Desarrollo de Software Dirigido por Modelos: conceptos, lenguajes y desafíos

Ivanna M. Lazarte[1]
(1) Departamento de Formación Básica, Facultad de Tecnología y Ciencias Aplicadas, UNCa.
ilazarte @tecno.unca.edu.ar

Fecha de recepción del trabajo: 10/12/2015
Fecha de aceptación del trabajo: 16/06/2016

RESUMEN: El Desarrollo de Software Dirigido por Modelos (MDSD, *Model-Driven Software Development*) es un nuevo paradigma para el desarrollo de software, en el cual los modelos son los principales artefactos en el proceso de desarrollo. La esencia de MDSD se basa en dos temas principales: (1) *elevar el nivel de abstracción* de las especificaciones para estar más cerca del dominio del problema y lejos del dominio de implementación mediante el uso de lenguajes de modelado específicos del dominio, (2) *elevar el nivel de automatización* usando tecnología informática para cerrar la brecha semántica entre la especificación (el modelo) y la implementación (el código generado). MDSD promete incrementar la productividad del desarrollador, reducir el costo (en tiempo y dinero) de la construcción de software, mejorar la reusabilidad del software, y hacer software más mantenible. En el presente trabajo se presentan los pilares básicos sobre los que se apoya MDSD que son: los modelos, los metamodelos y las transformaciones entre modelos. Además, se describen brevemente los desafíos en la adopción de MDSD en la industria.

PALABRAS CLAVES: Desarrollo de Software Dirigido por Modelos, Modelos, Metamodelos, Transformación de Modelos

MODEL-DRIVEN SOFTWARE DEVELOPMENT: CONCEPTS, LANGUAGES AND CHALLENGES

ABSTRACT: Model-Driven Software Development is a new paradigm for software development, in which the models are the primary artifacts in the development process. The essence of MDSD is based on two key topics: (1) *raising the level of abstraction* of specifications to be closer to the problem domain and further away from the implementation domain by using domain specific modeling languages, (2) *raising the level of automation* by using computer technology to fill in the semantic gap between the specification (the model) and the implementation (the generated code). MDSD promises to increase the developer productivity, to reduce the cost (in time and money) of building software, to improve the software usability, and to build more maintainable software. In this paper, the key pillars on which relies MDSD are presented: models, metamodels and model transformations. Also, the challenges in adopting MDSD in the industry are briefly described.

KEYWORDS: Model-Driven Software Development, Models, Metamodels, Model Transformations

1 INTRODUCCIÓN

El desarrollo de software es una tarea compleja y difícil que requiere la inversión de importantes recursos y conlleva mayor riesgo de fracaso. El Desarrollo de Software Dirigido por Modelos (MDSD, *Model-Driven Software Development*), surge como una manera de mejorar la forma en que se construye software.

MDSD es un nuevo paradigma para el desarrollo de software, en el cual los modelos son los principales artefactos en el proceso de desarrollo. La ventaja principal de este paradigma es que los modelos se expresan usando conceptos mucho más cercanos al dominio del problema, elevando el nivel de abstracción de los mismos, en lugar de usar conceptos ligados a la tecnología de implementación. Esto hace que los modelos sean más fáciles de especificar, entender y mantener; y sean menos sensibles a la tecnología de implementación elegida y a los cambios en dicha tecnología (Selic, 2003).

Otra característica importante de MDSD es que el código es generado (semi-) automáticamente (mediante transformaciones de modelos) a partir de sus correspondientes modelos (Selic, 2003). De esta manera, los modelos no sólo son utilizados para documentar el sistema sino también para construir el producto final. Esto es diferente al enfoque tradicional de desarrollo de software en donde los modelos son meramente utilizados para propósitos de documentación.

Si bien MDSD no es una panacea, ayuda a incrementar la productividad del desarrollador, reducir el costo (en tiempo y dinero) de la construcción de software, mejorar la reusabilidad del software, y hacer software más mantenible. Además, MDSD también ayuda en la detección temprana de defectos tales como defectos de diseño, omisiones, y malentendidos entre clientes y desarrolladores (Liddle, 2011).

El resto de trabajo se organiza de la siguiente manera: en la Sección 2 se describen los distintos enfoques

dirigidos por modelos y beneficios de MDSD. En la Sección 3 se describen los pilares en los que se basa MDSD: los modelos, los metamodelos y las transformaciones entre modelos. En la Sección 4 se describen los lenguajes de modelado más utilizados. En la Sección 5 se describen los lenguajes de transformación de modelos, tanto transformaciones modelo-a-modelo como transformaciones modelo-a-texto. En la Sección 6 se presentan los principales desafíos en la adopción de MDSD. Finalmente, en la Sección 7 se presentan las conclusiones.

2 DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS

Básicamente, MDSD se refiere al uso de lenguajes específicos de dominio para crear modelos que expresan la estructura o comportamiento del sistema usando conceptos cercanos al dominio del problema. Estos modelos se transforman subsecuentemente en código ejecutable mediante una serie de transformaciones de modelos (Völter, 2013).
En contraste al proceso de desarrollo de software tradicional, las transformaciones de modelos en MDSD se ejecutan siempre mediante herramientas.

2.1 *Enfoques dirigidos por modelos*

A continuación se describen brevemente los diferentes enfoques del universo dirigido por modelos (*model-driven*, en inglés) (ver Mura 1) (Brambilla, 2012).

- *Model-Driven Software Development (MDSD):* es un paradigma de desarrollo que usa modelos como artefactos primarios del proceso de desarrollo. Este enfoque, se centra principalmente en las fases de elicitación de requerimientos, análisis, diseño e implementación. Generalmente, la implementación se genera (semi-) automáticamente a partir de los modelos.
- *Model-Driven Architecture (MDA):* es la visión particular de MDSD propuesta por la OMG (2003), que se basa en el uso de los estándares de OMG. Por consiguiente, MDA puede considerarse como un subconjunto de MDSD, donde los lenguajes de modelado y de transformaciones son estandarizados por la OMG.
- *Model-Driven Engineering (MDE):* es un superconjunto de MDSD, ya que va más allá de las actividades de desarrollo y abarca otras actividades basadas en modelos del proceso de Ingeniería de Software (por ejemplo, evolución basada en modelos del sistema o ingeniería revesa dirigida por modelos de un sistema heredado) (da Silva, 2015).
- *Model-Based Development (MBE):* es una versión más suave de MDE, donde los modelos de software juegan un papel importante pero no son los artefactos primarios del desarrollo, es decir, no dirigen el proceso de desarrollo. Por ejemplo, un proceso de desarrollo donde en la fase de análisis los diseñadores especifican los modelos del dominio del sistema pero luego esos modelos son entregados directamente a los programadores que los usan como borradores para escribir manualmente el código (no hay generación de código automática ni una definición explícita de un modelo específico de la plataforma).

