

GENERACIÓN DE EDITORES GRÁFICOS DE MODELOS PARA UNA HERRAMIENTA MDA

Francisco Vargas Ruiz¹, José Luis Roda García², Antonio Estévez García¹,
Orlando Avila-García¹, E. Victor Sánchez Rebull¹

1: Open Canarias S.L.

Santa Cruz de Tenerife, España

e-mail: {francisco.vargas, aestevz, orlando, vsanchez}@opencanarias.com, web:

<http://www.opencanarias.com>

2: Escuela Técnica Superior de Ingeniería Informática

Universidad de La Laguna

Santa Cruz de Tenerife, España

e-mail: jlroda@ull.es, web: <http://www.etsii.ull.es>

Palabras clave: DSL, Edición, Editor, MDA, Modelo

Resumen. Debido al reciente interés suscitado por los Lenguajes Específicos de Dominio (DSLs) como lenguajes de modelado, es necesario que las herramientas de modelado permitan la incorporación de nuevos editores gráficos que los asistan o, si cabe, la posibilidad de definir editores personalizados para su uso dentro de estas herramientas. En este artículo se presenta un framework llamado MSGF (ModelSET Graphical Framework), que opera en el contexto de una herramienta MDA con el propósito de diseñar y generar editores gráficos destinados a modelar instancias de sus respectivos DSLs.

1. Introducción

El objetivo principal del Desarrollo de Software Dirigido por Modelos (DSDM) es construir software a partir de modelos, desplazando así el uso tradicional del código fuente como protagonista principal de los procesos de desarrollo.

La descripción formal en un sistema computacional de estos modelos se produce a través de la especificación de lenguajes de modelado. Un ejemplo muy extendido de estos lenguajes es UML [8], de propósito general, para el cual existen diversas herramientas en el mercado, tanto gratuitas (Poseidon [5]) como de pago (IBM Rational [7]) que dan soporte a la edición de modelos basados en él.

Pero UML adolece de ciertos problemas como lenguaje de modelado enfocado al DSDM, sobre todo en relación a su tamaño y complejidad. En muchas ocasiones sólo se requiere una pequeña parte o subconjunto de su especificación. UML permite definir Perfiles que podrían dar solución a este problema. Otras veces resultará muy conveniente

utilizar diagramas de edición particularizados a un escenario o dominio concreto, con requisitos, reglas de edición y elementos visuales involucrados particulares, que nos permitan formalizar modelos con un mayor grado de expresividad.

Es aquí donde juegan un papel importante los Lenguajes Específicos de Dominio (DSLs) [16] como lenguajes de modelado, y sus respectivos metamodelos como aportación sintáctica abstracta. Para poder dar un uso práctico a estos DSLs surge la necesidad de disponer de editores que faciliten la creación de instancias de dichos DSLs.

Además de ofrecer asistencia a la edición de DSLs predefinidos, sería ideal de cara a lograr la máxima libertad en el desarrollo que las herramientas DSDM dieran facilidades para la personalización de los editores registrados en ella, o contaran con una infraestructura para permitir al usuario definir nuevos DSLs y sus editores correspondientes de forma práctica y amigable.

A principios de 2005 la empresa Open Canarias S.L. toma la decisión de crear una herramienta de desarrollo denominada ModelSET (Models for Software Engineering Technologies), cuyo objetivo es dar soporte completo al ciclo de vida del desarrollo de software, siguiendo la aproximación MDA (Arquitectura Dirigida por Modelos) [9]. Como aproximación dirigida por modelos, MDA potencia el uso de estándares, dentro de los cuales se encuentra MOF (Meta-Object Facility) [11], como lenguaje de metamodelado para especificar metamodelos que representan lenguajes de modelado, y por lo tanto, DSLs.

ModelSET se construye sobre el entorno de desarrollo Eclipse, y se basa en EMF (Eclipse Modeling Framework) [13] y su lenguaje de metamodelado Ecore, compatible con MOF, para definir metamodelos para DSLs. Dentro de ModelSET se ha desarrollado un marco para la generación automática de editores gráficos para DSLs, denominado *MSGF* (ModelSET Graphical Framework), que es el objeto principal de este trabajo.

