub-apartado describe brevemente cómo se pueden reutilizar dichas TMs en la especificación de otro SMA, haciendo referencia al apéndice C.1, que contiene más detalles de dicha reutilización.

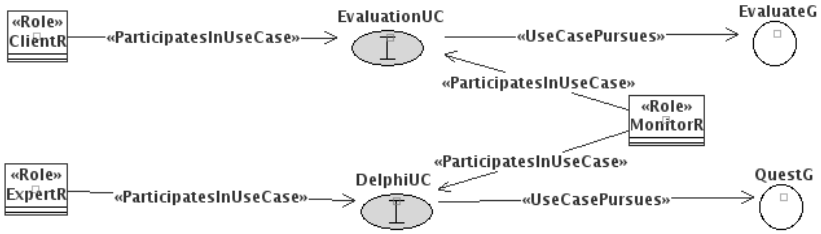


Figura 6.2: Casos de uso para el SMA sobre Delphi.

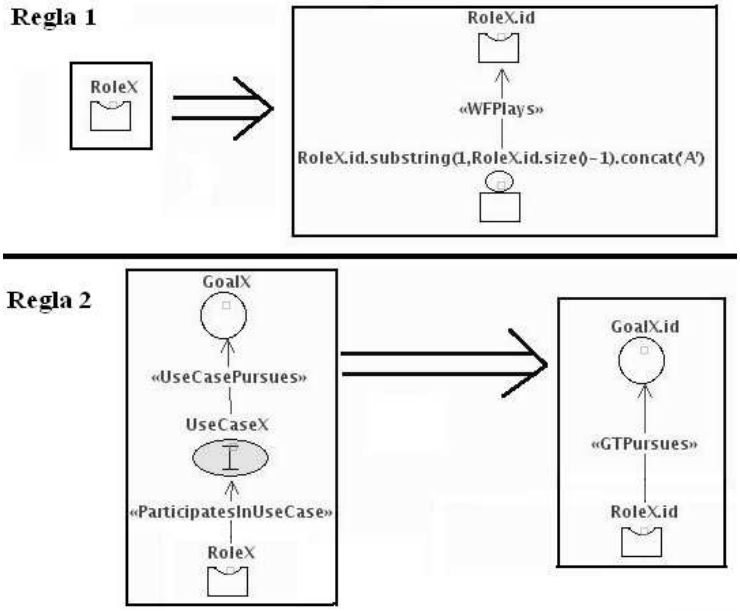


Figura 6.3: TM *UseCase2Agent* a partir de los casos de uso para la definición de los roles y agentes, con dos reglas que son respectivamente uno-a-muchos y muchos-a-muchos.

6.4.1 Una transformación para generar los roles a partir de los casos de uso

En INGENIAS, los diagramas de los casos de uso (véase un ejemplo en la Figura 6.2) generalmente incluyen roles, casos de uso donde los roles participan, y los objetivos satisfechos por estos casos de uso. Un paso posterior debe enlazar los roles y los objetivos que ellos persiguen. Además, debe haber al menos un agente que juegue cada rol (véase la Figura 6.4). Dado que estas tareas son comunes en los desarrollos INGENIAS, se propone crear automáticamente la definición de roles y agentes con una TM.

Las parejas de modelos de la Figura 6.3 pueden generar esta TM, que incluye dos reglas que son respectivamente uno-a-muchos y muchos-a-muchos. En la primera pareja, el modelo origen contiene un rol, y el modelo destino contiene el agente que juega este rol. El identificador del agente creado se compone a partir del identificador del rol usando el mecanismo de asociación de atributos propor-

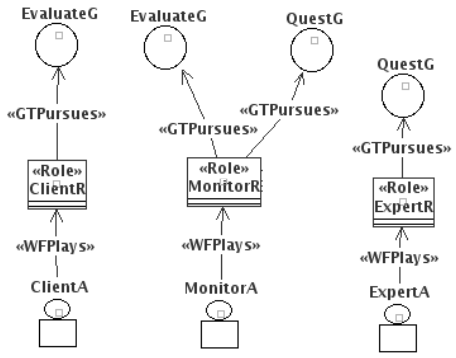


Figura 6.4: Los *Roles* y *Agentes* para Delphi.

cionado por la herramienta. En la segunda pareja, el modelo origen contiene un objetivo conectado a un rol a través de un caso de uso, y el modelo destino tiene el mismo objetivo directamente enlazado con el rol. De esta manera, la TM enlaza los roles con sus objetivos en concordancia con las casos de uso existentes. Por ejemplo, la TM genera las definiciones de roles de la Figura 6.4 a partir de los casos de uso de la Figura 6.2.

En esta primera TM generada, se puede ver un ejemplo práctico del mecanismo de asociación de atributos descrito en el apartado 5.2.6. En la Figura 6.3, los identificadores de los modelos prototipo de entrada (i.e. *GoalX, UseCaseX* y *RoleX*) se usan en los modelos prototipo de salida para indicar como se transmite la información. Por ejemplo, para copiar los valores de los identificadores del objetivo y del rol, se usan las expresiones "*GoalX.id*" y "*RoleX.id*". Estas se transforman en restricciones OCL que expresan en la regla que la información se toma de la entrada, tal y como se indicó en el apartado 5.3.3. En el resultado de aplicación de la TM en la Figura 6.4, se observa que las reglas de la TM encajaron con roles diferentes, tales como *ClientR, MonitorR* y *ExpertR*, y objetivos diferentes, tales como *EvaluateG* y *QuestG*.

En este ejemplo de TM, también se observa que se pueden crear nuevos identificadores o atributos modificando los atributos de entrada. Como se mencionó en el apartado 5.3.3, para la modificación de valores se deben usar las primitivas del LTM de la implementación, en este caso las primitivas de OCL, que se usa en ATL para las restricciones. En concreto, en este SMA, se adopta el convenio de que los roles tienen el sufijo *R* y los agentes tienen el mismo nombre que el rol que juegan pero con el sufijo *A*. Para ello, se usa las funciones *substring* y *concat* de OCL para quitar respectivamente el sufijo de los roles y añadir el sufijo de los agentes, como se muestra en la siguiente expresión:

```
RoleX.id.subsring(1,RoleX.id.size()-1).concat('A')
```

Como se observa en la Figura 6.4, en el resultado de la aplicación de la TM con la anterior expresión, se crearon los nuevos identificadores de agente *ClientA, MonitorA* y *ExpertA*.

Además, es importante destacar que la regla 2 de la Figura 6.3 se aplica varias veces para el rol *MonitorR*, asociando así dos objetivos (i.e. *EvaluateG* y *QuestG*) a dicho rol, como se puede observar en la Figura 6.4. El extremo de la relación *GTPursues* (una instancia de EClass, como se indicó en el apartado 4.2.1) copia el valor del extremo de la relación *ParticipatesInUseCase*. De

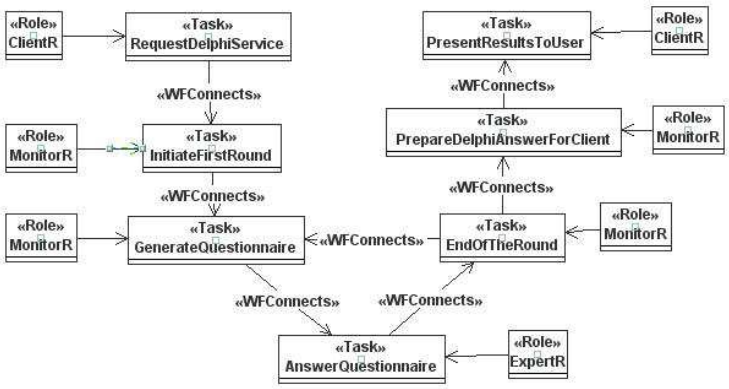


Figura 6.5: El flujo de trabajo para el SMA Delphi.

esta manera, todos los objetivos generados por la regla 2 se enlazan con el rol generado por la regla 1, dado que ambas reglas pertenecen a la misma TM.

Todos los elementos obtenidos como resultado de la aplicación de la TM se encuentran en la especificación original del SMA de Delphi. Sin embargo, para que la definición de los roles quede completa con respecto al SMA original, son necesarios más elementos: la asociación de los roles con las tareas de las que son responsables y los estados mentales iniciales de cada agente. Por tanto, la TM presentada genera elementos necesarios para el SMA, pero no genera todos los necesarios. Aun así, se podrían crear más elementos sin toda la información para que fueran rellenados por el usuario. Estos elementos no tendrían toda la información dado que no se puede obtener de otros modelos.

En general, tanto esta TM como las siguientes presentadas no pueden crear todos los elementos necesarios de una especificación de INGENIAS, debido a las limitaciones de la aproximación de GTBE presentada. Las operaciones para transformar los atributos están limitadas por la expresividad del lenguaje de restricciones, en este caso OCL. Además, el diseñador necesita modificar los modelos para añadir más información en cada paso del proceso. Sin embargo, sería recomendable que esta información se preguntara al diseñador de manera interactiva en vez de tenerse que modificar los modelos generados por las TMs, dado que este tipo de modificaciones suele crear problemas de trazabilidad.

6.4.2 Transformaciones para crear las tareas del flujo de trabajo

La definición de un *flujo de trabajo* en INGENIAS es una secuencia de tareas con posibles bifurcaciones y/o vueltas atrás (véase un ejemplo en la Figura 6.5), conectando las tareas con relaciones *WFConnects*. En INGENIAS, los diseñadores refinan esta definición proporcionando las entradas y salidas de cada tarea en diagramas diferentes. En concreto, la primera tarea de un flujo de trabajo es usualmente activada por un evento lanzado por alguna aplicación de usuario. La Figura 6.6 propone una TM para crear la tarea inicial. El modelo origen de la TM contiene la tarea inicial del flujo de trabajo, y el modelo destino incluye la tarea consumiendo un evento a partir de una aplicación interna que representa la aplicación de usuario. La tarea inicial de un flujo de trabajo es la única que no

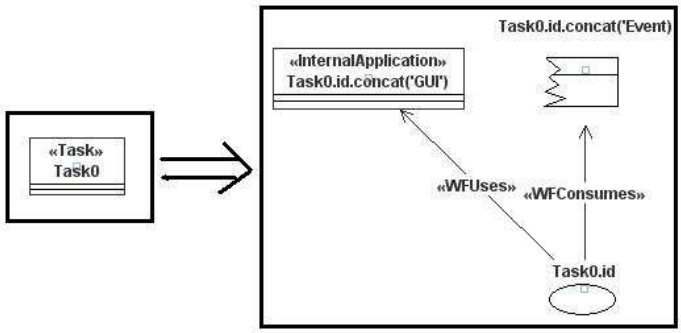


Figura 6.6: TM *InitialTaskWorkflow* para la tarea inicial de un flujo de trabajo, con una regla uno-a-muchos.

es precedida por ninguna otra tarea. Sin embargo, esto no se puede especificar dentro del modelo origen de la pareja. Los diseñadores de la TM deben definir una restricción para la pareja de modelos con el lenguaje OCL y de acuerdo con el metamodelo de INGENIAS. La restricción se debe añadir a la restricción de la entrada de la regla generada con el operador lógico *and*, para indicar que se debe satisfacer además de las restricciones generadas por la herramienta. La expresión de esta restricción es la siguiente:

```
MMA!WFConnects.allInstances()->select(e|WFConnectstarget.
      WFConnectstargettask.id=cin.id).isEmpty()
```

La restricción declara que no hay ninguna otra tarea conectada con la tarea actual (i.e. *cin*, que es el elemento principal de la entrada de la regla) con una relación *WFConnects*. Por tanto, la tarea es la primera del flujo de trabajo. Esta restricción no necesitaría ser definida si la herramienta de GTBE considerara *ejemplos negativos* (Varró y Balogh, 2007b), que evitan la ejecución de un regla en caso de que su entrada encaje con un ejemplo negativo. Para ello, el ejemplo negativo debería tener una tarea que precediera a la tarea en cuestión. Se planea incluir soporte para ejemplos negativos en una versión posterior del algoritmo y de la herramienta *MTGenerator*.

Las tareas no iniciales de un flujo de trabajo usualmente consumen un *fragmento de conocimiento* (i.e *frame fact*) producido por la tarea anterior, y producen otro fragmento de conocimiento para que la siguiente tarea lo consuma. La Figura 6.7 presenta la TM que crea estos fragmentos de conocimientos y sus conexiones con las tareas relacionadas. Cabe destacar que las dos reglas son muchos-a-muchos. La primera regla crea la relación *WFConsumes*, mientras que la segunda regla crea la relación *WFProduces*. La TM propaga los nombres de los modelos origen a los modelos destino por el mecanismo de asociación de atributos del algoritmo descrito en el apartado 5.2.6. El nombre de cada nuevo fragmento de conocimiento es la concatenación de los nombres de la tarea productora y la tarea consumidora con un sufijo, e.g. con la expresión *Task0.id.concat(Task1.id).concat('FF')* en la que *Task0* y *Task1* hacen referencia a identificadores del modelo prototipo de entrada. Es importante recalcar que se produce un atributo que recibe información desde atributos de varios elementos diferentes, usando el mecanismo de asociación de atributos, mien-

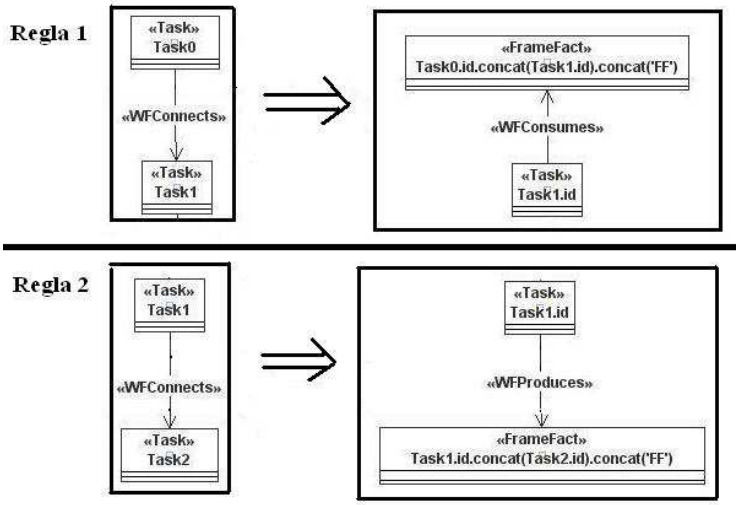


Figura 6.7: TM *NonInitialTaskWorkflow* para las tareas no iniciales de un flujo de trabajo, con dos reglas muchos-a-muchos.

tras que los mecanismos de asociación de atributos de otras aproximaciones de GTBE (Varró y Balogh, 2007b; Wimmer et al., 2007) no permiten establecer asociaciones con más de un elemento en la entrada de cada regla.

La primera regla de la TM especificada en la Figura 6.7 no es aplicada a la tarea inicial porque ésta no es precedida por ninguna otra tarea, gracias a la restricción introducida manualmente. Es importante recalcar que estas reglas pueden ser aplicadas varias veces a la misma tarea si precede o es precedida por varias tareas. Por ejemplo, el flujo de trabajo de la Figura 6.5 incluye la tarea *EndOfTheRound* que precede dos tareas diferentes, de tal forma que la segunda regla de la TM es aplicada dos veces para esta tarea.

En la segunda regla de la TM de la Figura 6.7, la relación *WFConnects*, que es una instancia de EClass según se explicó en el apartado 4.2.1, se transforma en la relación *WFProduces* copiando el valor para sus extremos. Tal y como se muestra en la Figura 6.7, esto hace que la relación *WFProduces* se conecte con el fragmento de conocimiento producido para el siguiente paso del flujo de trabajo.

En INGENIAS, una especificación también asocia un *componente de código* para cada tarea. La TM especificada en la Figura 6.8 crea estos componentes de código para las tareas.

Cuando se aplicaron las TMs propuestas en este apartado, se obtuvieron elementos que existían en la especificación original del SMA de Delphi, tal y como se esperaba. Sin embargo, la especificación original usa nombres más intuitivos que los nombres generados automáticamente para los fragmentos de conocimiento (e.g. *EmptyQuestionnaire* en lugar de *GenerateQuestionnaireAnswerQuestionnaireFF*). Además, la especificación original tiene más nivel de detalle en los elementos generados, tales como el tipo de información (i.e. *slots*) contenido en cada fragmento de conocimiento y las *conversaciones* que son lanzadas por cada

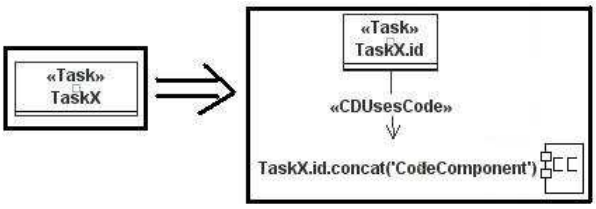


Figura 6.8: TM *Task2CodeComponent* para generar los *componentes de código* para las tareas, con una regla uno-a-muchos.

