# Integración de UML y Lenguajes de Modelado Específicos de Dominio Mediante la Generación Automática de Perfiles UML

**Tesis de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información**

**Giovanni Giachetti Herrera**

**Director:**
**Dr. Oscar Pastor López**

**Departamento de Sistemas Informáticos y Computación**

**Septiembre 2008**

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

**Tesis de Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información**

# Integración de UML y Lenguajes de Modelado Específicos de Dominio Mediante la Generación Automática de Perfiles UML

**Giovanni Giachetti Herrera**

Director: Oscar Pastor López

A mi amada Beatriz

# *Agradecimientos*

A Oscar por ser un gran director de tesis y un mejor amigo. Me has enseñado a navegar por los mares de la investigación cual corsario, y me has guiado por la jungla de la vida mejor que el propio Tarzán.

Pele, muchas gracias por tu franqueza, y por ser la mano amiga de la que puedes aferrarte cuando más lo necesitas.

A mi amiga Vicky, por su cariño, sus consejos y buenas vibras, eres la mejor!!

A Peter, por tu actitud siempre positiva ante la vida. Amigo, gente como tu es la hace de este mundo un lugar mejor para todos.

A la mejor gestora del mundo, Ana gracias por tu amistad y ayuda incondicional, no sé que haríamos sin ti.

Pau, después de tanto tiempo me es muy fácil decir que eres un gran amigo y espero que lo sigamos siendo mientras podamos :-P

A Manoli y Joan, gracias por vuestra confianza y por ser grandes compañeros de trabajo.

Fani y Carlos, su alegría y ternura, racionalidad y gran sentido del trabajo, son el Ying y el Yang de nuestro laboratorio.

A Paco, J-Lu y Sergio. Amigos, gracias por vuestras valiosas críticas constructivas, y por vuestro particular sentido del humor. Vuestra especial personalidad forma una parte esencial de la identidad de nuestro grupo

A Gonzalo, por ser el mejor lector de noticias de todos los tiempos. Gracias por tu amistad compadre.

A Barbe, Juan Carlos y todos nuestros amigos de CARE. El trabajo de esta tesis es un reflejo de la gran experiencia que hemos adquirido con vosotros, muchas gracias!.

A Natalie, Ignacio, Nelly, Marta, Luis, Paqui. Gracias a todos por enseñarme que la verdadera amistad supera cualquier frontera ;-)

Finalmente, quiero dar el mayor de los agradecimientos a mi amada Beatriz. Gracias por ser mi esposa y por darle a nuestro amor las alas que nos han permitido conocer a tanta gente maravillosa. Perri, eres sin lugar a dudas el mayor tesoro en la vida de este pirata.

# *Índice*

# *Introducción*

Al hacer un análisis de la evolución del software en los últimos años, es posible observar como las tecnologías de desarrollo de software se han enfocado cada vez más hacia el *Desarrollo Dirigido por Modelos* [29][58], también conocido como MDD por sus siglas en inglés: *Model Driven Development*. Este esquema de desarrollo ha impulsado la creación de una serie de tecnologías destinadas a dar soporte a todo el proceso de ingeniería de software centrado en el uso de modelos. Este conjunto de tecnologías están enmarcadas dentro de lo que se conoce como *Ingeniería Dirigida por Modelos*, o simplemente MDE (*Model Driven Engineering* en inglés) [25].

El posicionamiento de MDD se debe en gran medida a los beneficios que provee este enfoque, entre los que es posible destacar la reducción en los tiempos de desarrollo y la disminución de costos asociados al mantenimiento de las aplicaciones desarrolladas [4][62]. Por este motivo, no es extraño que aparezcan constantemente nuevas propuestas basadas en este esquema de desarrollo, enfocadas a los más diversos dominios de aplicación. Sin embargo, a pesar de los beneficios que provee MDD, esta propuesta no tendría el auge con el que cuenta actualmente sin un soporte adecuado que permita su difusión de forma masiva. Este soporte es sin lugar a dudas la *Arquitectura Dirigida por Modelos* [32][34][39] o MDA (por sus siglas en ingles que significan *Model Driven Architecture*). La propuesta MDA definida por el *Object Management Group* (OMG) [38] es actualmente la arquitectura más conocida y utilizada en entornos MDD [33].

El objetivo de MDA es separar la lógica de aplicación de los aspectos tecnológicos asociados a la plataforma de implementación, dividiendo el desarrollo de una aplicación en varios niveles de abstracción que son representados mediante modelos conceptuales.

Durante el proceso de desarrollo, estos modelos son transformados para pasar desde un nivel de mayor abstracción a un nivel de menor abstracción, y así sucesivamente hasta alcanzar el nivel de de abstracción más bajo que corresponde a la aplicación final. El esquema de desarrollo MDA está representado en la Figura 1.

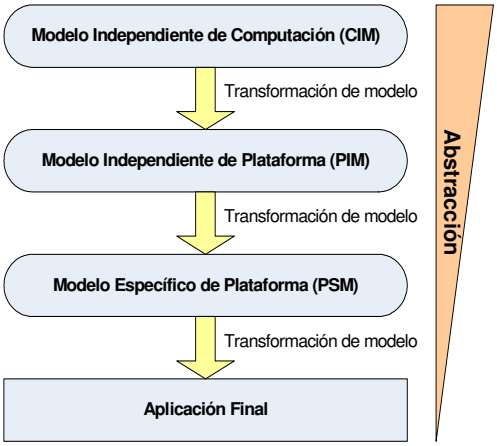


**Figura 1.** Esquema de desarrollo MDA

En términos generales, todas las tecnologías MDD poseen una filosofía similar a la de MDA, donde la generación de aplicaciones está centrada en transformaciones de modelos. Extrapolando esta idea a la *Ingeniería Dirigida por Modelos* (MDE), podríamos resumir que las tecnologías MDE transforman los modelos conceptuales en productos de software que apoyan los distintos pasos de un proceso de ingeniería (desarrollo de aplicaciones, medición de calidad, estimación de costo y esfuerzo, generación automática de documentación, etc.).

Dentro de este contexto de MDE, es fundamental contar con un lenguaje de modelado cuyos constructores conceptuales posean una semántica precisa para poder definir modelos conceptuales que representen sin ambigüedad los productos que serán generados. MDA recomienda el uso del lenguaje de modelado unificado UML (Unified Modeling Language en inglés) [13] para definir los modelos conceptuales requeridos [32]. Sin embargo, la semántica de UML no cuenta con la precisión suficiente para realizar de forma correcta un

proceso de transformación de modelos completo, siendo necesario extender UML para incorporar esta precisión semántica [14][59].

Extender UML no es una tarea sencilla, debido entre otras cosas a que no existe un proceso que indique como realizar correctamente las extensiones necesarias [14]. Esta situación ha motivado que muchas soluciones MDE opten por definir lenguajes de modelado específicos para su dominio de aplicación. Estos lenguajes de modelado específicos de dominio o DSML [27][57] (Domain Specific Modeling Language en inglés), definen de manera precisa el conjunto de constructores conceptuales requeridos por una propuesta MDE. En otras palabras, un DSML define sin ambigüedad los modelos conceptuales asociados a propuestas MDE, permitiendo realizar de forma correcta las transformaciones de modelos destinadas a generar los productos finales.

Por otra parte, UML es el principal estándar para el modelado de software a nivel mundial, por lo que contar con soporte para UML es una ventaja competitiva para aquellas propuestas MDE que deseen ser aceptadas de forma global por la industria del software. De esta manera, muchas propuestas MDE buscan la mejor manera para integrar la semántica de sus DSMLs en UML, sin perder la expresividad semántica del DSML original.

Dentro de la especificación de UML, se ha definido un mecanismo de extensión que permite incorporar en UML la precisión semántica requerida por propuestas MDE específicas. Este mecanismo de extensión es denominado *perfil UML* y presenta una serie de beneficios, tales como: la posibilidad de utilizar las herramientas UML ya existentes, y una mejora en la curva de aprendizaje [5][61]. Este último beneficio se debe principalmente a que las extensiones mediante perfiles UML no alteran la semántica original de UML, permitiendo a los desarrolladores de software aprovechar sus conocimientos de UML para definir los modelos conceptuales.

Dadas las ventajas que presenta el uso de perfiles UML, es particularmente atractivo el poder utilizar esta alternativa de extensión para resolver la brecha semántica que existe entre un

DSML y UML [5], o en otras palabras, para extender UML con la semántica de un DSML.

Trabajos relacionados establecen que la definición manual e intuitiva de un perfil UML no es una alternativa adecuada [26][63], ya que este tipo de definiciones están expuestas a introducir errores tecnológicos y conceptuales, además que pueden requerir demasiado tiempo de ejecución. Estos dos factores (tiempo y errores) causan un impacto negativo en la correcta aplicación de soluciones MDE, impacto que puede llegar a ser importante en propuestas enfocadas a un contexto industrial, donde el tiempo es oro y los errores de implementación impactan directamente en la satisfacción de los clientes. Por este motivo, contar con un mecanismo para generar de forma automática el perfil UML adecuado se convierte en la solución más indicada.

Para poder dar un soporte UML adecuado a propuestas MDE, el objetivo principal de esta Tesis de Master es: *La definición de un proceso completo que permita generar las extensiones necesarias, para integrar en UML la precisión semántica requerida por una propuesta MDE específica*. La precisión semántica estará dada por el metamodelo del DSML asociado a la propuesta MDE, mientras que las extensiones serán definidas mediante un perfil UML que es generado automáticamente. Este perfil permitirá la definición de los modelos conceptuales requeridos por propuestas MDE, manteniendo absoluta consistencia con la especificación de UML. De esta manera, conseguir un proceso completo que indique como definir extensiones sobre UML, y al mismo tiempo proveer un mecanismo automatizado para la generación de dichas extensiones.

El resto de este capítulo de introducción está organizado de la siguiente forma: La sección 1.1 establece el planteamiento del problema. La sección 1.2 detalla la motivación del trabajo realizado. La sección 1.3 muestra el contexto de desarrollo del trabajo presentado. La sección 1.4 muestra los objetivos específicos que dan soporte al trabajo realizado. Finalmente, la sección 1.5 presenta la estructura completa de esta Tesis de Master.

## 1.1. Planteamiento del Problema

El *Object Management Group* (OMG) ha definido el *lenguaje de modelado unificado* (UML), como un lenguaje de propósito general que cuenta con la flexibilidad semántica necesaria para adaptarse a distintos dominios de aplicación. Sin embargo, esta flexibilidad conlleva a que UML no tengan la precisión suficiente para definir modelos conceptuales que puedan ser utilizados como entrada en procesos de transformación asociados a entornos MDE. Esta falta de *precisión semántica* se puede observar en los *puntos de variación semántica* definidos en la especificación de UML. Estos puntos de variación presentan múltiples interpretaciones semánticas para un mismo constructor conceptual [59], estableciendo una definición ambigua o incompleta para estos constructores conceptuales. Por ejemplo, en el constructor UML de la *Asociación* hay un punto de variación que indica la falta de una semántica que establezca el orden y la forma en que se crean las partes de una composición [48].

Para poder utilizar UML en la definición de modelos que puedan ser aplicados en entornos MDE específicos, es necesario extender UML para darle la precisión semántica requerida en el dominio de aplicación de cada propuesta MDE. Extender UML no significa sólo incorporar nuevos constructores conceptuales o nuevas propiedades a los constructores conceptuales existentes, sino que también implica la definición de restricciones que limiten los posibles valores de las propiedades existentes. En otras palabras, se personaliza la especificación de UML para poder ser utilizado como un lenguaje de modelado específico de dominio o DSML.

Integrar en UML la semántica requerida por una propuesta MDE específica no es una tarea trivial. La principal dificultad radica en que no existe un proceso completo y estandarizado para definir extensiones sobre UML. Por otra parte, UML en su condición de

lenguaje de modelado multipropósito, posee un número de constructores conceptuales y modelos que suelen sobrepasar con creces las necesidades de modelado de propuestas MDE específicas. Esto dificulta la identificación de los constructores que deben ser extendidos, e implica un mayor esfuerzo para determinar el impacto que las extensiones definidas sobre un constructor conceptual tienen sobre la semántica del resto de los constructores.

Actualmente, la especificación de UML [47][48] incorpora una serie de características para mejorar las posibilidades de extensión de este lenguaje mediante el uso de *perfiles UML*. Las nuevas características de los perfiles UML permiten definir la precisión semántica requerida por la mayoría de las propuestas MDE existentes, provocando que en los últimos años la cantidad de perfiles UML asociados a propuestas MDE se incremente considerablemente.

Sin embargo, a pesar que los perfiles UML son cada vez más utilizados, aún no existe un proceso completo que indique como elaborar correctamente un perfil UML que integre en UML la semántica requerida por una propuesta MDE. Tal como señala Bran Selic en [59], la falta de un proceso para la definición de perfiles UML conlleva a que no se aprovechen todas las posibilidades de extensión que estos proveen, y que en muchos casos, los perfiles UML resultantes sean técnicamente inválidos al no cumplir con la especificación de OMG.

## 1.2. Motivación

Las propuestas MDE se ven impulsadas a utilizar perfiles UML para integrar en UML la precisión semántica que requieren, porque UML es el lenguaje más utilizado a nivel mundial para la definición de modelos conceptuales asociados al desarrollo de software. El contar con tecnologías MDE basadas en modelos UML les permite entrar con mayor fuerza en la industria del desarrollo de software,

pudiendo llegar a un mayor número de potenciales usuarios y aprovechar la gran variedad de tecnologías UML existentes. Sin embargo, las dificultades que existen para integrar en UML la precisión semántica requerida en procesos de desarrollo MDE, han motivado que distintas propuestas opten por no utilizar UML y definan su propio DSML. Estos DSMLs especifican sólo la semántica y estructura del conjunto de constructores necesarios para representar con precisión los modelos conceptuales requeridos, disminuyendo considerablemente el tamaño y la complejidad que tiene un DSML en relación a UML, lo que facilita su comprensión y uso por parte de usuarios que conozcan el dominio de aplicación. Otra ventaja de los DSMLs, es que su menor tamaño y complejidad (en relación a UML) facilita la implementación de herramientas MDE específicas, destinadas a optimizar las tareas de modelado, desarrollo y mantenimiento de los sistemas de software generados.

El trabajo presentado en esta Tesis de Master, comienza como una solución para resolver las necesidades de integración con UML de una propuesta MDA específica, esta propuesta es el método de producción automática de software OO-Method [53]. Este método, cuenta con soporte industrial mediante una suite de herramientas MDE y un compilador de modelos conceptuales [55] desarrollados por CARE Technologies [6].

La implementación industrial de OO-Method cuenta con un DSML propietario, por lo que hay especial interés en definir un mecanismo que permita integrar la semántica del DSML de OO-Method en UML. De esta manera, se persigue potenciar la expansión de OO-Method hacia el mundo de los usuarios de UML, y además poder compartir la experiencia de OO-Method con otras comunidades científicas, utilizando UML como lenguaje común.

Un primer intento de integrar OO-Method con UML ha sido el desarrollo de dos herramientas de intercambio entre modelos OO-Method y UML [28]: *XMI Importer* y *XMI Exporter*. La primera de estas herramientas (*XMI Importer*) realiza la importación de un modelo UML transformándolo en un modelo conceptual OO-Method.

Mientras que la segunda herramienta (*XMI Exporter)* realiza el proceso de exportación, es decir, transforma un modelo conceptual OO-Method en un modelo UML equivalente. Sin embargo, mucha información de modelado se pierde en el intercambio de modelos, dado que UML no cuenta con la precisión semántica necesaria para representar correctamente un modelo conceptual OO-Method.

La experiencia obtenida con esta primera solución de integración entre OO-Method y UML, es el punto de inicio para la elaboración de un proceso que permita generar automáticamente un perfil UML para dar soporte al modelo conceptual de OO-Method. Este proceso ha evolucionado y ha sido generalizado para obtener un proceso completo, que pueda ser aplicado por distintas propuestas MDE para integrar sus necesidades de modelado en UML.

De esta manera, el trabajo presentado en esta Tesis de Master pretende dar solución a aquellas propuestas MDE que deseen extender UML para utilizarlo en la definición de sus modelos conceptuales, mediante un proceso orientado a la generación de un perfil UML que cumpla con la especificación de OMG, que tome ventaja de las nuevas características introducidas en la última versión de la especificación de UML [47][48]. Además, dado que el uso de DSMLs y UML proveen beneficios para el desarrollo de las tecnologías MDE, la solución propuesta permitirá sacar provecho de estos beneficios, al posibilitar la integración de ambas alternativas de modelado.

## 1.3. Contexto

Esta Tesis de Master ha sido desarrollada dentro del Centro de Investigación Pros, perteneciente a la Universidad Politécnica de Valencia. El trabajo expuesto en esta Tesis de Master forma parte del convenio CONCOM, subscrito por el grupo de investigación OO-Method (perteneciente al centro de investigación Pros) y la empresa CARE-Technologies.

Tal como se puede apreciar en el establecimiento del problema y motivación, esta Tesis de Master adquiere un enfoque Científico-Industrial. Tiene un enfoque científico, ya que ha sido desarrollada para dar solución a un problema relevante dentro del área de la ingeniería de software dirigida por modelos. Y tiene un enfoque industrial, ya que los resultados obtenidos serán utilizados para dar una solución completa de integración con UML, a la propuesta industrial de OO-Method implementada por la empresa CARE-Technologies.

Finalmente, además del convenio CONCOM, el trabajo realizado se ha podido llevar a cabo gracias a proyectos financiados por el Centro de Investigación Científica y Tecnológica de España (CICYT), mediante el convenio SESAMO. A continuación se detallan ambos convenios:

- CONCOM: Construcción de Compiladores. Proyecto de I+D financiado por CARE Technologies desde junio de 2006 a mayo de 2008.
- SESAMO: Construcción de Servicios Software a partir de Modelos. Proyecto CICYT referenciado como TIN2007-62894 desde 2008 a 2010.

## *1.4. Objetivos Específicos*

Como se ha señalado anteriormente, el objetivo principal de esta Tesis de Master es la definición de un proceso completo que permita generar las extensiones necesarias, para integrar en UML la precisión semántica requerida por una propuesta MDE específica.

Este objetivo principal, está soportado por un conjunto de objetivos específicos que determinan las tecnologías utilizadas en el desarrollo del proceso propuesto, a la vez que dan forma a la estructura del mismo. Debido a que el trabajo desarrollado en esta tesis nace como una solución para integrar la propuesta industrial

de OO-Method, los objetivos específicos están enfocados a satisfacer las necesidades de propuestas MDE que se desarrollen en un ámbito industrial. Estos objetivos específicos son:

- *Definir un mecanismo de extensión que permita incorporar rápidamente los cambios que sufra la propuesta MDE*. Las propuestas MDE, especialmente las asociadas a contextos industriales, están cambiando constantemente sus esquemas conceptuales para definir nuevas características que permitan satisfacer las necesidades de sus clientes. Por este motivo es muy importante contar con un mecanismo de extensión que permita incorporar fácilmente los cambios en la semántica de los modelos conceptuales asociados.

- *Diseñar un mecanismo para extender UML con toda la expresividad semántica requerida por una propuesta MDE*. Para poder utilizar UML como lenguaje de modelado asociado a una propuesta MDE, es indispensable que UML tenga toda la expresividad semántica que requiere la propuesta MDE para definir modelos conceptuales de forma completa y precisa.

- *Definir un mecanismo para integrar en UML la semántica requerida por una propuesta MDA específica, sin perder la semántica original de UML*. De esta manera, los usuarios con conocimientos en UML pueden utilizar su experiencia de modelado para trabajar con tecnologías MDA específicas, reduciendo los tiempos de aprendizaje.

- *Poder definir extensiones de UML basadas en un estándar de intercambio que permita su interpretación por diferentes herramientas de modelado*. Esto posibilita que los usuarios de tecnologías MDA puedan trabajar con las herramientas de modelado que ya poseen y conocen, reduciendo los costos de implantación.

- *Definir un mecanismo que permita intercambiar los modelos UML con los modelos DSML*. Para las propuestas MDE, es de especial interés poder aprovechar los beneficios que proveen las

tecnologías basadas en DSML y en UML. Además, es muy probable que las propuestas MDE ya cuenten con soluciones basadas en DSMLs, que deseen aprovechar dentro de un contexto UML.

## *1.5. Estructura de la Tesis*

Sin considerar este primer capítulo de introducción, esta Tesis de Master posee la siguiente estructura:

- **Capítulo 2. Estado del arte.** Este capítulo presentan los mecanismos más relevantes para definir extensiones sobre UML, realizando un análisis de las ventajas y desventajas de cada uno. De esta manera, dar una visión más clara del por qué se ha optado por la utilización de un perfil UML y no otro mecanismo de extensión. Luego, se detallan nuevas características introducidas en UML para la definición de perfiles UML, considerando aquellas características relevantes para el diseño del proceso propuesto. Finalmente, se analizan distintos trabajos que proponen alternativas sistemáticas para la construcción de perfiles UML, indicando las ventajas y desventajas que presentan cada uno.
- **Capítulo 3. Generar Perfiles UML para Propuestas MDE**. Este capítulo utiliza el análisis de trabajos relacionados presentados en el de estado del arte (sección 2.3), para definir un proceso genérico que resuelve una serie de desafíos que deben afrontar las propuestas MDE que deseen construir un perfil UML correcto. Luego, a partir de este proceso genérico, es definido un proceso completo para la generación de perfiles UML. Este proceso completo está enfocado a dar solución a los objetivos de esta Tesis de Master y posee dos fases principales: 1) la definición del Metamodelo de Integración y 2) la generación automática del perfil UML.

- **Capítulo 4: Generación del Metamodelo de Integración.** Este capitulo detalla la primera fase del proceso completo para la generación de un perfil UML. En esta fase se requiere la definición del metamodelo del DSML que será integrado en UML. Para realizar correctamente esta definición, se dan una serie de pautas en relación a los elementos que deben ser definidos, y a las tecnologías que deben ser utilizadas para posibilitar una correcta integración con UML. Posteriormente se presenta el Metamodelo de Integración y junto con un conjunto de reglas que indican si el metamodelo se ha especificado correctamente. Finalmente, se presenta la propuesta sistemática diseñada para generar un Metamodelo de Integración correcto a partir del metamodelo de un DSML.

- **Capítulo 5: Generación Automática del Perfil UML.** Este capítulo detalla la segunda fase del proceso completo para la generación de un perfil UML, explicando como a partir del Metamodelo de Integración, se pueden obtener automáticamente las extensiones que deben ser representadas mediante el perfil UML. Luego se presenta un conjunto de reglas de transformación, para obtener el perfil UML adecuado a partir del Metamodelo de Integración y del conjunto de extensiones identificadas previamente.

- **Capítulo 6: Caso de estudio.** En este capítulo se aplica el proceso completo propuesto, para integrar la semántica de la asociación de OO-Method en UML. En este caso de estudio, se explica la propuesta OO-Method y la semántica de la asociación en OO-Method. Luego se construye el metamodelo del DSML para la asociación OO-Method, que es utilizado para generar el Metamodelo de Integración. Finalmente, sobre este Metamodelo de Integración se identifican las extensiones necesarias para integrar la semántica del DSML en UML, y se aplican una serie de reglas de transformación para obtener el perfil UML final que extiende a UML con la semántica de la asociación OO-Method.

- **Capítulo 7: Conclusiones.** En este capítulo se presentan las conclusiones generales del trabajo desarrollado en esta Tesis de Master, indicando las contribuciones obtenidas. También son presentados brevemente, una serie de proyectos MDE industriales que han sido relevantes para el desarrollo de este trabajo. En este capítulo también son presentados los trabajos actuales y futuros, así como las publicaciones relacionadas

# *Estado del Arte*

Las propuestas MDE requieren de modelos con la suficiente formalización semántica para representar a nivel conceptual, de forma completa, y sin ambigüedad, los productos de software que serán desarrollados. Entendiendo el concepto de software como: *El conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación* [22].

Para las propuestas MDE, el adoptar UML como lenguaje de modelado otorga una serie de ventajas, como el aprovechamiento de las herramientas UML existentes, reducción en los tiempos de aprendizaje y costos de implantación, etc. Sin embargo, UML no posee la suficiente precisión semántica para poder ser usado en un proceso MDE completo, siendo necesario definir extensiones sobre UML para poder incorporar esta precisión semántica.

A continuación revisaremos los mecanismos más relevantes para definir extensiones sobre UML, indicando las ventajas y desventajas que presentan cada uno.

## *2.1. Definir Extensiones en UML*

Para utilizar UML en la descripción de modelos conceptuales que sirvan en un proceso MDE, será necesario extender UML con la precisión semántica requerida. En otras palabras, convertir a UML en un DSML.

Para extender UML existen diferentes técnicas [5], entre las que destacan: las extensiones lightweight (o livianas) y las extensiones heavyweight (o pesadas). Ambas técnicas de extensión deben ser

aplicadas sobre el metamodelo de UML, que ha sido definido por OMG bajo el nombre de *Superestructura de UML* [48]. Esta superestructura es el metamodelo de referencia para cualquier tecnología basada en UML.

## Extensiones lightweight

Las extensiones lightweight son extensiones que no cambian el metamodelo de referencia, es decir, sólo pueden agregar nuevas restricciones, propiedades, y notación a los constructores conceptuales ya existentes. En UML este tipo de extensiones se realizan mediante el uso de perfiles, mejor conocidos como *Perfiles UML*.

La principal ventaja de los perfiles UML es que forman parte de la especificación de UML, por este motivo se encuentran bien documentados y las herramientas basadas en UML dan soporte a este tipo de extensiones.

La principal desventaja de los perfiles UML es que sus posibilidades de extensión son limitadas, ya que no pueden modificar la semántica original del metamodelo de UML. Esto implica que los perfiles UML:

- Sólo pueden definir nuevos constructores conceptuales que deriven de los constructores conceptuales de UML existentes.
- No pueden modificar las propiedades de los constructores conceptuales existentes.

La imposibilidad de los perfiles UML de modificar el metamodelo de UML, limita el conjunto de propuestas MDE que pueden utilizar esta técnica de extensión a aquellas en que los constructores conceptuales pueden ser considerados como un subconjunto de los constructores de UML. A pesar de lo fuerte que parezca esta restricción, la mayoría de las propuestas MDE existentes la cumplen, gracias a que UML se ha especificado como un lenguaje de modelado multi-propósito.

## Extensiones heavyweight

Las extensiones heavyweight, también conocidas como extensiones de primer nivel, son extensiones que pueden modificar el metamodelo de referencia, y por este motivo pueden incorporar nuevos constructores conceptuales al metamodelo de UML, o modificar las propiedades originales de los constructores conceptuales existentes. Para definir las extensiones, esta técnica utiliza la especificación MOF (Meta Object Facility) [36] de OMG, que es el lenguaje propuesto por OMG para la elaboración de metamodelos. Al utilizar MOF, las extensiones definidas mantienen consistencia con la sintaxis utilizada para definir el metamodelo de UML, ya que desde la versión 2.0 [35], MOF está completamente alineado con la superestructura de UML, en otras palabras, la superestructura de UML puede ser considerada una instancia de MOF.

La principal ventaja de las extensiones heavyweight es que brindan mayor flexibilidad para la definición de extensiones, dando soporte aquellas propuestas MDE que no pueden utilizar perfiles UML.

La principal desventaja de las extensiones heavyweight, es que al cambiar el metamodelo de UML el resultado obtenido no cumple con la especificación UML propuesta por OMG, y deja de ser compatible con las tecnologías UML basadas en dicha especificación. Por esta razón, este tipo de extensiones no forman parte la especificación de UML.

El trabajo de James Bruck y Kenn Hussey [5] muestra en mayor detalle los distintos tipos de mecanismos para definir extensiones sobre UML, indicando las ventajas y desventajas que presenta cada uno. Este trabajo también muestra una comparación entre las características soportadas por cada mecanismo de extensión. Finalmente, en este trabajo se señala que el uso de extensiones lightweight es el mecanismo más adecuado para definir extensiones sobre UML y sólo en caso que este no pudiera ser utilizado dada las

restricciones que presenta, se debe optar por extensiones heavyweight.

El caso de estudio presentado en [61] por Staron et al., realiza una comparación entre el uso de extensiones lightweight definidas mediante perfiles UML y extensiones heavyweight, ambas aplicadas a un contexto de desarrollo MDE industrial. En este estudio se ha concluido que el uso de los perfiles UML reporta mayores beneficios en la aplicación industrial, por sobre el uso de extensiones heavyweight. Considerando estos resultados y dado que el contexto de este trabajo es apoyar procesos de desarrollo MDE industriales, se ha optado por utilizar un perfil UML para definir las extensiones que serán generadas a mediante el proceso propuesto en esta Tesis de Master. A continuación se muestran en más detalle los perfiles UML y sus principales características.

## 2.2. Perfiles UML

El mecanismo de extensión de perfiles UML se encuentra definido en la *Infraestructura de UML* [47]. Este mecanismo está orientado a adaptar los metamodelos definidos mediante la especificación MOF [35] de OMG, para ajustar su semántica a: plataformas específicas, distintos dominios de aplicación, objetos de negocio, o proceso de modelado de software. Esta tesis está enfocada en la definición de extensiones sobre UML, por lo tanto el metamodelo que debe ser adaptado mediante un perfil UML es el metamodelo de UML. Este perfil UML debe integrar en la superestructura de UML la semántica particular asociada a los constructores conceptuales requerido por una propuesta MDE.

Un perfil UML se representa como un paquete UML estereotipado (con el tag <<profile>>), que extiende a otro metamodelo o perfil UML, y cuenta con tres constructores esenciales para definir extensiones: estereotipos, valores etiquetados y reglas OCL.

- Los estereotipos son el elemento para la especificación de un perfil UML, ya que representan las extensiones que serán aplicadas sobre las clases del metamodelo extendido por el perfil UML. Un estereotipo es identificado mediante un nombre, y para especificar las extensiones que realiza utiliza valores etiquetados y reglas OCL.
- Los valores etiquetados representan meta-propiedades que se incorporan al la clase extendida por el estereotipo. Es decir, permiten definir nuevos atributos o asociaciones que no existen en la definición original de la clase del metamodelo UML.
- Las reglas OCL permiten definir la forma en que interactúan los distintos constructores conceptuales que han sido extendidos mediante un estereotipo del perfil UML.

La Figura 2 muestra un breve ejemplo de un perfil UML llamado *PerfilEjemplo*, que posee un estereotipo llamado *ClaseIdentificada*. Este estereotipo extiende a la clase UML *Class*, agregándole un atributo identificador de tipo *string*. Además incorpora la regla OCL *oclIdentifier*, que indica que es necesario asignarle un valor al identificador, esta regla está definida como se muestra a continuación:

- oclIdentifier: `self.identificador->size() > 0`

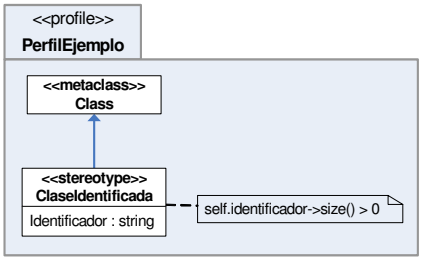


**Figura 2.** Ejemplo de un Perfil UML

# Características de los Perfiles en UML 2.1

En la última versión de UML se han incorporado nuevas características a los perfiles UML [59], que mejoran considerablemente su capacidad de definir extensiones sobre metamodelos. A continuación serán revisadas algunas de las nuevas características relevantes para el proceso propuesto. Para el mejor entendimiento de estas características, es necesario conocer el metamodelo definido en el paquete *Profiles* de la infraestructura de UML [47]. El diagrama de este metamodelo se puede apreciar en la Figura 3.

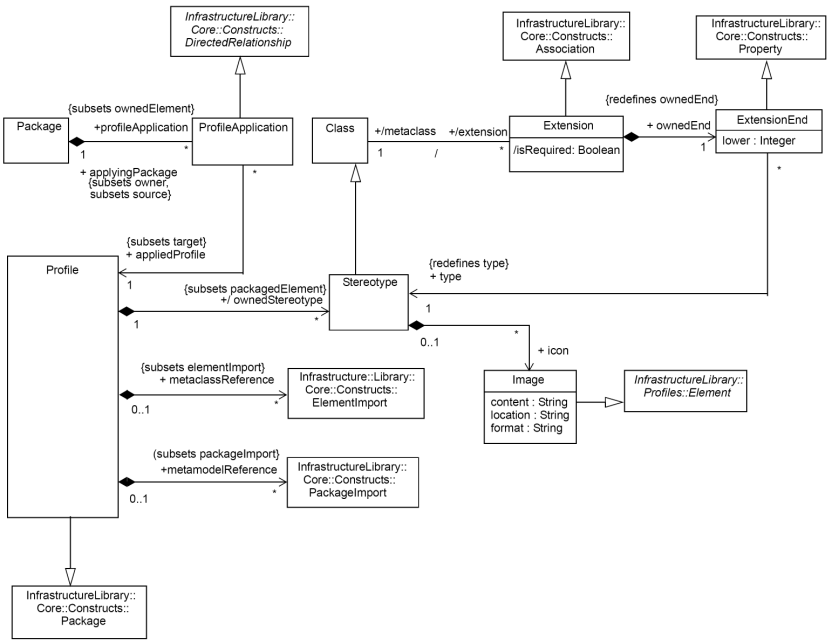


**Figura 3.** Metamodelo asociado al paquete *Profile* de la infraestructura de UML versión 2.1.2

### Estereotipos son Especializaciones de la Metaclase Class

Al ser la metaclase *Stereotype* una especialización de la metaclase C*lass*, se está definiendo a los estereotipos con una semántica similar al de una *Clase* de UML. En la versión 1.4.2 de UML [43] los estereotipos eran sólo elementos generalizables (Figura 4), que no

tenían una semántica bien definida. El cambio realizado en la nueva versión de UML, provee una semántica concreta para los estereotipos que facilita la definición de extensiones, ya que considera un estereotipo como un tipo especial de *Clase* que extiende otras clases mediante una asociación de extensión (*Extension*). El definir la extensión de estereotipos como una clase que especializa a *Association,* da una semántica más clara a la relación que existe entre el estereotipo y la clase que extiende. Si bien, la extensión definida entre una clase y un estereotipo corresponde a un refinamiento de la clase extendida, hay que tener muy claro que la extensión no es una asociación de generalización, y por lo tanto no presenta las mismas características, como pueden ser: la herencia, y la redefinición de propiedades.

**Valores Etiquetados como Atributos de una Clase**

En UML 1.4.2 los valores etiquetados (*tagged values*) son metaelementos independientes, esto se puede observar en la Figura 4 en la clase *TaggedValue*. En la versión actual de UML esta definición independiente de valores etiquetados ya no existe, ya que al ser los estereotipos especializaciones de la metaclase *Class*, los valores etiquetados pasan a ser los atributos de la metaclase *Stereotype*. En otras palabras, los valores etiquetados son propiedades y adquieren la semántica definida para la metaclase *Property*. En la versión 2.0 de UML [44], aún cuando los valores etiquetados son propiedades, se prohíbe que los estereotipos puedan participar en asociaciones, esta prohibición ha sido eliminada en la 2.1.1 [45] de UML. Con esto, los valores etiquetados al igual que las propiedades pueden ser de tipo objeto-valuado, es decir, pueden obtener valor a partir de las instancias de una clase, posibilitando la definición nuevas asociaciones, como son: asociaciones entre estereotipos y asociaciones entre estereotipos y cualquier otra clase del metamodelo de referencia.