En la literatura pueden encontrarse variantes de estos acrónimos, tales como *Model-Driven Software Engineering* (MDSE), *Model-Driven Development (MDD),* etc.

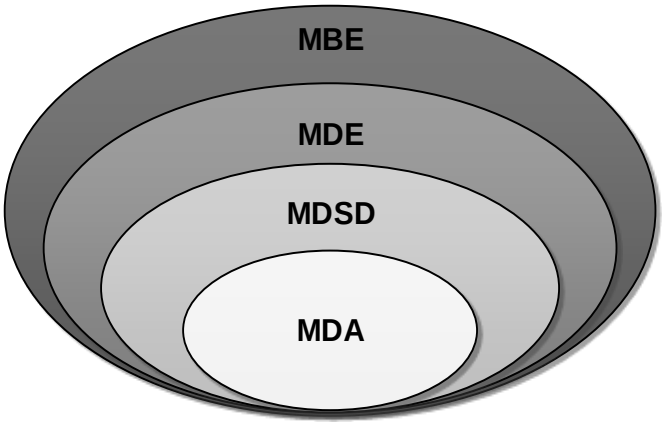

Figura 1. Relación entre los diferentes enfoques dirigidos por modelos

2.2 *Beneficios de MDSD*

MDSD tiene el potencial de mejorar en gran medida las prácticas de desarrollo de software convencionales. A continuación se listan los principales beneficios de aplicar MDSD (Swithinbank, 2005; Pons, 2010).

- *Incrementa la productividad:* MDSD reduce el costo de desarrollo de software, mediante la generación de código y otros artefactos a partir de modelos, lo cual incrementa la productividad del desarrollador. Si bien las transformaciones de modelos a código/artefactos tiene costo, este se puede amortizar mediante elmo reúso de dichas transformaciones.
- *Mantenibilidad:* el progreso tecnológico conduce a que los componentes de la solución queden atados a tecnologías de plataformas anteriores. MDSD ayuda a resolver este problema conduciendo a una arquitectura mantenible, donde los cambios se hacen rápida y consistentemente, permitiendo una migración más eficiente de los componentes hacia nuevas tecnologías.
  Los modelos de alto nivel se mantienen libres de detalles de implementación, lo cual facilita la adaptación a los cambios en la plataforma subyacente o la arquitectura de implementación. Los cambios en la arquitectura de implementación

se hacen actualizando la transformación, la cual se reaplica a los modelos originales para producir los nuevos artefactos de implementación.

- *Reúso de sistemas legados*: si existen sistemas legados en la organización, se pueden transformar los componentes de dichos sistemas a modelos UML, para luego usar esos modelos en (1) la migración hacia nuevas plataformas/tecnologías o (2) la generación de *wrappers* que permitan que los componentes legados puedan ser accedidos mediante tecnologías de integración, como los Servicios Web.
- *Adaptabilidad:* cuando se usa un enfoque MDSD, agregar o modificar una función de negocio es bastante sencilla, ya que la inversión en la automatización ya está hecha. Para agregar una nueva función, sólo se debe desarrollar el comportamiento específico para esa funcionalidad. El resto de la información necesaria para generar los artefactos de implementación ya está capturada en la transformación y puede reusarse.
- *Consistencia:* la aplicación manual de prácticas de codificación y diseño arquitectónico es una actividad propensa a errores. MDSD asegura que los artefactos se generan consistentemente.
- *Repetitividad:* MDSD es especialmente poderoso cuando se aplica a nivel de programa u organización. Este se debe a que el retorno de inversión de desarrollar transformaciones se incrementa cada vez que éstas son reusadas. El uso de transformaciones probadas además incrementa la confianza de desarrollar nuevas funciones y reduce el riesgo ya que los problemas técnicos y arquitecturales ya fueron resueltos.
- *Mejora la comunicación con los stakeholders:* los modelos de alto nivel omiten detalles de implementación que no son relevantes para entender el comportamiento lógico de un sistema. Por consiguiente, los modelos están más cercanos al dominio del problema, reduciendo la brecha semántica entre los conceptos que entienden los *stakeholders* y el lenguaje en el cual se expresa la solución. Esto facilita la entrega de soluciones de software que están mejores alineadas a los objetivos de negocio.
- *Mejora la comunicación de diseño:* los modelos facilitan el entendimiento y razonamiento sobre el sistema a nivel de diseño. Esto conduce a mejorar la toma de decisiones y la comunicación sobre un sistema. Además, el hecho de que los modelos son parte de la definición del sistema y no sólo documentación, hace que los modelos siempre permanezcan actualizados y confiables.
- *Captura experiencia:* los proyectos u organizaciones a menudo dependen de expertos claves quienes toman las decisiones prácticas. Al capturar su experiencia en los modelos y en las transformaciones, otros miembros del equipo pueden aprovecharla sin requerir su presencia. Además este conocimiento se mantiene aun cuando los expertos dejen la organización.
- *Modelos como activos a largo plazo:* en MDSD, los modelos son activos que capturan lo que hacen los sistemas IT de la organización. Los modelos de alto nivel son resistentes a los cambios a nivel plataforma y sólo sufren cambios cuando cambian los requisitos del negocio.
- *Posibilidad de retrasar las decisiones tecnológicas:* cuando se usa un enfoque MDSD, las etapas tempranas de desarrollo se enfocan en actividades de modelado. Esto significa que es posible retrasar la elección de una plataforma específica o versión de un producto hasta que se disponga de mayor información que permita realizar una elección más adecuada. En dominios con ciclos de desarrollo extremadamente largos, como sistemas de control de tráfico aéreo, esto es crucial, ya que la plataforma elegida puede incluso no existir cuando comience el desarrollo.

3 CONCEPTOS BÁSICOS

Los conceptos claves de MDSD son los modelos, meta-modelos y transformaciones de modelos. En las siguientes secciones se describen cada uno de estos conceptos.

3.1 *Modelos*

Un *modelo* es una representación de una parte de la funcionalidad, estructura y/o comportamiento de un sistema (OMG, 2003). Se construye desde un determinado punto de vista, expresado en un lenguaje bien definido y con un propósito determinado (Muñoz, 2007); con el fin de analizar la naturaleza del sistema que representa, desarrollar o comprobar hipótesis o supuestos y permitir una mejor comprensión del fenómeno real (Pons, 2010).