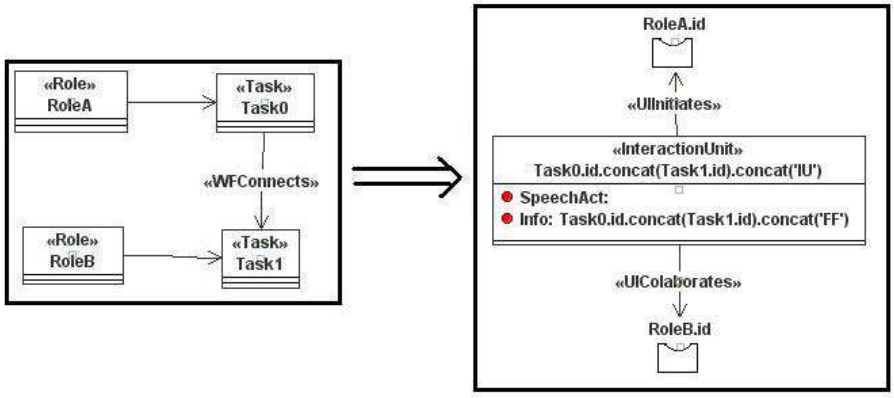


Figura 6.9: TM *InteractionUnit* para generar la *unidad de interacción* que comunica dos tareas, con una regla muchos-a-muchos.

tarea. Estas diferencias se deben a que la información necesaria no se encuentra disponible en otros diagramas y se debe especificar por el diseñador.

6.4.3 Una transformación para crear las unidades de interacción

Las *unidades de interacción* de INGENIAS transfieren información (i.e. fragmentos de conocimiento) entre los agentes. Las definiciones de las tareas indican la manera que los agentes producen y consumen los fragmentos de conocimiento. Si dos agentes diferentes son responsables de dos tareas consecutivas en un flujo de trabajo, entonces el fragmento de conocimiento que conecta estas tareas tiene que ser transferido dentro de una unidad de interacción. La TM especificada en la Figura 6.9 recibe como entrada dos tareas consecutivas y genera la correspondiente unidad de interacción con su fragmento de conocimiento. Los nombres de los nuevos elementos son el resultado de la concatenación de los nombres de las dos tareas involucradas con un sufijo (i.e. *IU* para la unidad de interacción y *FF* para el fragmento de conocimiento). En esta TM, se observa que el mecanismo de asociación de atributos del algoritmo (véase el apartado 5.2.6) permite tomar valores de atributos de varios elementos de entrada de cada regla y usarlos para establecer valores de atributos de los elementos de salida de la regla. Como se

Figura 6.10: TM *Agent2Deployment* para crear los *paquetes de despliegue*, con una regla uno-a-uno.

ha mencionado antes, esta propiedad no se encuentra en otras aproximaciones de GTBE (Varró y Balogh, 2007b; Wimmer et al., 2007).

La TM generada a partir de la especificación en la Figura 6.9 se dispara incluso si los dos agentes responsables son el mismo agente, aunque en este caso no es necesario crear la unidad de interacción. Para evitar esto, los diseñadores pueden añadir una restricción que establezca que los dos agentes deben ser diferentes. Esta restricción se debería añadir a la restricción de entrada de la regla generada con el operador lógico *and*, y sería similar a la siguiente expresión:

```
not(RoleA.id=RoleB.id)
```

Sin embargo, en la anterior restricción habría que sustituir manualmente las expresiones *RoleA* y *RoleB* por sus caminos desde el elemento principal u otro elemento raíz. Estos ajustes manuales se podrían haber evitado con un *ejemplo negativo* que incluyera dos tareas consecutivas relacionadas con el mismo rol.

Cuando se aplicó la TM propuesta en este apartado, se generaron las unidades de interacción que existían en la especificación original, pero con nombres menos apropiados (e.g. *AnswerQuestionnaireEndOfTheRoundIU* en lugar de *AnswerQuest*). Además, no se generaron todos los elementos necesarios para definir los protocolos de interacción: faltaba establecer el orden de las unidades de interacción y relacionar las mismas con las interacciones. Algunas de estos elementos no se crean porque se necesita TMs complejas que no se pueden generar con el algoritmo y herramienta propuestos, dado que requerirían que las TMs tuvieran en cuenta la semántica de los nombres y del problema. Por ejemplo, la agrupación y secuencia de unidades de interacción para formar protocolos de interacción requiere entender la semántica de la información que se va a transmitir para ver cuál es la mejor distribución.

6.4.4 Generación de despliegues y tests

Al final del modelado con INGENIAS, los diseñadores especifican el número de los agentes instanciados para el SMA por medio de los *despliegues*. Se propone automatizar este paso con una TM. La TM especificada en la Figura 6.10 crea un despliegue con una instancia para cada agente. Este despliegue creado por defecto por la TM puede ser alterado después por los diseñadores. En esta TM, se puede observar que un prototipo de modelo destino puede contener valores constantes tal y cómo se explicó en el apartado 5.3.3. En este caso, el atributo es el *número de instancias* (i.e. *ninstances*) y el valor es *1*.

La metodología INGENIAS también recomienda definir una configuración de test para cada despliegue. Por esta razón, se propone la TM especificada en la Figura 6.11, que puede crear los elementos de modelado requeridos para este propósito.

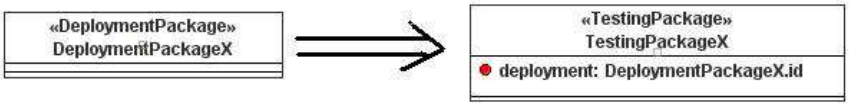


Figura 6.11: TM *Deployment2Testing* para crear los *paquetes de test*, con una regla uno-a-uno.

Cuando se aplicaron estas TMs al SMA de Delphi, se obtuvo tanto un despliegue como un test que consideraban un agente Cliente, un agente Monitor y un agente Experto. Se necesitaban varios agentes Expertos para que llegaran a una decisión consensuada y el despliegue por defecto generado por la TM sólo contiene un agente Experto. Por tanto, hubo que cambiar manualmente el número de agentes Expertos en el despliegue.

6.4.5 Reutilización de las transformaciones generadas

Para experimentar la aplicación de TMs creadas por GTBE en varios casos diferentes, las TMs presentadas en este capítulo fueron aplicadas en un SMA de *gestión de crisis* (véase el apéndice C.1). En este SMA, los nombres de todos los elementos eran diferentes al SMA de Delphi, pero el mecanismo de asociación de atributos (descrito en el apartado 5.2.6) permite transferir la información de los atributos y los nombres de los elementos, independientemente del SMA.

Además, las TMs se pudieron aplicar a diferentes estructuras. En este aspecto, es especialmente relevante la aplicación de las TMs referentes al flujo de trabajo especificadas en las Figuras 6.6 y 6.7, dado que los flujos de trabajo del SMA de Delphi y del SMA de gestión de crisis tienen estructuras diferentes. En concreto, el flujo de trabajo del SMA de Delphi genera iteraciones de tareas ejecutadas, mientras que el flujo de trabajo del SMA de gestión de crisis es una secuencia de tareas ejecutadas que incluye una bifurcación y nunca llega a repetir una tarea.

Las siete TMs presentadas en este capítulo se pudieron aplicar al SMA de gestión de crisis. Los detalles más relevantes de esta reutilización de TMs se indican en el apéndice C.1.3. Cabe mencionar que esta reutilización fue posible porque se usó el mismo proceso de desarrollo. Si se hubiera seguido otro proceso de desarrollo, se hubieran tenido que definir algunas nuevas TMs. Aun así, las TMs presentadas son genéricas y otros procesos de desarrollo podrían reutilizar algunas de ellas.

6.5 Tipos de transformaciones generadas

Algunas TMs generadas por la herramienta *MTGenerator* han sido aplicadas en el modelado de SMAs existentes. Además de las TMs mostradas, esta herramienta ha generado otras TMs que están recogidas en el apéndice C y que han sido probadas aplicándolas a SMAs existentes. Concretamente, como breve introducción a estas TMs, la TM *UseCase2Interaction* genera, a partir de un caso de uso, la interacción que lo implementa. Por otro lado, la TM *InteractionDefinition2-InteraactionProtocol* genera el protocolo de una interacción a partir de la defini-

ción de la misma. La TM *AddSkill* añade parte de los elementos de INGENIAS que se necesitan para añadir una nueva habilidad a un agente dado. Por último, la TM *RefactoringADELFE* crea los elementos de ADELFE referentes a las percepciones y acciones de comunicación a partir de las definiciones de los mensajes. Todas estas TMs se encuentran descritas con más detalle en el apéndice C.

Las TMs generadas por la aproximación de esta tesis como parte de su experimentación se resumen en la Tabla 6.1 indicando, entre otras características, el tipo de regla. En esta tabla, *1:1*, *1:N* y *N:N* denotan respectivamente los tipos de reglas *uno-a-uno*, *uno-a-muchos* y *muchos-a-muchos*. Considerando las once TMs estudiadas, siete de éstas no pueden ser generadas por otras herramientas existentes de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b), porque no dan un soporte a: una asociación de atributos desde varios elementos de entrada de una regla, y reglas que traten con grafos no conexos. La generación de este tipo de reglas es necesaria para desarrollar los SMAs de los ejemplos vistos.

Además, nueve de las once TMs (véase la Tabla 6.1) no requirieron ningún ajuste manual después de la generación. Sin embargo, dos TMs requirieron ajuste manual debido a que fue necesario añadir una restricción adicional. La restricción que fue añadida a la TM *InitialTaskWorkflow* se describe en el apartado 6.4.2, y la que fue añadida a la TM *InteractionUnit* se describe en el apartado 6.4.3. Para evitar estos ajustes manuales, en el futuro el algoritmo puede considerar *ejemplos negativos* (Varró y Balogh, 2007b), que evitan la ejecución de un regla en caso de encajar con el modelo origen. Para esto, el algoritmo podría incorporar la generación de restricciones negativas a partir de ejemplos negativos. Para ilustrar este tipo de prototipos, se proponen dos ejemplos negativos, uno para la TM *InitialTaskWorkflow* en el apartado 6.4.2 y otro para la TM *InteractionUnit* en el apartado 6.4.3.

Es importante mencionar que la TM *AddSkill* (véase el apéndice C.2.2) utiliza grafos no conexos en una regla de transformación, característica para la que ninguna de las otras aproximaciones de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b) da soporte. El algoritmo de GTBE presentado en esta tesis da soporte a este tipo de reglas, considerando todas las raíces lógicas del modelo prototipo origen, como se explicó en el apartado 5.2.4, y todas las raíces lógicas del modelo prototipo destino, como se explicó en el apartado 5.2.5.

Por último, la herramienta *MTGenerator* también ha sido aplicada sobre los metamodelos usados por metodologías diferentes de INGENIAS. En particular, este trabajo muestra su aplicación en la metodología ADELFE de ISOA, como se puede ver en la TM *RefactoringADELFE*, que se describe con más detalle en el apéndice C.3. En la generación de dicha TM, se usa una asociación de atributos que concierne a varios elementos de entrada y varias elementos de salida en las reglas de transformación, lo que no es posible con otras aproximaciones de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b).

6.6 Conclusiones

Este capítulo ha recogido la experimentación que permite evaluar la aproximación de GTBE propuesta en esta tesis. Para ello, se han presentado ejemplos de GTBE en un SMA existente (i.e. un SMA para la evaluación de documentos con el proceso *Delphi*), y las TMs obtenidas se han validado por comparación

	Tipo de regla	Puede ser generado por otra herramienta de GTBE	Requiere ajuste manual	Breve Descripción	Descrito con más detalle en
UseCase-2Agent	N:N	No	No	Define agentes a partir de casos de uso.	Apartado 6.4.1 y (García-Magariño et al., 2009,b)
Initial-Task-Workflow	1:N	Sí	Sí	Define la tarea inicial del flujo de trabajo.	Apartado 6.4.2 y (García-Magariño et al., 2009)
NonInitial-Task-Workflow	N:N	No	No	Define las tareas no iniciales del flujo de trabajo.	Apartado 6.4.2 y (García-Magariño et al., 2009)
Task2-Code-Component	1:N	Sí	No	Genera los componentes de código para las tareas.	Apartado 6.4.2 y (García-Magariño et al., 2009)
InteractionUnit	N:N	No	Sí	Genera la unidad de interacción que comunica dos tareas.	Apartado 6.4.3 y (García-Magariño et al., 2009)
Agent2-Deployment	1:1	Sí	No	Crea los despliegues.	Apartado 6.4.4 y (García-Magariño et al., 2009)
Deployment2-Testing	1:1	Sí	No	Crea los tests.	Apartado 6.4.4 y (García-Magariño et al., 2009)
UseCase2-Interaction	N:N	No	No	Crea interacciones a partir de casos de uso.	Apéndice C.2.1 y (García-Magariño et al., 2009a)
Interaction-Definition2-Interaction-Protocol	N:N	No	No	Crea protocolos de interacción a partir de las definiciones de las interacciones.	Apéndice C.2.1 y (García-Magariño et al., 2009a)
AddSkill	1:N	No	No	Añade una habilidad a un agente, con todos los elementos de modelado necesario.	Apéndice C.2.2 y (García-Magariño et al., 2009a)
Refactoring-ADELFE	N:N	No	No	Especifica la comunicación directa entre agentes a partir de un mensaje.	Apéndice C.3 y (García-Magariño et al., 2009b)

Tabla 6.1: TMs generadas por la herramienta MTGenerator.

de sus resultados con la especificación original del SMA obtenida por la manera tradicional de INGENIAS. Además, se ha descrito brevemente la reutilización de estas TMs en otro dominio (i.e. *gestión de crisis*), y se referenciado otros ejemplos prácticos de GTBE. Con la experimentación presentada en este capítulo, se ilustran las siguientes ventajas y limitaciones:

- *Se generan tipos de TMs que no se pueden generar por otras aproximaciones de GTBE* (Wimmer et al., 2007; Varró y Balogh, 2007b). En primer lugar, dentro de las reglas muchos-a-muchos, independientemente de si la entrada y la salida de la regla son grafos conexos o no-conexos, nuestra aproximación permite hacer asociaciones de atributos entre varios elementos de entrada y varios elementos de salida para que las reglas puedan transferir información. Esto permite combinar la información de atributos de diferentes elementos de entrada para formar atributos de los elementos de salida. Por el contrario, la asociación de atributos en otras aproximaciones de GTBE sólo permite recibir información de un elemento de la entrada de cada regla, siendo imposible generar TMs que combinen información de atributos de diferentes elementos. Además, esta aproximación permite generar reglas de transformación que trabajen con grafos no conexos tanto en la entrada como en la salida. Sin embargo, otras aproximaciones de GTBE no permiten generar reglas que encajen con grafos no conexos en la entrada.

- *Se puede ahorrar tiempo en algunos desarrollos.* El hecho de automatizar alguna tareas implica un ahorro en esfuerzo y la reducción de potenciales errores. Cuantificar este ahorro requiere realizar más pruebas de forma exhaustiva. Aunque estas pruebas aún no se han realizado, se planea hacer esta demostración en el futuro (véase el apartado 8.2). Los siguientes aspectos de los ejemplos de SMAs existentes usados en esta tesis indican que se podría haber ahorrado tiempo en sus desarrollos:

  - *La definición de modelos prototipo es más sencilla que la escritura de código de las TMs*, especialmente para los diseñadores de SMAs, que no suelen estar habituados a los LTMs existentes.
  - *La misma TM se pudo aplicar varias veces en un mismo desarrollo.* Esto es habitual en TMs definidas manualmente. Sin embargo, es necesario comprobarlo explícitamente en las TMs generadas a partir de ejemplos, dado que los ejemplos son muy concretos y es difícil garantizar que las TMs generadas se puedan aplicar a más casos y que estos sean los deseados. Por ejemplo en nuestra experimentación, dos transformaciones relativamente sencillas pudieron transformar un flujo de trabajo de complejidad media, generando elementos para cada paso del mismo.
  - *Las mismas TMs se pudieron aplicar en varios desarrollos diferentes.* Como se ha mencionado anteriormente, esto es habitual en las TMs definidas manualmente pero debe comprobarse explícitamente en aproximaciones GTBE. En la experimentación presentada, obsérvese que se aplicaron las siete TMs creadas para el SMA de Delphi en el SMA de gestión de crisis. Cabe destacar que se aplicaron a flujos de trabajo con una estructura diferente. Aun así, también hay que indicar que se pudieron reutilizar porque se siguió el mismo proceso de

desarrollo; en caso contrario, no hubiera sido posible la reutilización de todas las TMs.

- *El algoritmo y herramienta de GTBE presentados son útiles en algunos desarrollos de la ISOA*. Esto se puede observar en los ejemplos de TMs creadas por GTBE para el SMA de Delphi. Además, se han mencionado otros ejemplos prácticos de TMs creadas por GTBE, que se describen con más detalle en el apéndice C. Aunque la experimentación de esta tesis sugiere que la aproximación GTBE presentada se puede aplicar en la mayoría de desarrollos de ISOA, confirmarlo requiere aún experimentación adicional que es uno de los principales objetivos del trabajo futuro.

- *Fueron necesarios ajustes manuales limitados*. Los ajustes manuales necesarios después de la generación estuvieron relacionados con la inserción de ciertas restricciones. Sin embargo, las restricciones requeridas podrían haber sido generadas si el algoritmo diera soporte a los *ejemplos negativos* (Varró y Balogh, 2007b). En versiones futuras del algoritmo, se planea dar soporte a dichos ejemplos negativos.