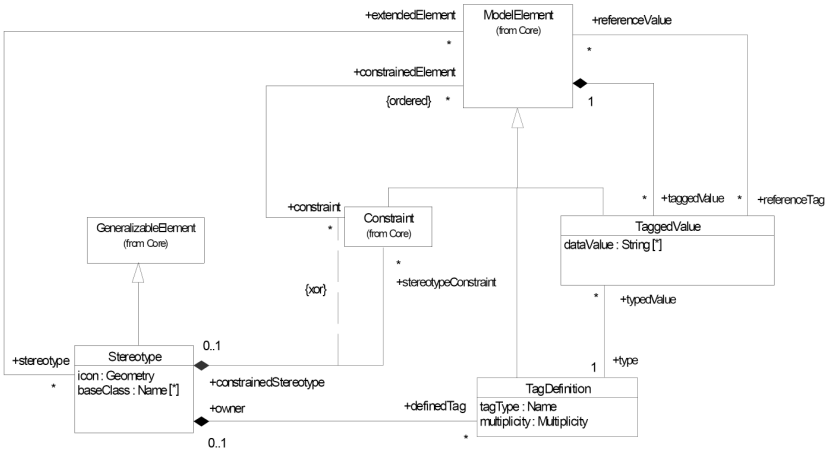


**Figura 4.** Metamodelo del paquete *Extension Mechanisms* definido en UML versión 1.4.2

### Definición de Estereotipos Obligatorios

En determinadas circunstancias, es necesario contar con estereotipos que se apliquen siempre que se defina una nueva instancia de la clase extendida, de esta manera evitar inconsistencias con la semántica del modelo conceptual que se intenta representar. En el metamodelo es posible observar que la obligatoriedad de un estereotipo sobre una clase, estará dado por la cardinalidad mínima que tenga *ExtensionEnd* (observar en la Figura 3 que ExtensionEnd es una especialización de *Property*). Cuando la cardinalidad mínima de *ExtensionEnd* es 1, entonces se establece la obligatoriedad entre el estereotipo y la metaclase. En este caso, el atributo derivado *isRequiered* de la metaclase *Extensión* toma valor *true*.

### Reglas Para la Definición de Perfiles Utilizando XMI

Actualmente los perfiles UML cuentan con una definición XMI estandarizada, el soporte de un mecanismo de intercambio estandarizado para la representación de los perfiles UML, permite la interpretación sin ambigüedad por diferentes herramientas de

modelado, y facilita su uso en procesos de transformación de modelos.

## 2.3. Propuestas para la Definición de Perfiles UML

Debido a los beneficios que presenta la definición de extensiones basada en perfiles UML, existen muchas propuestas para representar dominios de aplicación específicos mediante perfiles UML, tal es el caso de los perfiles UML cuya especificación ha sido adoptada por la OMG y que pueden ser encontrados en [31]. También es posible encontrar perfiles UML destinados a describir los modelos conceptuales requeridos por compiladores de modelos, este es el caso de las propuestas WebML [30] y OO-Method [53].

A pesar de que los perfiles UML se están utilizando de manera generalizada para permitir el uso de UML en la definición de modelos conceptuales asociados a dominios específicos, por parte de OMG aún no se ha definido un proceso enfocado a la definición de extensiones sobre UML, esto queda de manifiesto en el artículo de France, R.B et al. [14].

Antes de continuar con el análisis de los mecanismos para la definición de perfiles UML, es importante tener en claro que no todas las propuestas MDE pueden utilizar perfiles UML para integrar en UML sus necesidades particulares de modelado. La principal restricción radica en que los perfiles UML no pueden cambiar la semántica original del metamodelo que extienden. En el proceso propuesto en esta tesis, el metamodelo que debe ser extendido es el metamodelo de UML, que estará dado por la especificación de OMG que ha sido denominada *Superestructura de UML* [48]. Esta restricción limita el conjunto de propuestas MDE a aquellas cuyos constructores conceptuales pueden ser considerados como un subconjunto de los constructores de UML. Sin embargo, UML al ser formulado como un lenguaje de modelado de propósito general,

permite que la mayoría de las propuestas MDE existentes cumplan con esta restricción.

La literatura relacionada a la elaboración de perfiles UML es bastante escasa, sin embargo, analizando las implementaciones de perfiles existentes es posible identificar dos caminos bastante claros. El primer camino consiste en definir directamente sobre el metamodelo UML las extensiones necesarias, de forma manual e intuitiva, de acuerdo a los criterios y conocimientos con los que cuente el desarrollador del perfil UML. Un ejemplo de este esquema de definición es el perfil elaborado para SysML [41] [42]. Esta forma intuitiva de definir perfiles UML presenta un alto grado de complejidad, y el riesgo de no implementar correctamente las extensiones de acuerdo a la especificación de UML, o de cometer errores propios de una definición manual.

El segundo camino para la elaboración de perfiles UML consiste en un esquema más estructurado y formal. Este esquema está centrado en el metamodelo que describe los constructores conceptuales del dominio de la propuesta MDE. Este metamodelo define la semántica (sintaxis abstracta) del lenguaje específico de dominio (DSML) de la propuesta MDE, es decir, es el metamodelo del DSML. Luego, se establecen las correspondencias entre este metamodelo del DSML y el metamodelo de UML, para finalmente integrar la semántica descrita en el metamodelo del DSML en UML mediante la elaboración del perfil UML correspondiente. Este segundo camino es más adecuado para elaborar un perfil UML correcto por las siguientes razones:

- El metamodelo del DSML brinda una definición precisa de la semántica que debe ser integrada en UML, permitiendo establecer mecanismos de validación e incluso automatizar la definición de extensiones.
- Dado que el metamodelo del DSML representa un conjunto menor de constructores que el metamodelo de UML y está más ajustado al dominio de aplicación, su definición es más sencilla e

intuitiva que definir las extensiones directamente sobre el metamodelo de UML.

- Existe bastante documentación y herramientas orientadas a la definición de metamodelos, mientras que la literatura relacionada a la correcta definición de perfiles UML es bastante escasa y las herramientas UML prácticamente no proveen ayuda para su definición.
- El metamodelo del DSML y la identificación de correspondencias con el metamodelo de UML permiten determinar si la semántica requerida por la propuesta MDE puede ser integrada en UML, de acuerdo a las limitaciones que presentan los perfiles UML. Además, en caso que no sea posible la integración, el metamodelo del DSML puede ser utilizado en la construcción de herramientas de modelado específicas.

Siguiendo el segundo camino para la elaboración de perfiles UML, uno de los primeros trabajos en proponer el uso del metamodelo del DSML para la obtención de perfiles UML es el de Fuentes-Fernández et al. [15]. En este trabajo se proponen algunas guías básicas para la generación del perfil UML a partir del metamodelo del DSML (que en este trabajo es denominado *Metamodelo de Dominio*).

**Los puntos fuertes de este trabajo son:**

1. Propone el uso de Meta Object Facility (MOF) [8] para la definición del metamodelo del DSML. MOF es el lenguaje estándar de OMG para la definición de metamodelos y se encuentra alineado con la superestructura de UML. Se esta manera, el metamodelo del DSML estará definido en el mismo lenguaje que el metamodelo de UML, manteniendo una consistencia estructural entre ambos.
2. Propone algunas reglas con las que se pueden inferir las extensiones (estereotipos, valores etiquetados y restricciones) así como las metaclases que deben ser extendidas. Estas reglas

proveen una primera aproximación de cómo automatizar la generación de perfiles UML.

**Los puntos débiles del trabajo son:**

1. Las reglas propuestas para la construcción del perfil UML son demasiado básicas y no permiten aprovechar la semántica que ya existe en UML. Por ejemplo, propone incorporar todos los atributos definidos en el metamodelo del DSML como valores etiquetados. Sin embargo, muchos de estos atributos pueden tener equivalencia con atributos ya definidos en UML.
2. No se establecen pautas para la correcta definición del metamodelo del DSML, orientadas a obtener una correcta integración con el metamodelo de UML.

Selic en [14] presenta una aproximación sistemática para la definición de perfiles UML a partir del metamodelo del DSML (que denomina modelo de dominio). En este artículo se incluyen una serie de criterios que deben ser considerados al momento de elaborar el metamodelo del DSML para realizar una correcta integración con el metamodelo de UML, así como una lista de consideraciones para realizar el mapeo del metamodelo elaborado en un perfil UML.

**Los puntos fuertes de este trabajo son:**

1. Presenta las nuevas características de los perfiles UML.
2. Presenta guías para la correcta definición del metamodelo del DSML.
3. Propone realizar una identificación de equivalencias entre el metamodelo del DSML y el metamodelo de UML, para identificar correctamente los elementos que deben ser extendidos y las extensiones que deben ser realizadas.

**El punto débil del trabajo es:**

1. Las guías presentadas para la elaboración del perfil UML son demasiado generales y no permiten algún tipo de automatización.

Si bien los artículos orientados a la correcta definición de perfiles UML son escasos, lo son aún más los que proponen alternativas de automatización. Entre estos trabajos es posible destacar los trabajos de Lagarde et al. [26] y el de Wimmer et al. [63].

El trabajo de Lagarde et al. [26] propone realizar una identificación de equivalencias entre las clases del metamodelo del DSML y el metamodelo de UML mediante la definición de un esqueleto inicial del perfil UML. Luego, mediante la identificación de patrones, este esqueleto es refinado de manera automática para obtener el perfil UML final.

**Los puntos fuertes de este trabajo son:**

**Figura 1.** Propone un conjunto de reglas para la generación automática del perfil UML.

1. Propone la definición de un mapeo de equivalencias entre las clases de los metamodelos.
2. Propone la identificación de patrones para generar las extensiones necesarias, utilizando como referencia la información de mapeo.

**Los puntos débiles de este trabajo son:**

1. Para realizar el mapeo se define un esqueleto inicial del perfil UML, por lo que el diseñador del DSML requiere tener conocimientos de perfiles UML.
2. Los patrones definidos están centrados principalmente en el manejo de asociaciones entre clases, dejando de considerar

muchas otras situaciones de modelado, por lo que no es posible realizar una generación completa del perfil UML.

El trabajo de Wimmer et al. [63] propone un esquema semiautomático para la integración entre DSMLs y UML. En esta propuesta se incorpora un lenguaje específico para la definición de equivalencias (mapeo) entre los metamodelos.

**Los puntos fuertes de este trabajo son:**

1. Establece la definición de un mapeo entre los distintos elementos de los metamodelos (clases, atributos, asociaciones, etc.), lo que permite una mejor inferencia de las extensiones necesarias.
2. Establece pautas para definir las reglas de transformación para obtener el perfil UML, y como estas trasformaciones pueden ser implementadas mediante un lenguaje de transformación de modelos.

**El punto débil de este trabajo es:**

1. Sólo permite equivalencias 1:1 entre ambos metamodelos, es decir, un elemento del metamodelo del DSML está asociado sólo con un elemento del metamodelo de UML y viceversa. Esta es una limitación importante para poder aplicar este trabajo en propuestas MDE reales, donde las equivalencias ente metamodelos pueden ser 1:M, M:1 e incluso M:M.

Finalmente, destacaremos el trabajo de Abouzahra et al. [1], que propone una solución para el intercambio entre modelos definidos con perfiles UML y modelos definidos con DSMLs. Para el intercambio de modelos usa un mapeo entre la definición del perfil UML (que denomina metamodelo del perfil) y el metamodelo del DSML.

**Los puntos fuertes de este trabajo son:**

1. Permite aprovechar los beneficios de los DSMLs y los perfiles UML.
2. Es compatible con las propuestas antes señaladas, ya que hace uso del metamodelo del DSML.
3. Ofrece una herramienta de código abierto para realizar el intercambio.

**Los puntos débiles de este trabajo son:**

1. La propuesta está centrada en una herramienta específica, dificultando su generalización para otras propuestas MDE.
2. El mapeo entre el perfil UML y el metamodelo del DSML se debe realizar de forma manual.

# *Perfiles UML para Propuestas MDE*

Para conseguir un proceso completo que permita generar un perfil UML, que integre las necesidades de modelado de propuestas MDE en UML, será necesario resolver una serie de desafíos. Estos desafíos estarán dados por los puntos débiles que presentan cada una de las propuestas analizadas durante el capítulo de estado del arte de esta tesis. Por lo tanto, el primer paso para conseguir un proceso adecuado, será plantear los distintos desafíos que deben ser resueltos para generar un perfil UML asociado a una propuesta MDE y proporcionar soluciones a cada uno de estos desafíos. Con las soluciones propuestas para los distintos desafíos, será construida una maqueta, que denominaremos *proceso genérico,* donde se definen los pasos que deben ser considerados en un proceso completo para la generación de un perfil UML correcto. Posteriormente será construido un proceso completo, a partir del proceso genérico definido. En este proceso completo se detalla como deben ser implementados cada uno de los pasos del proceso genérico, para alcanzar los objetivos propuestos para esta Tesis de Master.

## *3.1. Desafíos y Soluciones*

En esta sección se consideran las propuestas analizada en el capítulo de estado del arte, para identificar los desafíos que deben ser resueltos en la obtención de un perfil UML, que se ajuste a las necesidades de modelado de propuestas MDE. Para cada desafío, se plantearán posibles soluciones que darán paso a la construcción de un esquema general para la generación de perfiles UML denominado

*Proceso Genérico*. En la definición de este proceso genérico el primer desafío que encontramos es:

**PRIMER DESAFÍO: Establecer el punto de partida.**

Al analizar los trabajos que proponen alternativas para la correcta definición de perfiles UML, parece claro que el mejor punto de partida será el metamodelo del DSML, siendo este es el primer paso del proceso para construir perfiles UML. El metamodelo del DSML debe ser definido sin considerar aspectos del perfil UML o del metamodelo de UML, con el fin de obtener una correcta representación del dominio y evitar cualquier tipo de restricción semántica proveniente de UML. El metamodelo del DSML debe tener los siguientes elementos:

- El conjunto de constructores conceptuales.
- El conjunto válido de relaciones que existen entre los constructores conceptuales.
- El conjunto de restricciones que controlan como pueden ser combinados los diferentes constructores conceptuales para definir modelos válidos.
- La notación asociada a los constructores conceptuales, cuando corresponda.
- La semántica o significado de los constructores conceptuales.

Para la elaboración del metamodelo se utilizará la especificación MOF [35] de OMG. El uso de MOF permitirá que el metamodelo del DSML sea elaborado con el mismo lenguaje que el metamodelo de UML. De esta manera, se evitan inconsistencias en la notación y el significado de los constructores utilizados en ambos metamodelos, y se facilita la identificación de equivalencias.

Una vez definido el metamodelo del DSML, será necesario identificar las equivalencias entre este metamodelo y el metamodelo de UML, encontrando dos nuevos desafíos.

**SEGUNDO DESAFÍO: Resolver diferencias estructurales entre metamodelos.**

Los trabajos presentados por Selic [59] y Wimmer et al. [63], mencionan que las diferencias estructurales entre los metamodelos del DSML y UML impiden una correcta integración semántica, provocando pérdida de expresividad semántica en el perfil UML resultante (en relación al DSML original). Esto es debido a que las diferencias estructurales dificultan la definición de una correcta identificación de equivalencias e integración entre los metamodelos, impidiendo, por ejemplo, que se puedan inferir las extensiones adecuadas que deben realizarse sobre el metamodelo de UML.

En particular, en el trabajo de Wimmer [63] es posible observar que sólo son soportadas las equivalencias entre un (1) elemento del metamodelo del DSML y un (1) elemento del metamodelo de UML (mapeo 1:1), debido a que su propuesta no permite identificar automáticamente las extensiones adecuadas en caso que existan equivalencias con una cardinalidad distinta (1:M, M:1, M:M).

Para entender como afectan las diferencias estructurales en la definición correcta de equivalencias e integración entre metamodelos, analizaremos el ejemplo presentado en la Figura 5.

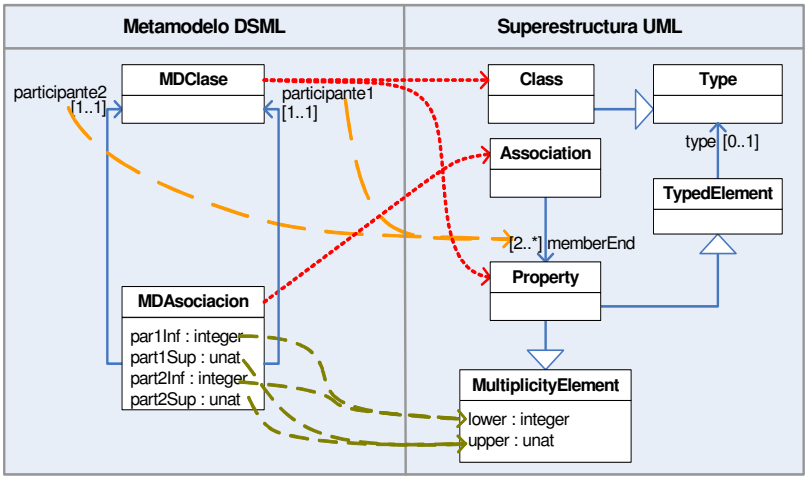


**Figura 5.** Mapeo simplificado entre una asociación binaria genérica y la Superestructura de UML

La Figura 5 presenta un metamodelo del DSML para representar una asociación binaria genérica entre clases (lado izquierdo de la figura). Este metamodelo está compuesto de dos clases: *MDClase* que representa la semántica de una clase tradicional en un esquema orientado a objetos y *MDAsociación* que representa la semántica de una asociación binaria entre dos clases. Para simplificar el ejemplo no se han representado toda la semántica de la clase *MDClase,* que corresponde entre otras cosas, a los atributos y servicios que pose una clase. Las clases que participan en la asociación representan los extremos de la asociación y son especificados mediante las asociaciones *participante1* y *participante2*. La cardinalidad mínima y máxima que tendrá el extremo de la asociación ligado a *participante1* estará dada por los atributos *part1Inf* y *part1Sup* respectivamente, y para *participante2* por los atributos *part2Inf* y *part2Sup*. Este metamodelo de DSML es mapeado con la Superestructura de UML (lado derecho de la figura) para identificar las equivalencias que existen entre ambos metamodelos. La Superestructura de UML también ha sido simplificada para facilitar la compresión del ejemplo. A continuación revisaremos los detalles del mapeo realizado en la Figura 5:

**Clase MDClase:** La clase *MDClase* es mapeada a las clases *Class* y *Property* de la Superestructura de UML.

La equivalencia entre *MDClase* y *Class* parece bastante clara, sin embargo, la equivalencia entre *MDClase* y *Property* puede resultar algo más complicada de entender a simple vista. La clase *Property* representa en UML las propiedades que puede tener un elemento de tipo *Classifier*, entre los que se encuentran las clases y las asociaciones. El mapeo definido corresponde a que en el metamodelo de UML, la clase *Property* es utilizada para identificar las clases que participan en una asociación. La clase representada por la propiedad es identificada mediante la asociación *type*, que representa el tipo soportado por una propiedad de UML. Esta asociación es heredada de la clase *TypedElement*.

El mapeo 1:2 definido entre la clase *DMClass* (1) y las clases *Class* y *Property* no permite que puedan ser identificadas automáticamente las extensiones que deben realizarse. Esto es debido a que no toda la semántica de la clase *MDClass* debe aplicarse a la clase *Property*, ya que en este caso particular, sólo se requiere representar la semántica para identificar las clases que participan en una asociación, y no sería correcto incorporar en la clase *Property* la semántica asociada a tributos y operaciones que puede contener un clase (que es parte de la semántica que tiene *DMClass*). En este caso lo correcto sería incorporar en la clase *Property*, sólo la semántica necesaria para identificar las clases que participan en una asociación binaria y no toda la semántica de la clase *DMClass*, sin embargo, el mapeo definido no brinda esta información, haciendo imposible identificar que parte de toda la semántica de la clase *DMClass* debe ser incorporada en la clase *Property*.

**Clase MDAsociacion:** La clase *MDAsociacion* es mapeada a la clase *Association* de la Superestructura de UML. Las asociaciones *participante1* y *participante2* (que corresponden a propiedades de la clase) son mapeadas a la asociación *memberEnd* que identifica los participantes de la asociación en UML. Finalmente, los atributos de la clase que representan las cardinalidades de los extremos de la asociación son mapeados a los atributos *lower* y *upper* de la clase *MultiplicityElement*, ya que como antes se ha señalado, la clase *Property* es utilizada en el metamodelo de UML para identificar las clases que participan en una asociación, y *Property* hereda de la clase *MultiplicityElement* los atributos para identificar las cardinalidades que corresponden a cada extremo de la asociación.

En este mapeo existe una serie de problemas para determinar como debe ser extendido el metamodelo de UML. En primer lugar, es posible observar una situación similar a la descrita en el mapeo de la clase *MDClase*, ya que la semántica de la clase *MDAsociacion* está distribuida entre dos clases de UML y estas clases no se encuentran relacionadas mediante una asociación de generalización. En este mapeo queda claro que los atributos de la clase

*MDAsociacion* son equivalentes a los atributos de la clase *MultiplicityElement*, pero no se puede determinar si estos atributos deben ser incorporados como nuevos atributos a la clase *Association* de UML.

Una vez analizados los problemas que tiene el mapeo de equivalencias del sencillo ejemplo presentado en la Figura 5, queda claro que deben ser resueltas las diferencias estructurales que impiden una correcta integración del DSML en UML.

Dado que el metamodelo de UML no puede ser modificado, debido a que se pierde compatibilidad con el estándar de UML, La única solución será redefinir el metamodelo del DSML para resolver las diferencias estructurales que impidan realizar un adecuado mapeo de equivalencias. Este mapeo debe proveer información suficiente para inferir de manera automática las extensiones necesarias, para integrar en el metamodelo de UML toda la semántica del metamodelo del DSML. De esta manera, el segundo paso del proceso será redefinir el metamodelo del DSML para obtener un nuevo metamodelo equivalente sin conflictos estructurales con el metamodelo de UML.

Una vez resueltas las diferencias estructurales que impiden realizar un mapeo adecuado entre metamodelos, será necesario definir el mapeo de equivalencias. Si bien este mapeo está orientado a inferir automáticamente las extensiones que deben ser representadas mediante un perfil UML, es recomendable que el mapeo se defina sin considerar aspectos asociados a la definición de perfiles UML. De esta manera se obtendrá una correcta identificación de equivalencias, independiente de los aspectos que deben ser considerados para implementar los perfiles UML, alcanzando los siguientes beneficios:

- **Un mapeo más sencillo de realizar**. Ya que se realizará a nivel de metamodelos manteniendo consistencia con la notación y constructores conceptuales utilizados.

- **No es necesario determinar cómo las extensiones afectan al resto de constructores de UML**. Ya que al momento de definir un perfil UML es necesario determinar como afectan las extensiones definidas, al resto de constructores conceptuales del metamodelo que está siendo extendido.
- **El diseñador del DSML no requiere tener conocimientos de perfiles UML**. La definición de perfiles UML no es una tarea trivial y que pueda realizarse de manera intuitiva. Al poder definir un mapeo independiente de la definición del perfil UML, el diseñador del DSML que es el que mejor conoce la semántica del lenguaje definido, podrá determinar la mejor equivalencia con la semántica de UML que sirva para generar un perfil UML adecuado, sin verse limitado por la falta de experiencia en la definición de perfiles UML.

Por lo tanto, el tercer desafío en la definición de un proceso para la generación de un perfil UML será:

**TERCER DESAFÍO: Identificar equivalencias entre metamodelos, sin tener considerar detalles relacionados con la definición de perfiles UML.**

Para realizar un mapeo de equivalencias que cumpla con el tercer desafío, se puede utilizar una solución como la propuesta por Wimmer [63] que realiza el mapeo mediante un lenguaje especialmente definido a partir de ATL (Atlas Transformation Language) [8]. Sin embargo, utilizar un lenguaje específico para la definición del mapeo puede significar una dificultad añadida a la definición de las equivalencias. En la sección 4.3 de esta tesis, se propone una solución más sencilla para especificar las equivalencias entre metamodelos, reduciendo la complejidad que implica utilizar un nuevo lenguaje específico de mapeo.

**Un mapeo correcto debe considerar lo siguiente:**

- Los elementos del metamodelo del DSML que participan en el mapeo son: clases, asociaciones, atributos, enumeraciones y tipos de datos. No es necesario que todos los elementos del metamodelo del DSML estén mapeados, ya que pueden existir elementos sin equivalencias con el metamodelo de UML. Estos elementos sin equivalencia son nuevos elementos que deberán ser incorporados en el metamodelo de UML mediante el perfil UML.
- Todas las clases del metamodelo del DSML deben ser mapeadas a clases del metamodelo de UML. Esto es debido a que los perfiles UML pueden ser aplicados correctamente solo cuando el DSML es un subconjunto de los constructores de UML.

La solución al tercer desafío propuesto da origen al tercer paso del proceso para la generación de un perfil UML que corresponde a definir un mapeo que indique las equivalencias entre el metamodelo del DSML y el metamodelo de UML.

Una vez identificadas las equivalencias (mapeo) entre los metamodelos, será necesario determinar las extensiones que deben ser realizadas en el metamodelo de UML para integrar la semántica del DSML. Una vez completado los primeros pasos del proceso (Pasos 1, 2 y 3), se tendrá: (1) un metamodelo EMOF que permite una correcta integración con el metamodelo de UML y (2) el mapeo de equivalencias entre ambos metamodelos. Por lo tanto el siguiente desafío es:

**CUARTO DESAFÍO: Identificar automáticamente las extensiones que deben ser definidas sobre el metamodelo de UML.**

La identificación automática de las extensiones evitará que existan inconsistencias semánticas entre el perfil UML resultante y el DSML original, ya que en un DSML de gran tamaño la

identificación manual de extensiones puede provocar que algunas de las extensiones no sean correctamente identificadas o simplemente no se lleguen a identificar. En las propuestas de Lagarde et al. [26] y Wimmer et al. [63] se muestran algunas reglas para identificar las extensiones que deben ser realizadas mediante perfiles UML para integrar la semántica del metamodelo del DSML. Sin embargo, estas reglas son dependientes de la definición de perfiles UML. Una identificación adecuada de las extensiones que deben realizarse sobre UML, debe ser independiente de la especificación de los perfiles UML, de esta manera se consigue:

- Permitir la validación de las extensiones sin necesidad de tener experiencia en la definición de perfiles UML. Con esto los expertos en el DSML, aún cuando no posean experiencia en la definición de perfiles UML, pueden corroborar si tanto el mecanismo para la identificación de extensiones así como las extensiones identificadas, son correctas para representar la semántica del DSML definido.
- Facilitar la adaptación a posibles cambios que sufra la especificación de los perfiles UML. Si la identificación de extensiones fuese dependiente de la definición del perfil UML, ante cualquier cambio en la especificación de los perfiles UML, también sería necesario modificar el esquema de identificación de extensiones.
- Permitir el uso de otros mecanismos de extensión de manera transparente. En caso que se decida cambiar el mecanismo de extensión de UML por otro distinto a los perfiles UML, puede ser reutilizada la identificación de extensiones de manera transparente sin necesidad de realizar modificaciones en el proceso (hasta este punto).

El cuarto desafío es en sí mismo el cuarto paso que se debe agregar al proceso propuesto. Por lo tanto, el cuarto paso del proceso

genérico corresponde a la identificación automática de las extensiones que deben aplicarse sobre el metamodelo de UML.

Una vez identificadas las extensiones que deben ser definidas en el metamodelo de UML, sólo queda la definición del perfil UML que implementa dichas extensiones. Si bien, para realizar este paso sí es necesario tener conocimientos de perfiles UML, la ventaja de separarlo en un paso independiente es que el diseñador del DSML no necesita tener conocimientos de perfiles UML, permitiendo la construcción del perfil UML final por parte de un especialista en perfiles UML o incluso de forma automática. Esto último corresponde al quinto desafío identificado:

**QUINTO DESAFÍO: Generar automáticamente el perfil UML.**

Al observar el quinto desafío se puede apreciar claramente que el quinto y último paso del proceso para la construcción de un perfil UML adecuado, es construir el perfil UML final. La generación automática del perfil UML se consigue a partir de los resultados obtenidos en cada uno de los pasos previos del proceso esto es: (1) el metamodelo del DSML alineado estructuralmente con la Superestructura de UML, (2) el conjunto de equivalencias definidas, y (3) las extensiones identificadas.

Mediante la automatización de este quinto paso, se reduce considerablemente el esfuerzo de mantener un perfil UML asociado a propuestas MDE, que están constantemente desarrollando mejoras que modifican la semántica de sus constructores conceptuales y que cada vez requerirán modificar el perfil UML para adaptarse a estos cambios.

Para realizar la generación automática, se pueden definir un conjunto de reglas de transformación que pueden ser implementadas mediante lenguajes de transformación de modelos como ATL [8] o QVT [40]. Para la correcta definición e implementación de estas reglas se propone separarlas de acuerdo a los constructores conceptuales del metamodelo del DSML (clases, atributos, asociaciones, etc.). Esta separación facilitará la correcta

identificación de las reglas necesarias para una generación correcta y completa del perfil UML, además de facilitar la detección de errores o la incorporación de mejoras en las reglas de transformación implementadas. Algunos aspectos que deben ser considerados en la correcta definición de las reglas de transformación son:

- Los tipos de datos que no son completamente equivalentes con los tipos de datos primitivos definidos en UML, es decir, que no tienen las mismas propiedades o presentan diferencias entre propiedades equivalentes, deben ser tratados igual que los tipos de datos no equivalentes. Esta consideración se debe a que los tipos de datos en UML son especializaciones de la metaclase *Classifier* y no de la metaclase *Class*, y los estereotipos sólo pueden extender elementos de tipo *Class*, por lo que no se pueden definir estereotipos sobre los tipos de datos de UML para ajustar su semántica a los tipos de datos equivalentes del metamodelo del DSML.
- Utilizar las facilidades de extensión que proveen los perfiles UML de acuerdo a la especificación actual de UML [45]. Por ejemplo, la especificación actual de UML permite establecer asociaciones a nivel de estereotipos, por lo que las asociaciones no equivalentes pueden ser incorporadas directamente en el perfil UML generado. Otras características interesantes incorporadas en la versión 2.1 de UML pueden ser revisadas en [59]. Igualmente, en el estado del arte de esta tesis se han analizado algunas de las nuevas características de los perfiles UML, que son relevantes para el desarrollo del proceso propuesto.
- Los nombres de los estereotipos no deben ser iguales a los nombres de las clases que participan en la definición del perfil UML. Los estereotipos son un tipo particular de clases (especialización de la metaclase Class), que al tener el mismo nombre que las clases que conforman el perfil UML definido, pueden presentar problemas al momento de identificar

correctamente los tipos que participan en las asociaciones definidas en el perfil UML.

- Determinar como tratar aquellas propiedades equivalentes del metamodelo del DSML que tengan diferencias de cardinalidad o de tipo que no puedan ser representadas mediante restricciones OCL.

Con este último paso (paso quinto), ya es posible contar con un proceso que genere correctamente un perfil UML asociado a una propuesta MDE. De esta manera, el proceso genérico propuesto para la definición de perfiles UML quedará estructurado de la siguiente forma:

1. Definir el metamodelo EMOF asociado al DSML de la propuesta MDE.
2. Verificar si existen diferencias entre la estructura del metamodelo del DSML y el metamodelo de UML que impidan una correcta integración. En caso de que existan problemas estructurales, redefinir el metamodelo del DSML para alinear su estructura con el metamodelo de UML (Superestructura de UML).
3. Mapear las equivalencias entre el metamodelo del DSML y el metamodelo de UML.
4. Identificar automáticamente las extensiones que deben ser definidas en el metamodelo de UML para integrar la semántica del DSML.
5. Generar automáticamente el perfil UML transformando el metamodelo del DSML. Para realizar la transformación se deben utilizar las equivalencias definidas entre metamodelos y las extensiones identificadas automáticamente.

Un esquema general del proceso obtenido se puede observar en la Figura 6. En esta figura se observa que el segundo paso es

opcional, además los pasos 4 y 5 están destacados para indicar que pueden ser automatizados.

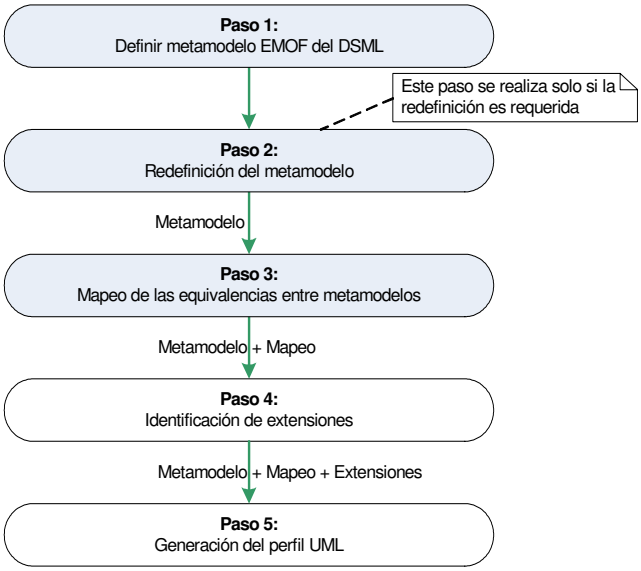


**Figura 6.** Proceso genérico para la construcción de un perfil UML asociado a una propuesta MDE

Con estos cinco pasos ya es posible generar un perfil UML completo, sin embargo, hemos querido incorporar un desafío adicional, este es:

**SEXTO DESAFÍO: Permitir el intercambio de modelos entre herramientas basadas en un DSML propietario y herramientas UML que utilicen el perfil UML generado.**

La solución a este último desafío está en las equivalencias definidas entre el metamodelo del DSML y el metamodelo de UML, y las reglas de transformación para obtener el perfil UML completo. Con esta información es posible conocer exactamente las equivalencias (mapeo) que existen entre el perfil UML y el metamodelo del DSML. Esta información puede ser utilizada para construir una herramienta como la presentada en [1], que mediante la información de mapeo entre el perfil UML y el metamodelo del DSML permite transformar modelos construidos con el perfil UML en modelos basados en el

DSML, y viceversa. Si las reglas de transformación para la generación del perfil UML son automatizadas, esta información de mapeo también se puede generar automáticamente durante la generación del perfil UML. Finalmente, el proceso integrado con la herramienta de intercambio queda representado en la Figura 7.

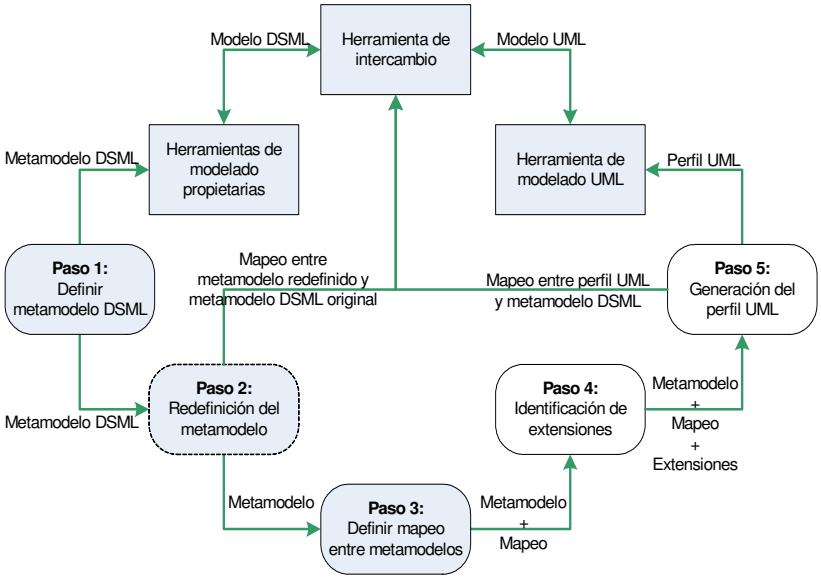


**Figura 7.** Proceso genérico integrado con herramienta de intercambio

En la Figura 7 se muestra como el proceso puede ser utilizado para permitir la integración con herramientas de modelado basadas en DSMLs, mediante una herramienta de intercambio que realice la conversión entre modelos. Esta herramienta de intercambio recibe como entrada la información de mapeo obtenida durante el paso 5, y en caso que sea necesario realizar el paso 2, también se incorpora el mapeo entre el metamodelo redefinido del DSML y el metamodelo original del DSML.

## 3.2. Proceso Completo para Generar Perfiles UML

En esta sección se presentan propuestas concretas para realizar cada uno de los pasos definidos en el proceso genérico obtenido en la sección anterior. Estas propuestas corresponden a los pasos que conformarán finalmente el proceso completo orientado a la generación de perfiles UML. Este proceso completo se ha definido para satisfacer las necesidades de tecnologías MDE con un enfoque industrial, tal como la propuesta OO-Method que da origen a este trabajo. Sin embargo, esto no impide que el proceso pueda ser utilizado por cualquier otro tipo de propuesta MDE que no posea un enfoque industrial, pero tal como se mostrará más adelante, es en las propuestas MDE industriales donde se maximizan los beneficios obtenidos por el proceso completo.

Las versiones de los estándares OMG que han sido considerados en esta Tesis de Master para definir el proceso completo son:

- Especificación MOF versión 2.0 [35].
- Especificación XMI versión 2.1.1 [50].
- Especificación UML versión 2.1.2 [47][48].