El artículo se estructura como sigue: en la sección 2 se introducen los elementos que participan en los componentes arquitectónicos y de ejecución del proyecto MSGF; en las secciones 3, 4 y 5 se desglosan en detalle los modelos de entrada, el componente *MSGF.Editor* y el componente *MSGF.Generator*, respectivamente; en la sección 6 se estudian aproximaciones similares a MSGF y se presenta una comparativa. Y finalmente se exponen las conclusiones.

2. Descripción general de MSGF

El propósito de nuestro framework *MSGF* es la generación automática de la totalidad del código Java de un editor gráfico para un DSL determinado (que podemos denominar DSL destino). Este proceso de generación se encuentra a su vez dirigido por modelos (DSDM), de manera que MSGF requiere de ciertos modelos que contendrán información indispensable para guiarlo en la generación del editor deseado. De este modo, podemos considerar que es una aproximación basada en programación generativa [3].

MSGF se construye en Eclipse, de donde aprovecha la potencia de EMF como facilidad para el metamodelado y la generación de código Java, y de GEF (Graphical Editing Framework) [14], marco de trabajo que ofrece la infraestructura necesaria para la creación

de editores gráficos sobre Eclipse. El editor destino generado por MSGF toma la forma de un conector o plug-in para Eclipse.

MSGF se compone de dos módulos: el editor de editores *MSGF.Editor* y el generador de código *MSGF.Generator*. El editor de editores sirve para editar aquellos modelos que guían la generación final del plug-in editor. Como *MSGF.Editor* es WYSIWYG¹, ofrece la posibilidad de ver en tiempo real el resultado de la representación gráfica que se da a los elementos del DSL. En el caso del generador de código *MSGF.Generator*, éste recibe como entrada los modelos que se obtienen a partir de *MSGF.Editor* para producir como salida el código Java final del plug-in editor.

Como veremos en las siguientes secciones, además del DSL destino, los distintos elementos implicados en la generación del editor mediante MSGF son dos modelos de entrada (y sus respectivos metamodelos), *MSGF.Editor*, para editar estos modelos, y *MSGF.Generator*, que genera el plug-in editor a partir de los modelos de entrada.

3. Modelos de entrada

Existen dos modelos en MSGF que contribuyen a la generación del plug-in por parte de *MSGF.Generator* y que se obtienen a través del editor *MSGF.Editor*, uno encargado de los aspectos estructurales de la edición, y otro que ejerce de vínculo o puente entre el metamodelo del DSL destino y este primer modelo. Los respectivos metamodelos a los cuales ambos conforman reciben el nombre de **Editor** y **Bridge** (ver figura 1).

El metamodelo *Bridge* permite especificar los puentes semánticos² que definen asociaciones entre elementos del DSL y elementos del modelo *Editor*.

El tratamiento que damos a la pirámide de metamodelado [4] en esta figura 1 no es muy habitual, con *Ecore* situado en el nivel M4, además de en M3. No existe ningún problema en el planteamiento de este esquema o variantes similares, siempre y cuando se respeten las relaciones de instanciación establecidas entre niveles adyacentes. El resto es una mera convención para comunicar una idea que en este caso ni siquiera afecta al desarrollo del framework MSGF o influye en su ejecución a la hora de obtener los editores destino.

En esta figura, hemos ubicado el modelo *Editor* en el nivel M2 con el fin de alinearlo al DSL destino con el cual se relaciona estrechamente. Por la misma razón el modelo *Bridge* se ha ubicado también en este nivel. Como consecuencia, sus respectivos metamodelos quedan emplazados en el nivel M3 y su meta-metamodelo (*Ecore*) en M4. Un caso clásico de esta disposición lo encontramos en los metamodelos QVT [10], aunque la existencia de MOF en M4 en estos esquemas suele omitirse por simplicidad. A pesar de no haberlos situado en el nivel M1 acostumbrado, los modelos *Bridge* y *Editor* han de ser interpretados como modelos finales, que intervienen únicamente en la generación del código del editor y no están pensados para ser instanciados.