3.1.1 *Características de los modelos*

Para que un modelo sea útil y eficaz, debe poseer las siguientes características (Selic, 2003, 2006):

- *Abstracto:* debe ocultar o remover los detalles irrelevantes con el fin de destacar los esenciales.
- *Comprensible:* expresados en un lenguaje fácilmente entendible para sus usuarios.
- *Preciso:* debe reflejar correctamente las propiedades de interés del sistema modelado.
- *Predictivo:* debe ser capaz de predecir con exactitud el comportamiento y otras propiedades del sistema modelado.

- *Económico:* debe ser significativamente más barato de construir y analizar que el sistema mismo.

3.1.2 *Principales funciones de los modelos*

En el ámbito de la Ingeniería de Software, los modelos deben cumplir con las siguientes funciones principales (Selic, 2003, 2006):

- Comprender el sistema.
  - La estructura, el comportamiento y cualquier otra característica relevante de un sistema y su entorno desde un punto de vista dado.
  - Separar adecuadamente cada uno de los aspectos, describiéndolos al nivel conceptual adecuado.
- Servir de mecanismo de comunicación.
  - Entre los distintos tipos de *stakeholder* del sistema (desarrolladores, usuarios finales, personal de soporte y mantenimiento, etc.).
  - Con las otras organizaciones (proveedores y clientes que necesitan comprender el sistema a la hora de interoperar con él).
- Validar el sistema y su diseño.
  - Detectar errores, omisiones y anomalías en el diseño tan pronto como sea posible.
  - Razonar sobre el sistema, infiriendo propiedades sobre su comportamiento.
  - Poder realizar análisis formales sobre el sistema.
- Guiar la implementación.
  - Servir como "planos" para construir el sistema y que permitan guiar su implementación de una forma precisa y sin ambigüedades.
  - Generar, de la forma más automática posible, tanto el código final como todos los artefactos necesarios para implementar, configurar y desplegar el sistema.

3.1.3 *Clasificación de modelos*

Los modelos se pueden clasificar como (Kleppe, 2003; Pons, 2010):

- *Modelos de negocio y modelos de software:* un modelo de negocio describe a un negocio o empresa (o parte de ellos). El lenguaje utilizado para modelarlos contiene un vocabulario que permite al modelador especificar los procesos del negocio, los clientes, los departamentos, las dependencias entre procesos, etc. Un modelo de software describe un sistema de software. Generalmente, los requisitos del sistema de software se derivan del modelo de negocio al cual el software brinda soporte.
- *Modelos estáticos y modelos dinámicos:* los modelos estáticos (estructurales) definen el conjunto de objetos que constituyen el sistema, sus propiedades y sus conexiones. Los modelos dinámicos definen el comportamiento que los objetos del sistema despliegan. Ambos tipos de modelos están fuertemente interrelacionados y juntos constituyen el modelo global del sistema.
- *Modelos independientes de la plataforma y modelos específicos de la plataforma:* El estándar MDA (OMG, 2003) clasifica los modelos dependiendo de si contienen o no conceptos técnicos relacionados a su implementación.
  - *Modelo independiente de la computación (CIM):* es una vista del sistema independiente de la computación que describe los requerimientos del mismo y el contexto de negocio en el cual será utilizado, sin mostrar detalles de la estructura del sistema. Utiliza un vocabulario que es familiar para los especialistas del dominio y también es conocido como modelo de dominio.
  - *Modelo independiente de la plataforma (PIM):* es una vista del sistema independiente de la plataforma tecnológica. Representa la lógica del negocio y su funcionalidad, independientemente de los detalles de la implementación. En este modelo se puede observar los aspectos que no cambiarán de una plataforma a otra, con el fin de permitir su mapeo a una o más plataformas tecnológicas.
  - *Modelo Especifico de la Plataforma (PSM):* es una vista del sistema desde la perspectiva de la plataforma tecnológica específica. Combina las especificaciones del PIM con los detalles y características propias del uso de dicha plataforma. En este modelo se puede observar la manera en la cual un sistema usa la plataforma para el cumplimiento de los objetivos trazados en el CIM.
  - *Modelo de la implementación:* es el código fuente, que implementa el sistema y satisface los requerimientos. Se obtiene en el último paso del proceso MDSD.

3.2 *Metamodelos*

Un modelo se compone de elementos de modelo y se ajusta a un metamodelo. Esto significa que el metamodelo describe los diversos tipos de elementos de modelos contenidos y la forma en que se organizan, relacionan y restringen (Bézivin, 2005).

Un *metamodelo* es un modelo de un lenguaje de modelado. Define la estructura (sintaxis), semántica, y restricciones para una familia de modelos (Mellor, 2004). Por ejemplo, el metamodelo de UML es un modelo que contiene los elementos para describir modelos UML, como *Package*, *Classifier*, *Class*, *Operation*, *Association*, etc. El metamodelo de UML también define las relaciones entre estos elementos, así como las restricciones de integridad de los modelos UML (Muñoz, 2007).

Por consiguiente, cada modelo se escribe en el lenguaje que define su metamodelo (su lenguaje de modelado). La relación entre un modelo y su metamodelo es de conformidad y se dice que un modelo es *conforme a* un metamodelo. Como analogía, se puede decir que un modelo es conforme a su metamodelo en la misma forma que un programa es conforme a la gramática del lenguaje de programación en el que está escrito (Bambrilla, 2012). Debido a que los metamodelos son a su vez modelos, están escritos en el lenguaje definido por su meta-metamodelo, los cuales son conformes a sí mismos, según la propuesta de la OMG, denominada *arquitectura de metamodelado* (OMG, 2015) (ver Figura 2). La OMG propone a MOF (*Meta Object Facility*) (OMG, 2015) como el lenguaje estándar de la capa M3.