**Primer Paso**

El primer paso para conseguir el proceso completo corresponde a la definición del metamodelo del DSML utilizando la especificación MOF [36]. La especificación de MOF está dividida en dos conjuntos. El primer conjunto corresponde a la especificación MOF completa o CMOF (por *complete MOF* en inglés). El segundo conjunto corresponde sólo a los constructores esenciales para la definición de metamodelos denominado MOF esencial o EMOF (*essential* MOF en inglés).

Al analizar la última especificación de UML [47] es posible observar que las capacidades de extensión soportadas por los perfiles UML son muy cercanas a las capacidades de metamodelado soportadas por EMOF. En cambio, muchas de las características de CMOF no son soportadas por los perfiles UML, como por ejemplo, las asociaciones n-arias o la redefinición de propiedades.

Una breve explicación de la similitud que hay entre EMOF y los perfiles UML, es que en EMOF el elemento principal para la construcción de metamodelos está dado por la metaclase *Class*, ver Figura 8. Mientras que en los perfiles UML, el elemento principal para la definición de extensiones es el estereotipo, que es definido como una especialización de la metaclase *Class*, tal como se ha mostrado en el capítulo de estado del arte (sección 2.2). En ambos metamodelo la metaclase *Class* posee la misma especificación, ya que es importada desde la Infraestructura de UML [47].

Con estas consideraciones, podríamos resumir que el primer paso del proceso para construir el perfil UML será la definición del metamodelo del DSML utilizando EMOF.

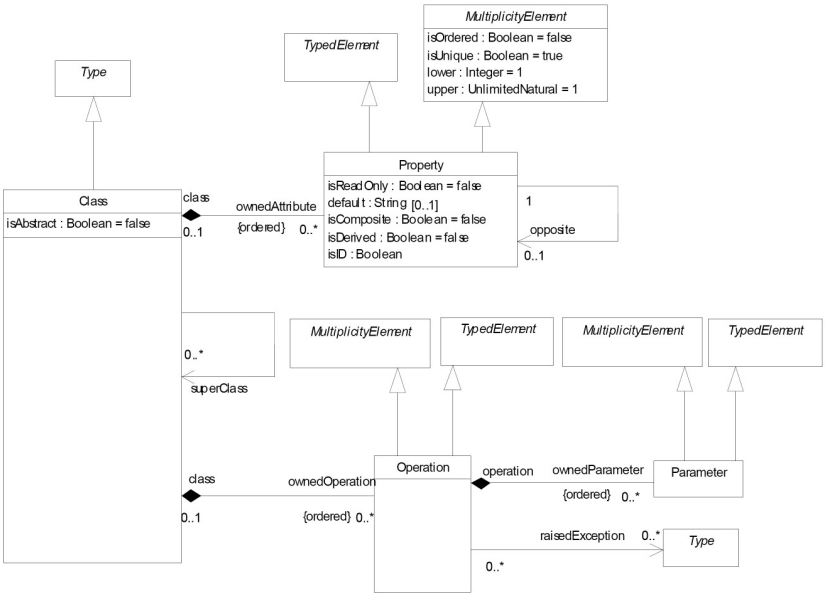


**Figura 8.** Metamodelo EMOF

**Segundo Paso**

De acuerdo a al proceso genérico, el segundo y tercer paso para la generación de un perfil UML, corresponden a:

- Resolver las diferencias estructurales que impidan realizar una correcta integración entre el metamodelo del DSML y el metamodelo de UML corresponde.
- Definir el mapeo de equivalencias entre el metamodelo del DSML y el metamodelo de UML.

Estos dos pasos del proceso genérico serán realizados en un solo paso (el segundo paso) del proceso completo. Para llevar a cabo este segundo paso se propone una solución que permite resolver los problemas estructurales que pueden existir para la integración de metamodelos, al mismo tiempo que se establece el mapeo de equivalencias entre metamodelos. Esta solución, está basada en construir a partir del metamodelo asociado al DSML, un nuevo metamodelo que sirva como entrada para un proceso que genere de manera completamente automática el perfil UML asociado. Este nuevo metamodelo se denomina *Metamodelo de Integración* (*Integration Metamodel* en inglés) y posee la misma semántica que el metamodelo del DSML a partir de cual es generado.

La estructura del Metamodelo de Integración hace posible la definición de un mapeo con el metamodelo de UML, que permite la generación automática de un perfil UML para integrar toda la expresividad semántica de un DSML en UML. Esta información de mapeo estará contenida dentro de la especificación del Metamodelo de Integración y se define utilizando los estándares de OMG. Con esto se consigue que la información necesaria para la generación del perfil UML esté integrada en un único archivo XMI que cumpla con los estándares de OMG y que pueda ser procesado electrónicamente por distintas herramientas. Por lo tanto, la definición del Metamodelo de Integración también implica definir el mapeo entre este metamodelo y el metamodelo del DSML. Esta

información de mapeo será utilizada para permitir el intercambio entre los modelos definidos a partir del DSML y los modelos que se construyan utilizando el perfil UML generado a partir del Metamodelo de Integración. La definición de esta información de mapeo se corresponde con el sexto desafío presentado en el proceso genérico (Sección 3.1).

Para obtener un correcto Metamodelo de Integración se ha definido una propuesta sistemática que facilita su definición. Esta propuesta sistemática, así como mayor detalle del Metamodelo de Integración son presentados en el capítulo cuarto de esta tesis. En este capítulo también se muestran gráficamente los beneficios que presenta el Metamodelo de Integración para la generación de perfiles UML que apoyen a propuestas MDE industriales.

Una vez obtenido el Metamodelo de Integración, se cuenta con toda la información necesaria para generar automáticamente el perfil UML, para conseguir esto, en el proceso completo se implementan los mismos definidos en el proceso genérico, es decir:

- Identificación automática de extensiones.
- Generación del perfil UML final.

**Tercer Paso**

El tercer paso del proceso completo, corresponde a la identificación automática de extensiones que deben ser definidas sobre el metamodelo de UML para integrar la semántica del DSML. En este tercer paso se utiliza el Metamodelo de Integración que incorpora la información del mapeo entre metamodelos.

En este documento, para diferencias los elementos mapeados de los elementos no mapeados, diremos que:

- Son *elementos equivalentes*, aquellos elementos del Metamodelo de Integración que se encuentran mapeados a un elemento del elemento de UML.

- Son *elementos nuevos*, aquellos elementos del Metamodelo de Integración que no están mapeados con algún elemento del Metamodelo de UML. Estos elementos se consideran como nuevos ya que corresponden a aquellos elementos del Metamodelo del DSML que deben ser incorporados dentro del Metamodelo de UML.

Para identificar las automáticamente las extensiones que se debe definir sobre el metamodelo de UML se propone el siguiente mecanismo: utilizar el mapeo de equivalencias para identificar diferencias entre elementos equivalentes, y para identificar los elementos nuevos que deben ser incorporados en el metamodelo de UML. De esta manera, las diferencias encontradas entre los elementos equivalentes, así como los elementos nuevos del metamodelo del DSML, corresponden a las extensiones que deben ser incorporadas en el metamodelo de UML

**Cuarto Paso**

El cuarto paso del proceso completo corresponde a la generación del perfil UML final. Este paso utiliza como entrada el Metamodelo de Integración, y la identificación de las extensiones que deben ser definidas en UML. Esta información es procesada por una serie de reglas que transformarán el Metamodelo de Integración en el perfil UML final. Además, este paso generará la información de Mapeo entre el perfile UML resultante y el Metamodelo de Integración. Con esta información de mapeo será posible construir herramientas de intercambio para tecnologías MDE basadas en metamodelo del DSML, y tecnologías UML que utilicen el perfil UML generado, tal como se muestra en la sección 3.1 que describe el proceso genérico.

El tercer y cuarto paso del proceso completo corresponden a la *Generación Automática del Perfil UML*. La forma en que se realiza esta generación automática es presentada en el capítulo 5. En este capítulo se indica en detalle como se lleva a cabo la identificación

automática de las extensiones y se presentan las reglas de transformación necesarias para obtener automáticamente el perfil UML final.

Finalmente, el proceso completo queda estructurado de la siguiente manera:

- Paso 1**:** Definición del Metamodelo del DSML. Utilizando una herramienta UML con soporte XMI, siguiendo el estándar EMOF definido dentro de la especificación MOF de OMG.
- Paso 2**:** Definición del Metamodelo de Integración de acuerdo a la Superestructura de UML. Este segundo paso corresponde a la solución propuesta para resolver las diferencias estructurales entre el metamodelo del DSML y el metamodelo de UML, así como para definir el mapeo de equivalencias entre metamodelos.
- Paso 3: Identificación de las extensiones que deben ser incorporadas en UML para integrar la semántica del DSML. La identificación de las extensiones se realiza mediante una comparación entre el Metamodelo de Integración y la Superestructura de UML.
- Paso 4: Transformación del Metamodelo de Integración en el Perfil UML final. Este paso se realiza mediante un conjunto de reglas de transformación que además de generar el perfil UML final, generan el mapeo entre el Metamodelo de Integración y el Metamodelo de UML extendido con el perfil UML, para posibilitar el intercambio entre modelos DSML y UML (estendidos con el perfil UML generado).

El primer y segundo paso del proceso corresponde a la *Generación del Metamodelo de Integración*, y requieren de algún tipo de intervención manual. Mientras que toda la complejidad asociada a la identificación de las extensiones requeridas en UML y la transformación del Metamodelo de Integración, es encapsulada en

los pasos tercero y cuarto del proceso que se realizan de manera automática (*Generación automática del perfil UML)*.

La secuencia de ejecución del proceso, así como las entradas y salidas obtenidas en cada paso se pueden observar en el esquema presentado en la Figura 9. En esta figura también se muestra como cada uno de los pasos definidos en el proceso genérico tienen su equivalencia en el proceso completo. En los capítulos siguientes se verán en más detalle los cuatro pasos que componen el proceso completo propuesto.

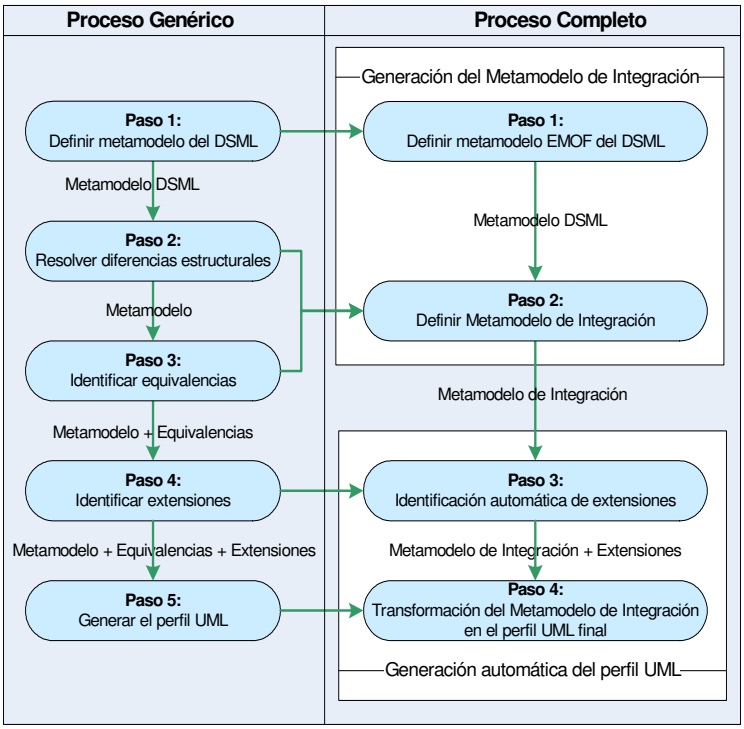


**Figura 9.** Proceso completo para la generación de un Perfil UML asociado a una propuesta MDE.

## 3.3. Conclusiones

Mediante el proceso genérico presentado, se propone un marco base que puede ser utilizado por distintas propuestas MDE para construir perfiles UML destinados a apoyar sus necesidades de modelado. Siguiendo las bases presentadas en este proceso es posible obtener perfiles UML correctos, que cumplen con los estándares dictados por OMG y que además pueden ser validados, ya sea de forma automática o por terceros.

La definición del proceso completo, propone una solución concreta que implementa el proceso genérico presentado. El proceso completo puede ser utilizado ya sea de forma directa por propuestas MDE para implementar sus perfiles UML, o como una propuesta inicial, que se adapte para obtener soluciones particulares más ajustadas a los requerimientos de las propuestas MDE específicas. De esta manera reducir el esfuerzo que otras propuestas MDE deban emplear para implementar un proceso de generación de perfiles UML, al no tener que partir desde cero para obtener una solución completa.

Debido a que el proceso completo se ha estructurado de acuerdo a los pasos definidos en el proceso genérico, este permite realizar validaciones a distintos niveles de la construcción del perfil UML:

- Validación de la correcta semántica del DSML, mediante el metamodelo que describe sus constructores conceptuales (metamodelo del DSML).
- Validación de la definición del perfil UML, mediante la validación de las reglas para la identificación de extensiones y las reglas de transformación para obtener el perfil UML final.

Además, la estructura definida para el proceso completo permite que éste pueda ser utilizado como referencia para otros

mecanismos de extensión o de intercambio entre metamodelos. Por ejemplo, el metamodelo de UML puede ser reemplazado por el metamodelo de otro DSML, o se pueden cambiar las reglas de transformación para implementar las extensiones en otro mecanismo de extensión distinto a un perfil UML [5].

En este capítulo, además de indicar como generar un perfil UML correcto, se muestra como puede ser utilizada la información obtenida durante la generación del perfil UML para permitir la integración con tecnologías basadas en el DSMLs, como por ejemplo, compiladores de modelo o herramientas de modelado propietarias que incorporen características para mejorar las posibilidades de modelado dentro del dominio de aplicación. De esta manera, es posible obtener un perfil UML adecuado y al mismo tiempo aprovechar los beneficios que proveen las herramientas basadas en DSMLs sin incurrir en esfuerzos adicionales.

# *Generación del Metamodelo de Integración*

El Metamodelo de Integración es un tipo especial de metamodelo que se define a partir del metamodelo de un DSML. Este metamodelo posee la misma semántica que el metamodelo del DSML utilizado para su definición, la diferencia radica en que el Metamodelo de Integración posee una estructura que permite identificar automáticamente las extensiones que deben ser definidas sobre el metamodelo de UML, de esta forma poder integrar automáticamente la semántica del DSML en UML.

Para generar el metamodelo de integración, es necesario realizar el primer paso definido en el proceso completo presentado en el capítulo anterior, que corresponde a la definición EMOF del metamodelo del DSML. Para especificar un metamodelo de DSML adecuado, que permita la correcta generación del Metamodelo de Integración, se deben considerar una serie de aspectos que son detallados a continuación.

## *4.1. El Metamodelo del DSML*

El metamodelo del DSML representa la sintaxis abstracta de lenguaje de modelo (DSML) requerido por una propuesta MDE. A la sintaxis abstracta representada por este metamodelo la denominaremos *semántica*, aún cuando autores como Harel [20], afirman que no es correcto utilizar el término semántica para definir las posibilidades de representación de un metamodelo. Sin embargo, en esta tesis es utilizado el término semántica para ser consistentes con la especificación de UML y los trabajos relacionados.

Para realizar la definición adecuada del metamodelo del DSML se puede utilizar cualquier herramienta UML que tenga soporte para el intercambio de modelos en formato XMI [50], ya que los constructores de MOF se corresponden con los constructores asociados al modelo de clases de UML. El contar con la descripción del metamodelo de acuerdo a la especificación XMI permitirá:

- Intercambiar la definición del metamodelo entre diferentes herramientas de modelado.
- Validar que la sintaxis del metamodelo de integración cumple con la especificación UML de OMG.
- Automatizar la generación del perfil UML mediante transformaciones definidas sobre la especificación XMI del Metamodelo de Integración.

A pesar de que muchas herramientas UML dicen soportar la especificación XMI de OMG, es importante señalar que cada herramienta suele realizar su implementación particular de este estándar, perdiendo las ventajas de contar con un estándar de intercambio estandarizado que pueda ser interpretado por diferentes herramientas UML. Esta situación se puede observar claramente en la implementación de la herramienta de intercambio de modelos UML y OO-Method descrita en [28]. Esta herramienta utiliza la especificación XMI para importar los modelos UML, pero ha requerido la implementación de transformaciones específicas para diferentes herramientas de modelado (Rational Rose, Poseidon, y Magic Draw) debido a que estas no cumplen a cabalidad la especificación XMI de OMG.

Por este motivo, para implementar el metamodelo del DSML recomendamos utilizar el proyecto UML2 de Eclipse [12], que bajo nuestro criterio es el que se mejor se ajusta al estándar de OMG.

Otra ventaja que provee el proyecto UML2 es que permite exportar la definición del metamodelo en formato *EMF Core* [9], más conocido como *ECORE*. El proyecto EMF, cuyas siglas significan

*Eclipse Modeling Framework*, es la implementación para eclipse de EMOF. El formato de definición de metamodelos ECORE es utilizado por diferentes tecnologías MDE, como por ejemplo, el proyecto *Eclipse GMF* (*Graphical Modeling Framework*) [11] que permite la generación de editores gráficos para DSMLs.

De todas maneras, el proceso puede ser implementado con cualquier otra herramienta UML que de soporte a XMI. En este caso habrá que asumir el riesgo de que el metamodelo definido no pueda ser intercambiado con otras herramientas UML o tecnologías MDE. Esto también conlleva a que la implementación del proceso sea dependiente de la de la herramienta UML y la forma en que esta maneje los estándares de OMG.

Siguiendo las indicaciones presentadas en el proceso genérico, donde se indican los elementos que deben ser definidos en un metamodelo de DSML, y considerando las posibilidades de representación de EMOF, es posible concluir que para la especificación del metamodelo del DSML los siguientes elementos deben ser incorporados:

- El conjunto de constructores conceptuales. Estos constructores serán definidos mediante clases en el metamodelo.
- El conjunto válido de relaciones que existen entre los constructores conceptuales. Representado mediante asociaciones binarias entre las distintas clases.
- El conjunto de restricciones que controlan como pueden ser combinados los diferentes constructores conceptuales para definir modelos válidos. Representados en cada clase mediante restricciones OCL [37].
- El conjunto de valores para propiedades acotadas a un conjunto finitos de opciones, representados mediante enumeraciones.
- Los tipos de datos soportados por las propiedades de los constructores conceptuales.

No todos los elementos requeridos para la correcta definición del metamodelo del DSML pueden ser especificados utilizando EMOF. En otras palabras, no se puede definir toda la semántica que requiere un DSML. Esto es debido a que la semántica que puede ser representada por EMOF corresponde solo a la *sintaxis abstracta* que gobierna la definición de los modelos y no es posible especificar el significado que tienen los constructores conceptuales. De esta manera, los aspectos que de acuerdo al proceso genérico deben ser considerados en la definición del metamodelo del DSML, y que no pueden ser incorporados en la especificación son:

- La notación asociada a los constructores conceptuales.
- La semántica o significado de los constructores conceptuales.

La información que no puede ser representada en el metamodelo del DSML, deberá ser documentada para entender correctamente la especificación del metamodelo y de esta manera pueda ser utilizada y validada por terceros. La documentación puede ser estructurada de forma similar a la Superestructura de UML, para facilitar de esta manera la identificación de equivalencias. La documentación se realizará por cada clase definida en el metamodelo que corresponde a cada uno de los constructores conceptuales del DSML. En la documentación de cada constructor conceptual del DSML sugerimos incluir la siguiente información:

- Nombre de la clase asociada.
- Descripción y semántica asociada al constructor conceptual.
- Atributos, asociaciones y relaciones de generalización.
- Restricciones OCL.
- Notación
- Representación gráfica de la clase y sus asociaciones.
- Guías de uso y ejemplos

Finalmente, es muy difícil dar guías claras de como se debe modelar el metamodelo del DSML, ya que su especificación es dependiente de los criterios del diseñador del DSML. Sin embargo, para apoyar una correcta especificación recomendamos considerar el artículo de Luoma et al. [27], donde se analizan distintas experiencias para la definición de DSMLs, y el artículo de Henderson-Seller [21], donde se hace un análisis de errores comunes que se cometen al momento de definir metamodelos.

Una vez especificado el metamodelo EMOF del DSML, es posible definir el Metamodelo de Integración a partir de este metamodelo inicial. En las secciones siguientes se verá en más detalle como realizar esta definición para obtener una correcta generación del Metamodelo de Integración.

## 4.2. El Metamodelo de Integración

El Metamodelo de Integración, es un metamodelo definido para integrar la semántica de un DSML en UML. Para realizar esta integración, el Metamodelo de Integración posee una estructura que permite la generación automática de un perfil UML. El Metamodelo de Integración se define a partir del metamodelo del DSML que se desea integrar y que ha sido especificado de acuerdo a los criterios presentados en la sección anterior.

Para la correcta definición del Metamodelo de Integración, es necesario identificar de forma precisa los problemas estructurales que impiden una correcta integración entre el metamodelo del DSML y el metamodelo de UML. Para identificar estos problemas estructurales se han definido un conjunto de reglas que deben ser satisfechas, en caso que alguna de estas reglas no se cumpla, nos encontramos ante un problema estructural que debe ser resuelto. Las reglas son aplicadas en función del mapeo de equivalencias que se haya realizado entre el metamodelo del DSML y el metamodelo de UML. De esta manera, una vez que todas las reglas se cumplan

para todos los elementos del metamodelo del DSML, obtendremos como resultado el Metamodelo de Integración.

A continuación se muestran las reglas que deben ser cumplidas por un Metamodelo de Integración correcto. Para entender mejor estas reglas, analizaremos brevemente el Metamodelo de Integración definido para el metamodelo de la asociación binaria presentado en la sección 3.1. Este Metamodelo de Integración se muestra en la Figura 10.

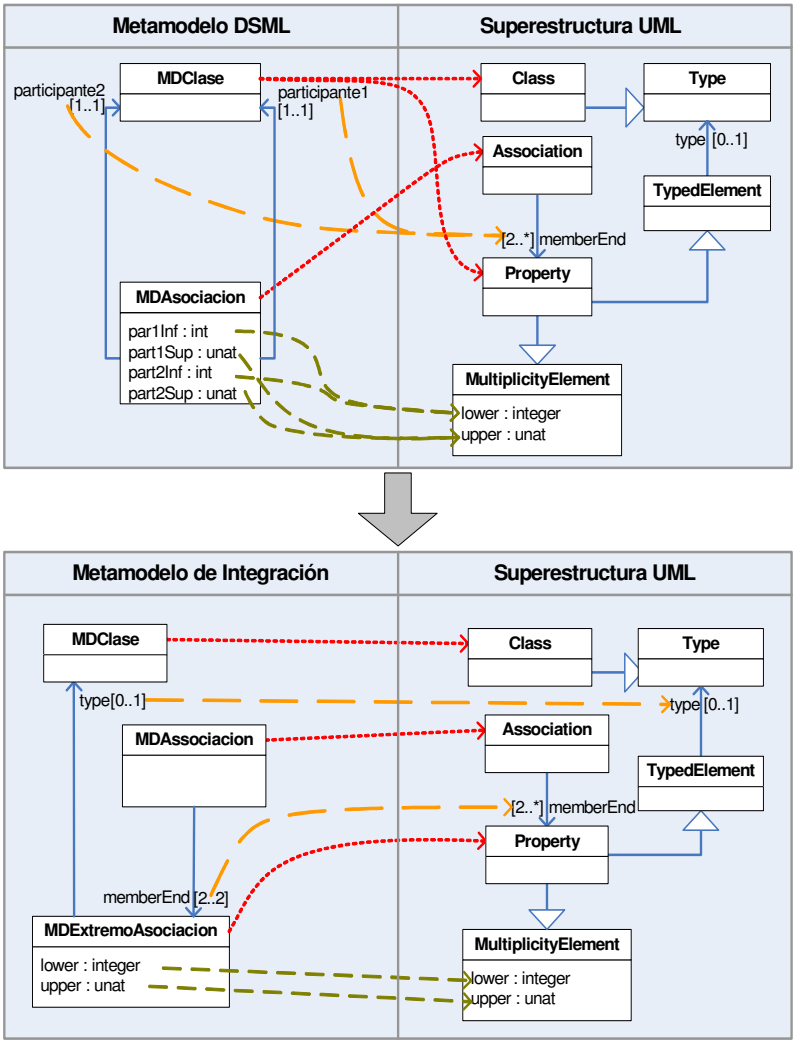


**Figura 10.** Metamodelo de Integración para el metamodelo de la asociación binaria

En el Metamodelo de Integración presentado en la Figura 10 se puede observar que una nueva clase llamada *MDExtremoAsociacion* se ha definido para resolver los problemas de mapeo entre metamodelos. Con esta nueva clase el mapeo resultante para todos los elementos es 1:1, es decir, 1 elemento del Metamodelo de Integración con 1 elemento del metamodelo del DSML.

Al enfocarnos en las clases equivalentes, nos daremos cuenta que los problemas de estructurales del metamodelo inicial ya no existen, permitiendo inferir con precisión la semántica que debe extender cada una de las clases del metamodelo de UML. Para el caso de la clase *MDClase*, el nuevo mapeo indica que toda la semántica de esta clase deberá ser incorporada en la clase *Clase* y en ninguna otra clase del metamodelo de UML. En la clase *MDAsociacion* se produce esta misma situación.

La nueva clase *MDExtremoAsociacion* representa la semántica de las clases *MDClase* y *MDAsociacion* que producía el mapeo 1:M entre los metamodelos. El mapeo 1:M está referido a que una clase o sus propiedades están mapeadas a dos o más clases del metamodelo de UML. La separación de la semántica original en una nueva clase transforma el mapeo 1:M en un mapeo 1:1, ya que es el mapeo 1:M el que impide determinar con precisión que clases de UML deben ser extendidas.

En la nueva clase definida *MDExtremoAsociacion* ocurre una situación interesante de analizar. Esta clase está mapeada con la clase *Property* mientras que sus propiedades están mapeadas con la clase *MultiplicityElement*, obteniendo como resultado un mapeo 1:M. En este caso particular, el mapeo 1:M no presenta problemas para una correcta integración del DSML, ya que la clase *MultiplicityElement* es una generalización de *Property*, con lo que el mapeo 1:M se puede considerar como un mapeo 1:1. En otras palabras, dado que *Property* es también un *Multiplicity element*, las propiedades mapeadas de *MultiplicityElement* son también propiedades de *Property*.

Con este breve análisis del Metamodelo de Integración presentado en la Figura 10, queda claro que los mapeos 1:M deben ser resueltos para conseguir una correcta integración.

Para identificar las situaciones que presentan problemas de integración, se ha definido el siguiente conjunto de reglas. Al cumplir estas reglas, se asegura la especificación de un Metamodelo de Integración correcto.

## Reglas para un Metamodelo de Integración

Las reglas presentadas a continuación son complementarias, y en ningún caso las restricciones definidas en una regla reemplazarán a las de otra:

1. Todas las clases del metamodelo del DSML deben estar mapeadas. Con esta regla se asegura que todos los constructores del DSML pueden ser representados utilizando los constructores de UML. En caso que esta regla no se cumpla, no se podrá obtener un perfil UML adecuado, ya que de acuerdo a las restricciones de extensión de los perfiles UML, su correcta definición requiere que los constructores del DSML puedan ser vistos como un subconjunto de los constructores de UML.
2. El mapeo de las propiedades (atributos y asociaciones) entre una clase equivalente y la clase UML relacionada debe ser 1:1. Esto debido a que los atributos de las clases de la superestructura de UML son univaluados y por lo tanto el valor del atributo de una instancia no pueden representar el valor de dos o más atributos equivalentes. En el caso de las asociaciones ocurre algo similar, no se puede representar con una asociación el valor de dos o más asociaciones equivalentes. Para el ejemplo presentado en la Figura 10 el doble mapeo de propiedades ha quedado resuelto con la definición de la clase *DMExtremoAsociación*.

3. El mapeo debe ser definido entre elementos de la misma naturaleza. Esta restricción es bastante evidente, ya que no tiene sentido que una clase del Metamodelo de Integración sea equivalente a un atributo del metamodelo de UML. De esta manera, el mapeo debe ser siempre de clases con clases, atributos con atributos, etc.

4. Un elemento del metamodelo de DSML debe estar mapeado sólo con un elemento de del metamodelo de UML. En este sentido, los mapeos soportados entre metamodelos pueden ser 1:1 (tal como el ejemplo de la Figura 10) o M:1, o dicho de otra manera X:1 (con $X \geq 0$). El mapeo M:1 corresponde a que varios elementos del Metamodelo de Integración pueden estar mapeados a un elemento del Metamodelo de UML, por ejemplo, varias clases del metamodelo de Integración podrían estar mapeadas a una clase del metamodelo de UML. Dado que las restricciones son complementarias, esta restricción se debe aplicar considerando el resto de restricciones definidas, como la asociada al mapeo de propiedades.

5. Las propiedades (atributos y asociaciones) equivalentes de una clase del metamodelo del DSML deben estar mapeadas a la misma clase UML a la que está mapeada la clase que contiene las propiedades. En caso contrario, el mapeo será válido sólo si las propiedades están mapeadas a propiedades de una clase que es generalización de la clase UML a la que está mapeada la clase equivalente que contiene las propiedades. Esta última regla corresponde a la situación presentada en el ejemplo de la Figura 10, para las propiedades de la clase *MDExtremoAsociación* que ha sido definida para resolver los problemas de integración.

Con este conjunto de reglas se simplifica enormemente la identificación de problemas que impiden la integración entre metamodelos, estas reglas están diseñadas para implementar una herramienta que automatice la identificación de problemas de

integración. Sin embargo, aún con este conjunto de reglas, el obtener un Metamodelo de Integración correcto sigue siendo una tarea complicada, especialmente para determinar como resolver los problemas una vez que han sido identificados. Por este motivo, se ha diseñado una propuesta sistemática que define un conjunto de pasos para generar un Metamodelo de Integración a partir del metamodelo del DSML.

## *4.3. Propuesta Sistemática*

La propuesta sistemática definida para generar el Metamodelo de Integración a partir del metamodelo del DSML está conformada por tres pasos, estos son:

1. Realizar el mapeo de equivalencias entre el metamodelo del DSML y la Superestructura de UML.
2. Validar las reglas OCL.
3. Resolver problemas de mapeo.

Los tres pasos que componen la propuesta sistemática para obtener el Metamodelo de Integración se ejecutan de forma iterativa hasta resolver todos los problemas estructurales. A continuación son detallados cada uno de estos pasos.

**PRIMER PASO: Realizar el mapeo de equivalencias entre el metamodelo del DSML y la Superestructura de UML.**

El mapeo entre metamodelos se debe realizar entre elementos del mismo tipo considerando:

- Clases
- Atributos
- Asociaciones

- Enumeraciones
- Valores literales de enumeraciones
- Tipos de datos.

La información de mapeo debe ser almacenada junto con el Metamodelo de Integración, ya que no tiene sentido fuera de este contexto. Para almacenar la información es conveniente utilizar un recurso que sea compatible con el estándar XMI. De esta manera, obtener un único archivo que pueda ser interpretado por una herramienta computacional, que automatice la generación del perfil UML y que cumpla con los estándares de OMG.

Dado que el Metamodelo del DSML es definido utilizando una herramienta UML, lo más recomendable para especificar el mapeo entre metamodelos, será utilizar un estereotipo definido sobre la clase UML *Element*. Este estereotipo sólo tendrá un valor etiquetado que almacenará el elemento UML asociado al elemento equivalente del Metamodelo de Integración. En el estereotipo se pueden definir una serie de restricciones para garantizar que los elementos relacionados serán siempre del mismo tipo. La definición del estereotipo es presentada en la Figura 11.

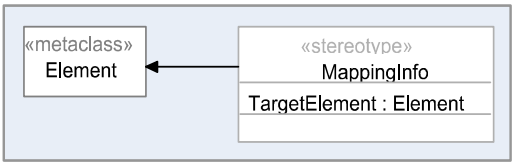


**Figura 11.** Estereotipo para almacenar la información de mapeo del Metamodelo de Integración

Las restricciones OCL para garantizar el correcto mapeo son:

1. Para el mapeo entre clases:

```
self.targetElement->isOclKindOf(Class) implies self-
>oclIsKindOf(Class)
```

2. Para el mapeo entre asociaciones:

```
self.targetElement->isOclKindOf(Property) and
self.targetElement.association->notEmpty() implies
self->oclIsKindOf(Property) and self.association-
>notEmpty()
```

3. Para el mapeo entre atributos:

```
self.targetElement->isOclKindOf(Property) and
self.targetElement.association->isEmpty() implies self-
>oclIsKindOf(Property) and self.association->isEmpty()
```

4. Para el mapeo entre enumeraciones:

```
self.targetElement->isOclKindOf(Enumeration) implies
self->oclIsKindOf(Enumeration)
```

5. Para el mapeo entre valores literales:

```
self.targetElement->isOclKindOf(EnumerationLiteral)
implies self->oclIsKindOf(EnumerationLiteral)
```

6. Para el mapeo entre tipos de datos:

```
self.targetElement->isOclKindOf(DataType) implies self-
>oclIsKindOf(DataType)
```

**SEGUNDO PASO: Validar las reglas OCL.**

Una vez definidas las equivalencias, hay que validar las reglas OCL definidas en el metamodelo del DSML para asegurar que no causan conflicto con las reglas OCL definidas en la Superestructura de UML. Para validar las reglas OCL, se deben considerar las equivalencias definidas en el paso anterior. Las reglas OCL que pueden presentar algún conflicto, son aquellas que restringen elementos que ya están restringidos por las reglas OCL del metamodelo de UML. Esta validación puede ser automatizada, ya que las reglas OCL están diseñadas para ser interpretadas de forma computacional.

**TERCER PASO: Resolver problemas de mapeo.**

Para resolver los problemas de mapeo, lo primero que hay que hacer es identificar aquellos mapeos que no cumplen las reglas definidas en la sección 4.2, para determinar si un Metamodelo de Integración es correcto. Una vez identificados los problemas de mapeo nos podemos encontrar ante alguna de las siguientes situaciones:

1. Clases no mapeadas
2. Mapeo M:1 de las propiedades de una misma clase.

3. Mapeo incorrecto de elementos, por ejemplo, atributos mapeados con asociaciones.
4. Mapeo 1:M.

De las cuatro situaciones conflictivas que se pueden presentar en el mapeo del metamodelo del DSML, las tres primeras pueden ser resueltas corrigiendo el mapeo definido, mientras que la cuarta situación requiere una redefinición del metamodelo del DSML. Es precisamente en esta cuarta situación donde nos centramos para resolver los problemas de mapeo, ya que suele ser la más difícil de resolver.

Un mapeo de tipo 1:M indica que la semántica de una clase del metamodelo del DSML está dividida entre varias clases del metamodelo UML. Esto también puede ser visto como que un constructor conceptual del DSML es en realidad una composición de varios constructores de UML, por lo tanto para alinear correctamente la estructura del metamodelo del DSML con el metamodelo de UML lo que se debe hacer es dividir la semántica de la clase equivalente con mapeo 1:M de acuerdo a la estructura de las clases UML involucradas en el mapeo. En la división deben ser consideradas las reglas OCL definidas en la clase que está siendo dividida para mantener consistencia con el DSML original.

Una vez realizada la división, cada nuevo elemento obtenido en el metamodelo del DSML debe ser mapeado a su respectivo elemento de UML. Luego, para los nuevos elementos y los elementos que han sido modificados, se deben realizar todos los pasos de la propuesta sistemática partiendo desde el primer paso. En la Figura 12 se muestra la aplicación de este tercer paso para obtener el Metamodelo de Integración presentado en la Figura 10.

Tal como se ha señalado anteriormente, esta propuesta sistemática es iterativa y acaba cuando no existen más conflictos de acuerdo al conjunto de reglas establecidas para el Metamodelo de Integración.

Cada vez que se aplique el tercer paso de la propuesta sistemática para resolver los mapeos 1:M, será necesario realizar un redefinición del metamodelo del DSML, por este motivo, antes de realizar la primera redefinición (división de la estructura de una clase) se realizará una copia del metamodelo del DSML y un mapeo entre la copia y el metamodelo original. Esta copia es la que será redefina para obtener el Metamodelo de Integración correspondiente y en cada redefinición se irá modificando el mapeo definido, especificando las operaciones que se deben realizar para mantener la equivalencia con el metamodelo original del DSML.