En suma, en la generación de un editor se requiere una instancia tanto del metamodelo

¹What You See Is What You Get

²la necesidad de estos puentes semánticos se justifica en la sección 3.2

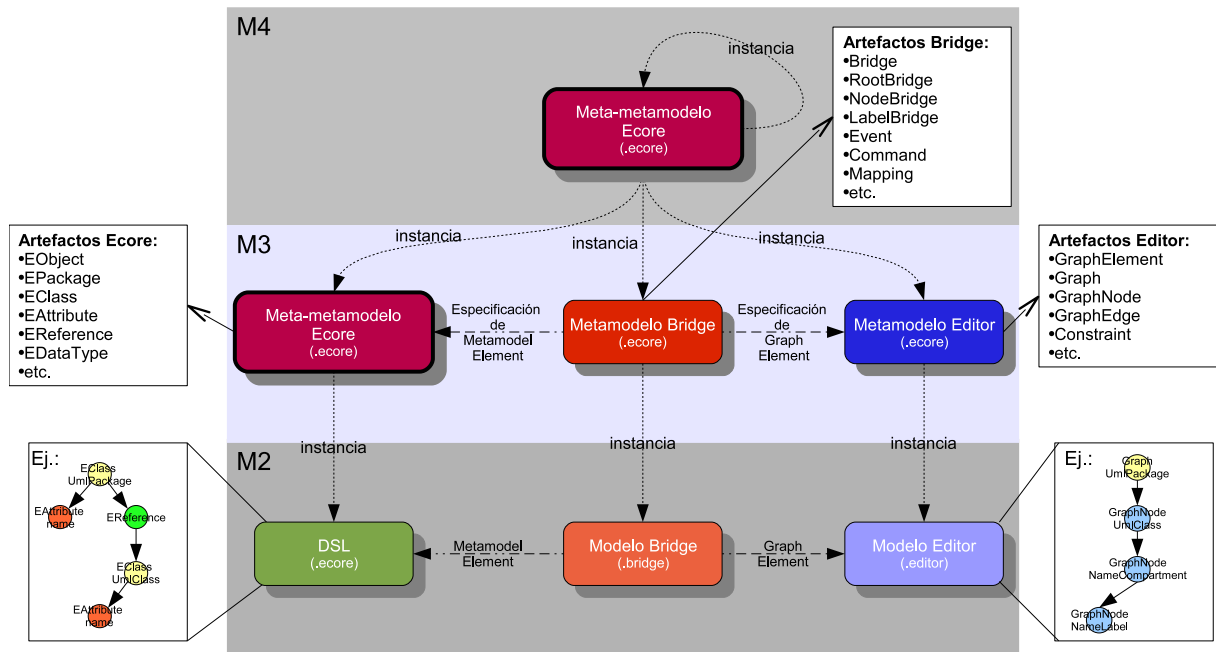


Figura 1: Disposición y relación entre los metamodelos *Bridge* y *Editor* con respecto al DSL y sus correspondientes modelos.

Editor como del *Bridge*.

3.1. Metamodelo Editor

Para poder ser editados, los elementos del DSL (que puede ser considerado como sintaxis abstracta) necesitan tener asociada una representación visual. El metamodelo *Editor* sirve para especificar esta representación y establecer criterios para la manipulación de dichos elementos a través del editor generado. Las instancias de este metamodelo contienen la lógica que representa la sintaxis concreta.

Dado que los editores gráficos generados están basados en grafos, el elemento raíz de este tipo de representación es un grafo que se compone de nodos y arcos. En los nodos a su vez se pueden anidar otros grafos, con sus respectivos nodos y arcos.

Visualmente los nodos se pueden representar como formas (rectángulos, elipses, etc.), etiquetas (para mostrar texto) e imágenes. A su vez, podemos formar figuras más complejas combinando nodos y arcos con distintas características.

En términos del metamodelo *Editor*, todo elemento gráfico es un *GraphElement*. Encontramos dos tipos de elementos gráficos: *GraphNode* (que representa un nodo) y *GraphEdge* (que representa un arco o arista entre dos nodos). A su vez tenemos *Graph* (que representa el grafo) que es el contenedor de todos los nodos y arcos, es decir, el nodo raíz.

Para dar las características de representación y de edición a nodos y arcos disponemos del elemento *Constraint* (restricción). Mediante restricciones asociadas a los elementos