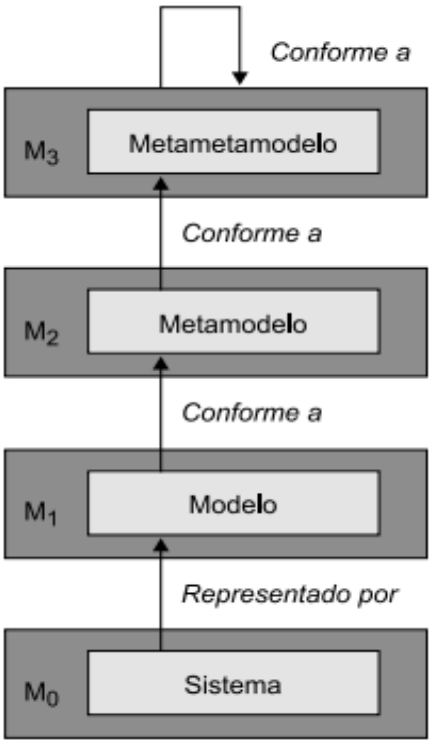

Figura 2. Arquitectura de metamodelado propuesta por la OMG.

Los metamodelos tienen diferentes usos, que pueden resumirse en:

- Definen la sintaxis y semántica de un lenguaje.
- Explican el lenguaje.
- Comparan lenguajes rigurosamente.
- Especifican los requerimientos para una herramienta del lenguaje.
- Especifican un lenguaje a ser usado en una meta herramienta.
- Permiten el intercambio entre herramientas.
- Permiten el mapeo entre modelos.

Como se verá en la Sección 3.3, la importancia de los metamodelos en MDSD radica en que se necesita de un mecanismo para definir inequívocamente lenguajes de modelado, a fin de que un motor de transformación pueda leer, escribir y entender los modelos. Además, las reglas de transformación usan los metamodelos de origen y destino para definir la transformación (Kleppe, 2003).

3.3 *Transformaciones de modelos*

Una *transformación de modelos* es el proceso de convertir un modelo de un sistema en otro modelo del mismo sistema. Mediante una serie de transformaciones, se reduce el nivel de abstracción de los modelos con el propósito de producir un modelo con suficientes detalles que permita la generación (semi-) automática de código ejecutable (Jouault, 2008).

Estas transformaciones tienen varias aplicaciones (Czarnecki, 2006):

- Generación de modelos a más bajo nivel, y eventualmente código, a partir de modelos a alto nivel.
- Mapeo y sincronización entre modelos del mismo o diferentes niveles de abstracción.
- Creación de vistas basadas en consultas (*queries*) sobre un sistema.
- Tareas de evolución de modelos, tales como refactorización de modelos.
- Aplicación de ingeniería inversa para obtener modelos de más alto nivel a partir de modelos de más bajo nivel.

En la transformación de modelos, se considera como potenciales sujetos de transformación a una amplia gama de artefactos de desarrollo de software, tales como modelos UML, especificaciones de interfaz, esquemas de datos, descriptores de componentes, el código del programa, entre otros (Czarnecki, 2006).

3.3.1 *Clasificación de transformaciones*

Las transformaciones de modelos más comunes son (Brown, 2005):

- *Transformación de tipo refactorización:* reorganiza un modelo basado en un criterio bien definido. La salida de esta transformación es una revisión del modelo original, denominado *modelo refactorizado*.
- *Transformación modelo-a-modelo:* generan modelos a partir de otros modelos.
- *Transformación modelo-a-texto:* generan cadenas de texto a partir de modelos. Se usan, por ejemplo, para generar código y documentos. Este tipo de transformación también es conocido en la literatura como *modelo-a-código*.

Dependiendo de los metamodelos origen y destino, las transformaciones se pueden clasificar como:

- *Transformaciones exógenas:* los metamodelos origen y destino son distintos. Es el caso más común.

- *Transformaciones endógenas:* los metamodelos origen y destino son el mismo, se trata de las llamadas transformaciones *in-place*.

Una transformación de modelos puede realizarse en forma *vertical*, en el cual la transformación se realiza en diferentes niveles de abstracción (por ejemplo, PIM a PSM), u *horizontal*, en el cual la transformación se realiza en el mismo nivel de abstracción (por ejemplo, PIM a PIM).

3.3.2 *Patrón de transformación de modelos*

En MDSD las transformaciones de modelos siguen un patrón común conocido como *patrón de transformación de modelos* (Jouault, 2008).

El patrón de transformación (Ver Figura 3) consta de una definición de transformación (DT) ejecutado por un motor de transformación (MT) para generar el modelo de destino/salida (Mb) a partir del modelo de origen/entrada (Ma).

Una *definición de transformación* (DT) está compuesta por un conjunto de reglas de transformación que describen cómo un modelo de origen Ma (definido con un lenguaje origen) se transforma en un modelo de destino Mb (definido con un lenguaje destino). Una *regla de transformación* es una descripción de cómo uno o más constructores del lenguaje origen (patrón origen) se transforma en uno o más constructores del lenguaje destino (patrón destino). El *motor de transformación* (MT) ejecuta la definición de transformación para generar automáticamente el modelo de salida a partir del modelo de entrada (Kleppe, 2003).

En general, una transformación puede tener varios modelos origen (Ma) y destino (Mb), siendo posible además que el metamodelo origen (MMa) y destino (MMb) sean el mismo (Czarnecki, 2006).

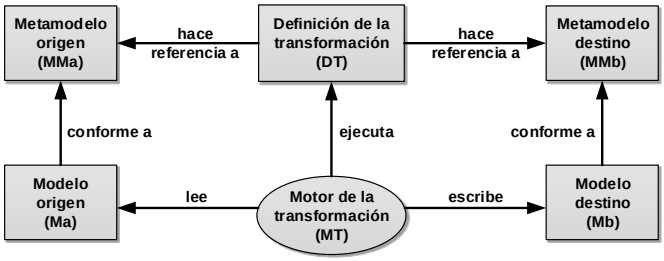

Figura 3. Patrón de transformación de modelos.

4 LENGUAJES DE MODELADO

Debido a que los modelos son los artefactos principales de MDSD, se requiere que estén escritos en un lenguaje bien definido. Para ello, los lenguajes usados deben tener tres elementos fundamentales (Brambilla, 2012; Muñoz, 2007):

- *Sintaxis abstracta:* describe la estructura del lenguaje y la forma en que las primitivas pueden combinarse. Es decir, describe el vocabulario con los conceptos del lenguaje, las relaciones entre ellos, y las reglas que permiten construir las sentencias válidas del lenguaje. La sintaxis abstracta se define mediante metamodelos.
- *Sintaxis concreta:* describe la representación explícita del lenguaje de modelado. Es decir, define la notación que se usa para representar los modelos que pueden describirse con ese lenguaje. Puede ser gráfica o textual.
- *Semántica:* describe el significado de los elementos definidos en el lenguaje y el significado de las distintas formas de combinarlos.