Además del mapeo 1:M, existe una situación especial que implica la redefinición del metamodelo del DSML. Esta situación especial ocurre cuando se desea igualar la semántica entre una propiedad del metamodelo del DSML y una propiedad del metamodelo de UML. Por ejemplo, cambiar el tipo de un atributo del metamodelo del DSML definido como booleano, por una enumeración (con dos valores literales), y de esta manera igualar su semántica con un atributo UML que está definido como enumeración. En estricto rigor, este tipo de diferencias no imposibilitan la generación automática de un perfil UML correcto, sin embargo, realizar este tipo de cambios mejora la integración, ya que permite aprovechar mayor cantidad de elementos definidos en el metamodelo de UML, reduciendo de esta manera el número de extensiones que deben ser incorporadas a UML. Lo ideal en estos casos sería refinar el metamodelo del DSML, para no tener que incorporar estos cambios en el Metamodelo de Integración, pero en caso que no sea posible modificar el metamodelo del DSML, se pueden incorporar estos refinamientos en el Metamodelo de Integración manteniendo siempre el mapeo con el metamodelo del DSML original.

Una vez termina de iterar la propuesta sistemática se obtiene un correcto Metamodelo de Integración que permite la integración automática del DSML en UML, y además, se obtiene el mapeo entre el Metamodelo de Integración y el metamodelo del DSML. Este mapeo permitirá el intercambio de modelos de acuerdo a la

propuesta definida en el proceso genérico para la generación de un perfil UML (Sección 3.1).

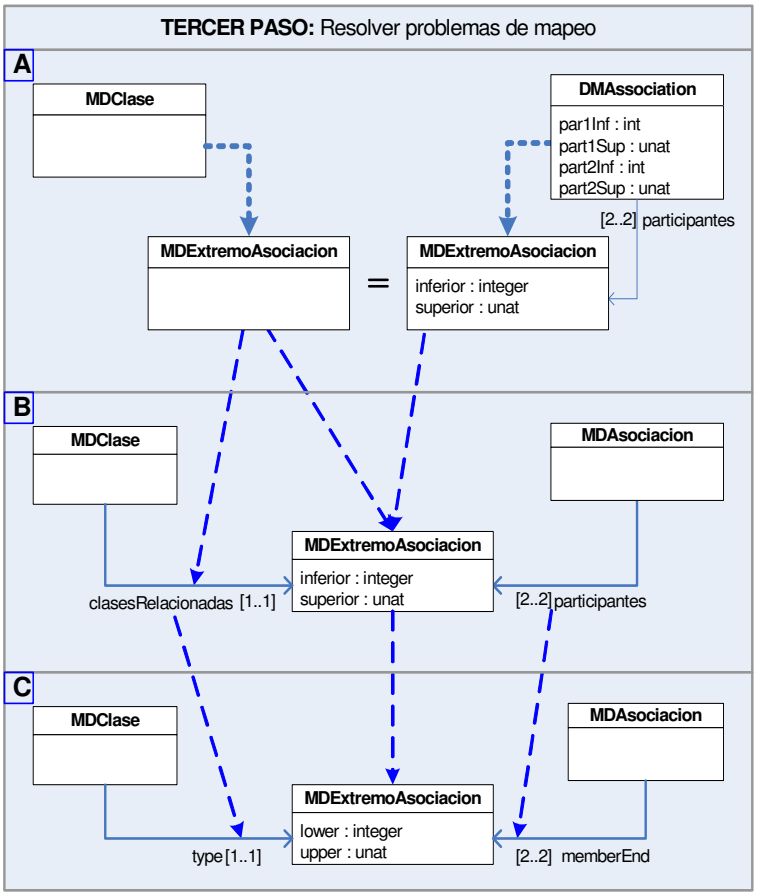


**Figura 12.** Ejemplo del tercer paso para la definición sistemática del Metamodelo de Integración

La Figura 12 muestra como son resueltos los problemas de mapeo 1:M para las clases *MDClase* y *MDAsociacion* presentados en el metamodelo de ejemplo de la Figura 10. Para resolver los problemas de mapeo, la semántica de la clase *Property* que se encontraba en las clases del metamodelo del DSML es separada en una nueva clase llamada *MDExtremoAsociacion*. Esta semántica corresponde a la representación de los extremos de una asociación en UML. Como en ambas clases del metamodelo del DSML la semántica que se extrajo corresponde al mismo constructor

conceptual, sólo se ha definido una nueva clase en el metamodelo del DSML, en caso contrario se habrían obtenido dos nuevas clases. Esta nueva clase contiene todas las propiedades relacionadas con la semántica del extremo de una asociación, es decir, las cardinalidades que representan las cotas superior e inferior de cada extremo (parte A). La relación con los constructores originales se realiza mediante dos asociaciones:

- La asociación *participantes*, que reemplaza a las asociaciones *participante1* y *participante2* del metamodelo original y mantiene la relación entre *MDAsociación* y los extremos de asociación correspondientes (Parte A).
- La asociación *clasesRelacionadas*, que mantiene la relación entre los extremos de asociación y las clases vinculadas (Parte B).

Finalmente, se renombran las asociaciones y propiedades para obtener una definición más cercana a la Superestructura de UML (Parte C). Este último paso es opcional, ya que no afecta la estructura del Metamodelo de Integración, sin embargo, es recomendable para facilitar la identificación de equivalencias entre metamodelos. En este último paso también se puede observar que la asociación *clasesRelacionadas* es equivalente a la asociación UML *type* ya que la clase UML *Class*, que está mapeada a la clase *MDClase*, es una especialización de la clase UML *Type*.

Luego de realizada la redefinición del metamodelo del DSML se deben volver a aplicar los tres pasos de la propuesta sistemática para generar el Metamodelo de Integración, considerando la nueva clase obtenida (*MDExtremoAsociacion*) así como las clases modificadas (*MDClase* y *MDAsociacion*). En la segunda iteración de la propuesta se puede observar que se cumplen todas las reglas que debe cumplir un Metamodelo de Integración, por lo que no es necesaria una nueva iteración. De esta manera, se obtiene el Metamodelo de Integración presentado en la Figura 10.

## *4.4. Beneficios*

La definición de un Metamodelo de Integración, provee una serie de beneficios orientados a facilitar el proceso de generación de los perfiles UML. El beneficio más claro parece ser la posibilidad de generar automáticamente el perfil UML, gracias a que la estructura del Metamodelo de Integración permite obtener un mapeo adecuado para inferir automáticamente las extensiones que deben definirse sobre UML, y de esta manera integrar la semántica del DSML en UML.

Además del beneficio de la integración con UML (que es el objetivo principal del Metamodelo de Integración), existen otros beneficios que no son tan evidentes, pero si son relevantes al momento de utilizar perfiles UML para satisfacer las necesidades de modelado de propuestas MDE, estos son:

- **La definición de un Metamodelo de Integración es más intuitivo que la definición de un perfil UML directamente sobre la Superestructura de UML.** Debido a que un metamodelo de DSML puede tener una especificación que no permita la integración automática con la Superestructura de UML, será necesario definir de forma manual las extensiones que no puedan ser inferidas automáticamente. El inferir y definir estas extensiones de forma manual es una tarea mucho más compleja que la definición del Metamodelo de Integración, no solo por que hay que resolver los problemas de la semántica que no puede ser integrada directamente, sino que además, la definiciones de perfiles UML suele ser más compleja que la definición de un metamodelo. Este aumento de complejidad se debe, entre otras cosas, a que es necesario tener conocimiento de cómo definir extensiones mediante perfiles UML, es necesario determinar el impacto que tendrá cada nueva extensión sobre el DSML que se está definiendo y además sobre los constructores conceptuales

del metamodelo de UML. Por ejemplo, algo tan simple como la definición de un estereotipo e identificar correctamente la clase que este extiende, puede resultar bastante complejo para alguien que no tenga experiencia en la definición de perfiles UML y más aún si el DSML que se intenta integrar en UML posee un número significativo de constructores conceptuales. De esta manera, el contar con un Metamodelo de Integración y una propuesta sistemática para su definición, permite realizar la especificación a nivel de metamodelos, que será mucho más intuitiva para el diseñador del DSML.

- **La definición del Metamodelo de Integración ayuda a aislar la complejidad asociada a las decisiones de diseño de los perfiles UML.** En el beneficio anterior ha quedado claro que la definición de un Metamodelo Integración es más intuitivo, debido a que es posible realizar una especificación a nivel de metamodelos que permita generar el perfil UML automáticamente. Sin embargo, la generación de este perfil UML no es trivial, ya que requiere considerar una serie de decisiones de diseño. Estas decisiones de diseño estarán ligadas a poder generar un perfil UML que mantenga consistencia con la semántica del DSML original para obtener un perfil UML correcto. Con el proceso de generación de perfiles UML propuesta en esta tesis (proceso completo), estas decisiones de diseño son encapsuladas en las reglas de transformación dentro del proceso de generación automática del perfil UML. De esta manera es posible definir el Metamodelo del DSML y el metamodelo de Integración de forma transparente, sin tener que considerar aspectos de diseño de los perfiles UML que suelen ser bastante complejos. Luego, estos aspectos de diseño pueden ser definidos por un experto en perfiles UML y gracias a que pueden ser automatizados mediante reglas de transformación, bastará con que solo se definan una vez sin tener que aplicar de forma manual estos criterios de diseño cada vez que se introduzcan cambios en el DSML. La automatización de la identificación de extensiones y la transformación del Metamodelo de Integración

en el perfil UML final, dan paso al tercer beneficio que queremos destacar.

- **El esfuerzo de incorporar cambios en el DSML se reduce considerablemente.** Este último beneficio está ligado especialmente a las propuestas MDE industriales, que incorporan constantemente mejoras para satisfacer las necesidades de sus clientes. Estas mejoras conllevan cambios en los DSMLs, los que que lógicamente deben ser reflejados en los perfiles UML asociados. De acuerdo a los dos beneficios previos, el definir estos cambios de forma manual e intuitiva sobre la Superestructura de UML es una labor muy cotosa, mientras que la definición de estos cambios sobre el Metamodelo de Integración resulta mucho más intuitiva y no requiere considerar aspectos de diseño de los perfiles UML. Pero existe otra razón por lo que el esfuerzo de introducir cambios se ve reducida gracias al Metamodelo de Integración, esta es: El número de modificaciones que se requieren para introducir un cambio en el DSML es menor en el Metamodelo de Integración que en el perfil UML. En otras palabras, Un cambio en el Metamodelo de Integración suele implicar varios cambios en el perfil UML.

Para ejemplificar estos tres beneficios planteados, a continuación se muestra un breve ejemplo donde se aplican dos reglas de transformación que forman parte del proceso de generación automática del perfil UML. En el ejemplo se presenta una situación inicial de un Metamodelo de Integración y el perfil UML asociado. Luego, el Metamodelo de Integración inicial es modificado para mostrar el impacto del cambio en el perfil UML.

La situación inicial de modelado y mapeo es representada de forma genérica en la Figura 13. A la izquierda de esta figura se muestra el Metamodelo de Integración con dos clases: *Clase1* y *Clase2*, cada clase contiene una propiedad: *propiedad1* y *propiedad2* respectivamente. Las dos clases presentadas en el

Metamodelo de Integración se encuentran mapeadas a una misma clase de la Superestructura de UML llamada: *UMLClase1*.

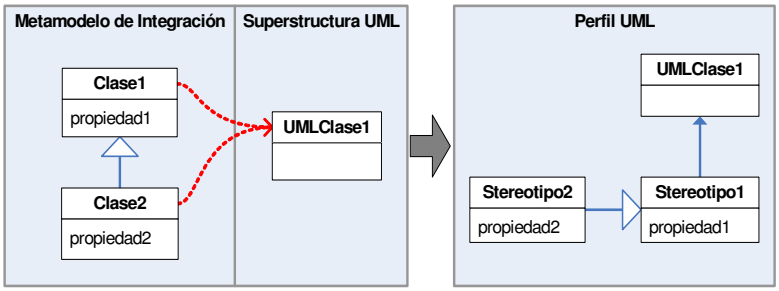


**Figura 13.** Ejemplo del impacto de las modificaciones en un Metamodelo de Integración – Situación inicial.

A la derecha de la Figura 13 se muestra el perfil UML obtenido a partir de este Metamodelo de Integración. Para obtener este perfil UML se ha aplicado la siguiente regla de transformación:

**PRIMERA REGLA:** Si hay dos *clases equivalentes* asociadas mediante una *nueva generalización* y ambas clases equivalentes están referenciando a la misma clase UML entonces:

- Definir un estereotipo por cada *clase equivalente* y una asociación de generalización entre los estereotipos de acuerdo a la *nueva generalización* definida en el Metamodelo de Integración.
- Sólo el estereotipo que representa a la *clase equivalente* padre extiende a la clase UML referenciada. El estereotipo que representa a la *clase equivalente* hija no extiende a la clase UML ya que la extensión está implícita en la asociación de generalización definida

Al introducir un cambio en el Metamodelo de Integración definido en la situación inicial de este ejemplo, entonces será necesario reflejar dicho cambio en el perfil UML. El cambio que será introducido en el Metamodelo de Integración es bastante simple y corresponde a cambiar el mapeo de la clase *Clase2* a una clase UML distinta

(*UMLClase2*). De esta forma, se obtiene el mapeo definido en la Figura 14. En esta Figura también se muestra el perfil UML resultante producto del cambio realizado.

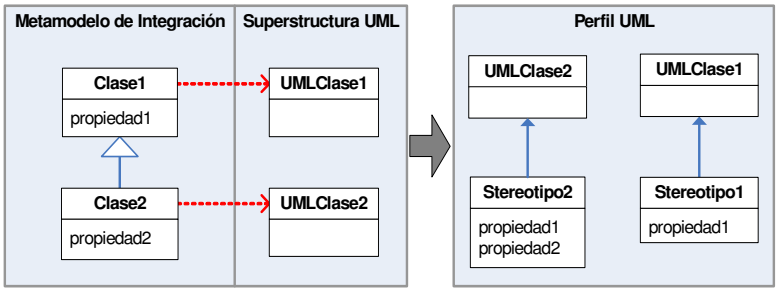


**Figura 14.** Ejemplo del impacto de las modificaciones en un Metamodelo de Integración – Metamodelo modificado.

Para obtener el perfil UML presentado en la Figura 14, ya no puede ser utilizada la primera regla de transformación presentada, en este caso la regla de transformación que es aplicada es la siguiente:

**SEGUNDA REGLA:** Si hay dos *clases equivalentes* asociadas mediante una *nueva generalización* y ambas *clases equivalentes* están referenciando a distintas clases UML entonces:

- Definir un estereotipo por cada *clase equivalente* y las propiedades (atributos y asociaciones) heredadas son duplicadas.
- Definir las extensiones de cada estereotipo con su correspondiente clase UML, de acuerdo a las *clases equivalentes* que representan.

En este caso la asociación de generalización no es representada, ya que conlleva a problemas en la correcta integración de la semántica del DSML en UML. Esta situación es explicada en más detalle en el capítulo de generación automática del perfil UML (Capítulo 5).

Analizando la Figura 14, se puede observar que un simple cambio en el mapeo del Metamodelo de Integración produce cuatro cambios en el perfil UML resultante, estos son:

1. La asociación de generalización entre estereotipos es eliminada.
2. Debe ser importada en el perfil UML la clase *UMLClass2.*
3. Se debe definir la extensión entre el estereotipo *Estereotipo2* y la clase UML *UMLClass2*.
4. Se debe incorporar el valor etiquetado *propiedad1* en el estereotipo *Estereotipo2*.

El ejemplo presentado, además de graficar como se reduce el esfuerzo de introducir cambios en el DSML (un cambio del Metamodelo de Integración corresponde a cuatro cambios en el perfil UML), también muestra que la definición de estos cambios es mucho más intuitiva ya que es mucho más simple definir un mapeo que una extensión mediante estereotipos. También en el ejemplo se puede observar como el cambio se ha realizado sin considerar aspectos de diseño asociados a perfiles UML, ya que estos se encuentran encapsulados en las reglas de transformación definidas.

En un contexto industrial, donde los cambios en los DSML suelen producirse con mayor frecuencia, y además estos cambios son de una magnitud mucho mayor en el ejemplo presentado, se hacen más evidentes los beneficios de definir un Metamodelo de Integración. De esta manera queda justificado el esfuerzo adicional necesario para definir el Metamodelo de Integración con el fin de automatizar la generación del perfil UML final.

## *4.5. Conclusiones*

En este capítulo se han presentado una serie de criterios que deben ser considerados para obtener un metamodelo de DSML adecuado.

Además, se indican las tecnologías que deben ser utilizadas para facilitar la aplicación de este metamodelo en un proceso de integración con UML.

El Metamodelo de Integración definido a partir del metamodelo del DSML, provee una solución para automatizar la integración de un DSML en UML. Además, el conjunto de reglas que debe satisfacer un Metamodelo de Integración y la propuesta sistemática presentada para su definición, dan una solución más robusta de cara a aplicar efectivamente el Metamodelo de Integración, posibilitando la construcción de herramientas que agilicen y den soporte su correcta definición.

Aún cuando los beneficios que provee el Metamodelo de Integración se maximizan en propuestas MDE con un enfoque industrial, este metamodelo puede ser igualmente aplicado en cualquier otro entorno de aplicación.

Finalmente, el Metamodelo de Integración marca una gran diferencia entre el proceso completo propuesto y otros trabajos relacionados que solo consiguen automatizar parcialmente la generación de un perfil UML. Aplicando el concepto de Metamodelo de Integración en estos trabajos, sería posible mejorar considerablemente estas propuestas, en busca de su automatización completa, como por ejemplo, la propuesta de Wimmer et al. presentada en [63].

# *Generación Automática del Perfil UML*

Este capítulo muestra como puede ser automatizada la generación del perfil UML, a partir del Metamodelo de Integración definido en el segundo paso del proceso completo propuesto.

La generación automática del perfil UML corresponde a los dos últimos pasos de proceso completo (tercer y cuarto paso). Estos pasos involucran: 1) la identificación automática de las extensiones que deben ser definidas sobre el metamodelo de UML y 2) la transformación del Metamodelo de Integración en el perfil UML final.

La identificación automática de extensiones, a pesar de identificar los elementos que deben ser definidos mediante un perfil UML, no requiere tener conocimientos en como definir perfiles UML para su correcta implementación. Esto es debido a que la identificación de las extensiones se basa en el reconocimiento de diferencias entre el Metamodelo de Integración y el Metamodelo de UML. Al ser la identificación de extensiones independiente de la definición del perfil UML, su validación por parte de terceros se simplifica, ya que su representación está ligada al Metamodelo de Integración, permitiendo ver de forma más clara como debe ser integrada la semántica en el Metamodelo de UML.

La transformación del Metamodelo de Integración en el perfil UML final, sí requiere tener conocimientos de cómo definir perfiles UML correctamente. Es precisamente en las reglas de transformación definidas en este cuarto paso, que son encapsulados todos los aspectos de diseño para la generación del perfil UML adecuado. Precisamente, por la complejidad que involucra el correcto diseño de un perfil UML, en este capítulo se definen un conjunto de reglas de transformación que pueden ser utilizadas por cualquier propuesta MDE para obtener un perfil UML completo y correcto.

## 5.1. Identificación Automática de Extensiones

Para integrar toda la semántica de un DSML en UML, será necesario identificar las extensiones que deben ser definidas sobre el metamodelo de UML. Esta identificación se realiza mediante una comparación entre el Metamodelo de Integración y el metamodelo de UML, que permite determinar las diferencias entre ambos metamodelos. Son precisamente estas diferencias, la clave para identificar las extensiones que deben ser definidas sobre UML, ya que las extensiones serán los cambios o personalizaciones que deben ser introducidos en el metamodelo de UML para que ya no existan diferencias entre los metamodelos.

Para efectuar una correcta comparación entre los metamodelos en busca de estas diferencias, se utiliza la información de mapeo definida en el Metamodelo de Integración, considerando:

- **La identificación de *elementos nuevos*.** Es decir, aquellos elementos que no tienen correspondencia con el metamodelo de UML. De acuerdo a las características del Metamodelo de Integración, estos elementos pueden ser: atributos, asociaciones, enumeraciones, valores literales, y tipos de datos. Las clases no pueden ser nuevos elementos, ya que en un Metamodelo de Integración correcto todas las clases se encuentran mapeadas con una clase del metamodelo de UML.
- **La identificación de diferencias en la *cardinalidad* de las *propiedades equivalentes*.** Considerando que las propiedades equivalentes son los atributos o asociaciones que tienen correspondencia con el metamodelo de UML. La identificación de las diferencias de cardinalidad es bastante intuitiva, ya que corresponde a una comparación entre valores numéricos.
- **La identificación de diferencias en el tipo de las propiedades equivalentes.** La identificación de las diferencias

de tipo en las propiedades equivalentes no resulta tan intuitivo como la identificación de diferencias de cardinalidad, ya que para realizar esta identificación se debe utilizar la información de mapeo, comparando si los tipos de las propiedades de UML están mapeados a elementos equivalentes en el Metamodelo de Integración. A continuación analizaremos en más detalle como se debe realizar la comparación de los tipos para propiedades equivalentes.

En un metamodelo EMOF el tipo de una propiedad puede estar determinado por: una clase del metamodelo en el caso de las asociaciones, y por un tipo de datos o enumeración en el caso de los atributos. Considerando el caso de las asociaciones, tendremos que una asociación equivalente no habrá diferencia de tipo cuando: El tipo de la asociación del Metamodelo de Integración (*asociación equivalente*) sea equivalente al tipo de la de la asociación UML referenciada. Esta situación se puede observar en la Figura 15. En esta figura el tipo de la asociación equivalente *asociacion1* corresponde a la clase equivalente *Clase1* y dado que *Clase1* es equivalente a *UMLClase1* que corresponde a la asociación UML *UMLasociacion1*, entonces se obtiene que el tipo de *asociacion1* es igual (equivalente) al tipo de *UMLasociacion1*. En caso que la clase equivalente *Clase1* estuviese mapeada a una clase UML distinta a *UMLClase1*, entonces habría una diferencia de tipo entre las asociaciones equivalente (Figura 16).

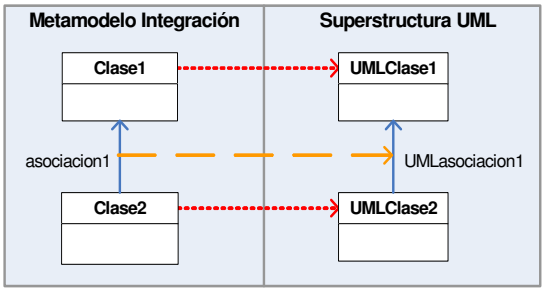


**Figura 15.** Ejemplo de asociación equivalente sin diferencia de tipo

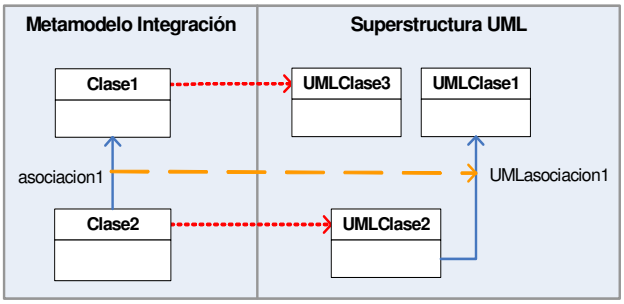


**Figura 16.** Ejemplo de asociación equivalente con diferencia de tipo

En el caso de los atributos, para identificar las diferencias de tipo se deben considerar los tipos de datos y enumeraciones. Para los tipos de datos es posible encontrar dos tipos de aplicación. La primera es que se utilicen los tipos de datos primitivos definidos en la infraestructura de UML, que tambiién son utilizados por la superestructura de UML y por MOF, estos son: Integer, Boolean, String, y UnlimitedNatural. La otra posibilidad es que se definan nuevos tipos de datos mediante el elemento *DataType*, los que también pueden ser equivalentes a un tipo primitivo de datos. En el ejemplo de la Figura 17, se muestra un Metamodelo de Integración con la clase *Clase1* que tiene tres atributos: *atr1*, *atr2* y *atr3*. Estos atributos tres atributos son equivalentes a *umlAtr1*, *umlAtr2* y *umlAtr3* de la clase UML *UMLClase1*. Para el caso de *atr1* y *umlAtr1* no existen diferencias de tipo, ya que ambos atributos utilizan el tipo primitivo *Integer*. Para el caso de *atr2* y *umlAtr2* si existe una diferencia de tipo, ya que *atr2* es de tipo *String*, mientras que *umlAtr2* es de tipo *Boolean*. Finalmente, para el caso de *atr3* y *umlAtr3* parece claro que también existe una diferencia de tipo, ya que el tipo de *atr3* es *Fecha*, mientras que el tipo de *umlAtr3* es *String*. Sin embargo, para asegurar que existe una diferencia de tipo habrá que revisar el mapeo para el tipo de dato *Fecha*, ya que si este tipo de dato es equivalente al tipo de dato primitivo *String* no habría diferencia de tipos. Como el tipo de dato *Fecha* no se encuentra mapeado, entonces confirmamos que se trata de un tipo de dato distinto.

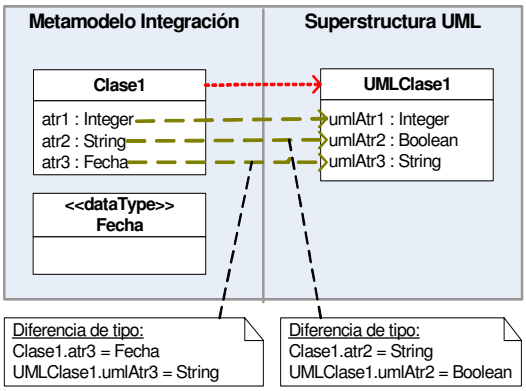


**Figura 17.** Ejemplo de comparación de tipos entre atributos

Para el caso en que los tipos de los atributos estén definidos mediante enumeraciones, hay que aplicar un criterio similar al de las asociaciones. En este caso, se debe utilizar la información de mapeo de las Enumeraciones para determinar si el tipo del *atributo equivalente* es igual al tipo del atributo UML referenciado, la Figura 18 ejemplifica esta situación.

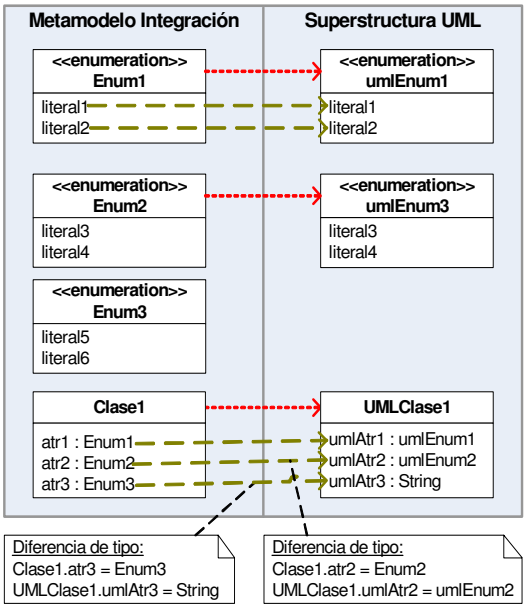


**Figura 18.** Ejemplo de comparación de tipos entre enumeraciones.

En el Metamodelo de Integración presentado en la Figura 18 se pueden observar 3 enumeraciones distintas, que corresponden a los

tipos de los tres atributos definidos en la clase *Clase1*. En el caso del atributo equivalente *atr1*, no existen diferencias de tipo con el atributo UML *umlAtr1*, ya que el tipo del atributo *atr1* corresponde a *Enum1*, que es equivalente al tipo de *umlAtr1* que corresponde a *umlEnum1*. Para el caso del atributo *atr2*, que es equivalente al atributo UML *umlAtr2*, se puede observar que existe una diferencia de tipo debido a que *Enum2*, que corresponde al tipo de *atr2*, no es equivalente a *umlEnum2*, que corresponde al tipo de *umlAtr2.* Finalmente, para el caso del atributo *atr3* del Metamodelo de Integración, queda claro que su tipo es distinto al del atributo de UML referenciado *umlAtr3*, ya que en el caso de *atr3* el tipo corresponde a una enumeración, mientras que en el caso de *umlAtr3* corresponde a un tipo primitivo de dato.

Para entender mejor cómo las diferencias encontradas mediante la comparación entre metamodelos permiten la correcta identificación de las extensiones que deben definirse en el metamodelo de UML, se analizará el ejemplo presentado en la Figura 19. En este ejemplo sólo se presentan dos tipos de diferencias, nuevos elementos y diferencias de cardinalidad de propiedades equivalentes.

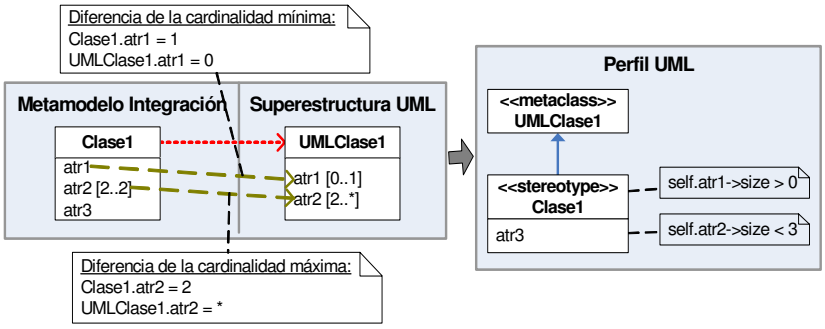


**Figura 19.** Ejemplo de identificación de extensiones – Elementos nuevos y diferencias de cardinalidad

En la Figura 19 se presenta un ejemplo genérico de una clase llamada *Clase1* definida en el Metamodelo de Integración que es equivalente a la clase *UMLClase1* del metamodelo de UML (Superestructura UML). Los atributos *atr1* y *atr2* son equivalentes

en ambas clases, mientras que el atributo *atr3* de *Clase1* no tiene equivalencia en el metamodelo de UML. Para extender la semántica de la clase UML *UMLClase1* se define el estereotipo *Clase1*, mediante este estereotipo se definen las extensiones que eliminan las diferencias entre ambos metamodelos.

La extensión definida para resolver la diferencia de cardinalidad entre *Clase1.atr1* y *UMLClase1.atr1*, implica aumentar la cardinalidad mínima de *UMLClase1.atr1* para que no pueda ser 0. Esta extensión es utiliza la siguiente regla OCL asociada al estereotipo *Clase1*: `self.atr1->size > 0`.

La segunda extensión necesaria estará dada por la diferencia de cardinalidad entre *Clase1.atr2* y *UMLCLase1.atr2*. Esta situación es similar a la anterior y se resuelve mediante la definición de una segunda regla OCL sobre el estereotipo *Clase1,* esta regla es: `self.atr2->size <3`.

Finalmente, una tercera extensión es definida para representar la la diferencia dada por el nuevo atributo *atr3*. Esta tercera extensión consiste en definir el valor etiquetado *atr3* en el estereotipo *Clase1*.

Con estas tres extensiones se eliminan las diferencias entre el Metamodelo de Integración y el Metamodelo de UML, en otras palabras, se ha integrado la semántica del DSML en UML.

La forma en que se define el perfil UML correcto a partir de las extensiones identificadas no es relevante en este punto, sin embargo, en el ejemplo se ha incorporado la definición del perfil equivalente para entender mejor como la identificación de diferencias permite determinar las extensiones que deben realizarse sobre el metamodelo de UML.

Siguiendo el ejemplo de la asociación binaria genérica, para el Metamodelo de Integración definido en la Figura 20 obtendremos la identificación de diferencias presentada en la Tabla 1. Para simplificar la información presentada en esta tabla, las propiedades heredadas son consideradas como propiedades de la clase que las hereda, sin hacer referencia a la clase padre sobre la que están

originalmente definidas las propiedades heredadas. La información de comparación muestra las diferencias identificadas entre propiedades equivalentes.

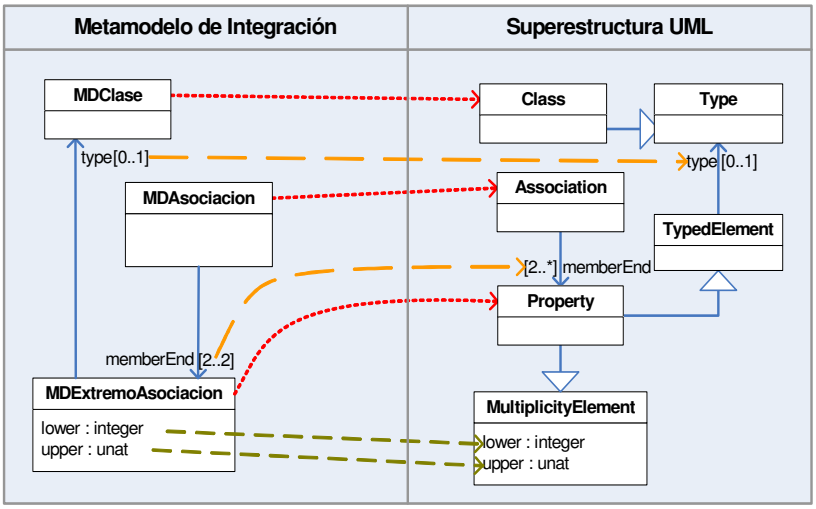


**Figura 20.** Metamodelo de Integración para la asociación binaria

**Tabla 1.** Resultado de la comparación para el Metamodelo de Integración presentado en la Figura 20

| Metamodelo Integración | Comparación |
|---|---|
| MDAsociacion | |
| .memberEnd | Diferente cardinalidad superior:<br>MDAsociacion.memberEnd = 2<br>Association.memberEnd = * |
| MDExtremosAsociacion | |
| .type | Diferente cardinalidad inferior:<br>MDExtremoAsociacion.type = 1<br>Property.type = 0<br><br>Diferente tipo:<br>MDExtremoAsociacion.type = DMClass<br>Property.type = Type |

•

Al observar los resultados de la comparación presentados en la Tabla 1, es posible observar que *MDAsociacion.memberEnd* tiene el

mismo tipo que *Association.memberEnd,* ya que de acuerdo a la información de mapeo definida en el Metamodelo de Integración, el tipo de *MDAsociacion.memberEnd*, que corresponde a *MDExtremoAsociacion*, es equivalente a la clase UML *Property*, que corresponde al tipo de *Association.memberEnd*. En cambio, en el caso de *MDExtremoAsociacion.type* y *Property.type*, el tipo es distinto porque la clase *MDClase* no es equivalente con la clase UML *Type*

Las diferencias de cardinalidad son identificadas analizando la cota superior e inferior de la cardinalidad del atributo equivalente con el atributo UML referenciado. De acuerdo a la tabla de resultados, para Metamodelo de Integración presentado en la Figura 20, los atributos equivalentes *MDAsociacion.memberEnd* y *MDExtremoAsociacion.type* presentan diferencias de cardinalidad en relación a los atributos UML referenciados, estos corresponden a *Association.memberEnd* y *Propert.type* respectivamente.

Finalmente, estas diferencias son utilizadas en el siguiente y último paso del proceso completo para generar el perfil UML. En este último paso el Metamodelo de Integración es transformado en el perfil UML final, mediante reglas de transformación que utilizan la información de mapeo y la identificación de diferencias. En la siguiente sección se explican en detalle las como se realiza la transformación del Metamodelo de Integración y de esta manera completar la generación automática del perfil UML.

## *5.2. Transformar el Metamodelo de Integración*

La transformación del Metamodelo de Integración para obtener el perfil UML final, corresponde al último paso de la generación automática del perfil UML. Para realizar esta transformación se han definido un conjunto de reglas de transformación que utilizan como información de entrada: El Metamodelo de Integración, y las diferencias identificadas en el paso previo del proceso (sección 5.1).

Es importante destacar que la información de mapeo definida en el Metamodelo de Integración, tiene especial importancia para ejecutar correctamente las reglas de transformación definidas.

Además del perfil UML, la transformación del Metamodelo de Integración genera la información de mapeo entre el Metamodelo de Integración y el Metamodelo de UML extendido con el Perfil UML generado. Esta información de mapeo permitirá la integración entre modelos definidos con el Perfil UML y modelos basados en el DSML.

A continuación se muestran las reglas para transformar el Metamodelo de Integración en un perfil UML completo. Para facilitar su compresión, las reglas han sido agrupadas por constructores conceptuales y las posibles situaciones de modelado. Este agrupamiento, además de facilitar la compresión, permite validar que se han definido todas las reglas necesarias para transformar correctamente Metamodelo de Integración.