En conclusión, el algoritmo de GTBE y su implementación, *MTGenerator*, han facilitado la generación de tipos de TMs que son necesarios en la ISOA y a los que no se les daba soporte en el resto de aproximaciones de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b). Esta tesis ha mostrado algunos ejemplos reales de SMAs para los que es útil la aproximación de GTBE presentada; sin embargo, se tiene como objetivo futuro determinar si realmente la GTBE es útil en la mayoría de desarrollos de ISOA, considerando un rango más amplio de SMAs.

Capítulo 7

Conclusiones

Este capítulo final recoge las principales aportaciones de esta tesis, e introduce brevemente las líneas futuras de investigación.

7.1 Aportaciones

Esta tesis hace dos aportaciones básicas: la guía de definición de metamodelos y el algoritmo de GTBE.

La guía permite que los diseñadores puedan definir metamodelos, siendo guiados en las tareas más complejas. Propone un armazón que separa en capas la información específica de las herramientas CASE, los aspectos de presentación y el LM.

La información específica de las herramientas CASE no se considera en los metamodelos de la literatura, pero en la práctica las herramientas CASE necesitan usarla y la mezclan con los modelos. Esta tesis propone modelar este tipo de información y separarla del resto de información.

En cuanto a la capa de presentación, se propone definir vistas (i.e. diagramas) que referencian a un modelo abstracto global. Esto se contrapone a la aproximación de UML-DI, que es la más extendida. La ventaja de nuestra aproximación es que la aparición de un mismo elemento en varios diagramas no genera problemas en el procesamiento del modelo abstracto, dado que éste se encuentra unido y sin elementos repetidos.

En cuanto a la sintaxis abstracta de los LMs, se propone un catálogo de representaciones para los elementos. Luego, una serie de actividades asocian la representación adecuada para cada elemento. En esta serie de actividades es necesario tomar también las siguientes decisiones relacionadas con los requisitos de la aplicación de los metamodelos. Primero, se debe elegir entre una representación heterogénea y una representación homogénea. Ambos tipos de representaciones se encuentran en los metamodelos existentes, pero nuestra guía es la primera que indica pautas para elegir entre una y otra, contraponiendo la facilidad de procesamiento de los modelos y el uso de menor número de elementos. La guía también contiene recomendaciones para decidir entre las representaciones redundantes y no-redundantes, que contraponen la navegabilidad de los modelos con el coste de garantizar la consistencia.

La guía de definición de metamodelos ha permitido definir el metamodelo del IDK con ECore, permitiendo aplicar las nuevas tecnologías del DSDM en el IDK. Además, con la guía se ha definido un metamodelo que ha permitido generar un editor de procesos de ISOA con bajo coste. El editor ha permitido modelar procesos de ISOA, tales como el proceso SCRUM para INGENIAS, usando conceptos que no estaban disponibles en otros editores existentes para procesos, tales como EPF y APES2.

Esta tesis también ha presentado un nuevo algoritmo para GTBE. Su novedad es que permite asociar atributos entre los modelos prototipo considerando varios elementos en cada modelo prototipo origen, generando reglas de transformación que transfieren información desde varios elementos de la entrada. Sin embargo, las otras aproximaciones sólo permitían asociar los atributos de un único elemento en cada modelo prototipo origen. Más aún, con las anteriores aproximaciones no se podía combinar atributos de elementos diferentes (e.g. concatenar cadenas de caracteres procedentes de dichos elementos), mientras que con la aproximación presentada esto es posible. La aproximación presentada también permite tratar con grafos no conexos dentro de los modelos prototipo origen y destino, lo que no se permitía en otras aproximaciones anteriores.

Esta tesis ha presentado este algoritmo basándose en un pseudo lenguaje de transformaciones, facilitando así su posible implementación en diferentes lenguajes de transformaciones existentes. Si bien existen lenguajes que soportan directamente algunas de las manipulaciones resueltas por el algoritmo presentado, tales como permitir definir reglas con varios elementos de entrada, no son los más extendidos. De hecho, algunos de los lenguajes de transformación más extendidos (e.g. ATL y XSLT) sólo permiten definir reglas de transformación con un elemento en la entrada. La manera de definir reglas muchos-a-muchos en estos lenguajes es por medio de restricciones, lo que produce habitualmente un código largo y verboso. Nuestro algoritmo genera este código, evitando que el usuario tenga que escribirlo manualmente. Como prueba de concepto, el algoritmo se ha implementado para ATL e incluido en la herramienta *MTGenerator*, que está públicamente disponible.

Para la experimentación de la aproximación de GTBE se usó la herramienta *MTGenerator* y la versión del IDK que adopta el metamodelo definido por la guía. En esta experimentación, se han presentado ejemplos de transformaciones generadas que se pueden aplicar en el desarrollo de SMAs existentes. Siete de las once transformaciones propuestas no se pueden generar con otras aproximaciones de GTBE.

En conclusión, esta tesis contribuye a la implantación del DSDM en la ISOA. Para ello proporciona nuevos mecanismos para la definición y transformación de modelos.

7.2 Futuras líneas de investigación

En primer lugar, la guía de definición de metamodelos todavía sólo se ha experimentado con la configuración más completa (i.e. metamodelos Objetos, Concreto, Vistas, Layouts, y Proyecto) y con la configuración más simple (i.e. metamodelo Objetos). Se planea definir metamodelos que usen el resto de las configuraciones propuestas con el objetivo de probar la utilidad de las mismas. Por ejemplo, se puede definir un metamodelo para una herramienta CASE con un

LM *basado en conexiones* que no necesite información espacial (i.e. Configuración 5).

Además, la guía de definición de metamodelos podría incorporar la posibilidad de incluir *diagramas anidados*. Éstos son útiles para detallar el elemento de un diagrama con otro diagrama y, así, tener varios niveles de detalle. Para cumplir este objetivo, se puede incorporar en el metamodelo un elemento llamado *Diagrama Anidado*, que se incluya en los diagramas (i.e. elementos *Graphs* en el metamodelo *Vistas*) e incluya una referencia a otros diagramas (i.e. otros elementos Graphs).

La guía también puede incluir restricciones para garantizar una instanciación correcta de los metamodelos. Por ejemplo, los elementos de un diagrama (i.e. *GraphElement*) sólo se deben conectar a elementos del mismo diagrama. Esta restricción se puede formalizar para garantizar la corrección de los modelos con, por ejemplo, OCL. En esta línea, se planea estudiar y definir otras restricciones que sean necesarias para forzar la correcta definición de los metamodelos, así como establecer pautas en la guía para definir restricciones al igual que se hace con el resto de la definición de los metamodelos.

También se planea realizar una guía para el intercambio de modelos entre herramientas CASE, basándose en el armazón de metamodelos propuesto en esta tesis. La guía seguiría la separación de información propuesta por el armazón: sintaxis abstracta, vistas, distribución espacial e información específica de cada herramienta. Para ello, se puede recomendar localizar cada uno de estos tipos de información en los modelos de cada herramienta, y transformar cada tipo de información por separado. En esta línea, se ha realizado un análisis preliminar del intercambio de modelos entre la herramienta IDK modernizada y otras dos herramientas usadas en la ISOA, que son PDT e IBM-RSD.

Una desventaja de la guía para la sintaxis abstracta (i.e. metamodelo Objetos) es su acoplamiento con el lenguaje ECore. Si bien se cree que existe la posibilidad de adaptar las recomendaciones de esta guía a otros lenguajes de meta-modelado, no se proporciona ninguna evidencia al respecto más que un estudio preliminar con MOF. Se planea producir nuevas versiones de esta guía adaptadas a otros lenguajes de meta-modelado, tales como MOF y KM3.

En cuanto a la GTBE, un objetivo a corto plazo es incluir la posibilidad de usar *ejemplos negativos* en el algoritmo propuesto. Como se ha observado, esta mejora puede ahorrar ajustes manuales para ciertas TMs. Para dar soporte a los ejemplos negativos, se planea generar restricciones de forma similar a como se hace para los modelos prototipo origen, y añadir el operador lógico de negación a dichas restricciones.

Otra línea de investigación es determinar si las TMs generadas por nuestra aproximación pueden soportar desarrollos completos de SMAs. Aunque posiblemente siempre se requerirá algún tipo de ajuste manual en algunos casos, se quiere determinar el esfuerzo que se necesita para realizar esos ajustes y completar el resultado producido por las TMs hasta obtener especificaciones que puedan generar SMAs funcionales. Estas pruebas se aplicarán a un rango más amplio de desarrollos de SMAs que en esta tesis. Estos resultados permitirán evaluar objetivamente las ganancias en la ISOA que resultan de un DSDM soportado por las herramientas presentadas en este trabajo.

Como parte del futuro, se implementará el algoritmo de GTBE para otros lenguajes de transformación distintos de ATL. Por un lado, esto constituiría una evidencia de que el algoritmo propuesto se puede aplicar a varios lenguajes; por

otro lado, nuestra aproximación se podría aplicar a más modelos y no estaría restringida a las instancias de los metamodelos expresados con ECore. En concreto, se planea empezar implementando el algoritmo para el lenguaje XSLT, lo que permitirá realizar la GTBE en documentos XML, los cuales son usados por algunas herramientas de ISOA, tales como PDT.

Por último, la aplicación de TMs sufre el problema de la *trazabilidad*; es decir, algunas inconsistencias pueden surgir cuando se modifican modelos que fueron generados por TMs. Esto podría ocurrir en los ejemplos de experimentación presentados en el capítulo 6, en los que el usuario puede necesitar cambiar los elementos creados. Por ejemplo, el *paquete de despliegue* generado suele tener que alterarse para cambiar el número de instancias de cada agente (e.g. tuvo que ser cambiado para el SMA de Delphi como se indicó en el apartado 6.4.4). En este caso, si se añade un nuevo tipo de agente y se regenera el paquete de despliegue, se pierde la información respecto al número de agentes de cada tipo. En un futuro, se planea incorporar mecanismos para solventar la trazabilidad en nuestra aproximación, permitiendo crear TMs que no sobrescriban cierta información o crear TMs en el sentido inverso a unas TMs dadas. Para esto será necesario incluir ciertas *trazas* que permitan evitar situaciones de ambigüedad almacenando la información necesaria.

7.3 Publicaciones relacionadas

A lo largo del trabajo de investigación que ha implicado el desarrollo de esta tesis, se han producido una serie de publicaciones que se detallan en los siguientes sub-apartados. Las puntuaciones de las publicaciones se han indicado para los siguientes índices:

- *Journal Citation Report* (JCR)
- *COmputing Research and Education* (CORE)
- *Computer Science Conference Ranking* (CSCR)
- *CiteSeer*

Artículos en revistas

1. Iván García-Magariño, Rubén Fuentes-Fernández, y Jorge J. Gómez-Sanz. Guideline for the definition of EMF metamodels using an Entity-Relationship approach. *Information and Software Technology*, 51 (8):1217–1230. Elsevier, 2009. doi: 10.1016/j.infsof.2009.02.003. **JCR 2008 1.200, Segundo Cuartil**.

2. Iván García-Magariño and Rubén Fuentes-Fernández. A Technique for Defining Metamodel Translations. *IEICE Transactions on Information and Systems*, 2009 (aceptado para publicación). **JCR 2008 0.369, Cuarto Cuartil**.

Artículos en congresos

1. Iván García-Magariño, Jorge J. Gómez-Sanz, y Juan Pavón. Representación de las Relaciones en los Metamodelos con el Lenguaje Ecore. En *Desarrollo del Software Dirigido por Modelos*, páginas 11–20, 2007. 11 Septiembre, 2007, Zaragoza España.

2. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. Reaching Consensus in a Multi-agent System. En *6th International Workshop on Practical Applications on Agents and Multi-agent Systems, IWPAAMS'07*, páginas 349–358, 2007. 12-13 Noviembre, 2007, Salamanca España. **CSCR 0.56 (pos. 52/701)**.

3. Iván García-Magariño, Alma Gónzalez-Rodríguez, y Juan C. González. Modelando el Proceso de Desarrollo de INGENIAS con EMF. En *6th International Workshop on Practical Applications on Agents and Multi-agent Systems, IWPAAMS'07*, páginas 369–378, 2007. 12-13 Noviembre, 2007, Salamanca España. **CSCR 0.56 (pos. 52/701)**.

4. Iván García-Magariño y Jorge J. Gómez-Sanz. Framework for Defining Model Language Metamodels for CASE Tools. En *The Fifth International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MOMPES'08*, páginas 14–23. IEEE Computer Society, 2008. 5 Abril, 2008, Budapest Hungría.

5. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. A Complete-Computerised Delphi Process with a Multi-agent System. En *Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volumen 5442 de *Lecture Notes in Computer Science*, páginas 120–135. Springer, 2009.

6. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. An Evaluation Framework for MAS Modeling Languages based on Metamodel Metrics. En *Agent-Oriented Software Engineering IX (Revised Selected Papers)*, volumen 5386 de *Lecture Notes in Computer Science*, páginas 101–115. Springer, 2009. **CORE B, CiteSeer 1.57 (top 7.45 %)**.

7. Iván García-Magariño, Alma Gónzalez-Rodríguez, y Juan C. González. Definition of Process Models for Agent-based Development. En *Agent-Oriented Software Engineering IX (Revised Selected Papers)*, volumen 5386 de *Lecture Notes in Computer Science*, páginas 60–73. Springer, 2009. **CORE B, CiteSeer 1.57 (top 7.45 %)**.

8. Jorge J. Gómez-Sanz, Rubén Fuentes-Fernández, Juan Pavón, y Iván García-Magariño. INGENIAS Development Kit: a visual multi-agent system development environment (Awarded as Best Academic Demo of AAMAS'08). En *The Seventh International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08*, páginas 1675–1676. ACM, 2008. 12-16 Mayo, 2008, Estoril Portugal. **CORE A+, CSCR 0.76 (pos. 21/701)**.

9. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. A Multi-Agent Based Implementation of a Delphi Process. En *The Seventh International Conference on Autonomous Agents and Multiagent Systems,*

AAMAS'08, páginas 1543–1546, ACM, 2008. 12-16 Mayo, 2008, Estoril Portugal. **CORE A+, CSCR 0.76 (pos. 21/701)**.

10. Iván García-Magariño, Celia Gutiérrez, y Rubén Fuentes Fernández. Organizing multi-agent systems for crisis management. En *7th Ibero-American Workshop in Multi-Agent Systems (Iberagents 2008)*, páginas 69–80, 2008. 14 Octubre, 2008, Lisboa Portugal.

11. Iván García-Magariño, Alma Gómez-Rodríguez, Juan C. González, y Jorge J. Gómez-Sanz. INGENIAS-SCRUM Development Process for Multi-Agent Development. En *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volumen 50 de *Advances in Software Computing*, páginas 108–117. Springer, 2009.

12. Iván García-Magariño, Alma Gómez-Rodríguez, y Juan C. González. Modeling Processes of AOSE Methodologies by Means of a New Editor. En *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volumen 50 de *Advances in Software Computing*, páginas 672–681. Springer, 2009.

13. Iván García-Magariño. Towards the Coexistence of Different Multi-Agent System Modeling Languages with a Powertype-Based Metamodel. En *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volumen 50 de *Advances in Software Computing*, páginas 189–193. Springer, 2009.

14. Iván García-Magariño, Rubén Fuentes-Fernández, y Jorge J. Gómez-Sanz. INGENIAS development process assisted with chains of transformations. En *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, volumen 5517 de *Lecture Notes in Computer Science*, páginas 514–521. Springer, 2009. **CORE B, CiteSeer 0.16 (top 79.85 %), CSCR 0.55 (pos. 55/701)**.

15. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. En *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, volumen 55 de *Advances in Soft Computing*, páginas 40–49. Springer, 2009. **CSCR 0.56 (pos. 52/701)**.

16. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages. En *International Conference on Model Transformation (ICMT 2009)*, volumen 5563 de *Lecture Notes in Computer Science*, páginas 52–66. Springer, 2009. **22 % de ratio de aceptación**.

17. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. Model Transformations for Improving Multi-agent Systems Development in INGENIAS. En *The 10th International Workshop on Agent-Oriented Software Engineering AOSE'09*, 2009. 11 Mayo, 2009, Budapest Hungría. **CORE B, CiteSeer 1.57 (top 7.45 %)**.

18. Iván García-Magariño, Celia Gutiérrez, y Rubén Fuentes-Fernández. The INGENIAS Development Kit: a practical application for crisis-management. En *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, volumen 5517 de *Lecture Notes in Computer Science*, páginas 537–544. Springer, 2009. **CORE B, CiteSeer 0.16 (top 79.85 %), CSCR 0.55 (pos. 55/701)**.

19. Iván García-Magariño, Sylvain Rougemaille, Rubén Fuentes Fernández, Frédéric Migeon, Juan Pavón Mestras, y Marie-Pierre Gleizes. Towards Agent-Oriented Model Driven Engineering. En *6th European Workshop on Multi-Agent Systems (EUMAS 2008)*, páginas 69–70, 2008. 18-19 Diciembre, 2008, Bath Reino Unido.

20. Iván García-Magariño, Sylvain Rougemaille, Rubén Fuentes-Fernández, Frédéric Migeon, Marie-Pierre Gleizes, y Jorge J. Gómez-Sanz. A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. En *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, volumen 55 de *Advances in Soft Computing*, páginas 70–79. Springer, 2009. **CSCR 0.56 (pos. 52/701)**.

Chapter 8

Conclusions

This chapter summarizes the main contributions of this thesis, and briefly introduces the future lines of research.

8.1 Contributions

The two main contributions of this thesis are: the guideline for defining metamodels and the algorithm for *Model Transformation By-Example* (MTBE).