Como puede verse en la Figura 4, la sintaxis abstracta especifica cuáles son los modelos válidos, la sintaxis concreta permite representar dichos modelos y la semántica les asigna un significado preciso y no ambiguo. Además, una misma sintaxis abstracta puede tener asociada más de una sintaxis concreta, como por ejemplo una textual y otra gráfica.

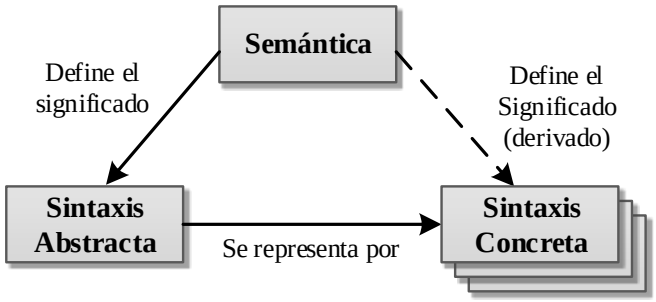

Figura 4. Elementos fundamentales de un lenguaje.

Los lenguajes de modelado pueden clasificarse en lenguajes de propósito general y específicos de dominio.

- *Lenguajes de propósito general (GPLs, General-Purpose Languages):* tienen notaciones que pueden aplicarse en cualquier dominio o sector. Por ejemplo, el lenguaje UML.
- *Lenguajes específicos del dominio (DSLs, Domain-Specific Languages)*: tienen notaciones específicas de un dominio, contexto o compañía para facilitar el modelado del dominio. Permiten a los expertos del dominio expresar el modelo de un sistema usando el vocabulario que normalmente utilizan, y de forma independiente de las plataformas de implementación (Mernik, 2005). Por ejemplo, el lenguaje BPMN.

En MDSD se apunta al uso de DSLs para elevar el nivel de abstracción de los modelos, usando conceptos más cercanos al dominio del problema.

Para que un DSL sea considerado útil debe seguir los siguientes principios (Brambilla, 2012):

- Debe proveer buenas abstracciones para el modelador, debe ser intuitivo y hacer el modelado más fácil.
- No debe depender de un único experto para su adopción y uso.
- Debe evolucionar y mantenerse actualizado basado en el contexto y las necesidades del usuario.
- Debe tener herramientas y métodos que faciliten su uso.

La creación de un nuevo DLS es una tarea que consume tiempo, requiere experiencia y generalmente es realizada por ingenieros especialistas en lenguajes. Hoy en día, está aumentando fuertemente la necesidad de nuevos DSL para diferentes dominios en crecimiento. En Karsai (2009) se pueden encontrar las directrices que ayudan a desarrollar DSLs de mejor calidad en el diseño del lenguaje y una mejor aceptación entre sus usuarios.

5 LENGUAJES DE TRANSFORMACIÓN DE MODELOS

Como parte del proceso de MDSD, los modelos son *fusionados* (para homogenizar diferentes versiones de un sistema), *alineados* (para crear una representación global del sistema desde diferentes vistas), *refactorizados* (para mejorar su estructura interna sin cambiar su comportamiento), *refinados* (para detallar modelos de alto nivel) y *traducidos* (a otros lenguajes/representaciones, por ejemplo, como parte de la generación de código). Todas estas operaciones son implementadas como transformaciones de modelos, ya sea como *Modelo-a-Modelo* o *Modelo-a-Texto* (Brambilla, 2012).

5.1 *Transformaciones Modelo-a-Modelo (M2M)*

Las transformaciones M2M proporcionan los mecanismos que permiten especificar el modo de producir modelos de salida a partir de modelos de entrada. Este tipo de transformaciones, se usan para modificar, crear, fusionar, entrelazar o filtrar modelos. En lugar de crear artefactos desde cero, las transformaciones M2M permiten el reúso de la información capturada en el modelo y construir a partir de éste.

En las transformaciones M2M, existen varios tipos de enfoques: manipulación directa, operacional, relacional, basada en grafos e híbridos (Czarnecki, 2006).

A continuación se describen los lenguajes de transformaciones M2M más utilizados en cada categoría.

5.1.1 *Enfoque de manipulación directa*

Este enfoque ofrece una representación interna de los modelos y una API para manipularla. Por lo general se implementa como un *framework* orientado a objetos, el cual puede proveer alguna infraestructura mínima para organizar las transformaciones (por ejemplo, la clase abstracta para transformaciones) (Czarnecki, 2006). Este enfoque es demasiado complejo debido a que se deben implementar las reglas de transformación desde cero, en un lenguaje de programación de propósito general, como Java.

5.1.1.1 SiTra

SiTra (Akehurst, 2006) es un *framework* que se enfoca en el desarrollo de transformaciones simples en Java. Permite introducir el concepto de reglas de transformación a los programadores que aún no están familiarizados con MDSD.

5.1.2 *Enfoque operacional (imperativo)*

Este enfoque es similar al enfoque de manipulación directa (Sección 5.1.1), pero ofrece soporte dedicado a transformación de modelos (Czarnecki, 2006). Los constructores y conceptos son similares a los lenguajes de programación de propósito general (Java, C/C++) y las transformaciones se describen como una secuencia de acciones, lo cual le da mayor flexibilidad al programador.

Una solución típica para aplicar este enfoque es extender la sintaxis de un lenguaje de modelado mediante el agregado de facilidades para expresar las transformaciones (Czarnecki, 2006), por ejemplo: OCL (*Object Constraint Language*) (OMG, 2012) con constructores imperativos. Un ejemplo de este enfoque es el lenguaje *Operational Mappings* de QVT (Sección 5.1.5.1)

5.1.2.1 Kermeta (Kernel Metamodeling)

Kermeta (Drey, 2009) es un DSL diseñado para permitir tanto la definición de metamodelos y modelos como la definición de consultas, vistas, transformaciones y acciones sobre los modelos

Kermeta es un lenguaje imperativo con soporte para las estructuras de control tradicionales. Es orientado a objetos con soporte para paquetes, clases, operaciones y métodos, herencia múltiple y *binding* dinámico. Estos rasgos pueden ser utilizados para encapsular transformaciones permitiendo la modularización de las mismas. La composición de transformaciones puede ser realizada por medio de llamadas a operaciones o sobrecarga de métodos.