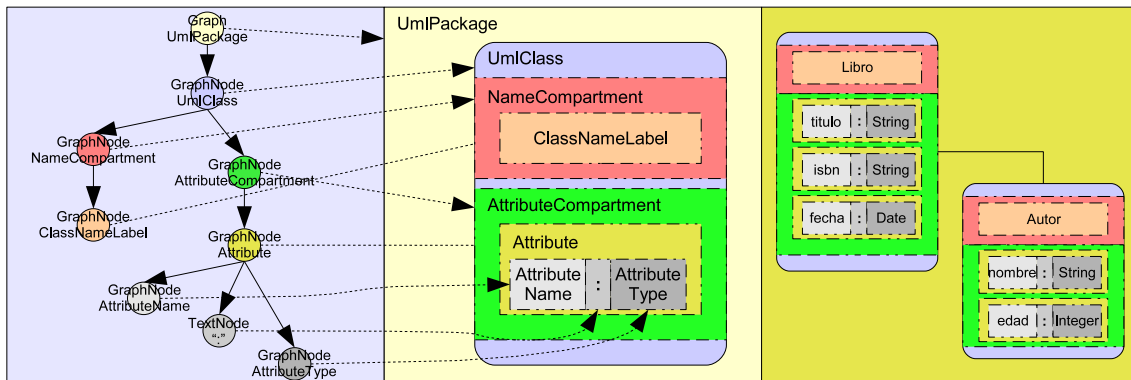


Figura 2: Representación gráfica (centro y dcha.) de la información del modelo *Editor* (izq.).

gráficos expresamos: la forma de un nodo (rectangular, con bordes redondeados, etc.), el tipo de fuente y estilo (negrita, etc.) empleado para mostrar un texto, el grosor y estilo (sólido, punteado, etc.) de bordes y líneas, los colores, la disposición de los nodos hijo o anidados, qué elementos son seleccionables, eliminables, redimensionables, etc.

En la parte izquierda de la figura 2 podemos observar un grafo, instancia del meta-modelo *Editor*, donde se especifica la representación de una clase UML simplificada. En el centro vemos el resultado visual de la información contenida en este grafo. A la derecha nos encontramos con un ejemplo de uso de esta representación, esto es, tal y como aparecería en el editor. Las restricciones asociadas a cada elemento del grafo se han obviado por simplicidad.

3.2. Metamodelo Bridge

El modelo *Editor* indica cómo y dónde se debe representar un nodo o arco, pero no qué elemento del DSL representa. Las asociaciones o puentes semánticos entre los nodos o arcos (elementos gráficos) y los elementos del DSL vienen especificadas en el modelo *Bridge*. Esta separación de los metamodelos *Editor* y *Bridge* ofrece flexibilidad en el sentido de que las características de representación y edición modeladas pueden ser reutilizadas. Para ello simplemente se instancian nuevos modelos *Bridge*, cada uno asociado a un DSL distinto, pero todos ellos relacionados con el mismo modelo *Editor*.

El elemento base de este metamodelo es *Bridge*, cuya instancia es un puente semántico que asocia o mapea un elemento concreto del DSL con un elemento gráfico definido en el modelo *Editor*. En función de cómo se desea realizar el mapeo existen distintos tipos de *Bridge*: *NodeBridge* (mapeo entre nodo - elemento del DSL), *RootBridge* (mapeo grafo - raíz del metamodelo DSL), *EdgeBridge* (mapeo arco - elemento del DSL), *LabelBridge* (mapeo nodo - texto contenido en la instancia del DSL en edición), etc.

A los elementos *Bridge* se les asocian eventos (*Events*) para especificar las acciones que se pueden realizar sobre la representación gráfica de cada elemento. A cada evento se le asocia un comando (*Command*), que especifica las modificaciones a ejecutar sobre la

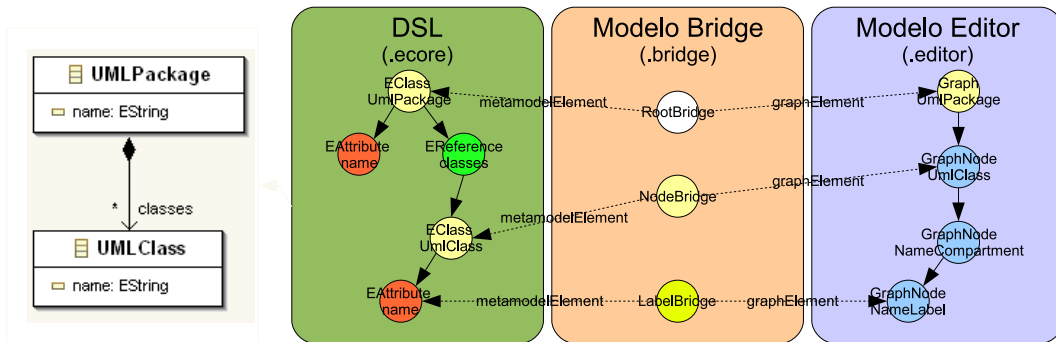


Figura 3: Ejemplo de puentes semánticos definidos en un modelo *Bridge* sencillo. A la izquierda podemos ver la representación de nuestro DSL en notación UML.

instancia del DSL cada vez que ocurre el evento. En contraposición, los elementos *Bridge* tienen *Mappings* asociados, para actualizar la representación cuando se producen cambios en la instancia del DSL manipulada en el editor, es decir, el modelo editado.