The guideline allows designers to define metamodels, advising them in the most complex tasks. The guideline proposes a meta-modeling framework that separates the tool-specific information, the presentation aspects and the *Modeling Language* (ML) in layers.

In the literature, other metamodels do not consider the information that is specific of *Computer-Aided Software Engineering* (CASE) tools, but CASE tools need to use this information and they usually mix it with models. This thesis proposes to explicitly model this kind of information and to keep it apart from the other information of the metamodel.

Concerning the presentation aspects, this thesis proposes to define views (i.e. diagrams) that refer to a global abstract model. This is different from the *UML Diagram Interchange* (UML-DI) approach, which is currently the most extended. The advantage of our approach is that the appearance of an element in several diagrams does not imply problems in processing the abstract model, because the abstract model is common and without repeated elements. In other words, our views are logical diagrams intended to ease the understanding of a unique abstract model. Traditional approaches slice this model in different diagrams, which causes problems in certain kinds of processing for nested information.

Related to the abstract syntax of MLs, this guideline provides a catalogue of representations for elements. Moreover, the guideline proposes a sequence of activities that associates the most appropriate representation with each element. This guideline also consider some additional design decisions. Firstly, designers decide between a homogeneous and a heterogenous representation. Both types of representations are found in existent metamodels, but our guideline is the first one that includes recommendations for choosing between them, presenting a trade-off between the easiness for processing models and the use of a lower

number of elements. The guideline also suggests the way for choosing between redundant and non-redundant representations by reaching a compromise between navigability of models and the cost for guaranteeing their consistency. A redundant representation allows bidirectional navigability of relationships when processing models, but requires couples of references. On the contrary, the non-redundant representation uses single references but does not allow inverse navigation.

The metamodel of the *INGENIAS Development Kit* (IDK) has been defined with the proposed guideline and the ECore language, allowing the use of the new technologies of *Model Driven Development* (MDD) in the IDK. In addition, the guideline has driven the definition of a metamodel from which a process editor tool has been generated with a low cost. This editor tool has facilitated the definition of *Agent Oriented Software Engineering* (AOSE) processes, such as the SCRUM process for INGENIAS, using concepts that are not available in existent process editors, like EPF and APES2.

This work has also introduced a new algorithm for MTBE, which has two main innovations. First, it allows designers to associate attributes between prototype models considering several elements in each source prototype model for generating transformation rules that receive information from several elements. However, other approaches only can map attributes from one element of each source prototype. They cannot combine attributes from different elements (e.g. concat strings belonging to several elements), while this is possible in the approach of this thesis. Second, the presented algorithm allows processing non-connected graphs in the prototype models, which is not possible in previous approaches.

This thesis has introduced this algorithm by means of a pseudo transformation language, facilitating its implementation in several existent transformation languages. Although there are several languages that can define transformation rules with several input elements, in some of the most widespread transformation languages (e.g. ATL y XSLT) each transformation rule can only have one input element. The way of defining many-to-many rules is to include constraints in rules, which usually produces long and verbose code. Our algorithm generates this kind of code preventing users from manually writing it. As a proof of concept, the algorithm has been implemented for ATL and this implementation is included in the *MTGenerator* tool, which is publicly available.

The experimentation of the presented MTBE approach used the *MTGenerator* tool and the IDK version that adopts the metamodel defined with the guideline. This thesis has presented examples of generated transformations that can be applied in the development of existent Multi-Agent Systems (MASs). Seven out of the eleven proposed transformations cannot be generated by other MTBE approaches.

On the whole, this thesis constitutes a step forward the incorporation of MDD in AOSE, providing new mechanisms for defining and transforming models.

8.2 Future lines of research

The guideline for defining metamodels has only been experimented with the configurations covering the complete framework (i.e. Objets, Concrete, Views;

Layouts and Project metamodels) and the simplest subset (i.e. Objects metamodel). This work is planning to define metamodels that use the remaining proposed configurations in order to prove their utility. For instance, the guideline can define a metamodel for a CASE tool that manages a connection-based ML and does not need to include spatial information (i.e Configuration 5).

Moreover, the guideline for defining metamodels can incorporate the possibility of including *nested diagrams*. These are useful for detailing a diagram element with another diagram and, consequently, including several levels of details. For this purpose, metamodels can incorporate an element called *Nested Diagram*, which is included in diagrams (i.e. *Graph* elements in the *Views* metamodel) and includes references to other diagrams (i.e. other *Graph* elements).

The guideline can also include constraints for guaranteeing a correct instantiation of metamodels. For instance, the elements of a diagram (i.e. *GraphElement*) must be only connected to the elements of the same diagram. This constraint can be formalized with, for example, OCL. The use of constraints can extend the guideline in two ways. First, the meta-modeling framework can include certain standard constraints to guarantee certain properties that are not considered in the current version, like the previously mentioned constraint. Second, the guideline can incorporate advices to extend metamodels with arbitrary constraints regarding the needs of designers.

Furthermore, our research group is planning to create another guideline for model interchange in CASE tools, based on the meta-modeling framework proposed in this thesis. The guideline will use the separation of information proposed by the framework: abstract syntax, views, spatial distribution and the tool-specific information. For this purpose, the new guideline can recommend to find each kind of information in the models of each tool, and to separately transform each kind of information. In this line of work, a preliminar study has analyzed the model interchange between the modernized version of IDK and two other tools, an AOSE tool called *Prometheus Design Tool* (PDT) and another general-purpose tool called *IBM Rational Systems Developer* (RSD).

A disadvantage of the guideline for the abstract syntax (i.e. Objects metamodel) is its coupling with the ECore language. Although the recommendations of this guideline are general to be adapted to other meta-modeling languages, no evidence is provided to show this. This guideline is planned to be adapted to other meta-modeling languages, such as MOF and KM3 to prove this aspect.

Coupled with MTBE, a short-term goal is to include the possibility of using *negative examples* in the proposed algorithm. As one can observe in this thesis, this improvement can avoid manual modifications in certain *Model Transformations* (MTs). For supporting negative examples, it is planed to generate constraints in a similar way that is done for source prototype models, and to add the logic operator of negation to these constraints.

Another future work is to determine whether the MTs that are generated by our approach can support complete developments of MASs. Bearing this goal in mind, practitioners will analyze MTs to determine the effort required for its manual adjustment if necessary (for instance, measuring the time that it takes). They also will analyze the models that are produced by these MTs, in order to determine the effort to complete them for obtaining the models that can generate functional MASs. This experiment is planned to use a wider range of MAS developments than the range used in this thesis. In this manner, these

experiments are expected to objectively evaluate the gain of applying MDD in AOSE when using the approach presented in this thesis.

The algorithm of MTBE is planned to be implemented in other transformation languages different from ATL. This would provide evidence that the proposed algorithm can be implemented for different transformation languages. Moreover, this will make that our approach can be applied to more models and not be restricted to the instances of metamodels that are expressed with ECore. In particular, it is planned to start with the implementation for the XSLT language, allowing one to apply MTBE in XML documents, which are used by some AOSE tools, such as PDT.

Finally, the problem of *traceability* is usually related to MTs; in other words, some inconsistencies may occur when modifying models that were generated by MTs. This could happen in the experimentation examples presented in (García-Magariño et al., 2009), in which users may need to change the created elements. For instance, the generated *Deployment Packages* usually need to be modified for changing the number of instances of each agent, e.g. this was changed for the Delphi MAS (García-Magariño et al., 2009). In this case, if a new type of agent is added and the deployment package is regenerated, then the information of the number of agents of each type will be lost. A long-term goal is to incorporate mechanisms for solving the traceability in our approach, allowing one to create MTs that do not overwrite certain information or are inverse of given MTs. For this purpose, it may be necessary to include certain *traces* that avoid ambiguous situations by storing some information.

8.3 Related publications

The work of this thesis has implied several publications, which are mentioned in te following sub-sections. The scores of the publications are indicated for the following rankings:

- *Journal Citation Report* (JCR)
- *COmputing Research and Education* (CORE)
- *Computer Science Conference Ranking* (CSCR)
- *CiteSeer*

Journals

1. Iván García-Magariño, Rubén Fuentes-Fernández, y Jorge J. Gómez-Sanz. Guideline for the definition of EMF metamodels using an Entity-Relationship approach. *Information and Software Technology*, 51 (8):1217–1230. Elsevier, 2009. doi: 10.1016/j.infsof.2009.02.003. **JCR 2008 1.200, Second Quartile**.

2. Iván García-Magariño and Rubén Fuentes-Fernández. A Technique for Defining Metamodel Translations. *IEICE Transactions on Information and Systems*, 2009 (accepted for publication). **JCR 2008 0.369, Fourth Quartile**.

Conferences and Workshops

1. Iván García-Magariño, Jorge J. Gómez-Sanz, y Juan Pavón. Representation of the Relationships in the Metamodels with the ECore language (in Spanish). In *Desarrollo del Software Dirigido por Modelos*, pages 11–20, 2007. September 11, 2007, Zaragoza España.

2. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. Reaching Consensus in a Multi-agent System. In *6th International Workshop on Practical Applications on Agents and Multi-agent Systems, IWPAAMS'07*, pages 349–358, 2007. November 12-13, 2007, Salamanca España. **CSCR 0.56 (pos. 52/701)**.

3. Iván García-Magariño, Alma Gónzalez-Rodríguez, y Juan C. González. Modeling the Development Process of INGENIAS with EMF (in Spanish). In *6th International Workshop on Practical Applications on Agents and Multi-agent Systems, IWPAAMS'07*, pages 369–378, 2007. November 12-13, 2007, Salamanca España. **CSCR 0.56 (pos. 52/701)**.

4. Iván García-Magariño y Jorge J. Gómez-Sanz. Framework for Defining Model Language Metamodels for CASE Tools. In *The Fifth International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MOMPES'08*, pages 14–23. IEEE Computer Society, 2008. Abril 5, 2008, Budapest Hungría.

5. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. A Complete-Computerised Delphi Process with a Multi-agent System. In *Sixth international Workshop on Programming Multi-Agent Systems, ProMAS'08*, volume 5442 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2009.

6. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. An Evaluation Framework for MAS Modeling Languages based on Metamodel Metrics. In *Agent-Oriented Software Engineering IX (Revised Selected Papers)*, volume 5386 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2009. **CORE B, CiteSeer 1.57 (top 7.45%)**.

7. Iván García-Magariño, Alma Gónzalez-Rodríguez, y Juan C. González. Definition of Process Models for Agent-based Development. In *Agent-Oriented Software Engineering IX (Revised Selected Papers)*, volume 5386 of *Lecture Notes in Computer Science*, pages 60–73. Springer, 2009. **CORE B, CiteSeer 1.57 (top 7.45%)**.

8. Jorge J. Gómez-Sanz, Rubén Fuentes-Fernández, Juan Pavón, y Iván García-Magariño. INGENIAS Development Kit: a visual multi-agent system development environment (Awarded as Best Academic Demo of AAMAS'08). In *The Seventh International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08*, pages 1675–1676. ACM, 2008. Mayo 12-16, 2008, Estoril Portugal. **CORE A+, CSCR 0.76 (pos. 21/701)**.

9. Iván García-Magariño, Jorge J. Gómez-Sanz, y José R. Pérez Agüera. A Multi-Agent Based Implementation of a Delphi Process. In *The Seventh*

International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08, pages 1543–1546, ACM, 2008. May 12-16, 2008, Estoril Portugal. **CORE A+, CSCR 0.76 (pos. 21/701)**.

10. Iván García-Magariño, Celia Gutiérrez, y Rubén Fuentes Fernández. Organizing multi-agent systems for crisis management. In *7th Ibero-American Workshop in Multi-Agent Systems (Iberagents 2008)*, pages 69–80, 2008. Octubre 14, 2008, Lisboa Portugal.

11. Iván García-Magariño, Alma Gómez-Rodríguez, Juan C. González, y Jorge J. Gómez-Sanz. INGENIAS-SCRUM Development Process for Multi-Agent Development. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50 of *Advances in Software Computing*, pages 108–117. Springer, 2009.

12. Iván García-Magariño, Alma Gómez-Rodríguez, y Juan C. González. Modeling Processes of AOSE Methodologies by Means of a New Editor. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50 of *Advances in Software Computing*, pages 672–681. Springer, 2009.

13. Iván García-Magariño. Towards the Coexistence of Different Multi-Agent System Modeling Languages with a Powertype-Based Metamodel. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50 of *Advances in Software Computing*, pages 189–193. Springer, 2009.

14. Iván García-Magariño, Rubén Fuentes-Fernández, y Jorge J. Gómez-Sanz. INGENIAS development process assisted with chains of transformations. In *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, volume 5517 of *Lecture Notes in Computer Science*, pages 514–521. Springer, 2009. **CORE B, CiteSeer 0.16 (top 79.85%), CSCR 0.55 (pos. 55/701)**

15. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, volume 55 of *Advances in Soft Computing*, pages 40–49. Springer, 2009. **CSCR 0.56 (pos. 52/701)**.

16. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages. In *International Conference on Model Transformation (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2009. **22% of acceptance ratio**.

17. Iván García-Magariño, Jorge J. Gómez-Sanz, y Rubén Fuentes-Fernández. Model Transformations for Improving Multi-agent Systems Development in INGENIAS. In *The 10th International Workshop on Agent-Oriented Software Engineering AOSE'09*, 2009. May 11, 2009, Budapest Hungría. **CORE B, CiteSeer 1.57 (top 7.45%)**.

18. Iván García-Magariño, Celia Gutiérrez, y Rubén Fuentes-Fernández. The INGENIAS Development Kit: a practical application for crisis-management. In *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, volume 5517 of *Lecture Notes in Computer Science*, pages 537–544. Springer, 2009. **CORE B, CiteSeer 0.16 (top 79.85%), CSCR 0.55 (pos. 55/701)**.

19. Iván García-Magariño, Sylvain Rougemaille, Rubén Fuentes Fernández, Frédéric Migeon, Juan Pavón Mestras, y Marie-Pierre Gleizes. Towards Agent-Oriented Model Driven Engineering. In *6th European Workshop on Multi-Agent Systems (EUMAS 2008)*, pages 69–70, 2008. December 18-19, 2008, Bath UK.

20. Iván García-Magariño, Sylvain Rougemaille, Rubén Fuentes-Fernández, Frédéric Migeon, Marie-Pierre Gleizes, y Jorge J. Gómez-Sanz. A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, volume 55 of *Advances in Soft Computing*, pages 70–79. Springer, 2009. **CSCR 0.56 (pos. 52/701)**.

Apéndice A

Comparación con UML Diagram Interchange

La idea de separar la información referente a los diagramas del resto de información en los metamodelos no es totalmente nueva. El *Object Management Group* (OMG) especificó un metamodelo para el intercambio de diagramas llamado *UML Diagram Interchange* (UML-DI) (OMG, Abril 2006), que mantiene la información de los diagramas separada del resto de información. Por tanto, el núcleo de UML puede cambiar sin tener efecto en el metamodelo UML-DI, y vice-versa. La causa es que las referencias de UML-DI sólo apuntan al objeto genérico de UML. Este objeto genérico está presente en todas las versiones de UML. Como aclaración, en la super-estructura (OMG, 2007a) y la infra-estructura (OMG, 2007b) de UML, no se proporciona información de cómo están distribuidos los elementos en los diagramas. Las diferencias más importantes entre el armazón de metamodelos presentado en esta tesis y la aproximación de UML-DI se mencionan en los siguientes sub-apartados.

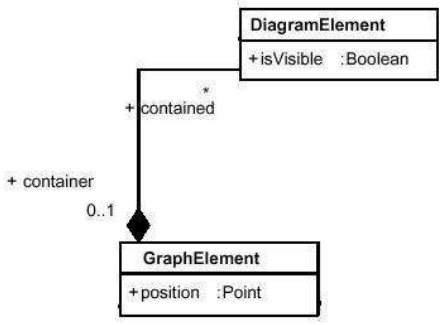


Figura A.1: Metamodelo UML-DI, raíz del diagrama.

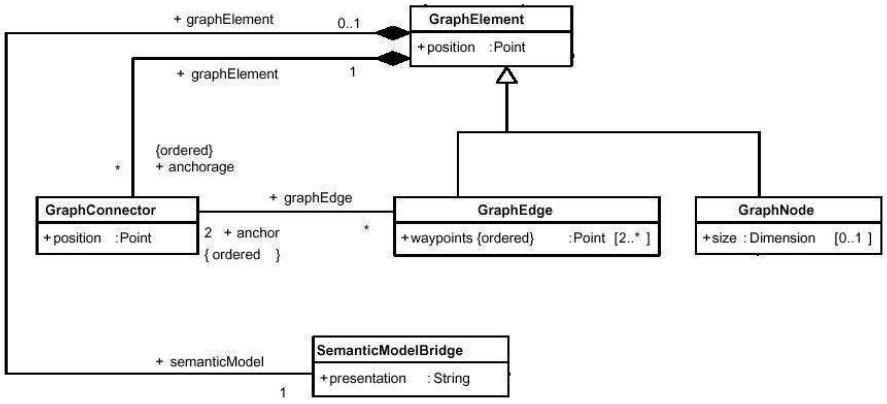


Figura A.2: Metamodelo UML-DI, nodos y aristas.