Las reglas de transformación son presentadas con la siguiente estructura:

- **Regla [*Número*]: [*Descripción*]:** Mantiene un correlativo de las reglas, seguido de una descripción que explica brevemente la operación de la regla.
- **Regla Identificación:** Indica los elementos que serán considerados en la regla de transformación.
- **Condición Identificación:** Condición adicional que se debe cumplir para aplicar la transformación.
- **Transformación:** Acción (es) que se ejecutan.
- **Consideraciones Adicionales:** Indica si existe algún elemento adicional que considerar para aplicar correctamente la regla de transformación.
- **Refinamiento perfil UML:** Indica si se debe realizar alguna operación adicional una vez generado el perfil UML. Este refinamiento se aplica una vez se han aplicado todas las reglas de transformación.

- **Mapeo Perfil UML:** Indica el mapeo entre el Metamodelo de Integración y Metamodelo de UML extendido por el perfil UML, una vez aplicada la regla de transformación.
- **Ejemplo:** Figura que grafica la regla de transformación. En caso que la figura sea utilizada para ejemplificar varias reglas, se indicará el elemento dentro de la figura sobre el que se aplica la regla.

## Clases

**Regla 1:** Por cada clase equivalente se definirá un estereotipo que extienda a la clase UML referenciada. El nombre del estereotipo debe ser el nombre de la clase equivalente.

Dado que en el Metamodelo de Integración las *clases equivalentes* están mapeadas sólo con una clase de UML, es posible asegurar que la semántica de las clases equivalente se puede representar mediante un estereotipo que extienda a la clase UML referenciada. En otras palabras, este estereotipo corresponde a la *clase equivalente* en el metamodelo de UML.

**Regla Identificación:** Clase equivalente.

**Condición Identificación:** <ninguna>

**Transformación:** Un estereotipo que extiende la clase UML referenciada. El nombre del estereotipo es igual al de la *clase equivalente*.

**Consideraciones adicionales:**

- Si el nombre de la *clase equivalente* coincide con el nombre de la clase referenciada, entonces un prefijo debe ser agregado al nombre del estereotipo para diferenciarlo de la clase extendida.

Es importante poder diferenciar el estereotipo de la clase extendida, ya que desde la especificación 2.1.1 de UML [45] es posible definir asociaciones mediante los valores etiquetados de los estereotipos. Si el nombre del estereotipo es igual al de la clase que

extiende, no se puede identificar correctamente el tipo de una asociación que referencia al estereotipo, por ejemplo, para definir una regla OCL.

**Refinamiento perfil UML:** Si hay sólo un estereotipo que extiende a la clase UML, entonces la asociación de extensión del estereotipo debe ser definida como obligatoria (*required*).

Si hay sólo un estereotipo que extienda a la clase UML, significa que en el contexto del DSML la clase UML sólo tiene la semántica de la clase equivalente representada por el estereotipo.

**Mapeo Perfil UML:** La clase equivalente estará mapeada al estereotipo generado por la regla de transformación.

**Ejemplo:** La Figura 21 muestra como se aplica la regla de transformación, y la Figura 22 muestra el mapeo del perfil UML resultante.

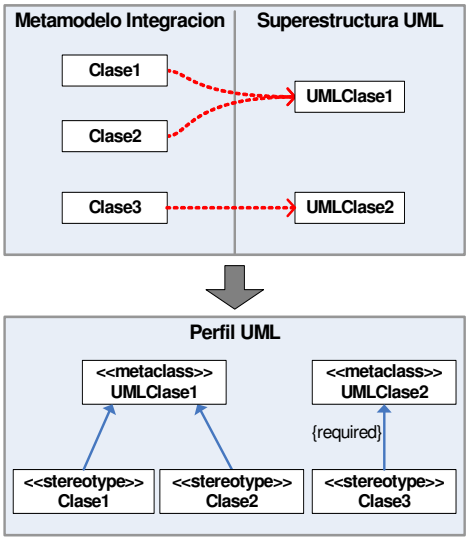


**Figura 21.** Ejemplo genérico para la regla de transformación asociada a clases equivalentes: Regla 1

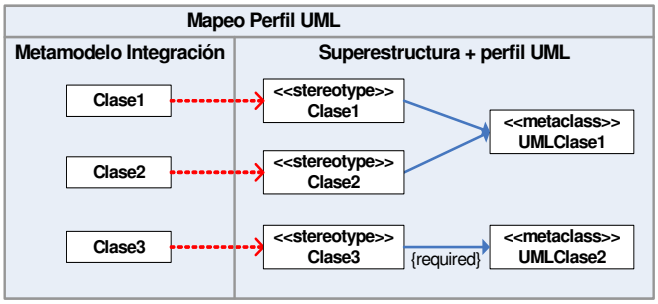


**Figura 22.** Mapeo del perfil UML asociado al ejemplo de la Regla 1

## Atributos

**Regla 2:** Por cada *atributo nuevo* se debe definir un valor etiquetado que represente a dicho atributo. El valor etiquetado se define dentro del estereotipo que corresponde a la *clase equivalente* que contiene el *atributo nuevo*. El nombre del valor etiquetado debe ser el nombre del *atributo nuevo*. Las propiedades del valor etiquetado deben ser iguales a las propiedades del *atributo nuevo*.

Un *atributo nuevo* corresponde a un atributo que no tiene equivalencia en el metamodelo de UML, por lo tanto, este *atributo nuevo* debe ser incorporado como un atributo de la clase UML correspondiente. Para agregar el atributo se utiliza un nuevo valor etiquetado definido sobre el estereotipo que extiende a la clase UML, que de acuerdo con la *Regla 1*, corresponde al estereotipo que representa a la *clase equivalente* que referencia a la clase UML que debe ser extendida.

**Regla Identificación:** Atributo nuevo

**Condición Identificación:** <ninguna>

**Transformación:** Un valor etiquetado definido en el estereotipo que corresponde a la clase equivalente que contiene el *atributo nuevo*. El nombre, cardinalidad y tipo del valor etiquetado serán igual que en el *atributo nuevo*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla1

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *atributo nuevo* es mapeado al valor etiquetado generado por la regla de transformación.

**Ejemplo:** El ejemplo de aplicación de esta regla se muestra en la Figura 23, y el mapeo del perfil UML se muestra en la Figura 24, para el atributo *atr3* de la clase equivalente *Clase1*.

**Regla 3:** Para los *atributos equivalentes* que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite inferior de la cardinalidad del *atributo equivalente* es mayor que el límite inferior del atributo UML referenciado, se debe definir una restricción para ajustar la cardinalidad del atributo UML a la del *atributo equivalente*.

La restricción debe ser definida mediante una regla OCL que impida que número de valores que puede almacenar el atributo UML puedan estar por debajo de la cardinalidad especificada en el *atributo equivalente*.

**Regla Identificación:** Atributo equivalente

**Condición Identificación:** Límite inferior de la cardinalidad del atributo equivalente mayor que el límite inferior del atributo UML referenciado (Diferencia de cardinalidad).

**Transformación:** Definir sobre el atributo UML referenciado una regla OCL con la siguiente estructura:

```
self.[atributoUML]->size() > [newLimInf – 1]
```

Donde:

- *atributoUML* es el atributo UML referenciado por el *atributo equivalente*.
- *newLimInf* es el límite inferior de la cardinalidad asociada al *atributo equivalente*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *atributo equivalente* es mapeado con el atributo UML referenciado.

**Ejemplo:** La Figura 23 muestra como se aplica la regla de transformación y la Figura 24 muestra el mapeo del perfil UML resultante, para el atributo *atr1* de la clase equivalente *Clase1*.

**Regla 4:** Para los *atributos equivalentes* que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite superior de la cardinalidad del *atributo equivalente* es menor que el límite superior del atributo UML referenciado, se debe definir una restricción para ajustar la cardinalidad del atributo UML a la del *atributo equivalente*.

La restricción debe ser definida mediante una regla OCL que impida que número de valores que puede almacenar el atributo UML puedan estar por sobre la cardinalidad especificada en el *atributo equivalente*.

**Regla Identificación:** Atributo equivalente

**Condición Identificación:** Límite superior de la cardinalidad del atributo equivalente menor que el límite superior del atributo UML referenciado (Diferencia de cardinalidad).

**Transformación:** Definir sobre el atributo UML referenciado una regla OCL con la siguiente estructura:

```
self.[atributoUML]->size() < [newLimSup + 1]
```

Donde:

- *atributoUML* es el atributo UML referenciado por el *atributo equivalente*.
- *newLimSup* es el límite superior de la cardinalidad asociada al *atributo equivalente*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *atributo equivalente* es mapeado con el atributo UML referenciado.

**Ejemplo:** La Figura 23 muestra como se aplica la regla de transformación y la Figura 24 muestra el mapeo del perfil UML resultante, para el atributo *atr2* de la clase equivalente *Clase1*.

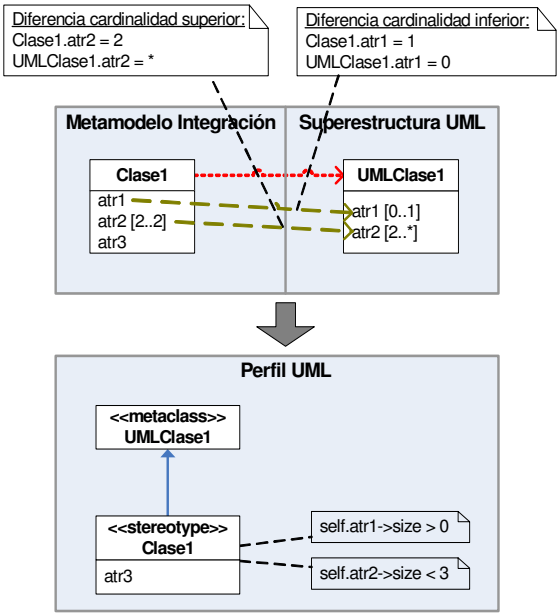


**Figura 23.** Ejemplo genérico para las reglas de transformación asociadas a atributos: Reglas 2, 3, y 4

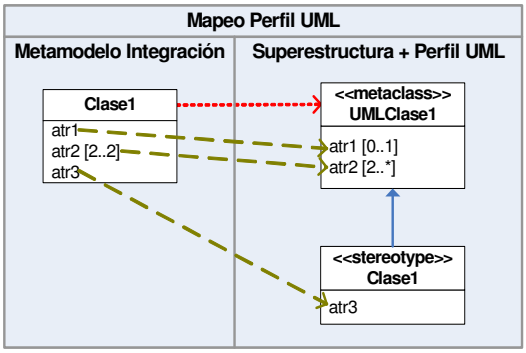


**Figura 24.** Mapeo del perfil UML asociado al ejemplo de las Reglas 2, 3, y 4

**Regla 5:** Para los *atributos equivalentes* que posean diferencias que no pueden ser representadas en UML utilizando un perfil UML, se debe definir un nuevo atributo en la clase UML que representa al *atributo equivalente* y reemplaza al atributo UML original.

La regla 5 se ha definido para resolver las limitaciones de extensión de los perfiles UML, que no permiten modificar la semántica de los metamodelos que extienden, en este caso particular, no permiten la redefinición de propiedades. Una vez aplicada esta regla, los procesos MDE que utilicen el perfil UML generado deben considerar el nuevo atributo definido en vez del atributo UML original.

Las diferencias de atributos equivalentes que no permiten una representación mediante un perfil UML son: (1) cardinalidad del atributo equivalente es menor que el limite inferior o mayor que el límite superior del atributo UML referenciado, y (2) tipo del atributo equivalente distinto al del atributo UML referenciado.

**Regla Identificación:** Atributo equivalente

**Condición Identificación:**

- Límite inferior de la cardinalidad del atributo equivalente menor que el límite inferior del atributo UML referenciado (Diferencia de cardinalidad). Ó

- Límite superior de la cardinalidad del atributo equivalente mayor que el límite superior del atributo UML referenciado (Diferencia de cardinalidad). Ó
- Tipo del atributo equivalente distinto al tipo del atributo UML referenciado (Diferencia de tipo).

**Transformación:** Un valor etiquetado definido en el estereotipo que corresponde a la clase equivalente que contiene el *atributo equivalente*. El nombre, cardinalidad y tipo del valor etiquetado serán igual que en el *atributo equivalente.*

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *atributo nuevo* es mapeado al valor etiquetado generado por la regla de transformación.

**Ejemplo:** La Figura 25 muestra como se aplica la regla de transformación, y la Figura 26 muestra el mapeo del perfil UML resultante

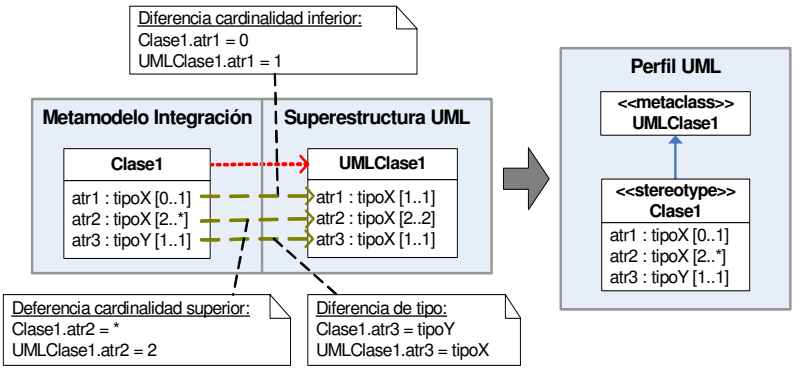


**Figura 25.** Ejemplo genérico para la Regla 5

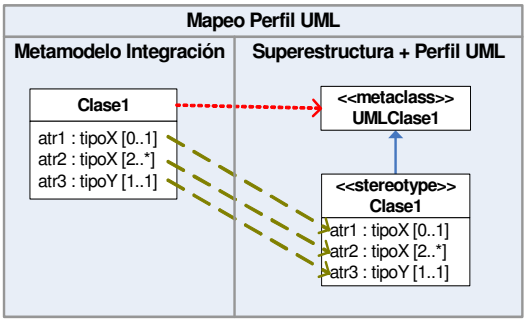


**Figura 26.** Mapeo del perfil UML obtenido para el ejemplo de la Regla 5

## Enumeraciones

**Regla 6:** En el perfil UML se debe definir una enumeración por cada *enumeración nueva*. La enumeración definida debe tener todos los valores literales de la *enumeración equivalente*.

La definición de una enumeración en el perfil UML, para representar una *enumeración nueva* del Metamodelo de Integración parece evidente, ya que es un elemento que no existe en UML y debe ser incorporado mediante el perfil UML.

**Regla Identificación:** Enumeración nueva

**Condición Identificación:** <ninguna>

**Transformación:** Una enumeración con el mismo nombre y valores literales que la *enumeración nueva*.

**Consideraciones Adicionales:** <ninguna>

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *enumeración nueva* y sus valores literales son mapeados a la enumeración y valores literales generados por la regla de transformación.

**Ejemplo:** La Figura 27 muestra el ejemplo para esta regla y la Figura 28 muestra el mapeo del perfil UML resultante, para la enumeración *Enum3*.

**Regla 7:** En el perfil UML se debe definir una enumeración por cada *enumeración equivalente* con *valores literales nuevos*. La enumeración definida debe tener todos los valores literales de la *enumeración nueva*.

En este caso, la enumeración definida debe reemplazar la definición UML original, ya que las enumeraciones no pueden ser extendidas mediante estereotipo, por lo que no es posible extender la enumeración UML con los valores literales que no posee.

**Regla Identificación:** Enumeración equivalente.

**Condición Identificación:** La enumeración equivalente posee *valores literales nuevos*.

**Transformación:** Una enumeración con el mismo nombre y valores literales que la *enumeración equivalente*.

**Consideraciones Adicionales:** <ninguna>

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *enumeración equivalente* y sus valores literales son mapeados a la enumeración y valores literales generados por la regla de transformación.

**Ejemplo:** La Figura 27 muestra el ejemplo para esta regla y la Figura 28 muestra el mapeo del perfil UML resultante, para la enumeración *Enum2*.

**Regla 8:** Para *cada enumeración equivalente* que contenga sólo *valores literales equivalentes*, y que estos valores literales sean menos que los valores literales de la enumeración UML, se debe definir una regla OCL para restringir los valores literales de la enumeración UML que no corresponden al contexto del DSML.

Cuando la enumeración UML posee más opciones que la *enumeración equivalente* que la referencia, significa que los valores literales no referenciados de la enumeración UML, no son válidos

para los atributos UML cuyo se corresponda con la *enumeración equivalente*. Por lo tanto, para estos atributos será necesario definir una regla OCL que limite las opciones que pueden ser asignadas para definir su tipo, a las opciones dadas por los *valores literales equivalentes* de la *enumeración equivalente*.

**Regla Identificación:** Atributo equivalente

**Condición Identificación:**

- El tipo del *atributo equivalente* es una *enumeración equivalente*. Y
- La enumeración equivalente asociada al tipo del atributo sólo contiene valores literales equivalentes. Y
- La enumeración UML referenciada por la *enumeración equivalente*, contiene más valores literales que la *enumeración equivalente*.

**Transformación:** Por cada valor literal no mapeado de la enumeración UML, definir sobre el atributo UML referenciado una regla OCL con la siguiente estructura:

```
self.[atributoUML] <> #[valorLiteralUMLNoMapeado]
```

Donde:

- *atributoUML* es el atributo UML referenciado por el *atributo equivalente*.
- *valorLiteralUMLNoMapeado* es el valor literal no mapeado de la enumeración UML.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *enumeración equivalente* y sus valores literales son mapeados a la enumeración UML referenciada y los valores literales correspondientes.

**Ejemplo:** La Figura 27 muestra el ejemplo para esta regla y la Figura 28 muestra el mapeo del perfil UML resultante, para la enumeración *Enum1*.

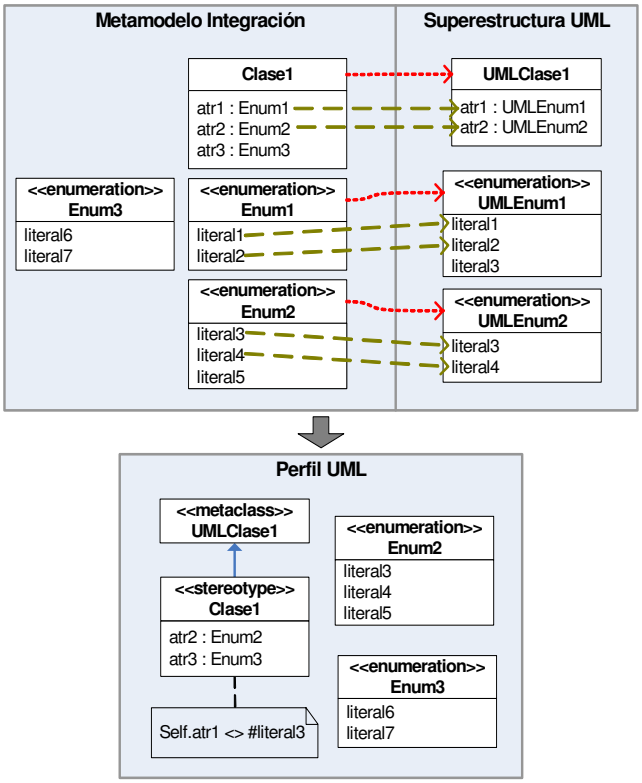


**Figura 27.** Ejemplo genérico para las reglas de transformación asociadas a enumeraciones: Reglas 6, 7, y 8

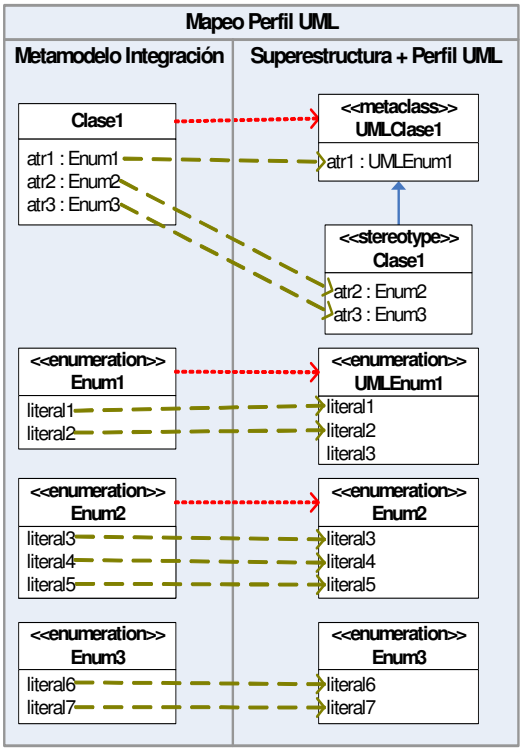


**Figura 28.** Mapeo del perfil UML obtenido para el ejemplo asociado a las Reglas 6, 7, y 8

## Generalizaciones

Para entender las reglas de transformación asociadas a las generalizaciones, es necesario conocer como son identificadas las *generalizaciones equivalentes*, ya que las generalizaciones no forman parte de los elementos que son mapeados en la definición del Metamodelo de Integración. Esto es debido a que con la información de mapeo del Metamodelo de Integración las *generalizaciones equivalentes* se pueden identificar automáticamente.

Una generalización del Metamodelo de Integración será equivalente a una generalización UML cuando:

- Las clases que participan en la generalización del Metamodelo de Integración son equivalentes a las clases UML que participan en la generalización UML.
- La generalización UML presenta la misma relación padre hijo entre las clases UML, que la generalización del Metamodelo de Integración con las clases equivalentes

**Regla 9:** Para las *clases equivalentes* asociadas por una *generalización nueva* y ambas clases están mapeadas a la misma clase UML, se debe definir un estereotipo por cada clase y entre los estereotipos se debe definir generalización idéntica a la *generalización nueva*. De los estereotipos definidos, sólo el estereotipo padre (de acuerdo a la asociación de generalización) extenderá a la clase UML referenciada.

Cuando existe una generalización entre dos estereotipos que extienden a la misma clase UML, no es necesario definir la extensión del estereotipo hijo a la clase UML, ya que la asociación de extensión es heredada a través de la generalización.

**Regla Identificación:** Generalización nueva

**Condición Identificación:** Las *clases equivalentes* que participan en la generalización están mapeadas a la misma clase UML.

**Transformación:**

- Un estereotipo por cada clase equivalente que participa en la generalización. El nombre de cada estereotipo es igual al nombre de la *clase equivalente* correspondiente.
- Una asociación de generalización definida entre los estereotipos generados, que es idéntica a la *generalización nueva*.
- Una asociación de extensión entre la clase padre de la generalización y la clase UML referenciada.

**Consideraciones adicionales:**

- Si el nombre de la *clase equivalente* coincide con el nombre de la clase referenciada, entonces un prefijo debe ser agregado al nombre del estereotipo para diferenciarlo de la clase extendida.
- Esta regla de transformación extiende a la Regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:**

- Las *clases equivalentes* y sus propiedades estarán mapeada a los estereotipos y valores etiquetados generados por la regla de transformación.
- La *nueva generalización* estará mapeada a la generalización generada por la regla de transformación.
- **Ejemplo:** La Figura 30 muestra el ejemplo para esta regla y la Figura 31 muestra el mapeo del perfil UML resultante, para la generalización definida entre las clases *Clase1* y *Clase3*.

**Regla 10:** Para dos *clases equivalentes* asociadas por una *generalización nueva* y las clases están mapeadas a distintas clases UML, se debe definir un estereotipo por cada clase y las propiedades heredadas deben ser incorporadas en el estereotipo que representa a la clase hija. Para cada estereotipo se define la asociación de extensión a la clase UML correspondiente.

En esta regla de transformación no es representada la *nueva generalización* entre los estereotipos, porque debido a que la asociación de extensión se hereda, si es definida la generalización entre los estereotipos, el estereotipo hijo podría extender a la clase UML que extendida por el estereotipo padre. Esta situación no es correcta ya que el estereotipo hijo, de acuerdo a la clase equivalente que representa, extiende a una clase distinta que la que extiende la clase padre.

Eventualmente, se podría pensar que una segunda alternativa de representación más adecuada es definir la asociación de generalización ente los estereotipos y luego mediante una regla OCL

evitar que el estereotipo hijo extienda a la clase extendida por el estereotipo padre, de esta manera no sería necesario duplicar las propiedades heredadas. Suponiendo que esta alternativa de representación es correcta, la regla OCL debería ser definida en el estereotipo hijo y tendría la siguiente estructura:

```
self->oclIsTypeOf([claseUMLHijo])
```

Donde:

- *claseUMLHijo* es la clase UML que debe ser extendida por el estereotipo hijo.

Con esta regla OCL se asegura que el estereotipo hijo sólo puede ser aplicado sobre la clase UML que extiende directamente, y no sobre la clase UML extendida por el estereotipo padre. La Figura 29 grafica esta situación.

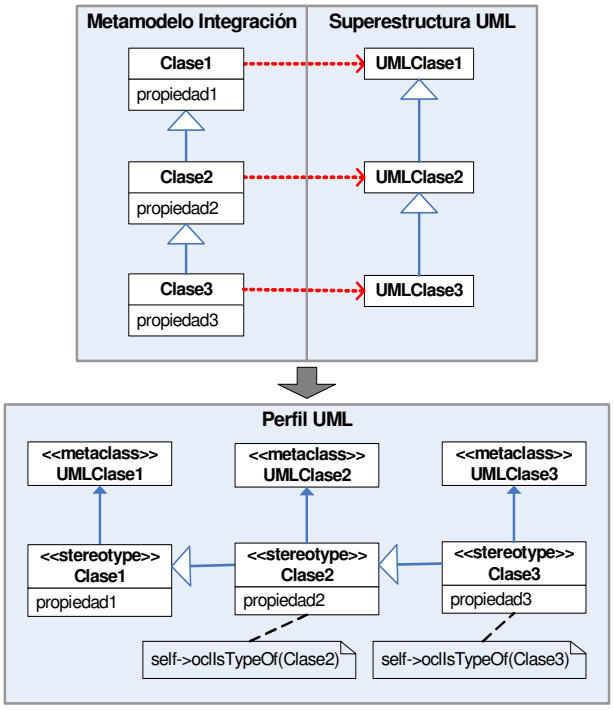


**Figura 29.** Ejemplo de transformación alternativa para la Regla 10

A pesar de que esta segunda alternativa de representación parece ser correcta y más sencilla que la duplicación de propiedades

propuesta para la regla 10, veremos que en realidad no es semánticamente correcta.

Debido a que la Infraestructura de UML [47] define a los estereotipos como un tipo particular de clase (la clase *Stereotype* es un especialización de la clase *Class*) y las reglas OCL corresponden a características de comportamiento de una clase (operaciones), entonces la generalización entre estereotipos no tan solo implica la herencia de propiedades sino que también implica la herencia de las reglas OCL definidas en el estereotipo padre. De esta manera, al observar la Figura 29 podremos apreciar que el estereotipo *Clase3* no podrá ser aplicado nunca debido a que también debe cumplir la regla OCL definida sobre el estereotipo *Clase2*, obteniendo una representación incorrecta del perfil UML resultante. Por otra parte, el tener un estereotipo que además de su asociación de extensión, herede la asociación de extensión de su estereotipo padre está implícitamente definiendo un mapeo 1:M entre un constructor conceptual del DSML y un constructor conceptual de UML, lo que puede dificultar la integración e intercambio de modelos UML y DSML, ya que obliga al análisis de las fórmulas OCL para determinar la semántica correcta del estereotipo en relación al DSML asociado.

Una vez justificada la decisión de diseño de la regla de transformación *Regla 10*, podemos continuar con su especificación.

**Regla Identificación:** Generalización nueva

**Condición Identificación:** Las *clases equivalentes* que participan en la generalización están mapeadas a distintas clases UML.

**Transformación:**

- Un estereotipo por cada clase equivalente que participa en la generalización. Cada estereotipo extiende a la clase UML referenciada por la clase equivalente correspondiente. El nombre de cada estereotipo es igual al nombre de la *clase equivalente* correspondiente.
- Las propiedades heredadas por la *clase equivalente* hija, son duplicadas en el estereotipo que representa a esta clase hija,

mediante valores etiquetados que poseen las mismas características que las propiedades heredadas.

**Consideraciones adicionales:**

- Si el nombre de la *clase equivalente* coincide con el nombre de la clase referenciada, entonces un prefijo debe ser agregado al nombre del estereotipo para diferenciarlo de la clase extendida.
- Esta regla de transformación extiende a la Regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:**

- Las clases equivalentes y sus propiedades estarán mapeada a los estereotipos y valores etiquetados generados por la regla de transformación.
- La *nueva generalización* estará mapeada a la generalización generada por la regla de transformación.

**Ejemplo:** La Figura 30 muestra el ejemplo para esta regla y la Figura 31 muestra el mapeo del perfil UML resultante, para la generalización definida entre las clases *Clase3* y *Clase4*.

**Regla 11:** Para las *clases equivalentes* asociadas por una *generalización equivalente*, se debe definir un estereotipo por cada *clase equivalente* que extienda a la respetiva clase UML y para el estereotipo que representa a la clase equivalente hija se debe definir una regla OCL que garantice que previamente se debe aplicar sobre la clase UML el estereotipo que representa a la *clase equivalente* padre.

En esta regla de transformación no se define en el perfil UML la generalización entre los estereotipos, ya que se utiliza la generalización definida en el metamodelo de UML. Sin embargo, debido a que no existe una generalización entre estereotipos al aplicar el estereotipo que representa a la clase equivalente hija no se estará incorporando la semántica de la clase equivalente padre,

tal como lo indica la generalización definida en el Metamodelo de Integración. De esta manera, la regla OCL se define para asegurar al momento de aplicar el estereotipo que corresponde a la *clase equivalente* hija, también sea incorporada la semántica representada por la *clase equivalente* padre.

```
self->oclIsTypeOf([parentStereotype])
```

**Regla Identificación:** Generalización equivalente

**Condición Identificación:** <ninguna>

**Transformación:**

- Un estereotipo por cada clase equivalente que participa en la generalización. El nombre de cada estereotipo es igual al nombre de la *clase equivalente* correspondiente. Cada estereotipo extiende a la clase UML correspondiente.
- En la *clase equivalente* hija se debe definir una regla OCL con la siguiente estructura:

```
self->oclIsTypeOf([estereotipoPadre])
```

Donde:

- *estereotipoPadre* corresponde al estereotipo que representa a la *clase equivalente* padre.

En esta transformación es correcto que la regla OCL se herede en caso de herencia múltiple, ya que para aplicar cualquier estereotipo que represente a una *clase equivalente* hija, será necesario que se aplique la semántica dada por la *clase equivalente* padre y la *clase equivalente* padre de esta última, y así sucesivamente de acuerdo a la jerarquía de herencia definida.

**Consideraciones adicionales:**

- Si el nombre de la *clase equivalente* coincide con el nombre de la clase referenciada, entonces un prefijo debe ser agregado al nombre del estereotipo para diferenciarlo de la clase extendida.
- Esta regla de transformación extiende a la Regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:**

- Las *clases equivalentes* y sus propiedades estarán mapeada a los estereotipos y valores etiquetados generados por la regla de transformación.
- La *generalización equivalente* estará mapeada a la generalización de UML correspondiente.

**Ejemplo:** La Figura 30 muestra el ejemplo para esta regla y la Figura 31 muestra el mapeo del perfil UML resultante, para la generalización definida entre las clases *Clase1* y *Clase3*.

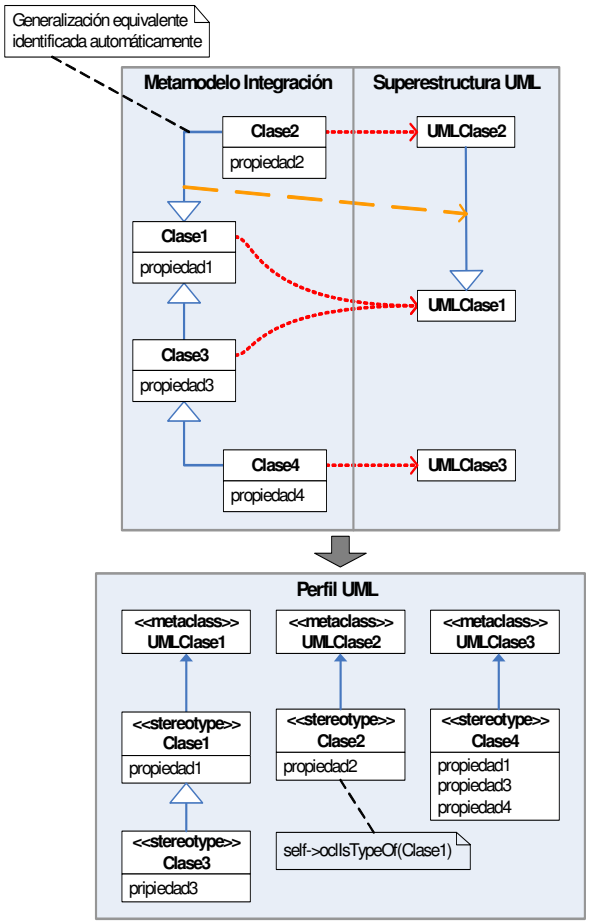


**Figura 30.** Ejemplo genérico para las reglas de transformación asociadas a generalizaciones: Reglas 9, 10, y 11

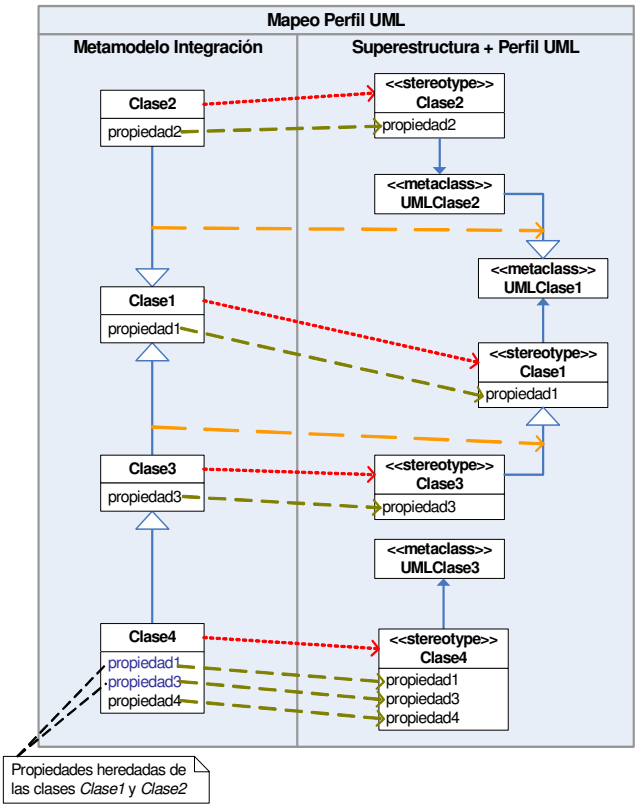


**Figura 31.** Mapeo del perfil UML obtenido para el ejemplo asociado a las Reglas 9, 10, y 11

En la Figura 31 las propiedades *propiedad1* y *propiedad3* que aparecen en la *clase equivalente Clase4* no están realmente especificadas en esta clase, sino que son heredadas de las *clases equivalentes Clase1* y *Clase3*. En la figura se han puesto de forma explícita estas propiedades para graficar mejor como es el mapeo resultante luego de aplicar la regla de transformación *Regla 10*.

## Asociaciones

**Regla 12:** Para las *asociaciones nuevas* se debe definir un valor etiquetado en el estereotipo que representa la clase equivalente que contiene la asociación. Este valor etiquetado tiene las mismas propiedades que la *asociación nueva* que representa.

Al igual que los nuevos atributos, las *asociaciones nuevas* son propiedades que no están presentes en las clases UML referenciadas y por lo tanto deben ser incorporadas como valores etiquetados definidos en el estereotipo que extiende dichas clases UML.

**Regla Identificación:** Asociación nueva

**Condición Identificación:** <ninguna>.

**Transformación:** Un valor etiquetado definido en el estereotipo que representa a la *clase equivalente* que contiene la *asociación nueva*. El nombre, cardinalidad y tipo del valor etiquetado serán igual que en la a*sociación nueva*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla1

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La asociación nueva es mapeada al valor etiquetado generado por la regla de transformación.

**Ejemplo:** El ejemplo de aplicación de esta regla se muestra en la Figura 32, y el mapeo del perfil UML se muestra en la Figura 33, para la asociación *rolClase2* de la clase equivalente *Clase1*.

**Regla 13:** Para las *asociaciones equivalentes* que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite inferior de la cardinalidad de la *asociación equivalente* es mayor que el límite inferior de la asociación UML referenciada, se debe definir una restricción para ajustar la cardinalidad de la asociación UML a la de la *asociación equivalente*.