En la figura 3 podemos ver un ejemplo simplificado de la relación entre las instancias de los diferentes metamodelos implicados en MSGF. En particular, los distintos puentes semánticos que el modelo *Bridge* establece entre el DSL destino y el modelo *Editor*.

Algo similar ha sido propuesto en [6], donde el metamodelo que relaciona los metamodelos de sintaxis abstracta y sintaxis concreta es denominado como metamodelo de correspondencia. Sin embargo, a diferencia de nuestro metamodelo *Bridge*, su metamodelo es muy simple ya que sólo permite establecer relaciones entre objetos, pero en ningún caso eventos y comandos asociados. Estos últimos son definidos mediante una gramática dirigida por eventos. Queda fuera del ámbito de este artículo establecer las ventajas o inconvenientes de esta separación.

4. Editor de modelos de entrada

MSGF ofrece un editor de editores denominado **MSGF.Editor**, basado en GEF, que permite modelar gráficamente el editor gráfico para un DSL. Su principal característica es que se trata de un editor WYSIWYG.

El objetivo de este editor es instanciar modelos de *Editor* y de *Bridge* (ver figura 4), los cuales serán empleados como parte de la entrada para el generador de código (*MSGF.Generator*), descrito en el siguiente apartado.

Para visualizar los elementos de estos modelos es necesario disponer de información de posición (coordenadas) y de dimensión (anchura y altura), que se gestiona de forma independiente y no interviene en la generación del código de los editores. Para el tratamiento de esta información se define el metamodelo **Diagram**, cuyas instancias son modelos decoradores (*decorator* o *wrapper*) de los modelos de entrada representados gráficamente, a los que añaden las características de posición y dimensión.

La infraestructura que ofrece GEF para crear editores sigue el patrón de diseño MVC

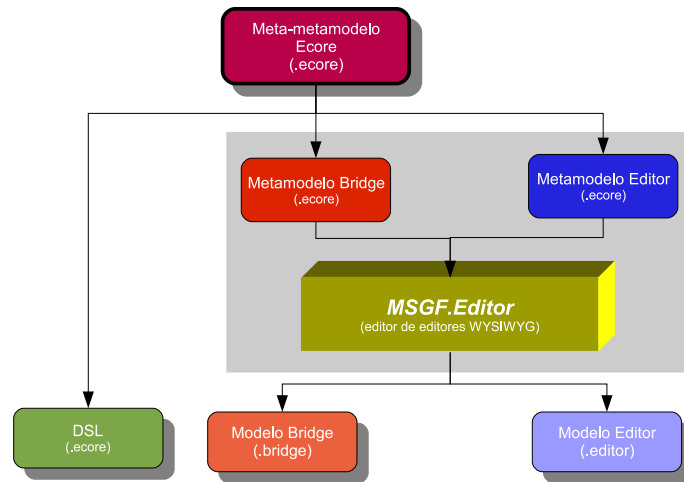


Figura 4: Esquema del papel que juega *MSGF.Editor* dentro del framework MSGF.

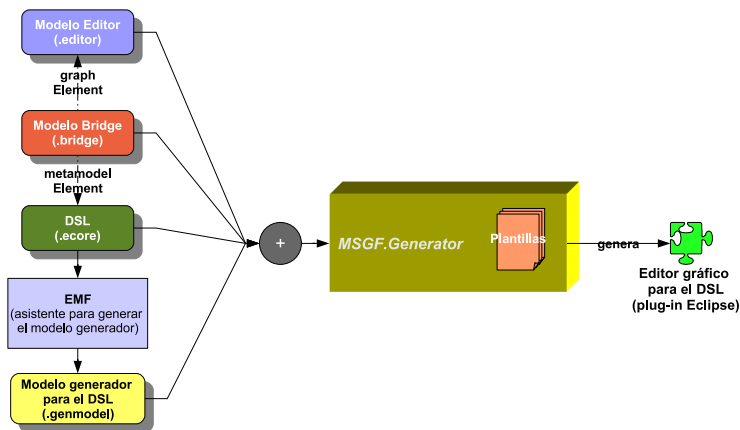


Figura 5: Esquema donde se representan las entradas y salidas del módulo *MSGF.Generator*.