5.1.3 *Enfoque relacional (declarativo)*

Este enfoque agrupa propuestas donde el concepto principal son las relaciones matemáticas, en el cual la idea básica consiste en especificar las relaciones entre los elementos del modelo origen y el modelo destino, usando restricciones (Czarnecki, 2006). De esta manera, el usuario debe enfocarse en *qué* debe mapearse en

lugar de *cómo* hacerlo. Un ejemplo de este enfoque es el lenguaje *Relations* de QVT (Sección 5.1.5.1)

5.1.3.1 Tefkat

Tefkat (Lawley, 2005) es un lenguaje diseñado específicamente para la transformación de modelos MOF usando patrones y reglas. El lenguaje adopta un paradigma declarativo, en el que los usuarios sólo deben preocuparse de las relaciones entre los modelos en lugar de tener que tratar explícitamente con cuestiones como el orden de ejecución de reglas y búsqueda de patrón/recorrido de los modelos de entrada. La sintaxis concreta de este lenguaje es parecida a SQL, y permite escribir transformaciones reusables y escalables, usando conceptos del dominio de alto nivel.

5.1.4 *Enfoque basado en transformación de grafos*

En este enfoque se aprovechan las técnicas y conceptos definidos en la teoría de grafos para aplicarlos en la transformación de modelos. En particular, este tipo de transformaciones actúa sobre grafos tipados, etiquetados y con atributos, que pueden pensarse como representaciones formales de modelos de clase simplificados (Czarnecki, 2006).

5.1.4.1 VIATRA (VIsual Automated model TRAns-formations)

VIATRA (Varró, 2007) es una herramienta para la transformación de modelos que forma parte del *framework* VIATRA (Eclipse, 2013). Este lenguaje está basado en patrones y reglas para manipular modelos de grafos combinando transformación de grafos y máquinas de estado abstractas. El lenguaje ofrece constructores avanzados para consultas y manipulación de modelos que permiten realizar tareas de verificación, validación y seguridad, así como una temprana evaluación de características no funcionales como fiabilidad, disponibilidad y productividad del sistema bajo diseño.

5.1.5 *Enfoque híbrido*

Este enfoque básicamente combina las distintas técnicas que proveen los enfoques anteriores (Czarnecki, 2006).

5.1.5.1 QVT (Query-View-Transformation)

QVT (OMG, 2011) es un lenguaje estándar propuesto por la OMG, que se compone de tres lenguajes (*Relations*, *Core* y *Operational Mappings*) capaces de expresar transformaciones, vistas y consultas entre modelos.

El lenguaje *Relations* permite especificar transformaciones como un conjunto de relaciones entre modelos. Esto permite al desarrollador crear elementos en el modelo destino a partir de elementos del modelo origen, y también aplicar cambios sobre modelos existentes.

El lenguaje *Core* es un lenguaje más simple que el lenguaje *Relations*, por tanto, las transformaciones escritas en *Core* suelen ser más largas que sus equivalentes especificadas en *Relations*. El lenguaje *Core* proporciona la base para especificar la semántica del lenguaje *Relations*.

El lenguaje *Operational Mappings* extiende al lenguaje *Relations* con constructores imperativos y constructores OCL. El lenguaje es puramente imperativo y similar a los lenguajes procedurales de programación tradicional. Además, incorpora elementos constructivos diseñados para operar (crear, modificar y eliminar) sobre el contenido de los modelos.

5.1.5.2 ATL (ATLAS Transformation Language)

ATL es un DSL que forma parte de la plataforma AMMA (*ATLAS Model Management Architecture*) (Jouault, 2008). ATL está inspirado en QVT y creado sobre el formalismo OCL (OMG, 2012).

ATL es un lenguaje híbrido, es decir, contiene constructores declarativos e imperativos, basado en OCL para definir sus tipos de datos y expresiones declarativas. Se recomienda usar el estilo declarativo ya que se basa en especificar las relaciones entre los patrones de origen y destino y, por lo tanto, tiende a ser más cercano a la forma en que los desarrolladores perciben una transformación. Esto permite ocultar algoritmos de transformación compleja en una simple sintaxis. Sin embargo, a veces es difícil proporcionar una solución declarativa completa para un problema de transformación dado. En ese caso, los desarrolladores pueden recurrir a las características imperativas del lenguaje (Jouault, 2008).

ATL fue desarrollado sobre la plataforma Eclipse (Eclipse, 2008).

ATL es uno de los lenguajes de transformación más usados tanto en el ámbito académico como en la industria debido a la madurez del lenguaje y las herramientas de soporte.

5.1.5.3 ETL (Epsilon Transformation Language)

ETL (Kolovos, 2009) es un lenguaje de transformación híbrido, similar a ATL, creado sobre la plataforma de gestión de modelos Epsilon (Eclipse b, 2005; Kolovos, 2006). Epsilon provee una arquitectura en capas que permite construir lenguajes para la gestión de tareas interoperables específicas tales como transformación de modelos, validación, comparación, fusión y refactorización.

ETL no sólo ofrece todas las características de un lenguaje de transformación estándar sino que también proporciona una mayor flexibilidad, ya que puede transformar muchos modelos de origen a muchos de

destino, y puede consultar/navegar/modificar ambos modelos.

5.1.5.4 RubyTL

RubyTL (Cuadrado, 2006) es un lenguaje híbrido extensible que proporciona expresividad declarativa a través de una construcción de *binding*. Tiene una sintaxis clara y un buen equilibrio entre concisión y verbosidad. Además, su aprendizaje no resulta complicado y requiere conocimientos mínimos de Ruby (Thomas, 2004). Entre las ventajas que proporciona RubyTL se pueden mencionar que el lenguaje puede ser adaptado a una familia particular de problemas de transformación, se pueden añadir nuevas construcciones sin modificar el núcleo básico, y proporciona un entorno para experimentar con las características de los lenguajes de transformación.

5.2 *Transformaciones Modelo-a-Texto (M2T)*

En las transformaciones M2T, la entrada es un modelo y la salida es una cadena de texto. Estas transformaciones están relacionadas a la generación de código para lograr la transición del modelo al código. La transformación M2T no sólo puede generar el código del sistema, sino también otras formas de código como documentación, casos de pruebas, scripts de despliegue, etc. Además, se pueden derivar código basado en *frameworks* formales, los cuales permiten analizar diferentes propiedades de un sistema. Este es uno de los mayores beneficios de usar modelos en el proceso de desarrollo de software, ya que éstos pueden usarse para derivar constructivamente el sistema y también analíticamente para explorar o verificar las propiedades del sistema (Brambilla, 2012).

En las transformaciones M2T, se distinguen dos tipos de enfoques: enfoque basado en el patrón *visitor* y enfoque basado en plantillas (Czarnecki, 2006).