La restricción debe ser definida mediante una regla OCL que impida que número de instancias referenciadas por la asociación UML puedan estar por debajo de la cardinalidad especificada en la *asociación equivalente*.

**Regla Identificación:** Asociación equivalente

**Condición Identificación:** Límite inferior de la cardinalidad de la *asociación equivalente* es mayor que el límite inferior de la asociación UML referenciada (Diferencia de cardinalidad).

**Transformación:** Definir sobre la asociación UML referenciada una regla OCL con la siguiente estructura:

```
self.[asocUML]->size() > [nuevoLimInf – 1]
```

Donde:

- *asocUML* es la asociación UML referenciada por la *asociación equivalente*.
- *nuevoLimInf* es el límite inferior de la cardinalidad asociada a la *asociación equivalente*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la *Regla 1*.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *asociación equivalente* es mapeada con la asociación UML referenciada.

**Ejemplo:** La Figura 32 muestra como se aplica la regla de transformación y la Figura 33 muestra el mapeo del perfil UML resultante, para la asociación *rolClase3* de la clase equivalente *Clase2*.

**Regla 14:** Para las *asociaciones equivalentes* que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite superior de la cardinalidad de la *asociación equivalente* es menor que el límite superior de la asociación UML referenciada, se debe definir una restricción para ajustar la cardinalidad de la asociación UML a la de la *asociación equivalente*.

La restricción debe ser definida mediante una regla OCL que impida que número de instancias referenciadas por la asociación UML puedan estar por sobre la cardinalidad especificada en la *asociación equivalente*.

**Regla Identificación:** Asociación equivalente

**Condición Identificación:** Límite superior de la cardinalidad de la *asociación equivalente* menor que el límite superior de la asociación UML referenciada (Diferencia de cardinalidad).

**Transformación:** Definir sobre la asociación UML referenciada una regla OCL con la siguiente estructura:

```
self.[asocUML]->size() < [nuevoLimSup + 1]
```

Donde:

- *asocUML* es la asociación UML referenciada por la *asociación equivalente*.
- *nuevoLimSup* es el límite superior de la cardinalidad asociada a la *asociación equivalente*.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *atributo equivalente* es mapeado con el atributo UML referenciado.

**Ejemplo:** La Figura 32 muestra como se aplica la regla de transformación y la Figura 33 muestra el mapeo del perfil UML resultante, para la asociación *rolClase3* de la clase equivalente *Clase2.*

**Regla 15:** Para las *asociaciones equivalentes*, se debe definir una restricción en la clase UML que contiene la asociación UML referenciada, para garantizar que a la clase UML que participa en la asociación esté extendida por el estereotipo correspondiente a la *clase equivalente* que participa en la *asociación equivalente*.

Partiendo de la premisa que una *asociación equivalente* siempre referencia a una *clase equivalente*, se tendrá por consiguiente que la asociación UML correspondiente debe referenciar a una clase con la semántica de la *clase equivalente* implicada. Por otra parte, dado

que una clase UML puede ser extendida por varios estereotipos, es necesario precisar mediante una restricción el estereotipo adecuado para que la asociación definida sea válida. Esta restricción debe asegurar que el tipo de la clase UML debe ser igual al estereotipo que representa a la *clase equivalente* asociada, y se debe definir mediante una regla OCL en la clase UML que contiene la asociación.

**Regla Identificación:** Asociación equivalente

**Condición Identificación:** <ninguna>

**Transformación:** Definir en el estereotipo que extiende la clase UML que contiene la asociación referenciada, una regla OCL con la siguiente estructura:

```
self.[asocUML]->oclIsTypeOf([estClaseAsoc])
```

Donde:

- *asocUML* es la asociación UML referenciada por la *asociación equivalente*.
- *estClaseAsoc* es el estereotipo que representa a la *clase equivalente* referenciada por la *asociación equivalente*.

**Consideraciones Adicionales:**

- Esta regla se aplica dentro del contexto de la Regla1.
- Esta regla también se aplica cuando la *asociación equivalente* vincula a una clase que es una especialización de la clase vinculada por la asociación UML. De esta manera se garantiza que las instancias vinculadas a la asociación UML serán siempre instancia de la especialización.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *asociación equivalente* es mapeada con la asociación UML referenciada.

**Ejemplo:** La Figura 32 muestra como se aplica la regla de transformación y la Figura 33 muestra el mapeo del perfil UML resultante. La asociación *rolClase3* de la clase *Clase3* muestra la aplicación normal de la regla y la asociación *rolClase1* de la clase

equivalente *Clase3* muestra como se aplica la regla a las asociaciones equivalentes definidas en las consideraciones especiales.

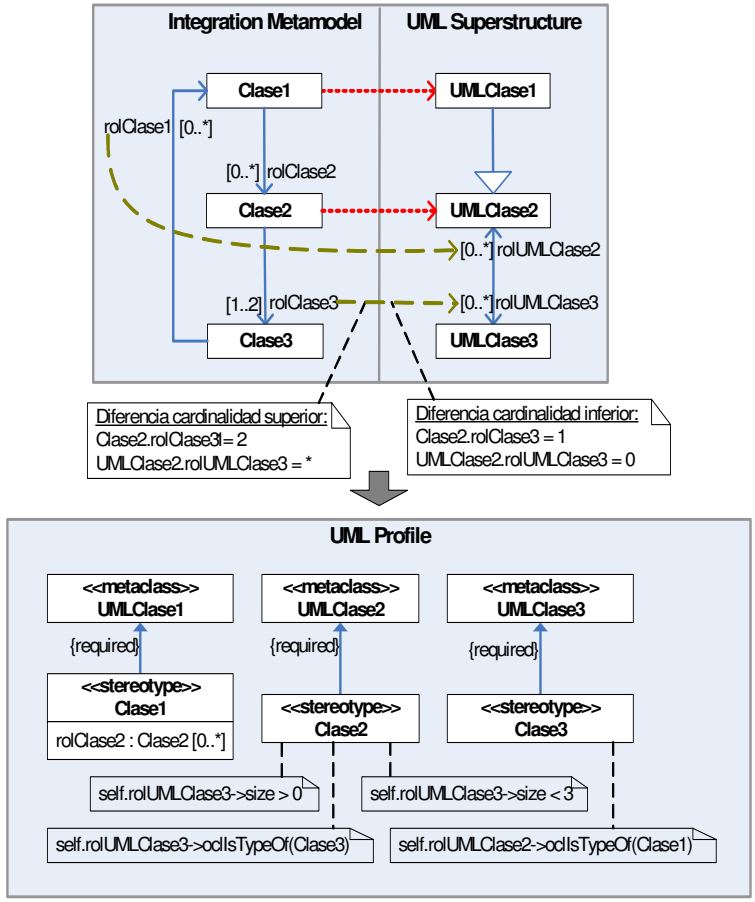


**Figura 32.** Ejemplo genérico para las reglas de transformación asociadas a asociaciones: Reglas 12 a la 15

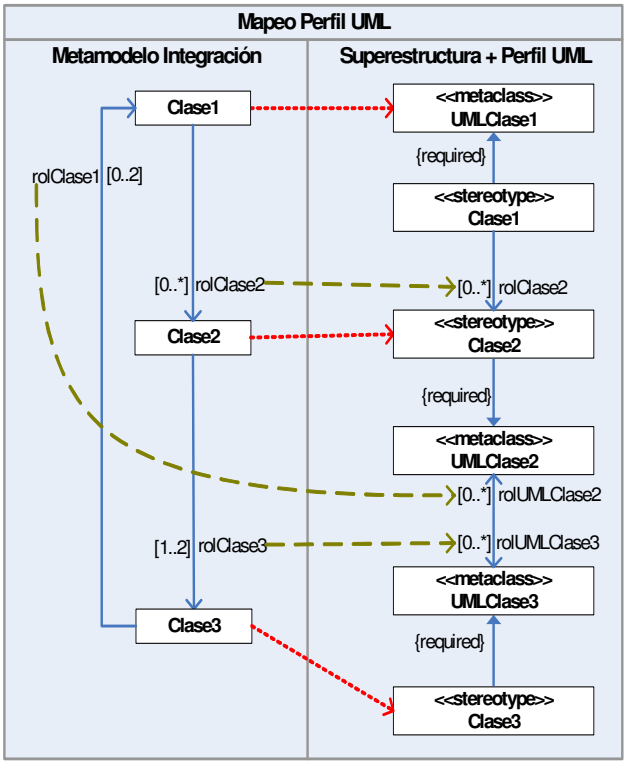


**Figura 33.** Mapeo del perfil UML obtenido para el ejemplo asociado a las Reglas 12 a la 15

**Regla 16:** Para las *asociaciones equivalentes* que posean diferencias que no pueden ser integradas en UML utilizando un perfil UML, se debe definir una nueva asociación en la clase UML que representa a la *asociación equivalente* y que reemplaza a la asociación UML original.

La definición de esta regla sigue el mismo criterio que la *Regla 5*, definida para el caso de *atributos equivalentes* que presentan diferencias que no pueden ser representadas mediante perfiles UML.

Las diferencias de *asociaciones equivalentes* que no permiten una representación mediante un perfil UML son: (1) cardinalidad de la *asociación equivalente* es menor que el limite inferior o mayor que el límite superior de la asociación UML referenciada, y (2) tipo de la asociación equivalente es distinto al de la asociación UML referenciada.

**Regla Identificación:** Asociación equivalente

**Condición Identificación:**

- Límite inferior de la cardinalidad de la *asociación equivalente* menor que el límite inferior de la asociación UML referenciada. Ó
- Límite superior de la cardinalidad de la *asociación equivalente* mayor que el límite superior de la asociación UML referenciada. Ó
- Tipo de la *asociación equivalente* distinto al tipo del atributo UML referenciado.

**Transformación:** Un valor etiquetado definido en el estereotipo que corresponde a la *clase equivalente* que contiene la *asociación equivalente*. El nombre, cardinalidad y tipo del valor etiquetado serán igual que en la *asociación equivalente.*

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla 1.

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** La *asociación nueva* es mapeada al valor etiquetado generado por la regla de transformación.

**Ejemplo:** La Figura 34 muestra como se aplica la regla de transformación, y la Figura 35 muestra el mapeo del perfil UML resultante

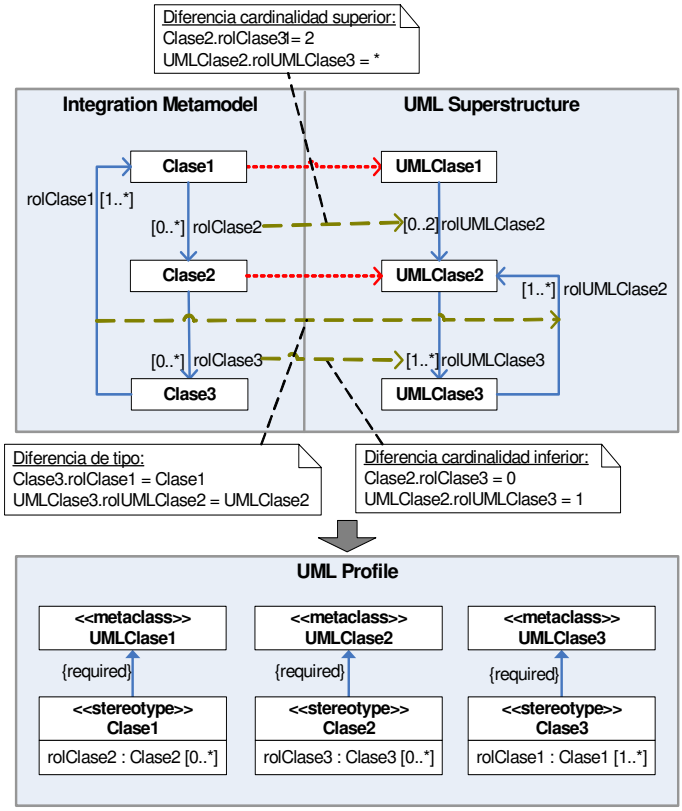


**Figura 34.** Ejemplo genérico para la regla de transformación 16

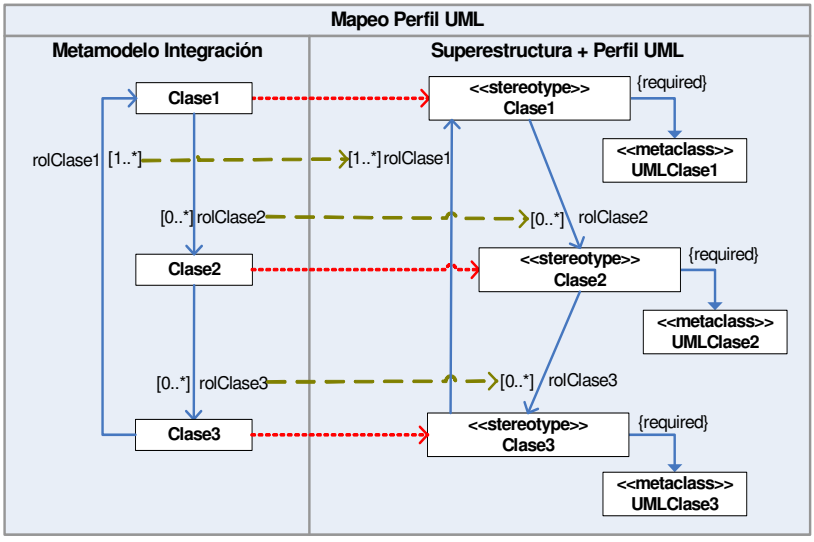


**Figura 35.** Mapeo del perfil UML obtenido para el ejemplo asociado a la Regla 16

## Reglas OCL

**Regla 17:** Las reglas OCL definidas en las *clases equivalentes* deben ser incluidas en los estereotipos generados a partir de estas clases.

En la elaboración sistemática del Metamodelo de Integración se ha asegurado que las reglas OCL no presentan inconsistencias ni problemas de integración con el Metamodelo de UML. Por este motivo, no existe inconveniente en incorporarlas para garantizar la correcta integración semántica del DSML en UML.

**Regla Identificación:** Reglas OCL

**Condición Identificación:** <ninguna>

**Transformación:** Una regla OCL definida en el estereotipo correspondiente a la clase equivalente que contiene la regla. La regla OCL debe tener la misma estructura que la regla original.

**Consideraciones Adicionales:** Esta regla se aplica dentro del contexto de la Regla 1.

**Refinamiento perfil UML:** Los elementos referenciados por las reglas OCL deben ser cambiados de acuerdo a la definición de estereotipos del perfil UML final.

**Mapeo Perfil UML:** <ninguno>

**Ejemplo:** La Figura 36 muestra como se aplica la regla de transformación. En este caso el mapeo del perfil UML resultante no se ve afectado por la regla de transformación.

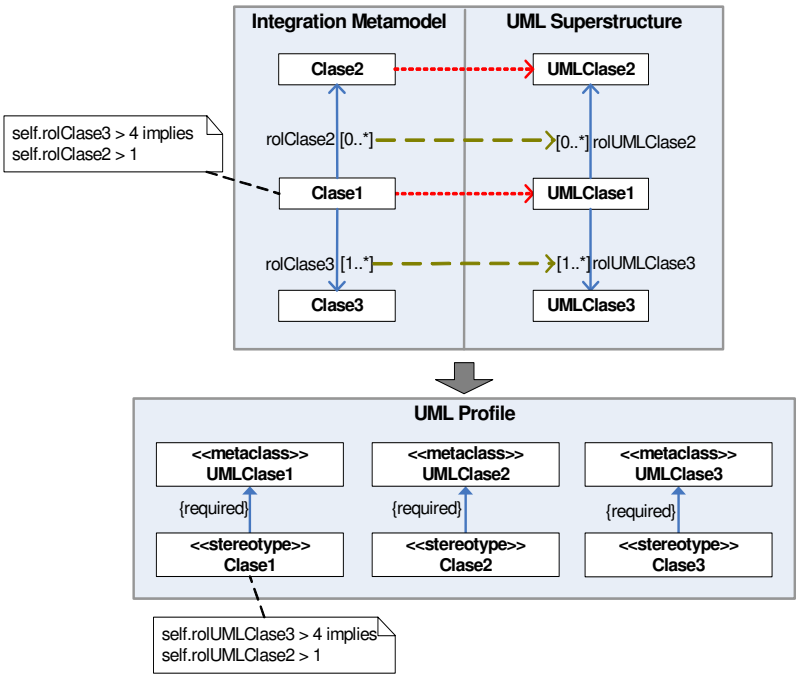


**Figura 36.** Ejemplo genérico para la regla de transformación 17

En la Figura 36 se puede observar que la regla OCL se aplica sobre el estereotipo correspondiente a la clase equivalente *Clase1* y que las *asociaciones equivalentes* referenciadas por la regla OCL son reemplazadas por las asociaciones UML correspondientes.

## Tipos de Datos

**Regla 18:** Los *tipos de datos nuevos* definidos en el Metamodelo de Integración deben ser definidos en una *librería de modelo*, que se representa mediante un paquete estereotipado con la etiqueta *<<ModelLibrary>>*. Esta librería debe ser importada en cada modelo UML que es diseñado utilizando el perfil UML generado.

La especificación de los perfiles UML no provee un mecanismo propio para la definición de nuevos tipos de datos, sin embargo, el uso de una librería de modelo permite incorporar elementos de UML que pueden interactuar con el perfil UML y ser incorporados como parte del modelo UML que se está diseñando.

**Regla Identificación:** Tipos de datos nuevos

**Condición Identificación:** <ninguna>

**Transformación:** Un tipo de dato definido como parte de una librería de modelo utilizada por el perfil UML. El tipo de dato generado es igual al *tipo de dato nuevo*.

**Consideraciones Adicionales:** <ninguna>

**Refinamiento perfil UML:** <ninguno>

**Mapeo Perfil UML:** El *tipo de dato nuevo* es mapeado al tipo de dato generado por la regla de transformación.

**Ejemplo:** La Figura 37 muestra como se aplica la regla de transformación y la Figura 37 muestra el mapeo del perfil UML resultante.

**Regla 19:** Para los *tipos de datos equivalentes* que presenten diferencias en las propiedades en relación a los tipos de datos UML referenciados, o que tengan definidas operaciones, se debe definir un tipo de dato en una librería de modelo que represente al *tipo de dato equivalente* y que reemplace al tipo de dato original de UML. Los tipos de datos originales de UML están definidos dentro del paquete *Kernel* de la Infraestructura de UML [47] y se identifican por la etiqueta <<*primitive*>>.

De acuerdo a la especificación de UML, los tipos de datos no pueden ser extendidos utilizando perfiles UML, por lo que diferencias semánticas que existan en los *tipos de datos equivalentes*, no pueden ser integradas en los correspondientes tipos de datos UML mediante un estereotipo. Para resolver esta situación, se define un nuevo tipo de dato que reemplace el tipo de dato UML original y para asegurar que ya no se puede utilizar el tipo de dato UML original, se debe definir una restricción sobre la clase *TypedElement* del metamodelo de UML. Se utiliza esta clase para definir la restricción, ya que es la generalización de todos aquellos elementos que tengan asociado un tipo de dato.

**Regla Identificación:** Tipos de datos equivalentes

**Condición Identificación:**

- El *tipo de dato equivalente* posee diferencias de propiedades en relación al tipo de dato UML referenciado. Ó
- El *tipo de dato equivalente* posee alguna operación.

**Transformación:** Una regla OCL definida sobre la clase *TypedElement* con la siguiente estructura:

```
self.type->oclIsTypeOf([tipoOriginalUML]) = False
```

Donde:

- *tipoOriginalUML* es tipo de dato de UML que será reemplazado por el tipo de dato generado por la regla de transformación.

**Consideraciones Adicionales:** <ninguna>

**Refinamiento perfil UML:** La regla OCL también debe ser aplicada sobre aquellos tipos de datos UML que no son referenciados por un *tipo de datos equivalente*, ya que al no ser referenciados implica que no son válidos dentro del contexto del DSML.

**Mapeo Perfil UML:** El *tipo de dato equivalente* es mapeado al tipo de dato generado por la regla de transformación.

**Ejemplo:** La Figura 37 muestra como se aplica la regla de transformación y la Figura 38 muestra el mapeo del perfil UML resultante.

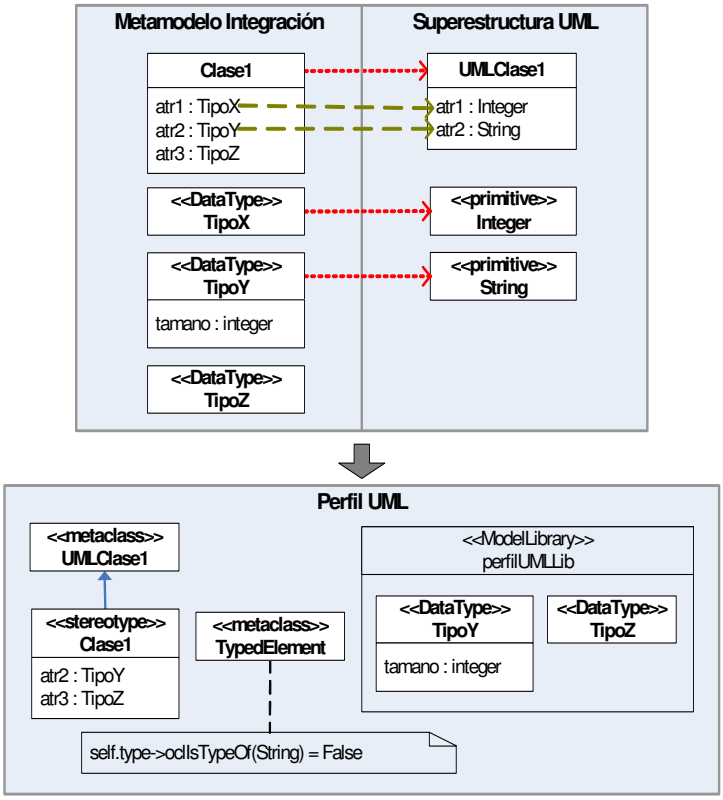


**Figura 37.** Ejemplo genérico para las reglas de transformación 18 y 19

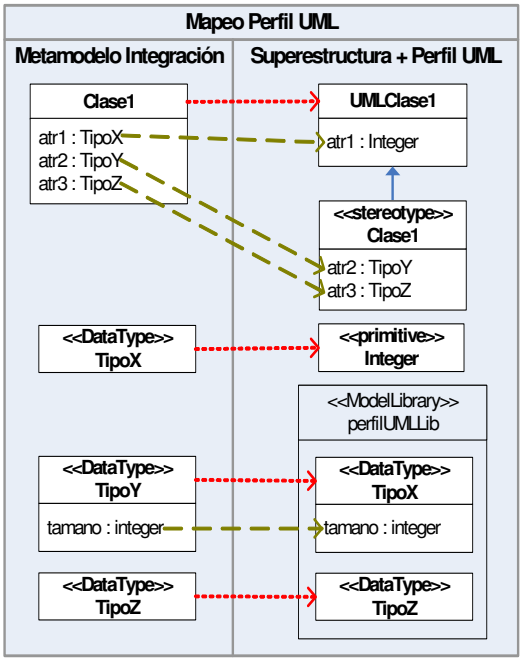


**Figura 38.** Mapeo del perfil UML obtenido para el ejemplo asociado a las reglas de transformación 18 y 19

Las 19 reglas de transformación presentadas en esta sección permiten la generación completa de un perfil UML a partir de un Metamodelo de Integración definido con la propuesta sistemática presentada en la sección 4.3.

Para obtener un mapeo del perfil UML completo, será necesario considerar aquellos *elementos equivalentes* que no cumplen ninguna regla de transformación. El hecho de que queden *elementos equivalentes* sin transformar es debido a que se utiliza directamente el elemento UML referenciado sin incorporar ninguna diferencia semántica. El mapeo de estos *elementos equivalentes* se define igual que en el Metamodelo de Integración.

El mapeo del perfil UML obtenido finalmente, provee una relación 1:1 entre las *clases equivalentes* del Metamodelo de Integración y los estereotipos del perfil UML generado. Este mapeo 1:1 permite una equivalencia en ambas direcciones entre el *Metamodelo de*

*Integración* y el metamodelo de UML extendido con el perfil UML, posibilitando el intercambio entre modelos UML y DSML.

Además, el mapeo del perfil UML generado presenta un mapeo completo de los elementos del *Metamodelo de Integración* con el metamodelo de UML, es decir, todos los elementos del Metamodelo de Integración están mapeados con algún elemento de la Superestructura de UML o del perfil UML generado. Esto garantiza que toda la semántica descrita mediante el metamodelo del DSML está integrada en UML mediante el perfil UML generado.

Finalmente, aplicando estas reglas de transformación sobre el Metamodelo de Integración definido para la asociación binaria genérica, obtendremos el perfil UML que permita integrar la semántica esta asociación binaria en UML. Para aplicar las reglas de transformación se ha utilizado el Metamodelo de Integración definido en la sección 4.2 y la información de diferencias presentadas en la Tabla 1, definida en la sección 5.1. La Figura 39 muestra el perfil UML generado para el ejemplo de la asociación binaria genérica.

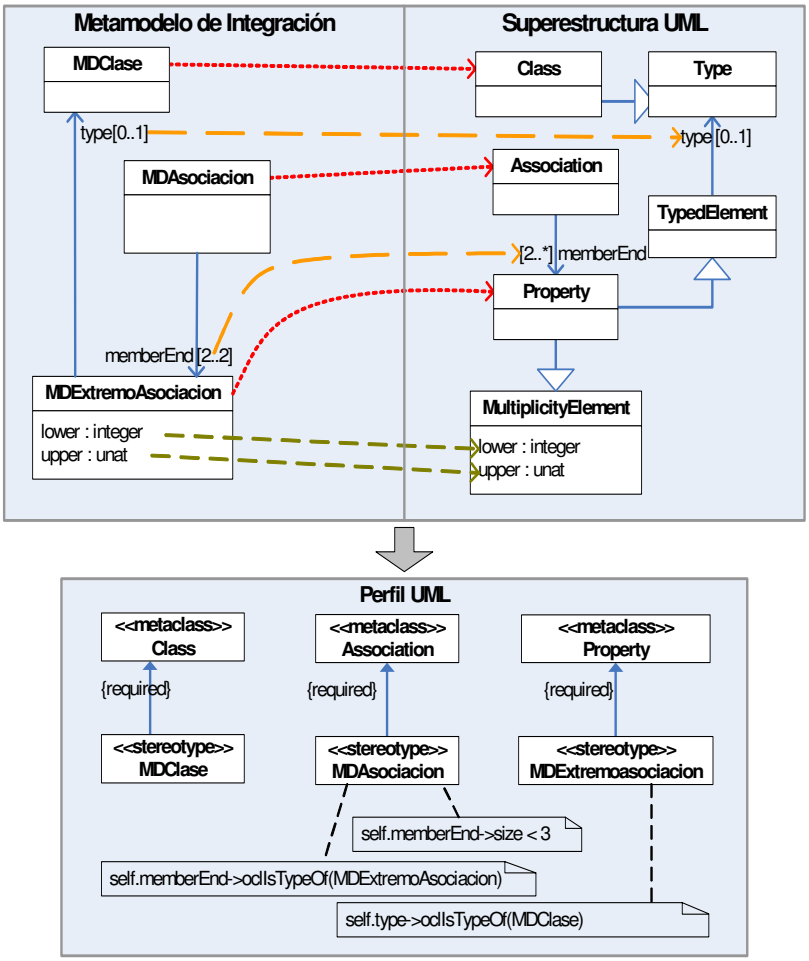


**Figura 39.** Perfil UML generado para el Metamodelo de Integración de la asociación binaria genérica.

Las reglas de transformación aplicadas sobre el Metamodelo de Integración presentado en este ejemplo son:

- Regla 1: Para la generación de los estereotipos.
- Regla 14: Para restringir la cardinalidad máxima de la asociación *memberEnd* de la clase UML *Association*.
- Regla 15: Para restringir que el tipo de las asociaciones equivalentes *memberEnd* y *type* sean las clases UML extendidas con los estereotipos *MDAsociacion* y *MDClase* respectivamente.

El mapeo del perfil UML obtenido en este ejemplo se presenta a continuación en la Figura 40.

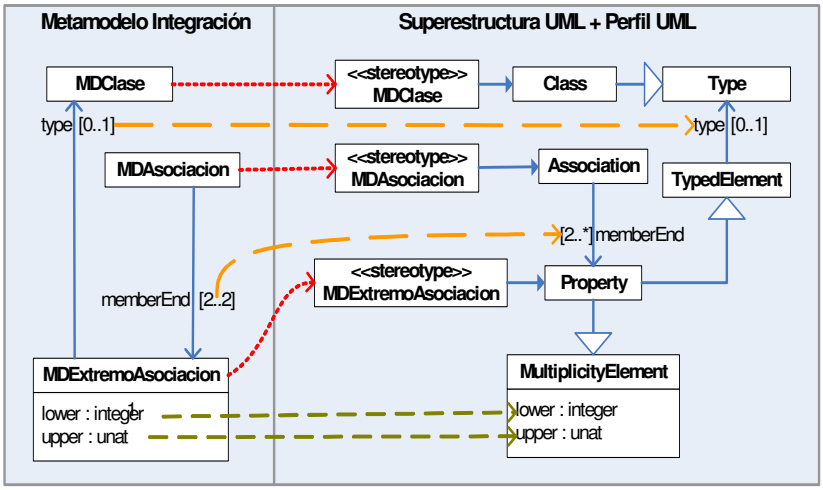


**Figura 40.** Mapeo del perfil UML generado para el ejemplo de la asociación binaria genérica.

## *5.3. Conclusiones*

En este capítulo, se ha presentado la generación automática de un perfil UML completo que permite integrar la semántica de propuestas MDE específicas en UML.

La generación automática del perfil UML y en particular las reglas de transformación propuestas, permiten resolver un problema que ha sido señalado por autores como Bran Selic en [59]. Este problema es la definición de perfiles UML incorrectos, o que no cumplen con la especificación de OMG.

De todas maneras, es importante señalar que las reglas de transformación presentadas, son solo una posible solución para la generación completa de un perfil UML. Es posible definir variaciones de estas reglas dependiendo de las decisiones de diseño que tomen las distintas propuestas MDE.

Las reglas de transformación se han presentado a nivel conceptual, evitando incluir aspectos de implementación, para facilitar la aplicación por parte de diferentes propuestas MDE que podrán implementar una solución acorde a sus tecnologías, por ejemplo: C#, Java, QVT, ATL, XSLT, etc.

Por otra parte, la información de mapeo del perfil UML con el Metamodelo de Integración, que se genera automáticamente durante el proceso de transformación, permitirá aprovechar tanto los beneficios de UML como de tecnologías basadas en DSMLs

# *Caso de estudio*

En este capítulo veremos en más detalle, mediante un caso de estudio, como aplicar el proceso completo propuesto para la generación de un perfil UML asociado a una propuesta MDE.

El caso de estudio está enfocado a precisar la semántica de la asociación de UML, ya que la semántica actual de la asociación de UML no permite que esta pueda ser utilizada en desarrollos MDE industriales.

En versiones anteriores de UML [43], distintos autores han mencionado la falta de precisión en la semántica de la asociación de UML, como por ejemplo: Opdahl et al. en [52], Graham et al. en [18] y Snoeck et al. en [60]. En la versión actual de UML, la semántica de la asociación ha sido mejorada, sin embargo, todavía presenta problemas de precisión semántica que deben ser resueltos, esto queda de manifiesto en los artículos de Albert et al. [3] y Guéhéneuc et al. [19]. Un claro ejemplo de esta situación, es la falta de una semántica precisa para la creación, eliminación o cambio de las asociaciones entre objetos. Otro ejemplo más claro aún, es la definición incompleta que existe para la semántica de la *agregación* [14], que es presentada en la Superestructura de UML [48] como un *punto de variación semántico* que señala:

*"The order and way in which part instances in a composite are created is not defined"*

Para precisar la semántica de la asociación de UML, integraremos en UML la semántica de la asociación definida en método de producción de software *OO-Method*. Para realizar la integración utilizaremos el proceso completo propuesto en esta Tesis de Master.

La semántica de OO-Method ofrece una buena solución para precisar la semántica de la asociación de UML en entornos MDE

industriales, ya que ha sido exitosamente aplicado al desarrollo industrial de software. La propuesta industrial de OO-Method cuenta con una suite de herramientas MDE, y con un compilador industrial de modelos. Estas tecnologías MDE han sido desarrolladas por la empresa CARE-Technologies [6].

La tecnología de compilación de modelos OO-Method, permite la generación automática de aplicaciones de software completas para distintas plataformas, a partir de un modelo conceptual OO-Method.

Para realizar la integración de la asociación de OO-Method en UML, en esta sección aplicaremos el proceso completo propuesto para la generación de un perfil UML. De acuerdo a los pasos que componen el proceso completo, para generar el perfil UML adecuado tendremos que llevar a cabo los siguientes pasos:

1. Definición del metamodelo del DSML para la asociación OO-Method, siguiendo las recomendaciones definidas en la sección 4.1.
2. A partir del metamodelo del DSML definido para la asociación OO-Method, generar el Metamodelo de Integración correspondiente siguiendo la propuesta sistemática presentada en la sección 4.3.
3. Identificación de diferencias entre el Metamodelo de Integración generado y la Superestructura de UML, siguiendo la solución presentada en la sección 5.1.
4. Utilizando las diferencias identificadas, transformar el Metamodelo de Integración en el perfil UML final, que integra la semántica de la asociación OO-Method en UML. Esta transformación se realiza aplicando las reglas de transformación presentadas en la sección 5.2.

En este capítulo se explica brevemente la propuesta OO-Method, luego revisaremos la semántica particular de la asociación de OO-Method para finalmente, aplicar los cuatro pasos del proceso completo definido en esta tesis.

## 6.1. La Propuesta OO-Method

El método de producción automática de software *OO-Method*, es un método orientado a objetos que permite generar automáticamente aplicaciones de software a partir de modelos definidos mediante un DSML propietario. Este DSML permite describir a nivel conceptual y de forma precisa, las aplicaciones que serán generadas.

OO-Method pone en práctica la propuesta MDA (Model Driven Architecture) [55] definida por OMG [32], realizando una clara separación entre la lógica de las aplicaciones y la plataforma tecnológica de implementación. Para lograr esta separación, OO-Method provee un conjunto de modelos conceptuales que permiten abstraer la lógica de las aplicaciones de la plataforma de implementación, centrando el proceso productivo en el espacio del problema (¿qué hace el sistema?), en lugar del espacio de la solución (¿cómo se implementa la solución?).

El proceso que utiliza OO-Method para la producción de software comprende cuatro grandes fases. Estas fases se corresponden con las etapas propuestas por MDA para el desarrollo de aplicaciones, y estarán dadas por los modelos involucrados en cada etapa: modelos independientes de computación (CIM), modelos independientes de plataforma (PIM), modelos específicos de plataforma (PSM) y modelos de implementación (IM), este último modelo corresponde a la aplicación final.

En la primera fase del proceso de desarrollo, OO-Method utiliza el *Modelo de Requisitos* [7][23] para describir los modelos independientes de computación (CIM), que corresponden a la primera etapa del desarrollo MDA. Este modelo es realizado por el analista del sistema con la ayuda de la herramienta *RETO* [51].

En la segunda fase del proceso, OO-Method utiliza el *Modelo Conceptual* para describir los modelos independientes de plataforma

(PIM). Este modelo es desarrollado por el analista del sistema con la ayuda de un conjunto de herramientas que dan soporte a distintas etapas de un proceso de ingeniería dirigido por modelos (MDE). Entre las características que presentan estas herramientas, es posible destacar: facilidades para el modelado [56], permitir el intercambio de modelos [28], validación de los modelos, y generación automática de documentación. Este conjunto de herramientas denominadas *Olivanova Suite* [6], han sido desarrolladas por la empresa CARE Technologies junto con el apoyo del grupo de investigación OO-Method.

En la fase de desarrollo de modelos específicos de plataforma (PSM) se encuentra el *Modelo de Ejecución* [17], que se genera de manera completamente automática a partir del *Modelo Conceptual* de OO-Method. La generación automática está soportada por un compilador de modelos desarrollado por la empresa CARE Technologies denominado: *The Olivanova Programming Machine* [6]. Finalmente, la fase de desarrollo del modelo de implementación (IM) que corresponde a la aplicación final, también se realiza de manera completamente automática por el compilador de modelos de OO-Method.