A.1 Estructura básica

En cuanto a la estructura básica, el metamodelo de UML-DI es bastante similar al metamodelo *Vistas* presentado en el apartado 3.2.1. Ambos metamodelos incluyen los conceptos de grafos, nodos y aristas.

UML-DI tiene un elemento raíz llamado *DiagramElement* (véase la Figura A.1). El elemento *DiagramElement* representa un contenedor de nodos y aristas. En concreto, contiene varios elementos *GraphElements*.

El elemento *GraphElement* (véase Figura A.2) puede ser tanto un *GraphNode* como un *GraphEdge*, que respectivamente representan un nodo y una arista. Las aristas apuntan directamente a los elementos *Graph Connectors*, que se enlazan a su vez con otros elementos *GraphElement*. Esto es bastante similar al metamodelo presentado en esta tesis. La EClass *GraphElement* del metamodelo presentado en esta tesis es equivalente al elemento *GraphElement* de UML-DI. Sin embargo, el armazón presentado (véase el apartado 3.2.1) mantiene de forma separada la información espacial (i.e. posiciones y dimensiones) en otro metamodelo, llamado *Layouts*.

Cada elemento *GraphElement* de UML-DI apunta al elemento *SemanticModelBridge* (véase Figura A.2). Por ejemplo, este *SemanticModelBridge* puede ser un elemento *Uml1SemanticModelBridge*. A su vez, este elemento apunta a un elemento UML del metamodelo núcleo. En nuestra aproximación, esta asociación entre los elementos del grafo y los elementos del metamodelo *Objetos* se consigue con la EReference *obj* (véase el apartado 3.2.1). En la segunda versión del metamodelo *Vistas*, el nombre de la EReference puede variar.

En conclusión, la especificación UML-DI de OMG es similar al metamodelo *Vistas*, pero el armazón presentado desacopla el contenido de los diagramas de su representación visual en el espacio, con el metamodelo *Layouts*.

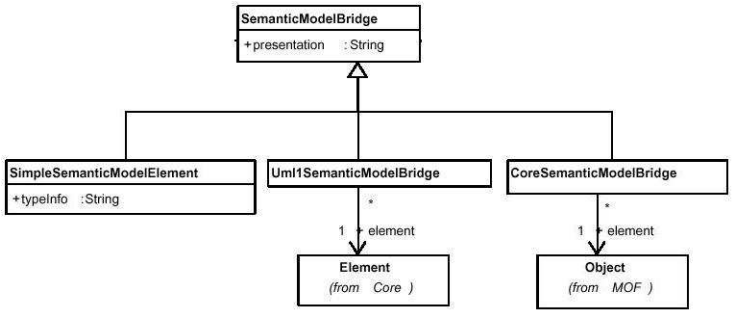


Figura A.3: Metamodelo UML-DI, semántica de cada nodo. Cada nodo apunta a un elemento del modelo núcleo.

A.2 Mostrando un recorte de un diagrama en otro diagrama

En la especificación de UML-DI, un diagrama puede ser usado para mostrar un extracto de otro diagrama existente sin la necesitad de crear los elementos del diagrama más de una vez. Sin embargo, la aproximación de UML-DI tiene las siguientes desventajas:

- *El procesamiento de los recortes es complicado.* Cada diseñador de herramienta CASE de modelado debe decidir qué hacer con los recortes. Si los recortes son procesados, un mismo elemento puede ser procesado dos veces. Si no son procesados los recortes, puede que algunas relaciones entre elementos no se procesen. El procesamiento es posible pero complicado. Probablemente, cada herramienta CASE de modelado use una representación interna diferente a la representación del modelo para resolver el problema mencionado. Sin embargo, los expertos en LMEDs (van Deursen et al., 2000) desaconsejan esta práctica.

- *Puede haber situaciones confusas.* Un diagrama puede tener un recorte de otro diagrama; y este segundo diagrama puede tener recortes de un tercer diagrama. En este caso, el diseñador de la herramienta CASE de modelado debe decidir si, en el primer diagrama, el recorte referencia al segundo o al tercer diagrama. Esta decisión es relevante porque, si el recorte del segundo diagrama se elimina, lo que ocurre en el primer diagrama no está determinado. Implementar todas estas situaciones confusas en una herramienta CASE de modelado es bastante costoso comparado con la aproximación de este trabajo.

Una de las ventajas de este trabajo de tesis frente a UML-DI es usar el concepto de *vistas* en el meta-modelado de herramientas CASE de modelado. En un modelo, hay un único contenedor (i.e. diccionario) de todos los objetos (i.e. entidades y relaciones) y hay varias vistas (i.e. diagramas) del contenedor único. Esta aproximación no tiene las desventajas de UML-DI mencionadas. Como aclaración, en la aproximación de esta tesis, un diagrama puede tener una parte igual a otra parte de otro diagrama, referenciando a los mismos elementos del diccionario y distribuyéndolos en el espacio de forma similar; de esta manera, se

consigue representar lo mismo que con los *recortes* de UML-DI. Sin embargo, los dos diagramas no están acoplados dado que no son *recortes*, sino *vistas* diferentes de los mismos elementos. Los cambios de un diagrama, en el grupo de elementos seleccionados o en la distribución espacial, no interfieren en el otro diagrama.

A.3 Visibilidad de los elementos del diagrama

En la especificación de UML-DI, el atributo *visible* indica si un elemento de un diagrama se muestra o no. Su valor se aplica recursivamente a todos los elementos contenidos. Si solo un elemento del diagrama en la jerarquía tiene su atributo *visible* a falso, todos los elementos anidados son también ocultados, independientemente de su propia visibilidad. Mientras un elemento está oculto, no se muestra pero existe.

En nuestra aproximación, no se necesita un atributo para la visibilidad. Puede haber objetos en el diccionario que no son mostrados en las vistas (i.e. diagramas). Por tanto, se puede tener elementos que existan pero que estén ocultos.

De hecho, algunas herramientas CASE de modelado para UML representan diccionarios de objetos, como por ejemplo *BoUML* o *Rational Rose*. Sin embargo, esto no se refleja en los metamodelos de UML y UML-DI. Por tanto, la aproximación de este trabajo de tesis es cercana a la representación de las herramienta CASE de modelado actuales para UML. Esta semejanza es positiva, especialmente para definir metamodelos de herramienta CASE de modelado, dado que acerca el metamodelo a la representación de la herramienta, práctica aconsejada por los expertos en creación de herramientas de modelado (Amyot et al., 2006).

Apéndice B

Transformación generada

B.1 Comienzo de la transformación

```
module a2b; – Module Template
create OUT : MMB from IN : MMA;
– For the two new built entities and relations
helper def: newEntities: Sequence(MMB!GeneralEntity)=Sequence{};
helper def: newRelations: Sequence(MMB!GeneralRelation)=Sequence{};
————————————- Rules Generated From Example Models ————————————-
```

B.2 Reglas generadas a partir de los modelos prototipo

```
—-Rule1———-—-
rule Rule1 {
from
    cin:MMA!UInitiates( true and
                cin.iniSource.multiplicity=1 and
                cin.iniSource.isInteractionUnit.id='IUA' and
                cin.iniTarget->select(t|
                    t.multiplicity=2 and
                    t.itRole.id='R1').notEmpty() and
            MMA!Role.allInstances()->select(e|
                e.id='R1').notEmpty() and
            MMA!InteractionUnit.allInstances()->select(e|
                e.id='IUA').notEmpty() )
to
    oute1:MMB!InteractionUnit(
            id<-cin.iniSource.isInteractionUnit.id),
    oute2:MMB!InteractionUnit(
            id<-'IUC'),
    oute3:MMB!Role(
            id<-'R2'),
```

```
    outa1:MMB!InteractionSource(
            multiplicity<-1,
            isInteractionUnit<-oute1),
    outa2:MMB!InteractionTarget(
            multiplicity<-1,
            itRole<-oute3),
    outr1:MMB!UColaborates(
            label<-'',
            colSource<-outa1,
            colTarget<-outa2),
    outa3:MMB!PrecedesEnd(
            multiplicity<-1,
            peInteractionUnit<-oute2),
    outa4:MMB!PrecedesEnd(
            multiplicity<-1,
            peInteractionUnit<-oute1),
    outr2:MMB!UPrecedes(
            label<-'',
            preSource<-outa3,
            preTarget<-outa4)
do{
            thisModule.newEntities<-
                thisModule.newEntities.append(oute1);
            thisModule.newEntities<-
                thisModule.newEntities.append(oute2);
            thisModule.newEntities<-
                thisModule.newEntities.append(oute3);
            thisModule.newRelations<-
                thisModule.newRelations.append(outr1);
            thisModule.newRelations<-
                thisModule.newRelations.append(outr2);
    }
}
– Rule2
rule Rule2 {
from
    cin:MMA!Goal( true )
to
    eout1:MMB!Goal(
            id<-cin.id)
do{
            thisModule.newEntities<-thisModule.newEntities.append(eout1);
    }
}
```

B.3 Final de la transformación

———————- Rule for the Allocation ———-

```
rule Specification{
    from
            cin:MMA!Specification
    to
            cout:MMB!Specification
    do{
            – We add the new entities and the new relations
            for(e in thisModule.newEntities){
                cout.entities<-e;
            }
            for(r in thisModule.newRelations){
                cout.relations<-r;
            }
    } }
```

Apéndice C

Ejemplos prácticos de la generación de transformaciones basadas en ejemplos en sistemas multi-agentes

El objetivo de este apéndice es completar la experimentación de GTBE presentada en el capítulo 6, mostrando más ejemplos de aplicación de GTBE en SMAs. En concreto, el siguiente sub-apéndice se centra en la reutilización de las TMs generadas, aplicando algunas de las TMs descritas en el capítulo 6 a un SMA diferente. Después, el sub-apéndice C.2 propone otros ejemplos de GTBE en el contexto de la metodología INGENIAS, y el sub-apéndice C.3 muestra un ejemplo de GTBE en la metodología ADELFE.

C.1 Reutilización de las transformaciones generadas

Este sub-apéndice muestra varios ejemplos de reutilización de las TMs descritas en el apartado 6.4. Para ello, se usa un SMA (García-Magariño et al., 2009a) existente para gestionar una situación de crisis en una ciudad, que fue desarrollado originalmente con la metodología INGENIAS sin usar la aproximación de esta tesis.

El siguiente sub-apartado presenta el SMA de gestión de crisis, mientras que los sub-apartados posteriores presentan la aplicación de las TMs generadas previamente (véase el apartado 6.4) más relevantes en el SMA de gestión de crisis.

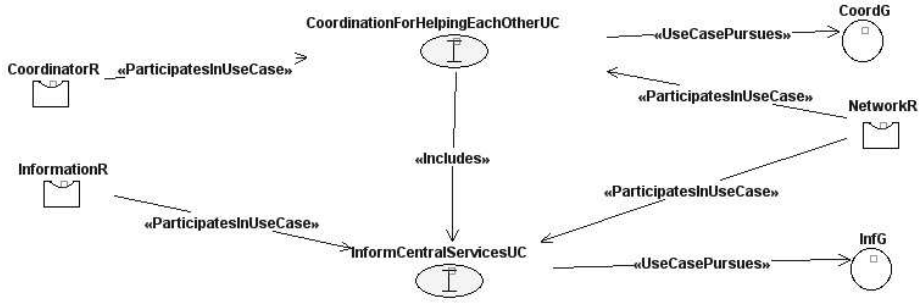


Figura C.1: Casos de uso para el SMA de Crisis

C.1.1 Sistema multi-agente para la gestión de crisis

En el caso de estudio de *Gestión de Crisis* (AgentLink, December 2005) con un SMA, un material altamente tóxico ha sido liberado accidentalmente en una ciudad. El número de personas afectadas es demasiado alto para ser atendidas con una solución centralizada. Los servicios médicos oficiales no son suficientes para curar todas las personas afectadas. Para permitir y coordinar la colaboración entre ciudadanos de manera eficiente, se propone una solución distribuida.

La solución de este caso de estudio se basa en los siguientes hechos: los ciudadanos con capacidades médicas pueden ayudar a los ciudadanos afectados que están por la misma zona; los ciudadanos deben ser rápidamente avisados de los lugares afectados por la sustancia tóxica para evitarlos; la central oficial debe ser informada de todos los lugares afectados por la sustancia; y las comunicaciones deben ser eficientes. Por tanto, los tres objetivos del SMA son: coordinar a los ciudadanos, mantener la eficiencia de la red de comunicación, e informar a los servicios centrales.

El SMA usado en este sub-apéndice fue inicialmente creado con la metodología INGENIAS sin GTBE tal y como se describe en (García-Magariño et al., 2009a), y está incluido en la distribución del IDK 2.8. Este SMA satisface los requisitos anteriormente mencionados con los siguientes tipos de agentes: *agentes coordinadores, agentes de red* y *agentes informadores*. Los *agentes coordinadores* son responsables de coordinar a los ciudadanos. En el dispositivo móvil de cada ciudadano, se ejecuta un agente coordinador que interacciona con su usuario. Los *agentes de red* están a cargo de hacer eficaz y eficiente la comunicación entre los otros tipos de agentes. El *agente informador* gestiona la información sobre los lugares afectados por la sustancia tóxica y los muestra a los servicios centrales oficiales.

C.1.2 Transformación para generar los roles a partir de los casos de uso

En el ejemplo de este sub-apartado, la TM especificada anteriormente en la Figura 6.3 se reutiliza para el SMA sobre gestión de crisis de una ciudad. De manera similar a como se usó para el SMA de Delphi en el apartado 6.4.1, los casos de uso del SMA de *Crisis* (véase la Figura C.1) se transforman en la definición de sus roles correspondientes (véase la Figura C.2). Es necesario mencionar que los modelos prototipo de la Figura 6.3 tienen vacíos los atributos

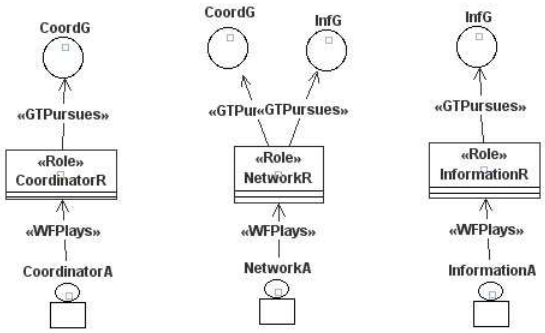


Figura C.2: Los roles y agentes del SMA de Crisis

referente a la descripción (i.e. no aparecen en la Figura 6.3), interpretándose como comodines. Esto permite que la TM se pueda aplicar a elementos que tengan cualquier valor en el atributo descripción.

Si bien se suele asumir que las TMs definidas manualmente pueden aplicarse a diferentes modelos, en la GTBE se debe comprobar que las TMs son lo suficientemente generales para aplicarse en más casos que los modelos prototipo. El ejemplo de este apartado muestra que la aproximación de GTBE presentada en esta tesis genera TMs aplicables a diferentes modelos. Esto se consigue gracias al mecanismo de asociación de atributos y al mecanismo de los comodines (respectivamente presentados en los apartados 5.2.6 y 5.2.4), que permiten que las TMs generadas puedan encajar con diferentes modelos y transformarlos.

C.1.3 Transformación para obtener las tareas relacionadas con un flujo de trabajo

Al *flujo de trabajo* del SMA de gestión de crisis (en la Figura C.3), también se le pueden aplicar las dos TMs para obtener las tareas relacionados con el flujo de trabajo, que fueron especificadas para la tarea inicial (en la Figura 6.6) y para el resto de tareas (en la Figura 6.7).

Se observa de nuevo que las reglas en la TM de la Figura 6.7 se pueden aplicar varias veces a la misma tarea si precede o es precedida varias veces. En el caso del SMA de Crisis, la tarea *CNNHelpT* precede dos tareas y la TM es aplicada dos veces sobre esta tarea.

Como se puede observar, los flujos de trabajos de los dos casos de estudio tienen diferentes estructuras. El flujo de trabajo de Delphi (especificado anteriormente en la Figura 6.5) puede producir varias iteraciones de tareas ejecutadas (i.e. varias rondas de cuestionarios), mientras que el flujo de trabajo de Crisis (en la Figura C.3) tiene una bifurcación. A pesar de sus diferencias estructurales, las mismas TMs pueden ser aplicadas a las especificaciones de ambos SMAs. El motivo probablemente sea que se ha escogido para la GTBE un modelo prototipo de entrada simple que se suele encontrar repetidas veces en los flujos de trabajo; en concreto, los modelos prototipo de entrada se han centrando en la relación de precedencia entre tareas.

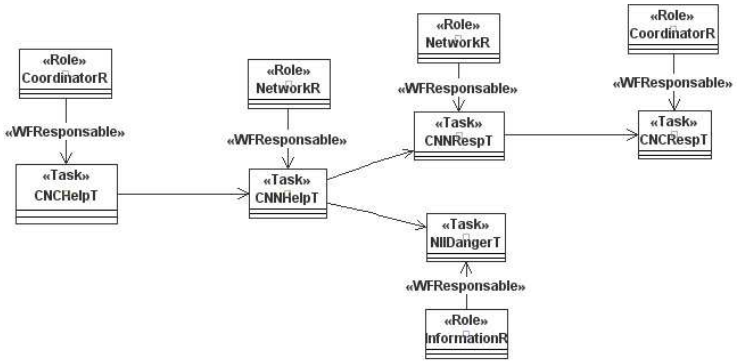


Figura C.3: Flujo de trabajo para el SMA de Crisis

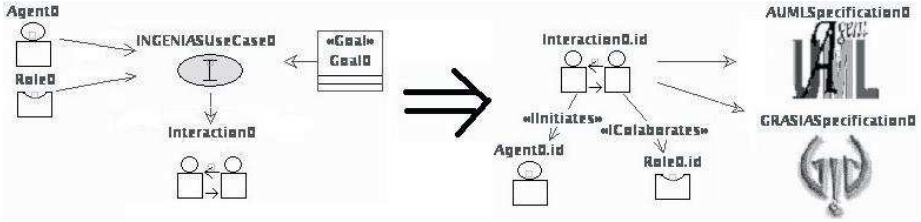


Figura C.4: Modelos prototipo de la TM *UseCase2Interaction* que generan una regla muchos-a-muchos.