(Model-View-Controller) [1]. Para componer la vista, el controlador toma parte de la información del modelo *Diagram* y otra parte de los modelos en edición (*Editor* y *Bridge*). Cuando se realiza una acción sobre la vista, el controlador modificará el modelo *Diagram* (si por ejemplo se trata de una acción de arrastrar y soltar o redimensionar) o el modelo editado (si por ejemplo lo que hacemos es crear/eliminar un elemento), según corresponda.

5. Generador de código

El módulo que se encarga de realizar la generación del código (transformación de modelo a texto) de los editores se denomina **MSGF.Generator**.

Como vemos en la figura 5, este módulo genera el código Java del editor gráfico a partir de los modelos *Editor* y *Bridge*, del DSL y del modelo *GenModel* generado a partir de dicho DSL (este último modelo se describe más adelante). Este código se empaqueta como

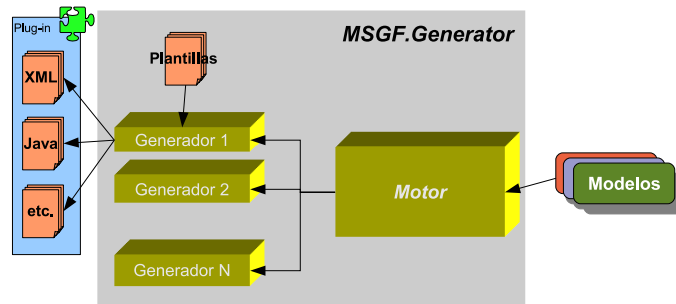


Figura 6: Motor para la generación de código que armoniza las plantillas con los modelos para producir recursos (clases Java, ficheros XML, directorios, etc.).

un proyecto de conector o plug-in para Eclipse que representa el editor generado.

Destacar que, al igual que el editor de editores *MSGF.Editor* (que es un editor gráfico), los editores generados se construyen aprovechando las facilidades que ofrece GEF para la creación de editores gráficos sobre Eclipse. Es por ello que el metamodelo *Diagram* también es usado por los editores generados, al objeto de enriquecer las instancias del DSL en edición con información sólo necesaria para la representación gráfica.

Para la generación de código, *MSGF.Generator* incorpora un motor compuesto por un conjunto de generadores cuya función es producir los distintos recursos o partes que conforman el editor (ver figura 6). Estos generadores se encargan de producir el código de la parte de los controladores y de las vistas (MVC). En cambio, el código para manipular el modelo es generado por EMF a partir del modelo generador o *GenModel* del DSL. Este modelo consiste en un decorador cuya finalidad es añadir una serie de propiedades o información extra al DSL para facilitar la generación del código del modelo.

Los generadores incluidos en el motor reciben como entrada los modelos y las plantillas generadas con JET (Java Emitter Templates) [12], y producen recursos (ficheros y directorios). En nuestro caso, estos recursos son clases Java, iconos para la paleta de herramientas, ficheros XML de configuración del plug-in, etc.

6. Trabajo relacionado

Otra aproximación a la problemática planteada en este artículo es el proyecto de Eclipse denominado GMF (Graphical Modeling Framework) [15], que surge con el objetivo de generar automáticamente editores gráficos como plug-ins para Eclipse a partir de modelos, aprovechando la infraestructura ofrecida por EMF y GEF.

El núcleo de GMF es el modelo de definición gráfica (Graphical Definition Model), que almacena información de los elementos gráficos que aparecen en el editor generado en tiempo de ejecución. Para la definición de la paleta de herramientas y otros elementos de la interfaz del editor se emplea un segundo modelo (Tooling Definition Model). Estos dos modelos no tienen una vinculación directa con el DSL destino, sino que la relación entre ambas partes se establece a través de un tercer modelo, conocido como modelo

de definición de mapeos (Mapping Definition Model). Finalmente, GMF dispone de un modelo generador (Generator Model) que permite definir los detalles de la implementación en la fase de generación del código Java del plug-in del editor para el DSL.