La Figura 41 esquematiza el proceso de producción de software de OO-Method, mostrando claramente los diferentes modelos que lo componen y las transformaciones entre modelos. Esta figura también refleja las correspondencias entre los modelos OO-Method y los modelos de la propuesta MDA.

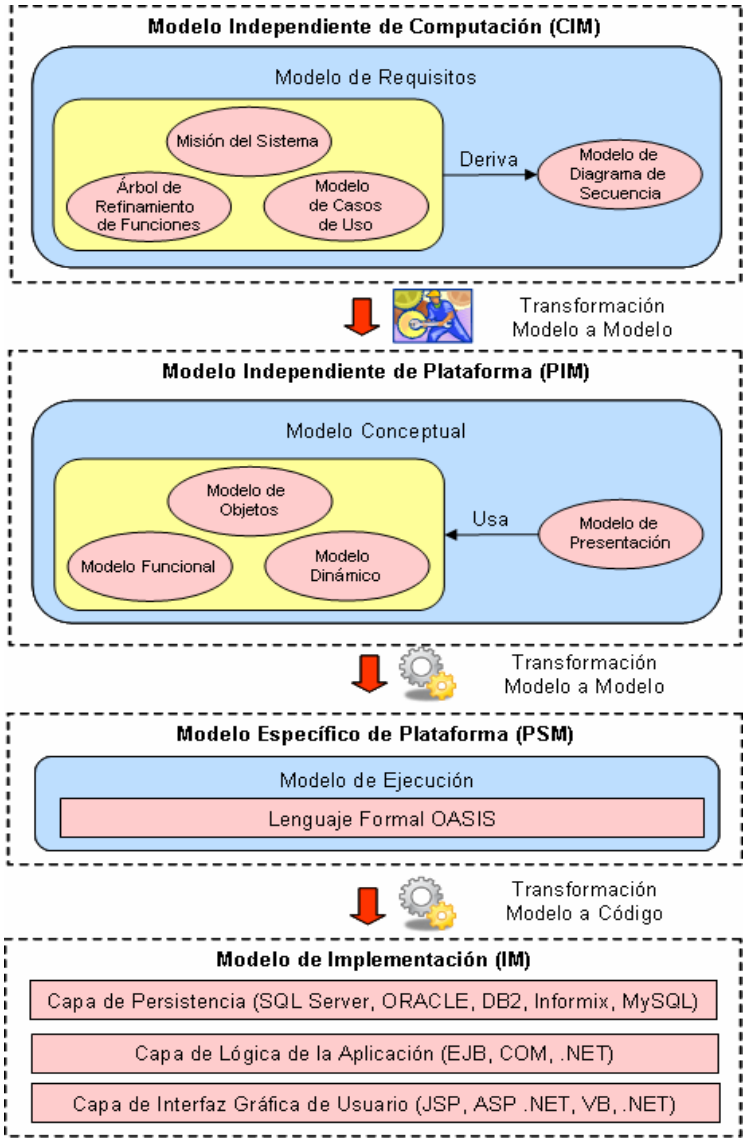


**Figura 41.** Proceso de Desarrollo de Software OO-Method

Tal como se puede observar en la Figura 41, el modelo de requisitos se compone de técnicas como la definición de la *Misión del Sistema*, el *Árbol de Refinamiento de Funciones*, los *Diagramas de Casos de Uso* y los *Diagramas de Secuencia*. Con la utilización de estas técnicas es posible definir los requisitos funcionales del sistema, que mediante transformaciones de modelos generan una

definición inicial de los modelos que componen el modelo conceptual.

El modelo conceptual (vea la Figura 41) captura las propiedades estáticas y dinámicas del sistema utilizando un *Modelo de Objetos*, un *Modelo Dinámico* y un *Modelo Funcional*. Además, el modelo conceptual permite la especificación de las interfaces de usuario de manera abstracta a través del *Modelo de Presentación*. Una vez que el analista ha especificado estos cuatro modelos, el modelo conceptual tiene todos los detalles necesarios para la generación automática de la aplicación de software. La definición completa de los elementos que componen el modelo conceptual de OO-Method está descrita en [55].

El modelo de ejecución permite especificar la forma en que será utilizado el sistema de manera abstracta, es decir, permite especificar cómo los usuarios accederán al sistema, qué acciones tendrán disponibles, y qué secuencia de acciones debe realizar para interactuar con el sistema. Para esto, el modelo de ejecución utiliza un lenguaje formal denominado OASIS [54], que permite la especificación exacta de las características de implementación de los objetos del sistema de acuerdo a la forma en que serán utilizados estos objetos.

A partir del modelo de ejecución se obtiene automáticamente el modelo de aplicación, que permite la transformación desde el espacio del problema (representado por el modelo conceptual) hacia el espacio de la solución (el producto de software correspondiente). Este modelo realiza correspondencias entre las primitivas conceptuales y sus representaciones de software para un entorno de desarrollo específico, por ejemplo Java o C#. Cabe destacar que el compilador de modelos OO-Method genera aplicaciones en arquitecturas de tres capas: una capa para el componente cliente, que contiene la interfaz gráfica; una capa para el componente servidor, que contiene las reglas de negocio y las conexiones a la base de datos; y una capa para el componente base de datos, que contiene los aspectos de persistencia de las aplicaciones.

El caso de estudio que será desarrollado en este capítulo, está centrado en la semántica de la asociación de OO-Method. Esta semántica es representada en el modelo de objetos que forma parte del modelo conceptual de OO-Method, y será revisada en la siguiente sección de este capítulo.

## 6.2. La asociación OO-Method

En esta sección se realizara una revisión general de la semántica de la asociación definida en OO-Method. Esta semántica será analizada en mayor profundidad, al explicar cada una de las clases que componen el metamodelo del DSML definido para la asociación OO-Method.

En OO-Method las asociaciones son binarias, es decir, están definidas como una relación entre dos clases. Además, la asociación en OO-Method es bidireccional, esto implica que puede ser navegada en ambas direcciones por las clases que participan en la asociación.

Para entender correctamente la semántica de la asociación OO-Method, es necesario tener en claro los siguientes conceptos básicos:

- Clases participantes de la asociación: Son las clases que están relacionadas mediante una asociación OO-Method.
- Identificador de una clase: Conjunto de atributos de la clase que permiten referenciar un objeto (instancia) particular de la clase.
- Link entre objetos: Representa una instancia de una asociación, es decir, es la relación que existe entre los objetos que pertenecen a cada una de las clases participantes.
- Asociación recursiva: Es la asociación de una clase consigo misma, o en otras palabras, las clases participantes son la misma clase.

A continuación veremos en más detalle los elementos que conforman una asociación en OO-Method y sus propiedades.

## Extremos de una asociación

- Los extremos de una asociación representan los puntos de conexión entre la asociación y las clases participantes. Dado que la asociación de OO-Method es binaria, sólo puede tener dos extremos que serán opuestos entre si.

Para definir correctamente un extremo de una asociación, es necesario especificar las siguientes tres propiedades:

- Cardinalidad: Especifica el número máximo y mínimo de objetos de una clase que pueden estar conectados con un objeto de la clase asociada.
- Temporalidad: Especifica cuando un objeto (durante su vida) puede ser conectado o desconectado de forma dinámica con uno o más objetos de la clase asociada. En este sentido se considera que si la temporalidad es *dinámica* permite la conexión y desconexión del objeto. Mientras que si la temporalidad es *estática*, una vez establecida la conexión para el objeto, esta no puede ser modificada durante la vida del mismo
- Rol: El rol indica como un extremo de la asociación es identificado por una instancia del extremo opuesto.

## Agregación

Una agregación es un tipo de asociación en la que la relación entre las clases es del tipo *"parte de"*, es decir las instancias (partes) de una clase están contenidas dentro de las instancias (todo) de la clase asociada. En este sentido, se tendrá una clase *compuesta* (sus instancias son el *todo* en la asociación), y una clase *componente* (sus instancias son las *partes* en la asociación).

En OO-Method la agregación tiene una característica particular que permite la correcta relación entre las clases participantes, esta es: la *dependencia de identificación* del todo en relación a la parte. Esta característica implica que el *identificador* de la clase *compuesta*

se define utilizando los identificadores de las clases *componentes*, más algún atributo adicional de la clase compuesta (si es necesario). La dependencia de identificación requiere que la cardinalidad asociada al extremo de la clase componente sea [1..1].

Dentro de la agregación existe un caso particular denominado *composición*, que presenta un tipo de agregación más fuerte y restrictiva. La semántica particular de la composición considera lo siguiente:

- Una instancia de la clase componente puede estar relacionada sólo en una instancia de una clase compuesta a la vez. Es decir, aún cuando una clase sea componente de varias clases, para un objeto de esta clase sólo puede establecerse un link de una de las asociaciones de composición definidas.
- Si una instancia de la clase compuesta es eliminada, entonces todas las instancias relacionadas de la clase componente también deben ser eliminadas.
- La dependencia de identificación es a la inversa que en el caso de la agregación, es decir, dependencia de identificación de la parte en relación al todo.

## Eventos compartidos

Un evento compartido es un tipo particular de servicio que define el comportamiento de una asociación dinámica. Estos eventos se denominan compartidos porque su definición está distribuida entre las clases participantes de la asociación, es decir, un evento compartido está definido al mismo tiempo en las dos clases participantes y se ejecuta al mismo tiempo en ambas clases. Esta definición distribuida se debe a que la asociación OO-Method es bidireccional, y por ejemplo, para realizar la operación de conexión de un objeto en un extremo de la asociación, se está realizando al mismo tiempo una operación de conexión con uno o varios objetos del extremo opuesto. Para representar estas operaciones

simultáneas entre clases asociadas, han sido definidos los eventos compartidos.

## 6.3. Metamodelo del DSML

En el metamodelo del DSML de la asociación OO-Method, no sólo especifica el constructor conceptual de la asociación, sino también todos aquellos constructores conceptuales que de alguna u otra manera participan en la correcta definición de una asociación en OO-Method.

Para definir este metamodelo se han seguido las recomendaciones presentadas en la sección 4.1. De manera que el metamodelo definido está basado en la especificación EMOF [35] y ha sido implementado utilizando la herramienta UML2 de Eclipse [12].

La Figura 42 muestra el Metamodelo DSML obtenido para la asociación de OO-Method. A continuación de esta figura son explicadas cada una de las clases definidas en el metamodelo del DSML.

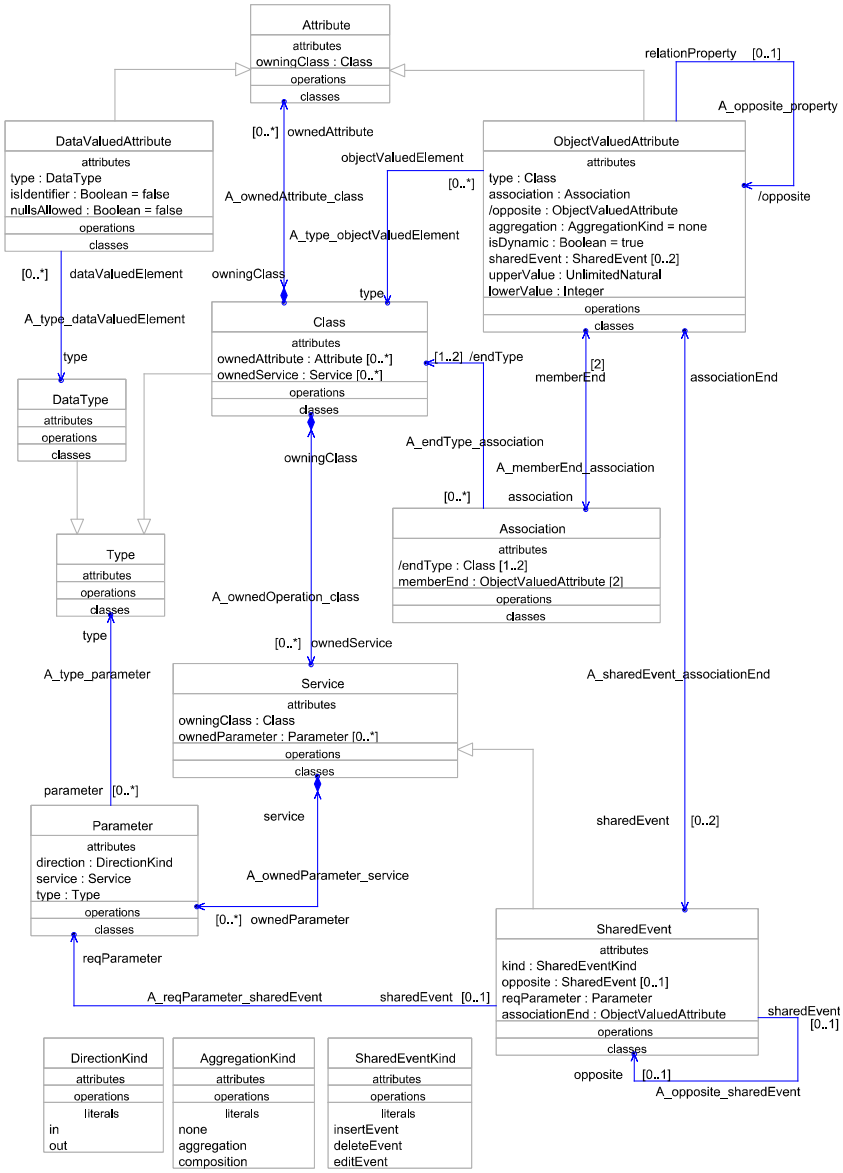


**Figura 42.** Metamodelo del DSML de la asociación OO-Method

Las clases presentadas en el metamodelo del DSML de la asociación OO-Method, representan cada uno de los constructores conceptuales involucrados en la correcta definición de la asociación de acuerdo al modelo conceptual de OO-Method. A continuación veremos en más detalles cada uno de estos constructores para

entender mejor la semántica que hay detrás de la asociación OO-Method.

La descripción de los constructores conceptuales tendrá la siguiente estructura:

- **Nombre Constructor:** Nombre del constructor conceptual de acuerdo a la especificación de OO-Method [55].
- **Nombre Clase:** Clase del Metamodelo DSML asociada al constructor conceptual.
- **Descripción:** Descripción del constructor conceptual y semántica asociada.
- **Atributos : tipo *[cardinalidad]*:** Los atributos que posee la clase, explicado la semántica de cada uno.
- **Asociaciones : tipo *[cardinalidad]*:** Al igual que los atributos, se muestra las asociaciones que posee la clase y la explicación de la semántica correspondiente.
- **Reglas OCL:** Las reglas OCL definidas en la clase, indicando como apoyan la semántica del constructor conceptual.

Antes de ver el detalle de cada una de las clases del metamodelo del DSML de la asociación OO-Method, es importante señalar que este corresponde sólo a una parte del metamodelo del DSML completo de OO-Method. En el metamodelo de OO-Method existe un número mayor de constructores y además, los constructores OO-Method considerados en este caso de estudio poseen más propiedades. Las propiedades que han sido quitadas, no son relevantes para representar la semántica de la asociación de OO-Method. Además, quitando los constructores y propiedades adicionales se obtiene un metamodelo más fácil de entender y representar.

## Asociación

**Nombre Clase:** Association

**Descripción:** Una asociación representa el link que existe entre los objetos de las clases participantes. El conjunto válido de objetos que participan en una asociación estará dado por los extremos de la asociación. En OO-Method las asociaciones son binarias, por lo tanto, en una asociación puede participar como mínimo una clase y a lo más dos clases. El caso particular donde sólo participa una clase es el de la *asociación recursiva*, que representa el link entre instancias de la misma clase.

**Atributos:** <ninguno>

**Asociaciones:**

- endType: Esta asociación indica las clases que participan en una asociación. Su valor deriva de los extremos de asociación definidos, y puede tener cardinalidad 1 en caso de las asociaciones recursivas y 2 en caso de las asociaciones normales. El valor de *endType* es derivado de los extremos de la asociación.
- memberEnd: Esta asociación identifica los extremos que participan en una asociación. Todas las asociaciones en OO-Method tienen dos extremos, incluso la asociación recursiva. En este último caso, los extremos de la asociación hacen referencia a la misma clase.

**Reglas OCL:** <ninguna>

## Atributo

**Nombre Clase:** Attribute

**Descripción:** Un atributo forma parte de una clase y describe los valores que pueden tener las instancias de la clase que lo contiene. Los valores de un atributo puede ser de dos tipos: *dato-valuados* y *objeto-valuados*. Los atributos dato-valuados representan los tipos de datos básicos de OO-Method (string, integer, date, etc.), mientras que los atributos objeto-valudos representan instancias de clases. Son precisamente los atributos objeto-valuados los que se

utilizan para establecer asociaciones entre clases (representan los extremos de la asociación).

**Atributos:** <ninguno>

**Asociaciones:**

- owningClass: Especifica la clase a la que pertenece el atributo.

**Reglas OCL:** <ninguna>

## Clase

**Nombre Clase:** Class

**Descripción:** Una clase describe las características comunes (atributos y servicios) de un conjunto de objetos que pertenecen al dominio de la aplicación. Una clase requiere un identificador que permita referenciar de forma única los objetos caracterizados. Este identificador está formado por uno o más atributos de la clase.

**Atributos:** <ninguno>

**Asociaciones:**

- ownedAttribute: Indica los atributos que posee una clase.
- ownedService: Indica los servicios que posee una clase.

**Reglas OCL:** <ninguna>

## Tipo de Dato

**Nombre Clase:** DataType

**Descripción:** Un tipo de dato representa los datos básicos (o primitivos) que existen en OO-Method, por ejemplo: string, real, integer, date, etc. Estos tipos de datos indican el conjunto de valores que pueden ser representados mediante los atributos dato-valuados.

**Atributos:** <ninguno>

**Asociaciones:** <ninguna>

**Reglas OCL:** <ninguna>

## Tipo

**Nombre Clase:** Type

**Descripción:** Un tipo establece el conjunto de valores que pueden ser representados por un parámetro. Este conjunto de valores puede ser un *tipo de dato* o una *clase*, por este motivo, la clase *Type* se representa como una generalización de las clases *DataType* y *Class*.

**Atributos:** <ninguno>

**Asociaciones:** <ninguna>

**Reglas OCL:** <ninguna>

## Atributo Dato-Valuado

**Nombre Clase:** DataValuedAttribute

**Descripción:** Un atributo dato-valuado es un atributo uni-valuado (puede tener solo un valor) y su valor debe estar dentro del conjunto de tipos de datos de OO-Method. Sólo los atributos dato-valuados pueden ser definidos como (parte del) identificador de una clase. Además, los atributos pueden tener un valor nulo (siempre y cuando se haya especificado) a excepción de aquellos atributos que participan como identificador de la clase.

**Atributos:**

- isIdentifier: Indica si el atributo dato-valuado participa como (parte del) identificador de la clase: isIdentifier = *True* → SI participa*,* isIdentifier = *False* → NO participa.

- nullsAllowed: Indica si el atributo permite valores nulos: nullsAllowed = *True* → SI permite. nullsAllowed = *False* → NO permite.

**Asociaciones:**

- type: Indica el tipo de dato OO-Method asociado al atributo dato-valuado.

**Reglas OCL:**

- Si el atributo es identificador, entonces no puede tener valores nulos:

```
self.isIdentifier = true implies self.nullsAllowed = false
```

## Atributo Objeto-valuado

**Nombre Clase:** ObjectValuedAttribute

- **Descripción:** El atributo objeto-valuado es probablemente el constructor conceptual más relevante del metamodelo DSML. Un atributo objeto-valuado representa un extremo de asociación, por lo tanto, sólo puede ser definido como participante de una asociación y no de forma independiente.

Si el extremo de asociación representado por el atributo objeto-valuado participa en un tipo de agregación (agregación o composición), entonces existe *identificación de dependencia* de la clase componente o compuesta, dependiendo del tipo de agregación especificada. En el caso de la *agregación*, la identificación de dependencia es de la clase compuesta en relación a las clases componentes. En el caso de la *composición*, la identificación de dependencia es de la clase componente en relación a la clase compuesta. La identificación de dependencia implica además que el extremo que referencia a la clase opuesta a la clase dependiente tiene cardinalidad [1..1]. La Figura 43 muestra un ejemplo de agregación en las clases: *Cliente*, *Alquiler*, y *Vehículo*.

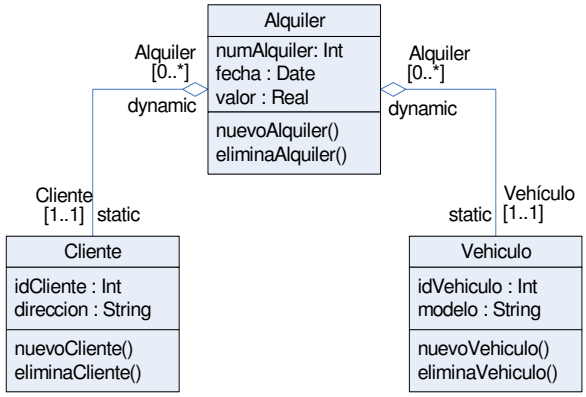


**Figura 43.** Ejemplo de una agregación

En esta figura, la clase *Alquiler* está agregada por las clases *Cliente* y *Vehículo*, es decir, un alquiler está conformado un cliente y un vehículo además de los datos del propio alquiler. Producto de la identificación de dependencia que implica la agregación, el identificador de la clase Alquiler estará compuesto por los identificadores de las clases Cliente y Vehículo más el atributo *numAlquiler* de la clase Alquiler. Este atributo adicional es definido debido a que la unión de los identificadores de *Cliente* y *Vehículo*, no asegura la referencia única de las instancias de *Alquiler*, por ejemplo: El caso en que un cliente alquile varias veces el mismo coche. En este ejemplo de agregación se puede ver claramente que debido a la identificación de dependencia asociada a la agregación, los extremos asociados a Cliente y Vehículo tienen cardinalidad [1..1] (Dependencia de identificación del todo en relación a sus partes).

**Atributos:**

- aggregation: Permite especificar (cuando sea necesario) el tipo de agregación asociado al extremo que representa el atributo objeto-valuado: aggregation = *#none* → no existe agregación para este extremo de asociación (valor por defecto), aggregation = *#aggregation* → agregación normal, aggregation = *#composition* → composición. El conjunto de posibles valores para este atributo están definidos mediante la enumeración *AggregationKind*.

- isStatic: Este atributo indica cuando un extremo de la asociación es dinámico o estático, es decir, que puede cambiar las conexiones durante la vida del o los objetos relacionado a este extremo: isStatic = *false* → extremo DINÁMICO, isStatic = *true* → extremo ESTÁTICO. Cuando ambos extremos de una asociación son dinámicos se dice que la asociación es *dinámica.*

- upperValue: Este atributo indica el límite *superior* de la cardinalidad del extremo de asociación representado por el atributo objeto-valuado.

- lowerValue: Este atributo indica el límite *inferior* de la cardinalidad del extremo de asociación representado por el atributo objeto-valuado.

- name: Este atributo permite especificar el nombre del extremo de la asociación, que corresponde al rol de este extremo en la asociación. Este atributo es heredado de la clase *NamedElement*. Sin embargo, para simplificar el modelo esta clase no es representada en la figura del caso de estudio, ya que la clase *NamedElement* es padre de todas las clases del metamodelo del DSML presentado y gráficamente, debido a todas las generalizaciones involucradas, se pierde claridad en el diagrama del metamodelo al incluir esta clase.

**Asociaciones:**

- type: Este atributo indica de que clase serán las instancias que participaran en el extremo de la asociación representado por el atributo objeto-valuado.

- opposite: Este atributo mantiene la relación que existe entre los extremos opuestos de una asociación. La cardinalidad [1..1] de este atributo obliga a que un extremo siempre tenga un opuesto, esto es debido a que en OO-Method la asociación es binaria y bidireccional.

- sharedEvent: Esta asociación permite especificar los eventos compartidos relacionados a un extremo de asociación. Los eventos compartidos permiten controlar las conexiones y

desconexiones de instancias que participan en una asociación, siempre y cuando ambos extremos de la asociación sean dinámicos (asociación dinámica). Los eventos compartidos pueden ser de tres tipos: 1) de creación de asociaciones (*insertEvent*) 2) de eliminación de asociaciones (*deleteEvent*) y 3) de edición de asociaciones (*editEvent*). Más detalle de la semántica de los eventos compartidos se encuentra en la descripción de la clase *SharedEvent*.

- **Reglas OCL:**
- Sólo un extremo de una asociación puede participar en una agregación:

```
self.aggregation <> #none implies
self.opposite.aggregation = #none
```

- Si el extremo de asociación participa en una *agregación,* entonces el extremo opuesto tiene cardinalidad [1..1]:

```
self.aggregation = #aggregation implies
self.opposite.lowerValue = 1 and
self.opposite.upperValue = 1
```

- El extremo de asociación que participa en una *composición* debe tener cardinalidad [1..1]:

```
self.aggregation = #composite implies self.lowerValue =
1 and self.upperValue = 1
```

- Un evento compartido se puede definir sólo cuando ambos extremos de la asociación son dinámicos (asociación dinámica).

```
self.isStatic = true or self.opposite.isStatic = true
implies self.sharedEvent.isEmpty()
```

- Sólo el evento compartido *editEvent* puede ser definido en asociaciones dinámicas, cuando uno de los extremos tiene cardinalidad [1..1]

```
((self.isStatic = false) and (self.opposite.isStatic =
false) and (self.lowerValue = 1) and (self.upperValue =
1)) implies self.sharedEvent.kind = #editEvent and
self.sharedEvent->size() = 1
```

- Los eventos compartidos de inserción y eliminación de asociaciones pueden ser definidos sólo cuando ambos extremos

de asociación tienen cardinalidad [X..*] (con X mayor o igual que 0).

```
((self.isStatic = false) and (self.opposite.isStatic =
false) and (self.upperValue > 1) and
self.opposite.upperValue > 1)) implies
self.sharedEvent->size() = 2 and self.sharedEvent-
>exists(kind = #insertEvent) and self.sharedEvent-
>exists(kind = #deleteEvent)
```

## Parámetro

**Nombre Clase:** Parameter

**Descripción:** Un parámetro es parte de un servicio y representa un valor de entrada que es requerido para la ejecución del servicio, o un valor de salida que es generado por la ejecución del servicio.

**Atributos:**

- direction: Este atributo especifica si el parámetro corresponde a un valor de entrada (parámetro de entrada) o salida (parámetro de salida): direction = *in* → Parámetro de ENTRADA, direction = *out* → Parámetro de SALIDA. El conjunto de posibles valores para este atributo están definidos mediante la enumeración *DirectionKind*.

**Asociaciones:**

- type: Esta asociación indica el conjunto de posibles valores que un parámetro puede representar. A diferencia de los atributos que se especializan para representar objetos (atributo objeto-valuado) o tipos de datos (atributo dato-valuados), los parámetros pueden representar ambos tipos (tanto objetos como tipos de datos).
- service: Esta asociación indica el servicio que contiene el parámetro.

**Reglas OCL:** <ninguna>

## Servicio

**Nombre Clase:** Service

**Descripción:** Un servicio describe un comportamiento específico que puede ser ejecutado por un objeto. En el metamodelo del DSML presentado, los servicios se especializan en eventos compartidos (clase *SharedEvent*). Sin embargo, en el metamodelo del DSML completo de OO-Method, los servicios (representados por la clase *Service*) son especializados en *eventos* y *transacciones* representados por las clases *Event* y *Transaction* respectivamente. Un evento es una unidad de procesamiento atómica que representa el cambio de estado de una clase. Una transacción es un servicio que puede agrupar eventos y otras transacciones y no puede cambiar el estado de la clase directamente, sino que lo hace a través de los eventos que contiene.

En el metamodelo de OO-method un evento compartido es en realidad una especialización de la clase *Event*, sin embargo, para simplificar el Metamodelo DSML, las clases que representan la semántica de eventos y transacciones no se han definido en el metamodelo del DSML de la asociación OO-Method, ya que no son relevantes para representar correctamente la semántica de la asociación. De esta manera, sólo la clase *SharedEvent*, que representa a los eventos compartidos, es definida en el metamodelo como una especialización directa de la clase *Service*.

**Atributos:** <ninguno>

**Asociaciones:**

- owningClass: Esta asociación indica la clase que contiene al servicio.
- ownedParameter: Esta asociación indica los parámetros que posee el servicio. En OO-Method, un servicio siempre tiene un parámetro de entrada definido por defecto que estará dado por la clase que contiene al servicio. El valor de este parámetro es obtenido en OO-Method mediante el comando *this*, que es

equivalente al comando *self* en OCL [37]. Este parámetro de entrada por defecto no necesita ser definido en el modelo conceptual OO-Method, porque su semántica está implícita en la semántica del servicio. De esta manera, aún cuando un servicio tiene al menos un parámetro (*this*), la cardinalidad mínima de la asociación *ownedParameter* es 0.

**Reglas OCL:** <ninguna>

## Evento Compartido

**Nombre Clase:** SharedEvent

**Descripción:** Un evento compartido es un servicio que representa el comportamiento de las *asociaciones dinámicas*. Una asociación toma la condición de *dinámica* cuando ambos extremos de la asociación son dinámicos, en caso contrario la asociación no es considerada dinámica, ya que los links establecidos en la creación de los objetos no pueden ser cambiados durante la vida de los objetos participantes. La Figura 44 ejemplifica esta situación.

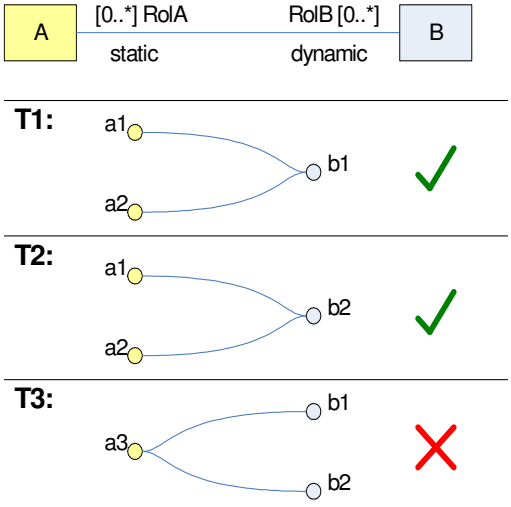


**Figura 44.** Ejemplo de temporalidad en la asociación

La Figura 44 muestra una asociación entre las clases *A* y *B* con cardinalidad [0..*] en ambos extremos. La temporalidad del

extremo de la asociación correspondiente a la clase *A* (*RolA*) es estático (*static*), mientras que el extremo de la asociación correspondiente a la clase *B* (*RolB*) es dinámico (*dynamic*). Para esta asociación se analizan tres posibles situaciones de *links* entre objetos. Estas tres situaciones son analizadas en tres instantes de tiempos consecutivos, representados en *T1*, *T2* y *T3*.

En el instante *T1* se crea una nueva instancia *b1* de la clase *B* y se asocia con dos instancias existentes de la clase *A*: *a1* y *a2*. Esta operación de creación y asociación se puede llevar a cabo sin problemas ya que el extremo *RolB* es dinámico, y por lo tanto, las instancias *a1* y *a2* pueden cambiar las asociaciones que poseen con instancias de la clase B. En este caso particular, el cambio ha implicado agregar una nueva asociación con la instancia *b1*.

En el instante *T2* se crea la instancia *b2* (de la clase *B*) y al igual que la instancia *b1* creada en T1, *b2* es asociada con las instancias *a1* y *a2*, cambiando nuevamente el conjunto de asociaciones definidas para *a1* y *a2.*

Finalmente, en *T3* se crea la instancia *a3* de la clase *A* y se asocia con las instancias *b1* y *b2* (de la clase *B*) creadas previamente. Esta vez la creación de la instancia *a3* con las asociaciones propuestas no es posible, debido a que *RolA* es estático y por lo tanto las instancias *b1* y *b2* no pueden cambiar el conjunto de asociaciones definidos al momento de su creación. Además, tampoco es posible cambiar los *links* establecidos entre las instancias de las clases *A* y *B*, y sólo podrán ser eliminados cuando una las instancias participantes sea eliminada. Entrando en un poco más de detalle, se puede apreciar que las instancias de la clase *A* no pueden ser eliminadas sin eliminar también las instancias relacionadas de la clase *B*, esto es debido a que la eliminación de una instancia de la clase *A* cambia el conjunto de asociaciones de las instancias relacionadas de la clase *B*. Entonces, al ser el extremo *RolA* estático este cambio no puede ser realizado, dejando como única alternativa posible para eliminar una instancia de la clase *A*, la

eliminación de las instancias relacionadas de la clase *B* (en caso que estas existan).

Los eventos compartidos son clasificados en tres tipos distintos de acuerdo al comportamiento que representan en el manejo de asociaciones:

- **Evento compartido de inserción:** Este tipo de evento compartido establece un link entre un objeto de la clase relacionada con un extremo de la asociación, y un objeto de la clase relacionada con el extremo opuesto.
- **Evento compartido de eliminación:** Los eventos compartidos de eliminación destruyen o eliminan un link que existe entre un objeto que pertenece a la clase relacionada a un extremo de la asociación, y un objeto perteneciente a la clase relacionada al extremo opuesto.
- **Evento compartido de edición:** Este tipo de evento compartido cambia un link existente entre un objeto que pertenece a la clase relacionada a un extremo de la asociación, y un objeto perteneciente a la clase relacionada al extremo opuesto. Este cambio se realiza reemplazando el objeto relacionado al extremo opuesto, por otro objeto de la misma clase. Viéndolo de una forma más sencilla, un evento compartido de edición puede ser considerado como la combinación de un evento de eliminación con uno de inserción.

Un evento compartido es contenido simultáneamente por las dos clases participantes de la asociación y su definición puede ser personalizada de forma independiente en cada clase participante. De esta manera, es posible decir que un evento compartido tiene una definición distribuida entre las clases que participan en la asociación. Para representar esta definición distribuida en el metamodelo del DSML, un evento compartido es representado como dos eventos dependientes con el mismo tipo y nombre. Cada uno de estos eventos dependientes es definido en las clases que participan en la asociación. Existe una situación particular para la dependencia

entre los eventos que representan un evento compartido, esta situación se presenta en la definición de asociaciones recursivas. En las asociaciones recursivas dinámicas la definición de un evento compartido no es distribuida, ya que las clases participantes son en realidad una misma clase y por lo tanto basta con la definición de un solo evento.

Un evento compartido siempre requiere de dos parámetros de entrada que representan: 1) los objetos que serán asociados, en el caso de un evento compartido de inserción, ó 2) los objetos que ya están asociados, en el caso de los eventos compartidos de edición y eliminación. Uno de estos parámetros está dado por el objeto que ejecuta el evento compartido. El valor de este objeto es obtenido automáticamente mediante el comando *this* en OO-Method, o *self* en OCL, por lo tanto no necesita ser especificado en el metamodelo. En cambio, el segundo parámetro si debe ser especificado en el metamodelo y corresponde al segundo objeto que participa en la creación, edición o eliminación de la asociación. Este objeto pertenece a la clase relacionada al extremo opuesto de la asociación, por lo tanto, la clase participante puede ser obtenida a través del servicio compartido opuesto (asociación *opposite*), mediante la asociación *owningClass* heredada de la clase *Service*.

Los eventos compartidos están relacionados a las asociaciones dinámicas, y su definición dependerá de la cardinalidad que tengan los extremos de la asociación. Para una asociación que posea un extremo de asociación con cardinalidad [1..1], no es posible eliminar un link existente o crear un nuevo link de forma independiente a la creación o eliminación de los objetos participantes. Esta situación se produce porque un extremo de asociación con cardinalidad [1..1] obliga a establecer un link (sólo uno) cuando un objeto de la clase asociada al extremo opuesto es creado. En este caso particular, la única alternativa de manipulación de la asociación es reemplazar el link existente por otro mediante un evento compartido de edición. En cualquier otro caso, las asociaciones son manejadas mediante los eventos compartidos de inserción y eliminación.

Un evento compartido sólo puede ser definido dentro del contexto de una asociación, y como antes se ha señalado, es definido mediante dos instancias dependientes de la clase SharedEvent (excepto para el caso particular de la asociación recursiva). Cada una de estas instancias debe estar ligada a cada extremo de la asociación, que son representados mediante instancias de la clase *ObjectValuedAttribute*.

**Atributos:**

- kind: Este atributo especifica si el evento compartido es de inserción, edición, o eliminación: kind = insertEvent → Evento compartido de inserción, kind = deleteEvent → Evento compartido de eliminación, kind = editEvent → Evento compartido de edición. El conjunto de posibles valores para este atributo están definidos mediante la enumeración *SharedEventKind*.