C.2 Otras transformaciones generadas para INGENIAS

Este sub-apéndice propone la generación de varias TMs que son aplicables a algunos SMAs desarrollados con INGENIAS y que son diferentes de las TMs presentadas en el capítulo 6.

C.2.1 Transformaciones para realizar el comportamiento de los casos de uso

La realización de los casos de uso en INGENIAS puede seguir varias alternativas. Una de éstas es la expansión de la definición de las interacciones que la realizan. Para cada interacción, se tiene que definir un protocolo. En este caso, la especificación del protocolo usa una notación particular de INGENIAS, que permite combinar las primitivas de paso de mensajes con las de ejecución de tareas. Usando TMs y empezando a partir del caso de uso inicial, es posible definir prototipos de los conceptos resultantes y su relación con los conceptos origen.

La primera TM crea la definición de una interacción a partir de un caso de uso, y los modelos prototipo de esta TM se muestran en la Figura C.4. En concreto, la TM define una interacción que comunica un agente y un rol ambos relacionados previamente con un caso de uso particular.

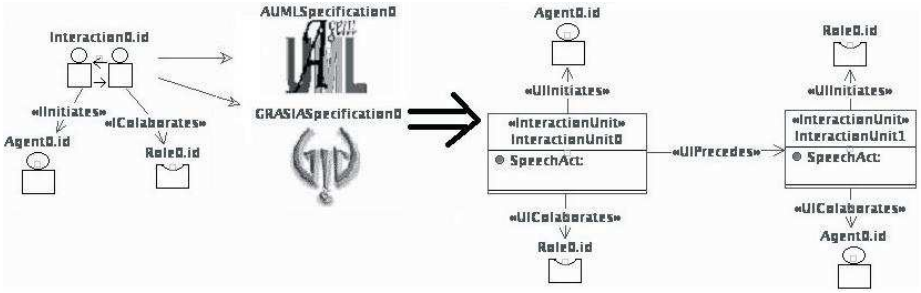


Figura C.5: Modelos prototipo para la TM *InteractionDefinition2Interaction-Protocol* que generan una regla muchos-a-muchos.

Después, otra TM crea una protocolo básico a partir de la definición de interacción, y esta TM se genera a partir de la pareja de modelos que aparecen en la Figura C.5. El protocolo creado incluye dos unidades de interacción, en las que hay una petición y una respuesta. Este protocolo es sólo un punto de partida, y los diseñadores pueden completarlo en caso de que sea necesario.

C.2.2 Una transformación para la definición de habilidades

La definición de las habilidades de un agente es una actividad básica en la mayoría de las metodologías de ISOA. Una habilidad se puede entender como la capacidad para cumplir con un objetivo. En INGENIAS, esto implica: definir una tarea; declarar que el agente juega un rol responsable de la ejecución de la tarea; y asociar el objetivo con la tarea. Esta tarea puede usar recursos o aplicaciones adicionales, donde las aplicaciones de INGENIAS son elementos de acceso a software externo para los agentes. Como resultado de la ejecución de una tarea, nueva información se añade al estado mental del agente.

Usando la GTBE, este conocimiento puede ser sintetizado en una pareja de ejemplos e incorporado dentro de una TM, llamada *AddSkill*. Los ejemplos se presentan en la Figura C.6 y corresponden a un agente que persigue un objetivo (i.e. *SkillGoal*) que se puede cumplir con una tarea (i.e. *SkillTask*) producida en el modelo ejemplo destino. La TM generada se aplica por defecto a todos los agentes. Los diseñadores pueden aplicar esta TM a un único agente si es necesario, añadiendo un restricción simple adicional con el identificador del agente, i.e. $cin.id = AgentID$, donde *cin* es el elemento principal de la entrada de la regla y *AgentID* es el identificador del agente.

La TM *AddSkill* tiene un grafo no conexo en la parte de salida de la regla de transformación, si sólo se considera el modelo de las *Vistas*, en el que cada elemento visualizado es un elemento diferente aunque se visualice con el mismo nombre. El algoritmo presentado en esta tesis da soporte a este tipo de reglas, considerando todas las raíces lógicas de los modelos prototipo, como se indicó en los apartados 5.2.4 y 5.2.5. El resto de aproximaciones de GTBE (Wimmer et al., 2007; Varró y Balogh, 2007b) no dan soporte a este tipo de reglas.

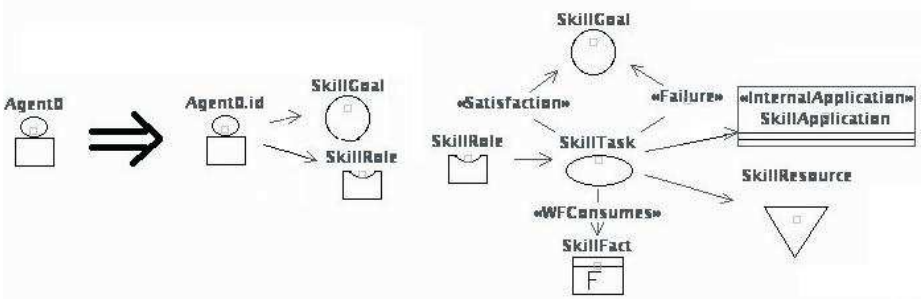


Figura C.6: Modelos ejemplo para la TM *AddSkill* que generan una regla uno-a-muchos con un grafo no-conexo en la salida.

C.3 Ejemplo de generación de transformaciones en ADELFE

ADELFE[1] (Bernon et al., 2005) es una metodología de ISOA para el diseño de *SMAs Adaptativos* (SMAAs) (Capera et al., 2003). Principalmente a *ADELFE* le concierne la especificación de sistemas que tratan la dinámica de entornos complejos y la auto-adaptación. El proceso de *ADELFE* se basa en el estándar del *Proceso Unificado de Desarrollo* (Jacobson et al., 1999) para metodologías orientadas a objetos. También integra una nueva fase de implementación siguiendo los principios del DSDM y basada en LMs específicos y TMs (Rougemaille et al., 2008). Por tanto, la última versión de *ADELFE* cubre el ciclo de vida completo del desarrollo del software, desde los primeros requisitos hasta la implementación.

ADELFE usa varios LMs en su desarrollo. ADELFE usa UML-2.0 (OMG, 2007a,b) para los requisitos y el análisis. Por ejemplo, se especifican las interacciones con los diagramas de secuencia de UML (Bauer y Odell, 2005). El LM para SMAAs (i.e. *AMAS-ML*, proveniente de las siglas de *Autonomous MAS Modeling Language*) se usa en varios pasos de la fase de diseño, por ejemplo en el diseño detallado de la estructura del agente y la definición de su comportamiento cooperativo. Los modelos resultantes se usan como entrada de la fase de implementación. En esta fase, *ADELFE* usa un lenguaje para la micro-arquitectura, llamado *micro-Architecture Description Language* (μADL), que es un LM centrado específicamente en la arquitectura de SMAAs. Por medio de varias TMs, esta fase permite generar el código del comportamiento del agente y de ciertos interfaces específicos de programación (Rougemaille et al., 2007). En conclusión, *ADELFE* proporciona un DSDM que genera el código de la aplicación a partir de un modelo abstracto definido con el lenguaje AMAS-ML.

Una de las tareas en la fase de diseño de la metodología *ADELFE* es la especificación de las interacciones de agentes, i.e. la comunicación directa entre agentes. Como se ha mencionado anteriormente, *ADELFE* hace la especificación inicial de las interacciones con diagramas de secuencia UML. Una TM ha sido especificada para traducir secuencias UML de mensajes en *protocolos cooperativos de interacción* en AMAS-ML. Una vez los protocolos se integran en un

[1] ADELFE es un acrónimo francés para "Atelier de Développement de Logiciels à Fonctionnalité Emergente". Éste era un proyecto francés financiado por RNTL (2000-2003)

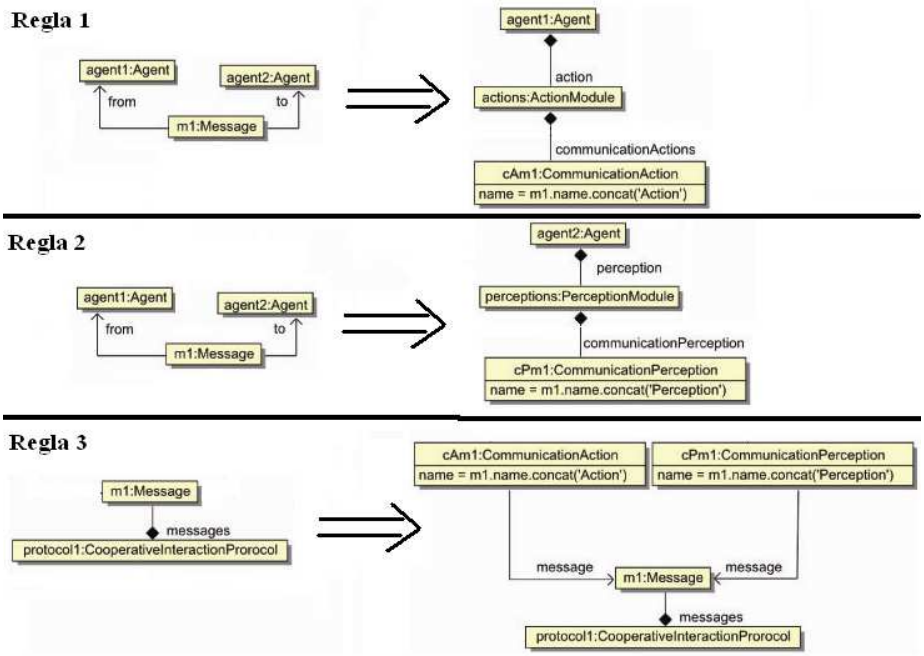


Figura C.7: Parejas de modelo prototipo en ADELFE para la TM *RefactoringADELFE* que generan tres reglas muchos-a-muchos.

modelo AMAS-ML, el envío y recepción de mensajes se debe declarar en los agentes involucrados en estos protocolos. Este ejemplo propone ayudar al diseñador en automatizar este proceso. La TM requerida es generada usando el algoritmo de GTBE y la herramienta *MTGenerator* presentados en esta tesis.

La Figura C.7 muestra la especificación de las parejas de ejemplo con una notación de meta-objetos, es decir, como instancias de las meta-clases de los lenguajes UML y AMAS-ML respectivamente. Los modelos prototipo origen representan un mensaje (i.e., *m1*) que pertenece a un protocolo (i.e., *protocol1*) y que el agente *agent1* envía al agente *agent2*. El propósito de los modelos prototipo destino es añadir una acción de comunicación (i.e., *cAm1*) al módulo de acción (i.e., *actions*) del agente *agent1* y la respectiva percepción de comunicación (i.e., *cPm1*) al módulo de percepción (i.e., *perceptions*) del agente *agent2*. Tanto la acción de comunicación como la percepción creadas se relacionan con el mensaje *m1* por medio de las relaciones *cAm1.message* y *cPm1.message* respectivamente, en conforme con el metamodelo de AMAS-ML. Como en el resto de los ejemplos, cada flecha numerada representa una pareja de modelos prototipo que se convertirá en una regla de transformación. La regla número 1 añade el modulo de acción (i.e., *actions*) al agente emisor (i.e., *agent1*). La regla número 2 añade el módulo de percepción (i.e., *perceptions*) al agente receptor (i.e. *agent2*). La regla número 3 integra la acción de comunicación (i.e., *cAm1*) y la percepción (i.e., *cPm1*) a los módulos respectivos y los enlaza al mensaje (i.e., *m1*).

Aunque los diseñadores de SMAAs pueden hacer este proceso, la TM presentada les libera de este trabajo repetitivo, mejorando su productividad. Con-

siderando el conjunto de protocolos y mensajes que usualmente se define en los modelos de AMAS-ML, la automatización de este proceso ahorra tiempo valioso en la fase de diseño. Además, esta TM evita los errores que se pueden cometer cuando se hace el proceso de forma manual.

Bibliografía

Abd-Ali, J. y El-Guemhioui, K. Horizontal Transformation of PSMs. *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Nuremberg, Germany*, páginas 299–315, 2005.

AgentLink. Multi-agent Systems in Crisis Management: The Combined Systems. *Case Study*, December 2005.

Amor, M., Fuentes, L. y Vallecillo, A. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. En *Agent-Oriented Software Engineering*, vol. 5, páginas 93–108. Springer, 2005.

Amyot, D., Farah, H. y Roy, J. Evaluation of Development Tools for Domain-Specific Modeling Languages. En *System Analysis and Modeling: Language Profiles*, vol. 4320 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 183–197. Springer, 2006.

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H., Kuske, S., Plump, D., Schuerr, A. y Taentzer, G. Graph transformation for specification and programming. *Science of Computer Programming*, vol. 34(1), páginas 1–54, 1999.

Atkinson, C. y Kuhne, T. Model-driven development: a metamodeling foundation. *Software, IEEE*, vol. 20(5), páginas 36–41, 2003.

Aufare-Portier, M. A. A High level interface language for GIS. *Journal of Visual Languages and Computing*, vol. 6(2), páginas 167–182, 1995.

Balsamo, S., Di Marco, A., Inverardi, P. y Simeoni, M. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, vol. 30(5), páginas 295–310, 2004.

Bardohl, R., Ehrig, H., de Lara, J. y Taentzer, G. Integrating Meta Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. En *Fundamental Approaches to Software Engineering*, vol. 2984 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 214–228. Springer, 2004.

Bauer, B. y Odell, J. UML 2.0 and agents: how to build agent-based systems with the new UML standard. *Engineering Applications of Artificial Intelligence*, vol. 18(2), páginas 141–157, 2005.

Bellifemine, F., Poggi, A. y Rimassa, G. Developing multi-agent systems with a FIPA-compliant agent framework. *Software-Practice and Experience*, vol. 31(2), páginas 103–28, 2001a.

Bellifemine, F., Poggi, A. y Rimassa, G. JADE: a FIPA2000 compliant agent development environment. *Proceedings of the fifth International Conference on Autonomous Agents*, páginas 216–217, 2001b. Montreal, Quebec, Canada.

Bernon, C., Camps, V., Gleizes, M.-P. y Picard, G. Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. En *Agent-Oriented Methodologies* (editado por B. Henderson-Sellers y P. Giorgini), vol. ISBN1-59140-581-5, páginas 172–202. Idea Group Pub, NY, USA, 2005.

Bernon, C., Gleizes, M., Peyruqueou, S. y Picard, G. ADELFE: A Methodology for Adaptive Multi-agent Systems Engineering. *Engineering Societies in the Agents World III: Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002: Revised Papers*, 2003.

Beydoun, G., Gonzalez-Perez, C., Henderson-Sellers, B. y Low, G. Developing and Evaluating a Generic Metamodel for MAS Work Products. En *Software Engineering for Multi-Agent Systems IV*, vol. 3914 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 126. Springer, 2006.

Bézivin, J., Breton, E., Dupé, G. y Valduriez, P. The ATL Transformation-based Model Management Framework. *TR03-08, University of Nantes, September*, 2003.

Bézivin, J., Dupe, G., Jouault, F., Pitette, G. y Rougui, J. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003a. Anaheim, California, USA.

Bézivin, J., Gérard, S., Muller, P. y Rioux, L. MDA Components: Challenges and Opportunities. *International Workshop on Metamodelling for MDA, Kings Manor, York, England, November*, páginas 24–25, 2003b.

Bézivin, J., Hammoudi, S., Lopes, D. y Jouault, F. Applying MDA Approach to B2B Applications: A Road Map. En *Workshop on Model Driven Development (WMDD 2004) at ECOOP 2004*, vol. 3344 de *LECTURE NOTES IN COMPUTER SCIENCE*. 2004.

Bodeveix, J., Chemouil, D., Filali, M. y Strecker, M. Towards formalising AADL in Proof Assistants. *Electronic Notes in Theoretical Computer Science*, vol. 141(3), páginas 153–169, 2005.

Boronat, A., Carsí, J., Ramos, I. y Letelier, P. Formal Model Merging Applied to Class Diagram Integration. *Electronic Notes in Theoretical Computer Science*, vol. 166, páginas 5–26, 2007.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. y Mylopoulos, J. Modeling Early Requirements in Tropos: A Transformation Based Approach. En *Agent-Oriented Software Engineering II*, vol. 2222 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 151–168. Springer, 2002.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. y Mylopoulos, J. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, vol. 8(3), páginas 203–236, 2004.

Budinsky, F. *Eclipse Modelling Framework: Developer's Guide*. Addison Wesley, 2003.

Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R. et al. Agent Oriented Analysis Using Message/UML. En *Agent-Oriented Software Engineering II*, vol. 2222 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 119–135. Springer, 2002.

Capera, D., Georgé, J.-P., Gleizes, M.-P. y Glize, P. The AMAS Theory for Complex Problem Solving Based on Self-organizing Cooperative Agents. En *Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003 (TAPOCS 2003) at WETICE 2003*. IEEE CS, 2003. Linz, Austria.

Chang, S., Polese, G., Thomas, R. y Das, S. A Visual Language for Authorization Modeling. *Proceedings of the IEEE Symposium on Visual Languages*, páginas 110–118, 1997. Isle of Capri, Italy.

Chella, A., Cossentino, M., Sabatucci, L. y Seidita, V. Agile PASSI: An Agile Process for Designing Agents. *International Journal of Computer Systems Science & Engineering. Special issue on"Software Engineering for Multi-Agent Systems". May*, vol. 21(2), páginas 133–144, 2006.

Chen, K., Sztipanovits, J. y Neema, S. Toward a semantic anchoring infrastructure for domain-specific modeling languages. *Proceedings of the 5th ACM international conference on Embedded software*, páginas 35–43, 2005.

Clark, J., DeRose, S. et al. XML path language (XPath) version 1.0. *W3C recommendation*, vol. 16, página 1999, 1999a.

Clark, J. et al. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, vol. 16(11), 1999b.

Clark, J. et al. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, vol. 16(11), 1999c.

Clavel, M., Duran, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Talcott, C. The Maude 2.0 system. En *Rewriting Techniques and Applications*, vol. 2706 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 76–87. Springer, 2003.

Collier, N. Repast: An extensible framework for agent simulation. *Internet site of RePast: http://repast. sourceforge. net/projects. html*, 1999.

Cossentino, M., Gaglio, S., Sabatucci, L. y Seidita, V. The PASSI and Agile PASSI MAS Meta-models Compared with a Unifying Proposal. En *Multi-agent Systems And Applications IV*, vol. 3690 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 183–192. Springer, 2005.

COSTAGLIOLA, G., DELUCIA, A., OREFICE, S. y POLESE, G. A Classification Framework to Support the Design of Visual Languages. *Journal of Visual Languages and Computing*, vol. 13(6), páginas 573–600, 2002.

CUADRADO, J. y MOLINA, J. A Plugin-Based Language to Experiment with Model Transformation. En *9th International Conference Model-Driven Engineering. Languages and Systems (MoDELS06)*, vol. 4199 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 336–350. Springer, 2006.

CUADRADO, J., MOLINA, J. y TORTOSA, M. RubyTL: A practical, extensible transformation language. En *European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA'06*, vol. 14 de *LECTURE NOTES IN COMPUTER SCIENCE*. Springer, 2006.

CZARNECKI, K. y HELSEN, S. Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003. Anaheim, California, USA.

CZARNECKI, K. y HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, vol. 45(3), páginas 621–645, 2006.

DARIMONT, R. Requirements Engineering with GRAIL/KAOS. *Industrial presentation, IEEE Joint International Conference on Requirements Engineering Industrial Presentations*, páginas 135–146, 2002.

DATE, C. An introduction to the unified database language (UDL). En *Proceedings of the sixth international conference on Very Large Data Bases-Volume 6*, páginas 15–32. VLDB Endowment, 1980. Montreal, Quebec, Canada.

DELOACH, S. Multiagent systems engineering of organization-based multiagent systems. *ACM SIGSOFT Software Engineering Notes*, vol. 30(4), páginas 1–7, 2005.

DELOACH, S. Engineering Organization-Based Multiagent Systems. En *Software Engineering for Multi-Agent Systems IV*, vol. 3914 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 109. Springer, 2006.

DELOACH, S., WOOD, M. y SPARKMAN, C. Multiagent Systems Engineering. *International Journal of Software Engineering and Knowledge Engineering*, vol. 11(3), páginas 231–258, 2001.

VAN DEURSEN, A., KLINT, P. y VISSER, J. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, vol. 35(6), páginas 26–36, 2000.

DOMOKOS, P. y VARRÓ, D. An Open Visualization Framework for Metamodel-Based Modeling Languages. *Electronic Notes in Theoretical Computer Science*, vol. 72(2), páginas 69–78, 2002.

DUDDY, K., GERBER, A., LAWLEY, M., RAYMOND, K. y STEEL, J. Model transformation: a declarative, reusable patterns approach. *Proceedings of Seventh IEEE International Enterprise Distributed Object*, páginas 174–185, 2003.

ECLIPSE, C. Eclipse consortium, eclipse graphical modeling framework (gmf). 2005.

EPF. Eclipse Process Framework (EPF). `http://www.eclipse.org/epf/`, 2009.

FAUCHER, C. y LAFAYE, J. Model-driven Engineering for Implementing the ISO 19100 Series of International Standards. 2007.

FERBER, J. *Multi-agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

FIPA, A. Message Structure Specification. *Foundation for Intelligent Physical Agents*, 2000.

FONDEMENT, F. y BAAR, T. Making Metamodels Aware of Concrete Syntax. En *Model Driven Architecture - Foundations and Applications*, vol. 3748 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 190. Springer, 2005.

FRANKLIN, S. y GRAESSER, A. Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents. En *Intelligent Agents III Agent Theories, Architectures, and Languages*, vol. 1193 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 21–36. Springer, 1997.

GARCÍA-MAGARIÑO, I. y FUENTES-FERNÁNDEZ, R. A Technique for Defining Metamodel Translations. *IEICE Transactions on Information and Systems, En Prensa*, 2009.

GARCÍA-MAGARIÑO, I., FUENTES-FERNÁNDEZ, R. y GÓMEZ-SANZ, J. J. Guideline for the definition of EMF metamodels using an Entity-Relationship approach. *Information and Software Technology*, vol. 51(8), páginas 1217–1230, 2009.

GARCÍA-MAGARIÑO, I., FUENTES-FERNÁNDEZ, R. y GÓMEZ-SANZ, J. J. INGENIAS development process assisted with chains of transformations. En *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, vol. 5517 de *Lecture Notes in Computer Science*, páginas 514–521. Springer, 2009.

GARCÍA-MAGARIÑO, I., GÓMEZ-RODRÍGUEZ, A. y GONZÁLEZ, J. C. Modeling Processes of AOSE Methodologies by Means of a New Editor. En *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, vol. 50 de *Advances in Software Computing*, páginas 672–681. Springer, 2009a.

GARCÍA-MAGARIÑO, I., GÓMEZ-RODRÍGUEZ, A., GONZÁLEZ, J. C. y GÓMEZ-SANZ, J. J. INGENIAS-SCRUM Development Process for Multi-Agent Development. En *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, vol. 50 de *Advances in Software Computing*, páginas 108–117. Springer, 2009b.

GARCÍA-MAGARIÑO, I. y GÓMEZ-SANZ, J. J. Framework for Defining Model Language Metamodels for CASE Tools. En *The Fith International Workshop*

on Model-based Methodologies for Pervasive and Embedded Software, MOMPES'08, páginas 14–23. IEEE Computer Society, 2008. April 5, 2008, Budapest Hungary.

García-Magariño, I., Gómez-Sanz, J. J. y Agüera, J. R. P. A Multi-Agent Based Implementation of a Delphi Process. En *The Seventh International Conference on Autonomous Agents and Multiagent Systems, AAMAS'08*, páginas 1543–1546. 2008. May 12-16, 2008, Estoril Portugal.

García-Magariño, I., Gómez-Sanz, J. J. y Fuentes-Fernández, R. INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. En *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, vol. 55 de *Advances in Soft Computing*, páginas 40–49. Springer, 2009.

García-Magariño, I., Gómez-Sanz, J. J. y Fuentes-Fernández, R. Model Transformations for Improving Multi-agent Systems Development in INGENIAS. En *The 10th International Workshop on Agent-Oriented Software Engineering AOSE'09*. 2009a. May 11, 2009, Budapest Hungary.

García-Magariño, I., Gónzalez-Rodríguez, A. y González, J. C. Modeling the INGENIAS Development Process with EMF (in spanish). En *6th International Workshop on Practical Applications on Agents and Multi-agent Systems, IWPAAMS'07*, páginas 369–378. 2007. November 12/13, 2007, Salamanca Spain.

García-Magariño, I., Gónzalez-Rodríguez, A. y González, J. C. Definition of Process Models for Agent-based Development. En *Agent-Oriented Software Engineering IX*, vol. 5386 de *Lecture Notes in Computer Science*, páginas 60–73. Springer, 2009b.

García-Magariño, I., Gutierrez, C. y Fuentes-Fernández, R. The INGENIAS Development Kit: a practical application for crisis-management. En *The 10th International Work conference on Artificial Neuronal Networks (IWANN2009)*, vol. 5517 de *Lecture Notes in Computer Science*, páginas 537–544. Springer, 2009a.

García-Magariño, I., Rougemaille, S., Fuentes-Fernández, R., Migeon, F., Gleizes, M.-P. y Gómez-Sanz, J. J. A Tool for Generating Model Transformations By-Example in Multi-Agent Systems. En *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, vol. 55 de *Advances in Soft Computing*, páginas 70–79. Springer, 2009b.

Gerber, A., Lawley, M., Raymond, K., Steel, J. y Wood, A. Transformation: The Missing Link of MDA. *Graph Transformation: First International Conference, Icgt 2002, Barcelona, Spain, October 7-12, 2002: Proceedings*, 2002.

Gerber, A. y Raymond, K. MOF to EMF: there and back again. *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, páginas 60–64, 2003.

Gómez-Sanz, J., Fuentes, R. y Pavón, J. Enabling Rapid Prototyping using Decoupling of Code Skeletons and Code generation Process. *INFOCOMP Journal of Computer Science*, páginas 26–34, 2006.

Gómez-Sanz, J. y Pavón, J. Defining coordination in multi-agent systems within an agent oriented software engineering methodology. *Proceedings of the 2006 ACM symposium on Applied computing*, páginas 424–428, 2006.

Gómez-Sanz, J. J., Fuentes-Fernández, R., Pavón, J. y García-Magariño, I. Ingenias development kit: a visual multi-agent system development environment (best academic demo of aamas'08). páginas 1675–1676. 2008. May 12-16, 2008, Estoril Portugal.

Grasia. *Grasia* web - Additional Material for Papers: `http://grasia.fdi.ucm.es` (el el apartado "Software" → "Additional Material for Papers"). 2009a.

Grasia. *Grasia* web - Full Development Examples: `http://grasia.fdi.ucm.es` (en el apartado "Training" → "Full Development Examples"). 2009b.

Grasia. *Grasia* web - IDK with EMF: `http://grasia.fdi.ucm.es` (el el apartado "Software" → "INGENIAS Development Kit" → "IDK with EMF"). 2009c.

Grasia. *Grasia* web - The MTGenerator Tool: `http://grasia.fdi.ucm.es` (en el apartado "Software"→ "Model-Transformation Generator"). 2009d.

Henderson-Sellers, B. y Giorgini, P. *Agent-oriented Methodologies, Chapter IV: From Requirements to Code with the PASSI methodology*. Idea Group Pub., 2005.

IBM. Telelogic TAU G2. *http://www.telelogic.com/products/tau/*, 2008.

INRIA. ATLAS Trasnformation Language(ATL): User Manual. *ChemEng Software Ltd., Beaminster, UK*, 1986.

IPSquad. APES2: A Process Engineering Software. `http://apes2.berlios.de/en/links.html`, 2009.

ISO/IEC JTC1. Information technology - Syntactic metalanguage - Extended BNF. *ISO/IEC Internationl Standard*, vol. 14977:1996(E), 1996.

Jacobson, I., Booch, G. y Rumbaugh, J. *The Unified Software Development Process*. Addison-Wesley, 1999. ISBN 0201571692.

Jouault, F. y Bézivin, J. KM3: A DSL for Metamodel Specification. En *Formal Methods for Open Object-Based Distributed Systems*, vol. 4037 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 171. Springer, 2006.

Jouault, F. y Kurtev, I. On the architectural alignment of ATL and QVT. *Proceedings of the 2006 ACM symposium on Applied computing*, páginas 1188–1195, 2006a.

JOUAULT, F. y KURTEV, I. Transforming Models with ATL. En *Satellite Events at the MoDELS 2005 Conference - Proceedings of the Model Transformations in Practice Workshop at MoDELS*, vol. 3844 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 128–138. Springer, 2006b.

KALNINS, A., CELMS, E. y SOSTAKS, A. Simple and Efficient Implementation of Pattern Matching in MOLA Tool. *Proceedings of the 7th International Baltic Conference on Databases and Information Systems (Baltic 92006) pp*, páginas 159–167, 2006.

KAPPEL, G., KAPSAMMER, E., KARGL, H., KRAMLER, G., REITER, T., RETSCHITZEGGER, W., SCHWINGER, W. y WIMMER, M. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. En *Model Driven Engineering Languages and Systems*, vol. 4199 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 528. Springer, 2006.

KAPPEL, G., KARGL, H., KRAMLER, G., SCHAUERHUBER, A., SEIDL, M., STROMMER, M. y WIMMER, M. Matching Metamodels with Semantic Systems-An Experience Report. En *Workshop Proc. of Datenbanksysteme in Business, Technologic und Web (BTW 2007), Germany*. 2007.

KELLY, S. GOPRR Description. *PhD dissertation*, vol. Appendix 1, 1997.

KERN, H. y KUHNE, S. Model Interchange between ARIS and Eclipse EMF. En *7th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*, vol. 2007. 2007.

KOLOVOS, D., PAIGE, R. y POLACK, F. Aligning OCL with Domain-Specific Languages to Support Instance-Level Model Queries. *Electronic Communications of the EASST*, vol. 5, 2006.

KOVSE, J. y HARDER, T. Generic XMI-Based UML Model Transformations. En *Object-Oriented Information Systems*, vol. 2425 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 183–190. Springer, 2002.

KRISHNAMURTHI, S., GRAY, K. y GRAUNKE, P. Transformation-by-Example for XML. *Practical Aspects of Declarative Languages: Second International Workshop, PADL 2000, Boston, MA, USA, January 2000: Proceedings*, 2000.

LAWLEY, M. y STEEL, J. Practical Declarative Model Transformation with Tefkat. En *Satellite Events at the International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005*, vol. 3844 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 139–150. Springer-Verlag, 2006.

LEDECZI, A., BAKAY, A., MAROTI, M., VOLGYESI, P., NORDSTROM, G., SPRINKLE, J. y KARSAI, G. Composing domain-specific design environments. *Computer*, vol. 34(11), páginas 44–51, 2001a.

LEDECZI, A., MAROTI, M., BAKAY, A., KARSAI, G., GARRETT, J., THOMASON, C., NORDSTROM, G., SPRINKLE, J. y VOLGYESI, P. The Generic Modeling Environment. En *Workshop on Intelligent Signal Processing (WISP 2001)*, vol. 17. 2001b. Budapest, Hungary.

Leroux, D., Nally, M. y Hussey, K. Rational Software Architect: A tool for domain-specific modeling-Author Bios. *IBM Systems Journal*, vol. 45(3), 2006.

Lopes, D., Hammoudi, S., Bézivin, J. y Jouault, F. Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. En *Advanced Information Systems Engineering*, vol. 3520 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 309–325. Springer, 2005.

Magyari, E., Bakay, A., Lang, A., Paka, T., Vizhanyo, A., Agrawal, A. y Karsai, G. UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. *Workshop on Domain-Specific Modeling, at OOPSLA 2003*, 2003. Anaheim, California, USA.

Matula, M. NetBeans Metadata Repository. *http://mdr.netbeans.org/MDR-whitepaper.pdf, accessed on 6/11/2008*, 2003.

Mei, J., Xie, G., Liu, S., Zhang, L. y Pan, Y. A System to enable Relational Persistence and Semantic Web style access simultaneously for Objects. *7th International Semantic Web Conference (ISWC2008)*, 2008.

Milicev, D. Automatic model transformations using extended UML object diagramsin modeling environments. *Software Engineering, IEEE Transactions on*, vol. 28(4), páginas 413–431, 2002.

Mohamed, M., Romdhani, M. y Ghedira, K. MOF-EMF Alignment. En *Proceedings of the Third International Conference on Autonomic and Autonomous Systems (ICAST 2007)*, páginas 1–6. IEEE Computer Society Washington, DC, USA, 2007. Athens, Greece.

Moore, B., Organization, I. T. S. y Corporation, I. B. M. *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM, International Technical Support Organization, 2004.

Murray-Rust, P., Rzepa, H. y Wright, M. Development of chemical markup language (CML) as a system for handling complex chemical content. *New J. Chem*, vol. 25, páginas 618–634, 2001.

Murzek, M., Kappel, G. y Kramler, G. Model Transformation in Practice Using the BOC Model Transformer. *online Proceedings of Model Transformations in Practice Workshop*, 2006.

Newcomb, P. Architecture-Driven Modernization (ADM). En *Reverse Engineering, 12th Working Conference on*, páginas 237–237. 2005.

Noy, N., Musen, M., Mejino, J. y Rosse, C. Pushing the envelope: challenges in a frame-based representation of human anatomy. *Data and Knowledge Engineering Journal*, vol. 48(3), páginas 335–359, 2002.