A continuación se describen los lenguajes de transformaciones M2T más utilizados en cada categoría.

5.2.1 *Enfoque basado en el patrón visitor*

Es un enfoque muy simple que consiste en proporcionar algún mecanismo basado en el patrón *visitor* para recorrer la representación interna del modelo y escribir la salida en un archivo de texto (Czarnecki, 2006). Este enfoque requiere que los elementos del modelo origen y destino coincidan uno a uno.

5.2.1.1 Jamda

Jamda (Jamda, 2003) es un *framework* orientado a objetos que proporciona un conjunto de clases para representar modelos UML, una API para manipular modelos y un mecanismo *visitor* para generar código.

5.2.1.2 Stratego

El lenguaje Stratego (Bravenboer, 2008) asume que los programas se pueden representar como árboles sintácticos abstractos. Para esto se emplean *parsers* que los generan a partir de una definición formal de la sintaxis de los mismos. Para definir la transformación se definen reglas para transformar los términos de los árboles. Algunas aplicaciones de transformación de programas con Stratego son la compilación, la generación de documentación, el análisis de código y la ingeniería inversa.

5.2.2 *El enfoque basado en plantillas*

En este enfoque, por lo general, una plantilla consiste en el modelo (texto) destino conteniendo fragmentos de metacódigo, que permiten acceder a la información del modelo origen y llevar a cabo una selección y expansión iterativa de código (Czarnecki, 2006). La estructura de una plantilla se asemeja mucho al código que se quiere generar.

Este enfoque es el más utilizado para las transformaciones M2T.

5.2.2.1 MOFM2T (MOF Model to Text)

MOFM2T (OMG, 2008) es un lenguaje estándar para transformar un modelo a varios artefactos de texto, tales como especificaciones de código, reportes, documentos, etc. Esencialmente, este estándar está pensado para transformar un modelo en una representación textual plana. En MOFM2T una transformación es considerada una plantilla donde el texto que se genera desde los modelos se especifica como un conjunto de bloques de texto parametrizados con elementos de modelos.

5.2.2.2 MOFScript

MOFScript (Oldevik, 2005) es un lenguaje fácil de usar, con pocos constructores, similar a los lenguajes de programación y *script*. La semántica de ejecución de las reglas es secuencial; las reglas son llamadas explícitamente. Permite al usuario definir transformaciones M2T desde instancias de cualquier metamodelo.

MOFScript está implementado como un *plugin* de Eclipse (Eclipse, 2011), usando *Eclipse Modeling Framework* (EMF) (Eclipse, 2003) para el manejo de modelos y meetamodelos. Es compatible con otros estándares de la OMG: QVT (OMG, 2011) y MOF (OMG, 2015).

5.2.2.3 The Epsilon Generation Language (EGL)

EGL (Rose, 2008) es un lenguaje para la generación de código, documentación y otros objetos de texto a partir de modelos. Fue creado en la cima de la plataforma Epsilon (Eclipse b, 2005; Kolovos, 2006), la cual soporta modelos escritos en cualquier lenguaje y puede generar código a partir de estos modelos con EGL. EGL soporta modelos diseñados en EMF, XML, CSV o Bibtex, pero además permite interactuar con otras

tecnologías de modelado que son conformes a la interface de la capa *Epsilon Model Connectivity* (EMC).

5.2.2.4 Acceleo

Acceleo (Eclipse, 2005) es un lenguaje que surgió para proveer una versión pragmática de las transformaciones M2T de modelos EMF. Acceleo provee una poderosa API para soportar OCL así como otras operaciones útiles para trabajar con documentos basados en texto, como por ejemplo funciones avanzadas para manipular *strings*.

6 DESAFÍOS EN LA ADOPCIÓN DE MDSD

MDE, de la cual MDSD forma parte, está ganando cada vez más aceptación en la comunidad de ingeniería de software, sin embargo, su adopción por parte de la industria está lejos de ser un éxito. El número de compañías que aplican MDE es todavía muy limitada (Cuadrado, 2014). Esto se debe a que la adopción de tecnologías MDE en un contexto industrial implica importantes beneficios pero también riesgos sustanciales. Los beneficios en términos de aumento de la productividad, la calidad y la reutilización son fácilmente previsibles. Por otro lado, las preocupaciones más importantes planteadas en MDE son las de escalabilidad, el costo de la introducción de tecnologías MDE al proceso de desarrollo (formación, la curva de aprendizaje) y la longevidad de las herramientas y lenguajes (Kolovos, 2009).

Otro de los desafío de MDSD es desarrollar lenguajes apropiados para la comunicación entre expertos técnicos y no técnicos, y para el modelado de varios aspectos del sistema. El mayor desafío es tener la experiencia en ingeniería de lenguajes requerido para crear perfiles o metamodelos propios, más aún para sistemas complejos que probablemente necesitan varios lenguajes. También se requieren herramientas que soporten dichos metamodelos para que puedan reusarse. Pero las herramientas actuales para el desarrollo de metamodelos y editores no son fáciles de usar, la curva de aprendizaje es empinada y la documentación y el soporte no es satisfactorio (Mohagheghi, 2009).

También, se requieren varias herramientas para el modelado y transformaciones M2M y M2T, verificación y simulación, y otras herramientas para almacenar, reusar y componer modelos. No existen cadenas de herramientas que permitan realizar estas operaciones, por lo que se deben integrar varias herramientas y hacer las adaptaciones necesarias (Mohagheghi, 2009).

7 CONCLUSIONES

En el presente trabajo se han presentado los principales conceptos y pilares en el que se apoya MDSD, un nuevo paradigma para el desarrollo de software, en el cual los modelos son los principales artefactos en el proceso de desarrollo.

La ventaja principal de este paradigma es que los modelos se expresan usando conceptos mucho más cercanos al dominio del problema, elevando el nivel de abstracción de los mismos, en lugar de usar conceptos ligados a la tecnología de implementación. Otra gran ventaja de MDSD es que el código se genera (semi-) automáticamente (mediante transformaciones de modelos) a partir de sus correspondientes modelos. De esta manera, los modelos no sólo se utilizan para documentar el sistema sino también para construir el producto final.

Si bien MDSD promete incrementar la productividad del desarrollador, reducir el costo (en tiempo y dinero) de la construcción de software, mejorar la reusabilidad del software, y hacer software más mantenible, presenta varios desafíos para su adopción, tales como escalabilidad, el costo de la introducción de tecnologías MDE al proceso de desarrollo (formación, la curva de aprendizaje), dificultad para crear metamodelos y herramientas que los soporten.