Entre MSGF y GMF, a pesar de que son proyectos independientes, surgiendo el primero a principios de 2005 y el segundo a finales del mismo año, existen las siguientes similitudes: (1) los editores generados son plug-ins para Eclipse y se basan en EMF y GEF, (2) la generación de editores se realiza a partir de modelos, (3) existe un modelo que sirve de vínculo o puente entre el DSL y el modelo del editor (modelo de definición de mapeos en GMF y modelo *Bridge* en MSGF), y (4) los DSLs son instancias de Ecore. En lo que se refiere a las diferencias más destacadas encontramos que (1) GMF carece de un editor gráfico que permita crear y manipular visualmente los modelos que se emplearán para generar el editor, cuando MSGF dispone de *MSGF.Editor*; (2) los editores generados con GMF requieren, a parte de EMF y GEF, librerías extra en tiempo de ejecución, lo que facilita la generación de código pero aumenta las dependencias hacia otros plug-ins, a diferencia de MSGF, que genera todo el código necesario para la ejecución de los editores, con la única dependencia de EMF y GEF; y (3) en GMF se necesita especificar los elementos de la interfaz del editor mediante el *Tooling Definition Model*, cuando en MSGF esa información se extrae de los modelos *Editor* y *Bridge*.

7. Conclusiones

Como consecuencia de la importancia que ha ido adquiriendo el empleo de DSLs como lenguajes de modelado en el DSDM, surge la necesidad de un nuevo tipo de herramientas o facilidades, al objeto de dar soporte a la manipulación de modelos para estos DSLs. En este artículo hemos propuesto la solución MSGF, que explota las características comunes a los editores basados en grafos para DSLs. Mediante este framework generamos el 100 % del código de los editores a partir de modelos. Nuestra solución puede ser entendida como una línea de productos de software [2], donde los productos a generar son editores. Como cualquier línea de producto, este framework nos ayuda a desarrollar productos, en este caso editores gráficos, empleando menos tiempo y cometiendo menos errores.

Otra aproximación, denominada GMF, ha seguido un camino similar a MSGF. Consideramos GMF como un framework poco práctico debido a la complejidad que supone modelar los editores, aunque los editores gráficos que genera son muy usables y completos en cuanto a funcionalidad. MSGF, en cambio, sí proporciona medios productivos para la definición de los editores, a través del editor de editores WYSIWYG.

8. Agradecimientos

Este trabajo ha contado con la participación del *Ministerio de Educación y Ciencia* (PTQ2004-1495 y PTR1995-0928-OP) y del *Fondo Social Europeo*. Y ha sido parcialmente financiado por la *DG de Universidades e Investigación del Gobierno de Canarias* (PI042005/007).

REFERENCIAS

- [1] Sam A. Adams. MetaMethods: The MVC Paradigm. *HOOPLA!*, Volumen 1 Número 4, Jul 1988.
- [2] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, Ago 2001.
- [3] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and FragmentBased Component Models*. PhD thesis, Department of Computer Science and Automation, October 1998.
- [4] J.M. Favre. Foundations of meta-pyramids: Languages and metamodels - episode ii: Story of thotis the baboon. *postproceedings of Dagstuhl Seminar on Model Driven Approaches for Language Engineering*.
- [5] Gentleware. Poseidon for UML. <http://www.gentleware.com/>.
- [6] Esther Guerra and Juan de Lara. Event-driven grammars: Towards the integration of meta-modelling and graph transformation. In *ICGT 2004: Proceedings of the 2nd International Conference on Graph Transformation*, volume 3256 of *LNCS*. Springer-Verlag, 2004.
- [7] IBM. Rational Software Architect. <http://www.rational.com/>.
- [8] OMG. UML (Unified Modeling Language). <http://www.uml.org/>.
- [9] OMG. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, Jun 2003.
- [10] OMG. MOF QVT Final Adopted Specification. ptc/05-11-01, Nov 2005.
- [11] OMG. MOF Core Specification 2.0. formal/06-01-01, Jan 2006.
- [12] Adrian Powell. Model with the Eclipse Modeling Framework, Part 2 (Generate code with Eclipse's Java Emitter Templates). IBM white paper available from <http://www-123.ibm.com/developerworks/library/os-ecmf2/>, Apr 2004.
- [13] The Eclipse Foundation. EMF. <http://www.eclipse.com/emf/>.
- [14] The Eclipse Foundation. GEF. <http://www.eclipse.com/gef/>.
- [15] The Eclipse Foundation. GMF. <http://www.eclipse.com/gmf/>.
- [16] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.