Nytun, J., Prinz, A. y Tveit, M. Automatic Generation of Modelling Tools. En *Model Driven Architecture - Foundations and Applications*, vol. 4066 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 268. Springer, 2006.

OMG. MOF QVT Final Adopted Specification, pct/05-11-01. *Object Management Group OMG*, 2005a.

OMG. UML 2.0 Diagram Interchange Specification. Abril 2006.

OMG, O. M. G. CWMM 1.1, Common Warehouse Metamodel. 2003a.

OMG, O. M. G. Model-driven Architecture (MDA), Guide Version 1.0.1, omg/03-06-01. 2003b.

OMG, O. M. G. Software process engineering metamodel specification. version 1.1. 2005b.

OMG, O. M. G. CORBA Component Model Specification, Version 4.0. 2006a.

OMG, O. M. G. Meta object facility (mof) core specification, v 2.0. 2006b.

OMG, O. M. G. UML 2.2, Unified Modeling Language Infrastructure. 2007a.

OMG, O. M. G. UML 2.2, Unified Modeling Language Superstructure. 2007b.

OMG, O. M. G. Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), v1.0. 2008a.

OMG, O. M. G. BMM 1.0, Business Motivation Model. 2008b.

OMG, O. M. G. Software Process Engineering Metamodel Specification. Version 2.0. 2008c.

Opdahl, A. y Henderson-Sellers, B. Grounding the OML metamodel in ontology. *The Journal of Systems & Software*, vol. 57(2), páginas 119–143, 2001.

Padgham, L., Thangarajah, J. y Winikoff, M. The Prometheus Design Tool-A Conference Management System Case Study. En *Agent-Oriented Software Engineering VIII*, vol. 4951 de *LECTURE NOTES IN COMPUTER SCIENCE*, página 197. Springer, 2008.

Padgham, L. y Winikoff, M. Prometheus: A Methodology for Developing Intelligent Agents. *Proceedings of the Third International Workshop on Agent Oriented Software Engineering, at AAMAS*, 2002.

Padgham, L., Winikoff, M. y Poutakidis, D. Adding debugging support to the Prometheus methodology. *Engineering Applications of Artificial Intelligence*, vol. 18(2), páginas 173–190, 2005.

Pan, Y., Xie, G., Ma, L., Yang, Y., Qiu, Z. y Lee, J. Model-Driven Ontology Engineering. En *Journal on Data Semantics VII*, vol. 4244 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 57–78. Springer, 2006.

Pavón, J., Arroyo, M., Hassan, S. y Sansores, C. Agent-based modelling and simulation for the analysis of social patterns. *Pattern Recognition Letters*, vol. 29(8), páginas 1039–1048, 2008.

Pavón, J., Corchado, J., Gómez-Sanz, J. y Ossa, L. Mobile Tourist Guide Services with Software Agents. En *Mobility Aware Technologies and Applications*, vol. 3284 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 322–330. Springer, 2004.

Pavón, J. y Gómez-Sanz, J. Agent Oriented Software Engineering with INGENIAS. *Multi-Agent Systems and Applications III*, vol. 2691, páginas 394–403, 2003.

Pavón, J., Gómez-Sanz, J., Fernández-Caballero, A. y Valencia-Jiménez, J. Development of intelligent multisensor surveillance systems with agents. *Robotics and Autonomous Systems*, vol. 55(12), páginas 892–903, 2007.

Pavón, J., Gómez-Sanz, J. y Fuentes, R. The INGENIAS Methodology and Tools. *Agent-Oriented Methodologies. Idea Group Publishing*, páginas 236–276, 2005.

Pavón, J., Gómez-Sanz, J. y Fuentes, R. Model Driven Development of Multi-Agent Systems. En *Model Driven Architecture - Foundations and Applications*, vol. 4066 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 284–298. Springer, 2006.

Perini, A. y Susi, A. Automating Model Transformations in Agent-Oriented Modelling. En *Agent-Oriented Software Engineering VI*, vol. 3950 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 167–178. Springer, 2006.

Porres, I. Model refactorings as rule-based update transformations. En *UML*, páginas 159–174. 2003.

Rector, A., Bechhofer, S., Goble, C., Horrocks, I., Nowlan, W. y Solomon, W. The GRAIL concept modelling language for medical terminology. *Artificial Intelligence in Medicine*, vol. 9(2), páginas 139–171, 1997.

Reina, A. y Torres, J. Using Aspect-orientation Techniques to Improve Reuse of Metamodels. *Electronic Notes in Theoretical Computer Science*, vol. 163(2), páginas 29–43, 2007.

Ren, W. y Beard, R. Consensus seeking in multiagent systems under dynamically changing interaction topologies. *IEEE Transactions on Automatic Control*, vol. 50(5), páginas 655–661, 2005.

Richters, M. y Gogolla, M. Validating UML Models and OCL Constraints. *UML 2000–the Unified Modeling Language: Advancing the Standard: Third International Conference, York, UK, October 2-6, 2000: Proceedings*, 2000.

Roser, S. y Bauer, B. An approach to automatically generated model transformations using ontology engineering space. *Proceedings of Workshop on Semantic Web Enabled Software Engineering (SWESE)*, 2006.

Rougemaille, S., Arcangeli, J.-P., Gleizes, M.-P. y Migeon, F. ADELFE Design, AMAS-ML in Action. En *International Workshop on Engineering Societies in the Agents World (ESAW), Saint-Etienne, 24/09/08-26/09/08*. Springer-Verlag, 2008.

ROUGEMAILLE, S., MIGEON, F., MAUREL, C. y GLEIZES, M.-P. Model Driven Engineering for Designing Adaptive Multi-Agent Systems. En *The 8th annual International Workshop on Engineering Societies in the Agents World: ESAW*, página (on line). Springer, 2007.

ROUGEMAILLE, S., MIGEON, F., MILLAN, T. y GLEIZES, M.-P. Methodology Fragments Definition in SPEM for Designing Adaptive Methodology : a First Step. En *Agent-Oriented Software Engineering IX*, vol. 5386 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 74–85. Springer, 2009.

RUSSOMANNO, D. A plausible inference prototype for the Semantic Web. *Journal of Intelligent Information Systems*, vol. 26(3), páginas 227–246, 2006.

SCHEKKERMAN, J. Enterprise Architecture Good Practices Guide: How to Manage the Enterprise Architecture Practice. *Trafford Publishing, Canada, ISBN: 1-4251-5687-8*, 2008.

SELIC, B. The pragmatics of model-driven development. *Software, IEEE*, vol. 20(5), páginas 19–25, 2003.

SENDALL, S. y KOZACZYNSKI, W. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, vol. 20(5), páginas 42–45, 2003.

SOMMERVILLE. *Software Engineering, 7th ed.*. Addison Wesley, Wokingham, UK, 2004.

SOTO, J. P., VIZCAÍNO, A., PORTILLO, J. y PIATTINI, M. Modelling a Knowledge Management System Architecture with INGENIAS Methodology. *Proceedings of the 15th International Conference on Computing*, 2006.

STEEL, J. y LAWLEY, M. Model-based test driven development of the Tefkat model-transformation engine. En *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, páginas 151–160. 2004.

STEIN, D., HANENBERG, S. y UNLAND, R. A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. En *Model Driven Architecture*, vol. 3599 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 77–92. Springer, 2005.

STROMMER, M., MURZEK, M. y WIMMER, M. Applying Model Transformation By-Example on Business Process Modeling Languages? En *Advances in Conceptual Modeling Foundations and Applications*, vol. 4802 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 116–125. Springer, 2007.

SUSI, A., PERINI, A. y MYLOPOULOS, J. The Tropos Metamodel and its Use. *Informatica*, vol. 29(4), páginas 401–408, 2005.

SUSS, J., MCCOMB, T., KIM, S., WILDMAN, L. y WATSON, G. MDA-Based Re-engineering with Object-Z. En *Model Driven Engineering Languages and Systems, MoDELS'06*, vol. 4199 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 291–305. Springer, 2006.

TOLVANEN, J. y ROSSI, M. MetaEdit+: defining and using domain-specific modeling languages and code generators. *Conference on Object Oriented Programming Systems Languages and Applications*, páginas 92–93, 2003.

TOLVANEN, J.-P. GOPRR metamodeling language. 2000.

TOMBELLE, C. y VANWORMHOUDT, G. Dynamic and Generic Manipulation of Models: From Introspection to Scripting. En *Model Driven Engineering Languages and Systems*, vol. 4199 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 395–409. Springer, 2006.

TRAN, H., ZDUN, U. y DUSTDAR, S. View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. *BPSC, Leipzig, Germany*, 2007.

TRATT, L. Model transformations and tool integration. *Software and Systems Modeling*, vol. 4(2), páginas 112–122, 2005.

TRATT, L. Model transformations in MT. *Science of Computer Programming*, vol. 68(3), páginas 169–186, 2007.

TURNER, R. Seven pitfalls to avoid on the hunt for best practices. *Software, IEEE*, vol. 20(1), páginas 67–69, 2003.

VARRO, D. Model transformation by example. En *Model Driven Engineering Languages and Systems*, vol. 4199 de *LECTURE NOTES IN COMPUTER SCIENCE*, páginas 410–424. Springer, 2006.

VARRÓ, D. y BALOGH, A. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, vol. 68(3), páginas 187–207, 2007a.

VARRÓ, D. y BALOGH, Z. Automating model transformation by example using inductive logic programming. *Proceedings of the 2007 ACM symposium on Applied computing*, páginas 978–984, 2007b.

VARRO, D., GYAPAY, S. y PATARICZA, A. Automatic transformation of UML models for system verification. *WTUML*, vol. 1, páginas 123–127, 2001.

W3C, W. W. W. C. OWL, Web Ontology Language. Overview. *http://www.w3.org/*, 2004.

WARMER, J. y KLEPPE, A. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Professional, 2003.

WEISS, G. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.

WILLINK, E. UMLX: A graphical transformation language for MDA. En *Model Driven Architecture - Foundations and Applications*, páginas 03–27. 2003.

WIMMER, M., STROMMER, M., KARGL, H. y KRAMLER, G. Towards Model Transformation By-Example. *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, 2007.

WOOLDRIDGE, M. y JENNINGS, N. Pitfalls of agent-oriented development. En *Proceedings of the second international conference on Autonomous agents*, páginas 385–391. ACM New York, NY, USA, 1998.

WOOLDRIDGE, M., JENNINGS, N. y KINNY, D. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, vol. 3(3), páginas 285–312, 2000.

Glosario

ADELFE - *Atelier de Développement de Logiciels à Fonctionnalité Emergente*
ADM - *Architecture-Driven Modernization*
APES - *A Process Engineering Software*
ATL - *Atlas Transformation Language*
AUML - *Agent Unified Modeling Language*
CASE - *Computer-aided Software Engineering*
CML - *Chemical Markup Language*
DSDM - Desarrollo del Software Dirigido por Modelos
EJB - *Enterprise JavaBeans*
EMF - *Eclipse Modeling Framework*
EMOF - *Essential Meta Object Facilities*
EPF - *Eclipse Process Framework*
ER - Entidad-Relación
FAML - *FAME Agent-oriented Modeling Language*
FIPA - *Foundation for Intelligent Physical Agents*
GRAIL - *GALEN Representation And Integration Language*
GEF - *Graphical Editing Framework*
GME - *Generic Modeling Environment*
GMF - *Graphical Modeling Framework*
GOPRR - Grafo, Objeto, Propiedad, Relación, y Rol
GTBE - Generación de Transformaciones Basadas en Ejemplos
IDK - *INGENIAS Development Kit*
ISOA - Ingeniería del Software Orientada a Agentes
JADE - *Java Agent DEvelopment framework*
KM3 - *Kernel Meta-MetaModel*
LM - Lenguaje de Modelado
LMED- Lenguaje de Modelado Específico de Dominio
LTM - Lenguaje de Transformación de Modelos
MDA - *Model Driven Architecture*
MEP - Modelo Específico de Plataforma
MIC - Modelo Independiente de Computación
MIP - Modelo Independiente de Plataforma
MOF - *Meta Object Facilities*
OCL - *Object Constraint Language*
OMG - *Object Management Group*
PDT - *Prometheus Design Tool*
QVT - Query View Transformation

RSA - *Rational Software Architect*
RUP - *Rational Unified Process*
SMA - Sistema Multi-agente
SMAA - Sistema Multi-agente Adaptativo
SPEM - *Software Process Engineering Metamodel*
SVG - *Scalable Vector Graphics*
TM - Transformación de Modelos
UML - *Unified Modeling Language*
UML-DI - *Unified Modeling Language - Diagram Interchange*
URI - *Uniform Resource Identifier*
W3C - *World Wide Web Consortium*
XHTML- *eXtensible Hypertext Markup Language*
XML - *Extensible Markup Language*
XSL - *Extensible Stylesheet Language*
XSLT - *XSL Transformation*

Appendix D

Publications with the thesis content

The details of this investigation, as well as the experimentation and the obtained results, are collected in the following papers:

- **Paper 1: Framework for Defining Model Language Metamodels for CASE Tools, I. García-Magariño and J.J. Gómez-Sanz, In *5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software, MOMPES'08*, pp. 14–23. IEEE Computer Society, 2008.** *This paper presents a framework for defining metamodels for CASE tools of MASs. The presented framework contains four internal metamodels. Firstly, a metamodel specifies the abstract syntax of the modeling language. Secondly, another metamodel specifies the particular elements for the connection-based languages. Thirdly, a metamodel specifies the spatial information. At last, a metamodel defines the necessary elements for the CASE tools. Each internal metamodel can be changed without having effect on the others. These four internal metamodels can be combined in several ways (denoted as configurations). Each configuration has its particular scope. The presented framework has been already used for defining the INGENIAS language and modernizing the INGENIAS Development Kit (IDK). The presented framework is compared to the UML Diagram Interchange (UML-DI) specification and other relevant works.*

- **Paper 2: Guideline for the definition of EMF metamodels using an Entity-Relationship approach, I. García-Magariño, R. Fuentes-Fernández, and J.J. Gómez-Sanz, Information and Software Technology, 51 (8):1217–1230. Elsevier, 2009, (JCR 2008 1.200, Second Quartile).** *This paper presents a guideline for defining the metamodels for the abstract syntax of the framework presented in the previous paper. This guideline proposes to begin by determining the structural features of the language, such as types of relationships and elements with attributes. Subsequently, it offers alternative representations for these features aimed at satisfying different requirements, such as changeability*

or optimized model processing. Two case studies illustrate the use of the guideline and its trade-offs.

- **Paper 3: Definition of Process Models for Agent-based Development, I. García-Magariño, A. Gónzalez-Rodríguez, and J.C. González. In *Agent-Oriented Software Engineering IX (Revised Selected Papers)*, volume 5386 of Lecture Notes in Computer Science, pp. 60–73. Springer, 2009, (CORE B; CiteSeer 1.57, top 7.45%).** *This paper presents a successful application of the guidelines of the previous paper. A metamodel is defined with the guideline, and a process editor tool is generated from this metamodel. The presented editor can edit modeling concepts that cannot be edited by other similar tools, such as EPF and APES2.*

- **Paper 4: Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages, I. García-Magariño, J.J. Gómez-Sanz, and R. Fuentes-Fernández. In *International Conference on Model Transformation (ICMT'09)*, volume 5563 of Lecture Notes in Computer Science, pp. 52–66. Springer, 2009, (22% of acceptance ratio).** *This paper presents a MTBE algorithm, which can generate many-to-many rules in several transformation languages even if the transformation language cannot directly represent these kinds of rules. The algorithm can associate attributes for transferring information, considering both groups of elements in the input and output. Moreover, each transformation rule can transform a group of non-connected graphs. These features are not available in the existent MTBE approaches.*

- **Paper 5: A Tool for Generating Model Transformations By-Example in Multi-Agent Systems, I. García-Magariño, S. Rougemaille, R. Fuentes-Fernández, F. Migeon, M.P. Gleizes, and J.J. Gómez-Sanz. In *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS'09)*, volume 55 of Advances in Soft Computing, pp. 70–79. Springer, 2009, (CSCR 0.56, pos. 52/701).** *This work describes the implementation of the MTBE algorithm for ATL. The implementation is included in a tool called MTGenerator, which is publicly available. This paper presents two examples that are useful respectively in the ADELFE and INGENIAS agent-oriented methodologies.*

- **Paper 6: INGENIAS development process assisted with chains of transformations, I. García-Magariño, R. Fuentes-Fernández, and J.J. Gómez-Sanz. In *The 10th International Work conference on Artificial Neuronal Networks (IWANN'09)*, volume 5517 of *Lecture Notes in Computer Science*, pp. 514–521. Springer, 2009, (CORE B; CiteSeer 0.16, top 79.85%; CSCR 0.55, pos. 55/701).** *This paper proposes several examples of MTs that can be generated with the MTGenerator tool. These examples are useful in INGENIAS methodology, and the paper shows the manner that these MTs can be applied in the development of two existent MASs.*

- **Paper 7: Model Transformations for Improving Multi-agent Systems Development in INGENIAS, I. García-Magariño, J.J. Gómez-Sanz, and Rubén Fuentes-Fernández, In *The 10th International Workshop on Agent-Oriented Software Engineering, AOSE'09*, 2009, (CORE B; CiteSeer 1.57, top 7.45%).** *This work contains three examples of MTBE in the INGENIAS methodology, which are different from the examples presented in the previous papers.*

European researchers can read the content of this thesis in English by reading the aforementioned papers.