8 REFERENCIAS

Akehurst, D. H., Bordbar, B., Evans, M. J., Howells, W. G. J., y McDonald-Maier, K. D. SiTra: Simple transformations in java. En *International Conference on Model Driven Engineering Languages and Systems* (pp. 351-364). Springer Berlin Heidelberg, 2006.

Brambilla, M., Cabot, J., y Wimmer, M. Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 1(1), 1-182, 2012.

Bézivin, J., Jouault, F., Rosenthal, P., y Valduriez, P. Modeling in the Large and Modeling in the Small. *Lecture Notes in Computer Science*, 3599, pp. 33-46. Springer Berlin Heidelberg, 2005.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1), 52-70, 2008.

Brown, A. W., Conallen, J., y Tropeano, D. Introduction: Models, Modeling, and Model-Driven Architecture (MDA). *Model-Driven Software Development*, 1-16. Springer Berlin Heidelberg, 2005.

Cuadrado, J. S., Molina, J. G., y Tortosa, M. M. RubyTL: a practical, extensible transformation language. En *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 158-172). Springer Berlin Heidelberg, 2006.

Cuadrado, J. S., Izquierdo, J. L. C., y Molina, J. G. Applying Model-Driven Engineering in Small Software Enterprises. *Science of Computer Programming*, 89, 176-198, 2014.

Czarnecki, K., y Helsen, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 621-645, 2006.

da Silva, A. R. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*, 43, 139-155, 2015.

Drey, Z., Faucher, C., Fleurey, F., Mahé, V., y Vojtisek, D. Kermeta language. *Reference Manual*. 2009. Documento en línea: http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.documentation/build/pdf.fop/KerMeta-Manual/index.pdf.

Eclipse Foundation (a). Acceleo, 2005. Accedido: Diciembre 2015. https://eclipse.org/acceleo/

Eclipse Foundation. ATL, 2008. Accedido: Diciembre 2015. https://eclipse.org/atl/

Eclipse Foundation. Eclipse Modeling Framework (EMF), 2003. Accedido: Diciembre 2015.
https://eclipse.org/modeling/emf/

Eclipse Foundation (b). Epsilon, 2005. Accedido: Diciembre 2015. http://www.eclipse.org/epsilon/

Eclipse Foundation. MOFScript, 2011. Accedido: Agosto 2016. http://www.eclipse.org/gmt/mofscript/

Eclipse Foundation. VIATRA, 2013. Accedido: Agosto 2016. http://www.eclipse.org/viatra/

Jamda Project, 2003. Accedido: Agosto 2016. http://jamda.sourceforge.net

Jouault, F., Allilaire, F., Bézivin, J., y Kurtev, I. ATL: A Model Transformation Tool. *Science of computer programmin*g, 72(1), 31-39, 2008.

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., y Völkel, S. Design guidelines for domain specific languages. *OOPSLA Workshop on Domain-Specific Modeling* (DSM' 09), 2009.

Kleppe, A. G., Warmer, J. B., y Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

Kolovos, D. S., Paige, R. F., y Polack, F. A. Eclipse Development Tools for Epsilon. *Eclipse Summit Europe, Eclipse Modeling Symposium*, 20062, 2006.

Kolovos, D. S., Paige, R. F., y Polack, F. A. The Grand Challenge of Scalability for Model Driven Engineering. *Lecture Notes in Computer Science*, 5421, 48-53. Springer Berlin Heidelberg, 2009.

Lawley, M., y Steel, J. Practical declarative model transformation with Tefkat. En *International Conference on Model Driven Engineering Languages and Systems* (pp. 139-150). Springer Berlin Heidelberg, 2005.

Liddle, S. W. Model-driven software development. *Handbook of Conceptual Modeling*. Springer Berlin Heidelberg, pp. 17-54, 2011.

Mellor, S. J. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, 2004.

Mernik, M., Heering, J., y Sloane, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR), 37*(4), 316-344. 2005.

Mohagheghi, P., Fernandez, M. A., Martell, J. A., Fritzsche, M., y Gilani, W. MDE Adoption in Industry: Challenges and Success Criteria. *Lecture Notes in Computer Science*, 5421, 54-59. Springer Berlin Heidelberg, 2009.

Muñoz, F. D., Castilla, J. T., y Moreno, A. V. *Desarrollo de Software Dirigido por Modelos*. Fundación para la Universitat Oberta de Catalunya, 2007.

Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., & Berre, A. J. Toward standardised model to text transformations. En *European Conference on Model Driven Architecture-Foundations and Applications* (pp. 239-253). Springer Berlin Heidelberg, 2005.

OMG. Meta Object Facility (MOF), Versión 2.5, 2015. Accedido: Diciembre 2015. http://www.omg.org/spec/MOF/2.5/

OMG. Model-Driven Architecture (MDA) Guide, Versión 1.0.1, 2003. Accedido: Diciembre 2015. http://www.omg.org/cgi-bin/doc?omg/03-06-01

OMG. MOF Model to Text Transformation Language (MOFM2T), Versión 1.0, 2008. Accedido: Diciembre 2015. http://www.omg.org/spec/MOFM2T/1.0/

OMG. Object Constraint Language (OCL), Versión 2.3.1, 2012. Accedido: Diciembre 2015.
http://http://www.omg.org/spec/OCL/2.3.1/

OMG. Query/View/Transformation Specification (QVT), Versión 1.1, 2011. Accedido: Diciembre 2015. http://www.omg.org/spec/QVT/1.1

Pons, C., Giandini, R., y Pérez, G. *Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica*. EDULP & McGraw-Hill Educación, 2010.

Rose, L. M., Paige, R. F., Kolovos, D. S., y Polack, F. A. The Epsilon Generation Language. *Lecture Notes in Computer Science*, 5095, 1-16. Springer Berlin Heidelberg, 2008.

Selic, B. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5), pp. 19–25, 2003.

Selic, B. Model-Driven Development: Its Essence and Opportunities. *Object and Component-Oriented Real-Time Distributed Computing* (ISORC 2006), 313-319. IEEE Computer Society, 2006.

Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., y Yusuf, L. *Patterns: Model-Driven Development Using IBM Rational Software Architect*. IBM, International Technical Support Organization, 2005.

Thomas, D., Fowler, C., & Hunt, A. *Programming Ruby. The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

Varró, D., y Balogh, A. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3), 214-234, 2007.

Völter, M., Stahl, T., Bettin, J., Haase, A., y Helsen, S. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, 2013.