**Asociaciones:**

- opposite: Esta asociación representa la dependencia que existe entre las dos instancias de la clase *SharedEvent*, que son necesarias para definir correctamente un evento compartido.
- *reqParameter*: Indica el argumento de entrada necesario para identificar el objeto perteneciente a la clase relacionada al extremo opuesto de la asociación.
- *associationEnd*: Esta asociación indica el extremo de la asociación que está relacionado con cada instancia de la clase *SharedEvent*. La cardinalidad de esta asociación es [1..1], debido a que un evento compartido sólo puede ser definido dentro del contexto de una asociación.

**Reglas OCL:**

- Un evento compartido es definido mediante dos instancias de la clase *SharedEvent* dependientes entre si, excepto en asociaciones recursivas. La dependencia se establece mediante la asociación *opposite*.

```
self.associationEnd.type <>
self.associationEnd.opposite.type implies
self.opposite.notEmpty and self.kind =
sel.opposite.kind and self.name = self.opposite.name
```

- En asociaciones recursivas una instancia de la clase *SharedEvent* se define de forma independiente, es decir, no requiere tener una segunda instancia relacionada mediante la asociación *opposite*.

```
(self.associationEnd.type =
self.associationEnd.opposite.type)) implies
self.opposite.isEmpty
```

- Un evento compartido no puede ser dependiente de si mismo, es decir, las instancias de la clase *SharedEvent* que representan un evento compartido deben ser distintas una de la otra.

```
self.opposite.notEmpty implies self <> self.opposite
```

Con la especificación de la clase *SharedEvent* se completa el metamodelo del DSML correspondiente a la asociación de OO-Method. Continuando la ejecución del proceso completo, el siguiente paso corresponde a la generación del Metamodelo de Integración a partir del metamodelo del DSML definido.

## *6.4. Metamodelo de Integración*

Para generar correctamente el Metamodelo de Integración que permite integrar en UML la semántica de la asociación OO-Method, seguiremos la propuesta sistemática presentada en la sección 4.3 que contempla los siguientes pasos:

1. Realizar el mapeo de equivalencias entre el metamodelo del DSML y la Superestructura de UML.
2. Validar las reglas OCL.
3. Resolver problemas de mapeo.

Tal como se señala la propuesta sistemática, estos pasos son iterativos y se repiten secuencialmente hasta resolver todos los problemas de mapeo.

## Mapeo de equivalencias

El mapeo de equivalencias entre el metamodelo del DSML definido para la asociación OO-Method y la Superestructura de UML [48], es presentado a continuación en la Tabla 2.

**Tabla 2.** Equivalencias entre el metamodelo del DSML de la asociación OO-Method y la Superestructura de UML.

| **Metamodelo DSML** | **Superestructura UML** |
|---|---|
| **Association** | **Association** |
| .endType | .endType |
| .memberEnd | .memberEnd |
| **Attribute** | **Property** |
| .owningClass | .class |
| **Class** | **Class** |
| .ownedAttribute | .ownedAttribute |
| .ownedService | .ownedOperation |
| **DataType** | **DataType** |
| **DataValuedAttribute** | **Property** |
| .type | .type |
| .isIdentifier | -- |
| .nullsAllowed | -- |
| **ObjectValuedAttribute** | **Property** |
| .type | .type |
| .association | .association |
| .opposite | .opposite |
| .aggregation | .aggregation |
| .isStatic | .isReadOnly |
| .sharedEvent | -- |
| .upperValue | .upper |
| .lowerValue | .lower |
| **Parameter** | **Parameter** |
| .direction | .direction |

| **Metamodelo DSML** | **Superestructura UML** |
|---|---|
| .service | .operation |
| .type | .type |
| **Service** | **Operation** |
| .owningClass | .class |
| .ownedParameter | .ownedParameter |
| **SharedEvent** | **Operation** |
| .kind | -- |
| .opposite | -- |
| .reqParameter | -- |
| associationEnd | -- |
| **Type** | **Type** |
| **DirectionKind** | **ParameterDirectionKind** |
| . in | .in |
| .out | .out |
| **AggregationKind** | **AggregationKind** |
| .none | .none |
| .aggregation | .shared |
| .composition | .composite |
| **SharedEventKind** | **--** |
| .insertEvent | -- |
| .deleteEvent | -- |
| .editEvent | -- |

## Validar reglas OCL

De acuerdo al segundo paso de la propuesta sistemática, es necesario validar que las reglas OCL definidas en el metamodelo del DSML no produzcan conflictos con las reglas OCL definidas en el metamodelo de UML. Para realizar esta validación, es necesario considerar las equivalencias entre las clases del metamodelo del DSML y las clases de la superestructura de UML definidas en al paso previo.

Las clases equivalentes DSML que poseen reglas OCL son: *DataValuedAttribute*, *ObjectValuedAttribute* y *SharedEvent*. De estas clases sólo se analizarán las reglas OCL que presentan algún

tipo de coincidencia con las reglas de UML, ya que precisamente estas reglas pueden producir algún tipo de conflicto.

En la clase *DataValuedAttribute* la única regla OCL definida no produce conflictos, ya que se aplica sobre un atributo que no tiene equivalencia en el metamodelo de UML.

En la clase equivalente *ObjectValuedAttribute*, será necesario validar la regla que restringe la cardinalidad en la composición, ya que la restricción coincide con la regla OCL definida en la clase UML *Property* destinada a validar la cardinalidad máxima en caso de composición. En este caso, el conflicto se puede producir debido a que ambas reglas validan la cardinalidad del extremo de la asociación que participa en la composición, estas reglas son:

- Restricción del Metamodelo del DSML:

```
self.aggregation = #composite implies self.lowerValue =
1 and self.upperValue = 1
```

- Restricción del la Superestructura de UML:

```
isComposite implies (upperBound()->isEmpty() or
upperBound() <= 1)
```

La regla de UML valida que el límite superior de la cardinalidad del extremo de la asociación definido como composición debe tener como máximo cardinalidad 1, permitiendo especificar un valor nulo o cero. La regla del metamodelo del DSML es más estricta, ya que obliga a que la cardinalidad sea solamente 1 y no otro valor, además también restringe la cardinalidad mínima a 1, ya que lo exige la identificación de dependencia. Aún cuando ambas reglas validan un elemento común (cardinalidad del extremo de la asociación), su aplicación conjunta no produce conflictos. El resto de las reglas OCL de la clase equivalente *ObjectValuedAttribute* no presentan coincidencia con los elementos restringidos en las reglas del metamodelo de UML, por lo tanto no presentan conflictos de validación.

En la clase equivalente *SharedEvent* no existen conflictos con las reglas OCL de la clase UML referenciada (*Operation*), ya que todas

las reglas OCL de *SharedEvent* tienen efecto sobre la asociación o*pposite* que no tiene equivalencia en el metamodelo de UML.

## Resolver problemas de mapeo

De acuerdo a la propuesta sistemática para generar el Metamodelo de Integración (sección 4.3), para identificar los problemas de mapeo se deben utilizar las reglas definidas en la sección 4.2. Con estas reglas se identificarán:

1. Clases no mapeadas
2. Mapeo M:1 de las propiedades de una misma clase.
3. Mapeo entre elementos de distinto tipo, por ejemplo, atributos mapeados con asociaciones.
4. Mapeo 1:M.

Al evaluar estas reglas en el metamodelo del DSML elaborado para la asociación OO-Method, obtenemos como resultado que todas las reglas definidas para determinar si un Metamodelo de Integración es correcto se cumplen satisfactoriamente. De esta manera, no es necesario realizar una nueva iteración de de la propuesta sistemática para la generación del Metamodelo de Integración. Finalmente, el Metamodelo del DSML junto con el mapeo de equivalencias definido, se convierten en el Metamodelo de Integración que permitirá generar automáticamente el perfil UML para integrar en UML la semántica de la asociación de OO-Method.

La generación automática del perfil UML a partir del Metamodelo de Integración se realiza mediante los pasos 3 y 4 del proceso completo.

## 6.5. Perfil UML de la Asociación OO-Method

El proceso de generación automática del perfil UML comienza por la identificación de las extensiones que deben ser definidas sobre la Superestructura de UML, para integrar la semántica descrita en el metamodelo del DSML de la asociación OO-Method. Luego, a partir del Metamodelo de Integración más las extensiones identificadas, se obtendrá el perfil UML final.

### Identificación de extensiones.

De acuerdo a la propuesta para la identificación automática de extensiones presentada en la sección 5.1, las extensiones se identifican mediante las diferencias que existan entre el Metamodelo de Integración y la Superestructura de UML, considerando: 1) los elementos del Metamodelo de Integración que no poseen equivalencia con la Superestructura de UML (elementos nuevos), y 2) las propiedades equivalentes del metamodelo del DSML que presentan diferencias de cardinalidad o tipo, en relación a las propiedades referenciadas de la Superestructura de UML. De esta manera, se obtienen el resultado presentado en la Tabla 3. Esta tabla muestra los elementos nuevos definidos en el Metamodelo de Integración, así como las propiedades equivalentes que presentan diferencias. Los elementos identificados en el Metamodelo de Integración que presentan información de las extensiones son presentados en la columna *Metamodelo DSML*, mientras que las diferencias encontradas son presentadas en la columna *Diferencia*. En la comparación se utilizan las siglas *M.I.* para identificar el metamodelo de integración, y las siglas *UML* para la Superestructura de UML.

**Tabla 3.** Resultado de la comparación para el Metamodelo de Integración de la asociación OO-Method.

| **Metamodelo Integración** | **Diferencia** |
|---|---|
| **Association** | |
| .endType | Diferente cardinalidad superior:<br>M.I. = 2<br>UML = *<br>Diferente tipo:<br>M.I. = Class<br>UML = Type |
| .memberEnd | Diferente cardinalidad superior:<br>M.I. = 2<br>UML = * |
| **DataValuedAttribute** | |
| .type | Diferente tipo:<br>M.I. = DataType<br>UML = Type |
| .isIdentifier | Atributo Nuevo |
| .nullsAllowed | Atributo Nuevo |
| **ObjectValuedAttribute** | |
| .type | Diferente tipo:<br>M.I. = Class<br>UML = Type |
| .association | Diferente cardinalidad inferior:<br>M.I. = 1<br>UML = 0 |
| .opposite | Diferente cardinalidad inferior:<br>M.I. = 1<br>UML = 0 |
| .sharedEvent | Asociación Nueva |
| **SharedEvent** | |
| .kind | Atributo Nuevo |
| .opposite | Asociación Nueva |
| .reqParameter | Asociación Nueva |
| .associationEnd | Asociación Nueva |
| **SharedEventKind** | Enumeración nueva |
| .insertEvent | Valor Literal Nuevo |
| .deleteEvent | Valor Literal Nuevo |
| .editEvent | Valor Literal Nuevo |

Una vez identificadas las extensiones que deben ser definidas en el metamodelo de UML, se utilizará este resultado para transformar el Metamodelo de Integración en el perfil UML final.

## Transformación del Metamodelo de Integración

La transformación del Metamodelo de Integración se llevará a cabo utilizando las reglas de transformación definidas en la sección 5.2. Estas reglas de transformación requieren además del Metamodelo de Integración, las extensiones identificadas previamente.

Para aplicar las reglas de transformación primero se deben aplicar las reglas que no presentan dependencia de otras reglas, es decir, no son extensiones o deben ser aplicadas dentro del contexto de otras reglas de transformación. Las reglas que cumplen esta condición son: la regla 1 asociada a clases equivalentes y las reglas 6 y 7, asociadas a enumeraciones. En particular, las reglas 1 y 6 son la que poseen una condición de identificación que se aplica al Metamodelo de Integración de la asociación OO-Method:

- **Regla 1:** *Por cada clase equivalente se definirá un estereotipo que extienda a la clase UML referenciada. El nombre del estereotipo debe ser el nombre de la clase equivalente*. Esta regla se debe aplicar a todas las clases del Metamodelo de Integración. Esta regla se aplica a todas las clases definidas en el Metamodelo de Integración. Además para aplicar correctamente la regla 1, se requiere de un prefijo para los casos en que el nombre del estereotipo coincida con el nombre de la clase, este prefijo será *OOm*.
- **Regla 6:** *En el perfil UML se debe definir una enumeración por cada enumeración nueva. La enumeración definida debe tener todos los valores literales de la enumeración equivalente*. Esta regla se aplica a la enumeración *SharedEventKind*.

Con la aplicación de estas reglas de transformación obtenemos el resultado presentado en la Figura 45.

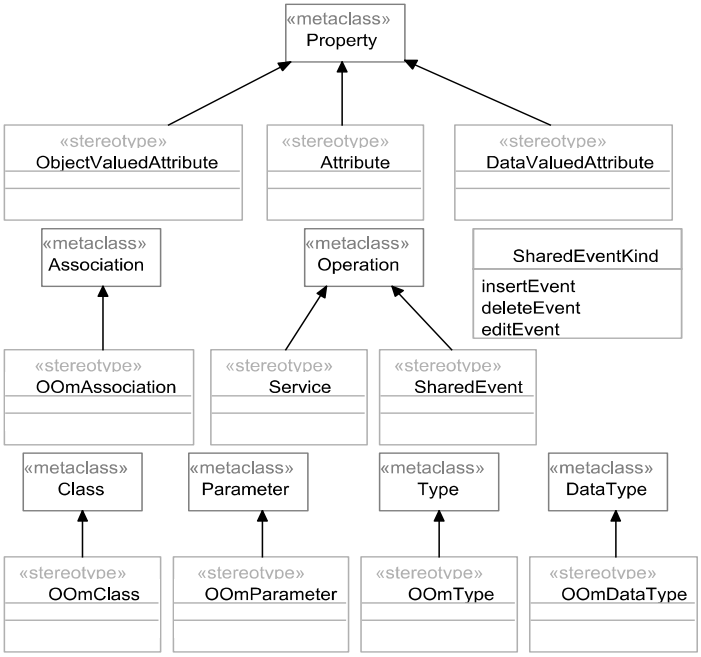


**Figura 45.** Aplicación de reglas de transformación 1 y 6.

Sin embargo, el resultado presentado en la Figura 45 no es completo, ya que deben ser aplicadas aquellas reglas de transformación que extienden a la regla 1, estas son las reglas de transformación asociadas a las generalizaciones (reglas 9, 10 y 11). De estas reglas, sólo las reglas 9 y 11 pueden ser aplicadas sobre el Metamodelo de Integración, la descripción de estas reglas se muestra a continuación:

- **Regla 9:** *Para las clases equivalentes asociadas por una generalización nueva y ambas clases están mapeadas a la misma clase UML, se debe definir un estereotipo por cada clase y entre los estereotipos se debe definir generalización idéntica a la generalización nueva. De los estereotipos definidos, sólo el estereotipo padre (de acuerdo a la asociación de generalización) extenderá a la clase UML referenciada*. Esta regla se aplica a la

generalizaciones entre la clase *Attribute* y las clases *DataValuedAttribute* y *ObjectValuedAttribute*, y a la generalización entre la clase *Service* y *SharedEvent*.

- **Regla 11:** *Para las clases equivalentes asociadas por una generalización equivalente, se debe definir un estereotipo por cada clase equivalente que extienda a la respetiva clase UML y para el estereotipo que representa a la clase equivalente hija se debe definir una regla OCL que garantice que previamente se debe aplicar sobre la clase UML el estereotipo que representa a la clase equivalente padre.* Esta regla se aplica a las generalizaciones definidas entre la clase *Type* y as clases *DataType* y *Class*.

Al extender la regla de transformación número 1 con las reglas 9 y 11, se obtiene el resultado presentado en la Figura 46.

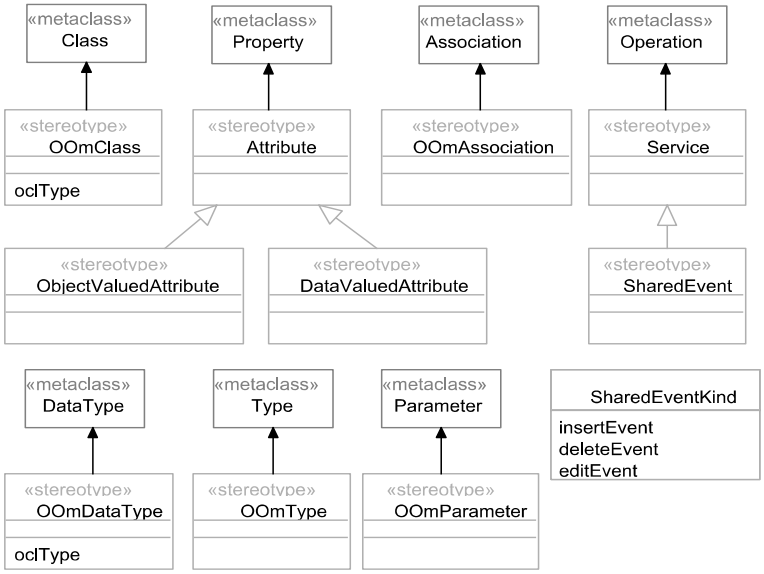


**Figura 46.** Aplicación de las reglas de transformación 9 y 11

Tal como se puede apreciar en la Figura 46, de acuerdo a la regla 11, se ha generado la regla OCL *oclType* para las clases DataType y Class. La definición de esta regla se puede observar a continuación:

- oclType: `self->oclIsTypeOf(OOmType)`

Una vez aplicadas las reglas de transformación 9 y 11, continuaremos con la aplicación de las reglas asociadas a atributos (reglas 2 a la 5) y asociaciones (reglas 12 a la 16), que se encuentran dentro del contexto de la regla 1. También se encuentra dentro de este contexto la regla de transformación número 8 (asociada a enumeraciones). Para el caso de las reglas asociadas a atributos sólo la regla 2 debe ser aplicada al Metamodelo de Integración:

- **Regla 2:** *Por cada atributo nuevo se debe definir un valor etiquetado que represente a dicho atributo. El valor etiquetado se define dentro del estereotipo que corresponde a la clase equivalente que contiene el atributo nuevo. El nombre del valor etiquetado debe ser el nombre del atributo nuevo. Las propiedades del valor etiquetado deben ser iguales a las propiedades del atributo nuevo*. Esta regla se aplica a los atributos: *DataValuedAttribute.isIdentifier*, *DataValuedAttribute.nullsAllowed*, y *SharedEvent.kind*.

En el caso de las reglas de transformación ligadas a asociaciones, las reglas que serán aplicadas son:

- **Regla 12:** *Para las asociaciones nuevas se debe definir un valor etiquetado en el estereotipo que representa la clase equivalente que contiene la asociación. Este valor etiquetado tiene las mismas propiedades que la asociación nueva que representa*. Esta regla se aplica a las asociaciones: *ObjectValuedAttribute.sharedEvent*, *SharedEvent.opposite*, *SharedEvent.reqParameter* y *SharedEvent.associationEnd*.

- **Regla 13:** *Para las asociaciones equivalentes que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite inferior de la cardinalidad de la asociación equivalente es mayor que el límite inferior de la asociación UML referenciada, se debe definir una restricción para ajustar la cardinalidad de la asociación UML a la de la asociación equivalente*. Esta regla se aplica a las asociaciones *association* y *opposite*, pertenecientes a la clase *ObjectValuedAttribute*. Las reglas OCL generadas por

esta regla de transformación serán *oclAssociationLower* y *oclOppositeLower*, estas reglas quedarán definidas como se muestra a continuación:

- oclAssociationLower: `self.[association]->size() > 0`
- oclOppositeLower: `self.[opposite]->size() > 0`

- **Regla 14:** *Para las asociaciones equivalentes que presenten diferencias de cardinalidad, y esta diferencia corresponda a que el límite superior de la cardinalidad de la asociación equivalente es menor que el límite superior de la asociación UML referenciada, se debe definir una restricción para ajustar la cardinalidad de la asociación UML a la de la asociación equivalente*. Esta regla se aplica a las asociaciones *endType* y *memberEnd*, pertenecientes a la clase *Association*. Las reglas OCL generadas por esta regla de transformación serán *oclEndTypeUpper* y *oclMemberEndUpper*, estas reglas quedarán definidas como se muestra a continuación:
  - oclEndTypeUpper: `self.endType->size() < 2`
  - oclMemberEndUpper: `self.memberEnd->size() < 2`

- **Regla 15:** *Para las asociaciones equivalentes, se debe definir una restricción en la clase UML que contiene la asociación UML referenciada, para garantizar que a la clase UML que participa en la asociación esté extendida por el estereotipo correspondiente a la clase equivalente que participa en la asociación equivalente*. Las asociaciones afectadas por esta regla son todas las asociaciones del Metamodelo de Integración, excepto aquellas identificadas como asociaciones nuevas en la tabla de comparación (Tabla 3). Las reglas OCL generadas por esta transformación se muestran en la Tabla 4.

**Tabla 4.** Reglas OCL generadas por la regla de transformación número 15

| **Nombre:** oclEndType | **Asoc:** Association.endType |
|---|---|
| `self.endType->oclIsTypeOf(OOmClass)` | |
| **Nombre:** oclMemberEnd | **Asoc:** Association.memberEnd |
| `self.memberEnd->oclIsTypeOf(ObjectValuedAttribute)` | |
| **Nombre:** oclOwningClass | **Asoc:** Attribute.owningClass |
| `self.class->oclIsTypeOf(OOmClass)` | |
| **Nombre:** oclOwnedAttribute | **Asoc:** Class.ownedAttribute |
| `self.ownedAttribute->oclIsTypeOf(Attribute)` | |
| **Nombre:** oclOwnedService | **Asoc:** Class.ownedService |
| `self.ownedOperation->oclIsTypeOf(Service)` | |
| **Nombre:** oclType | **Asoc:** DataValuedAttribute.type |
| `self.type->oclIsTypeOf(OOmDataType)` | |
| **Nombre:** oclType | **Asoc:** ObjectValuedAttribute.type |
| `self.type->oclIsTypeOf(OOmClass)` | |
| **Nombre:** oclAssociation | **Asoc:** ObjectValuedAttribute.association |
| `self.association->oclIsTypeOf(OOmAssociation)` | |
| **Nombre:** oclOpposite | **Asoc:** ObjectValuedAttribute.opposite |
| `self.opposite->oclIsTypeOf(ObjectValuedAttribute)` | |
| **Nombre:** oclService | **Asoc:** Parameter.service |
| `Self.operation->oclIsTypeOf(Service)` | |
| **Nombre:** oclType | **Asoc:** Parameter.type |
| `self.type->oclIsTypeOf(OOmType)` | |
| **Nombre:** oclOwningClass | **Asoc:** Service.owningClass |
| `self.class->oclIsTypeOf(OOmClass)` | |
| **Nombre:** oclOwnedParameter | **Asoc:** Service.ownedParameter |
| `Self.ownedParameter->oclIsTypeOf(OOmParameter)` | |

La regla de transformación número 8 se debe aplicar a la enumeración *DirectionKind*, esta regla señala que:

- **Regla 8:** *Para cada enumeración equivalente que contenga sólo valores literales equivalentes, y que estos valores literales sean menos que los valores literales de la enumeración UML, se debe*

*definir una regla OCL para restringir los valores literales de la enumeración UML que no corresponden al contexto del DSML*. Esta regla será aplicada al atributo UML *Parameter.direction*, generando las siguientes reglas OCL:

- oclInout: `self.[direction] <> #inout`
- oclReturn: `self.[direction] <> #return`

La Figura 47 muestra el resultado obtenido una vez aplicadas las reglas de transformación asociadas a atributos, asociaciones y la regla 8 de enumeraciones.

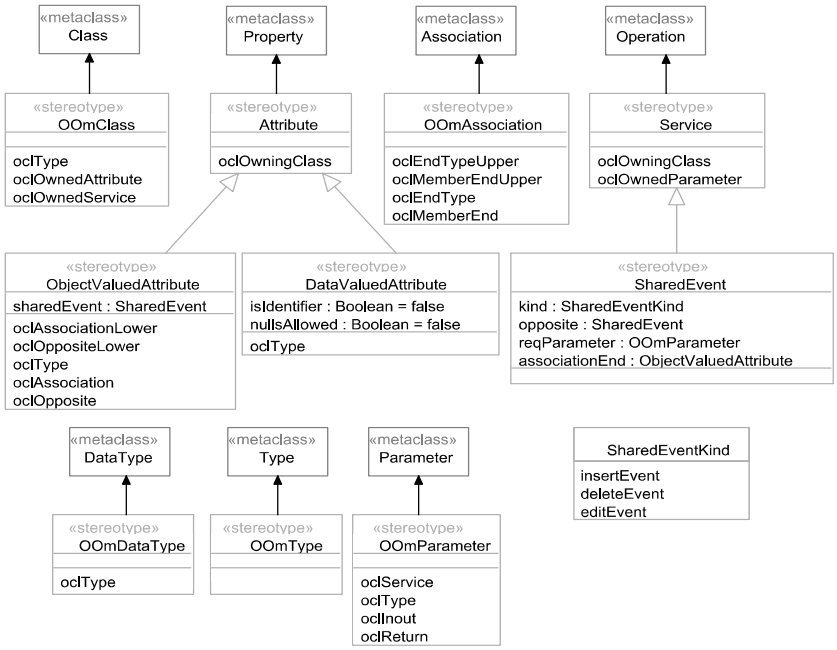


**Figura 47.** Aplicación de las reglas de transformación para atributos y asociaciones

Una vez aplicadas las reglas de transformación para atributos y asociaciones, corresponde aplicar la regla de transformación para las reglas OCL definidas en el Metamodelo de Integración (Regla 17). La descripción de esta regla la podemos ver a continuación:

- **Regla 17:** *Las reglas OCL definidas en las clases equivalentes deben ser incluidas en los estereotipos generados a partir de*

*estas clases*. Esta regla se debe aplicar a todas las reglas OCL definidas en el Metamodelo de Integración.

Para simplificar el caso de estudio la aplicación de esta regla será omitida, ya que simplemente consiste en repetir las reglas OCL definidas para el metamodelo del DSML de la asociación OO-Method. El refinamiento de esta regla implica cambiar cuando corresponda, las clases y propiedades definidas en las reglas OCL, por los respectivos estereotipos y valores etiquetados.

Las reglas de transformación asociadas a tipos de datos no pueden ser aplicadas (reglas 18 y 19), ya que en el Metamodelo de Integración no se han definido tipos de datos.

Finalmente, el proceso de transformación termina cuando son aplicados los refinamientos de las distintas reglas. En este caso de estudio el único refinamiento que puede ser aplicado pertenece a la Regla 1: *Si hay sólo un estereotipo que extiende a la clase UML, entonces la asociación de extensión del estereotipo debe ser definida como obligatoria (required)*. Con este refinamiento se obtiene el perfil UML final generado para la asociación OO-Method, presentado en la Figura 48.

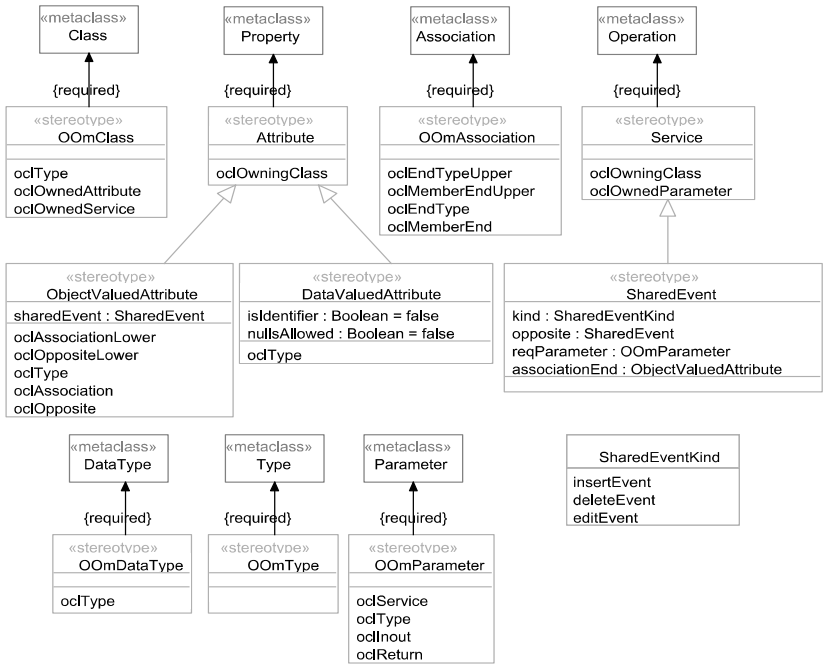


**Figura 48.** Perfil UML de la asociación OO-Method

El mapeo obtenido entre el perfil UML generado y la superestructura de UML es presentado en la Tabla 5. En esta tabla se presentan todos los elementos definidos en el Metamodelo de Integración (columna *Metamodelo Integración*) y su correspondencia con el metamodelo de UML extendido con el perfil UML generado (columna *Perfil UML*). Es importante destacar que en la columna *Perfil UML* no se muestran las clases UML, sólo se muestran los estereotipos que extienden las clases UML participantes. La columna U/P se utiliza para diferenciar los elementos definidos en el perfil UML, de aquellos definidos en la Superestructura de UML, donde: la letra *U* identifica los elementos de la superestructura UML, y la letra *P* los elementos definidos en el perfil UML.

**Tabla 5.** Mapeo del Perfil UML de la asociación OO-Method

| Metamodelo Integración | Perfil UML | U/P |
|---|---|---|
| **Association** | **OOmAssociation** | P |
| .endType | .endType | U |

| Metamodelo Integración | Perfil UML | U/P |
|---|---|---|
| .memberEnd | .memberEnd | U |
| **Attribute** | **Attribute** | P |
| .owningClass | .class | U |
| **Class** | **OOmClass** | P |
| .ownedAttribute | .ownedAttribute | U |
| .ownedService | .ownedOperation | U |
| **DataType** | **OOmDataType** | P |
| **DataValuedAttribute** | **DataValuedAttribute** | P |
| .type | .type | U |
| .isIdentifier | .isIdentifier | P |
| .nullsAllowed | .nullsAllowed | P |
| **ObjectValuedAttribute** | **ObjectValuedAttribute** | P |
| .type | .type | U |
| .association | .association | U |
| .opposite | .opposite | U |
| .aggregation | .aggregation | U |
| .isStatic | .isReadOnly | U |
| .sharedEvent | .sharedEvent | P |
| .upperValue | .upper | U |
| .lowerValue | .lower | U |
| **Parameter** | **OOmParameter** | P |
| .direction | .direction | U |
| .service | .operation | U |
| .type | .type | U |
| **Service** | **Service** | P |
| .owningClass | .class | U |
| .ownedParameter | .ownedParameter | U |
| **SharedEvent** | **SharedEvent** | P |
| .kind | .kind | P |
| .opposite | .opposite | P |
| .reqParameter | .reqParameter | P |
| .associationEnd | .associationEnd | P |
| **Type** | **OOmType** | P |
| **DirectionKind** | **ParameterDirectionKind** | U |
| . in | .in | U |
| .out | .out | U |
| **AggregationKind** | **AggregationKind** | U |
| .none | .none | U |

| Metamodelo Integración | Perfil UML | U/P |
|---|---|---|
| .aggregation | .shared | U |
| .composition | .composite | U |
| **SharedEventKind** | **SharedEventKind** | P |
| .insertEvent | .insertEvent | P |
| .deleteEvent | .deleteEvent | P |
| .editEvent | .editEvent | P |

En la 0 es posible apreciar que todos los elementos del metamodelo de DSML tienen equivalencia con el metamodelo de UML extendido con el perfil generado, de esta manera se consigue una integración completa del DSML en UML. Además, tal como se ha señalado en la sección 5.2, la equivalencia 1:1 entre las clases del Metamodelo de Integración y los estereotipos del perfil UML, permite identificar de forma inequívoca las correspondencias que hay entre un modelo construido a partir del DSML original, y un modelo UML extendido con el perfil UML generado.

En el caso de estudio presentado, para obtener un intercambio correcto y completo entre modelos DSML y UML, no es necesario incorporar información de mapeo entre el Metamodelo de integración y el metamodelo del DSML, ya que para generar el Metamodelo de Integración no fue necesario redefinir el metamodelo del DSML.

## 6.6. Conclusiones

El caso de estudio presentado muestra de forma más detallada la aplicación del proceso completo diseñado para la generación de perfiles UML, exponiendo claramente: la manera que se define el Metamodelo del DSML, la generación Metamodelo de Integración, y finalmente, la generación automática del perfil UML. Además, este caso de estudio permite observar como el perfil UML generado presenta absoluta consistencia con el metamodelo del DSML

original, permitiendo el intercambio entre modelos DSML y UML sin pérdida de precisión.

Este caso de estudio forma parte del proyecto de integración de la tecnología de modelado OO-Method en UML, con esta integración se pretenden alcanzar los siguientes objetivos:

1. Facilitar el aprendizaje y fomentar el uso de OO-Method como esquema conceptual para modelar sistemas de información sin ambigüedad.
2. Fomentar la participación de distintas comunidades y grupos de investigación en futuras versiones y mejoras de OO-Method, permitiendo de esta manera el trabajo colaborativo e interdisciplinario.
3. Permitir que los usuario de UML puedan aprovechar las ventajas que proporciona la tecnología de compilación de modelos de OO-Method, pudiendo generar a partir de sus modelos conceptuales aplicaciones completamente ejecutables para distintas plataformas.

# *Conclusiones*

Al término de esta Tesis de Master es posible concluir que se ha alcanzado el objetivo principal propuesto: La definición de un proceso completo que permita generar las extensiones necesarias para integrar en UML la precisión semántica requerida por una propuesta MDE específica. El proceso completo definido, realiza esta integración mediante un perfil UML que es generado automáticamente.

Este proceso completo ha sido diseñado para poder ser aplicado por distintas propuestas MDE, y de esta manera, no ser tan solo una herramienta para permitir la integración con UML, sino que además sirva para intercambiar conocimiento y experiencias entre las distintas tecnologías MDE utilizando UML como lenguaje de intercambio.

Si bien el proceso completo puede ser aplicado en propuestas MDE asociadas a distintos contextos, es importante recalcar que ha sido especialmente diseñado para ser aplicado en propuestas MDE industriales. Por este motivo, es en este tipo de propuestas MDE donde se obtienen los mayores beneficios de su aplicación.

Para elaborar un proceso completo, que pueda ser aplicado eficazmente en propuestas MDE industriales, la experiencia adquirida con OO-Method y su aplicación en el desarrollo industrial de software ha sido fundamental. Con esta experiencia se ha definido un proceso que además de generar un perfil UML correcto, brinda un entorno que permite integrar los resultados obtenidos con distintas soluciones MDE, basadas tanto en UML como en DSML. Además, esta experiencia ha ayudado a definir un proceso que cuenta con una estructura que facilita la validación de los resultados obtenidos en cada paso de la generación del perfil UML.

## 7.1. Contribuciones

Las contribuciones del trabajo presentado, derivan directamente de las soluciones diseñadas para alcanzar los objetivos presentados en esta Tesis de Master. Estas contribuciones son:

1. **La definición de un marco base para que las propuestas MDE puedan implementar soluciones destinadas a integrar sus necesidades de modelado en UML.** Este marco base es lo que se ha denominado proceso genérico y presenta los pasos que deben ser considerados para la correcta generación de un perfil UML correcto. En este proceso genérico, se definen guías y consejos para resolver los desafíos que es necesario resolver en un proceso de generación completo.

2. **La definición de una propuesta sistemática para la generación de un metamodelo enfocado a la integración completa de un DSML en UML.** Este metamodelo, denominado *Metamodelo de Integración*, permite la integración completa y automática de un DSML en UML. Esta es probablemente, la mayor ventaja que presenta el proceso propuesto en relación a otras propuestas destinadas a la construcción de perfiles UML, ya que en términos generales, el resto de propuestas no proveen una solución completa para generar de forma totalmente automática perfiles UML que integren toda la precisión semántica requerida por propuestas MDE.

3. **La definición de una propuesta para generar de forma completamente automática un perfil UML.** Hasta ahora, debido a las diferencias estructurales entre los metamodelos de DSMLs y UML, no se había podido definir un proceso que generara de forma completamente automática un perfil UML correcto. Gracias al Metamodelo de Integración estas diferencias se han resuelto, permitiendo elaborar un proceso que identifica

automáticamente las extensiones que deben ser incorporadas en UML para integrar toda la semántica de un DSML. Además, se han definido un conjunto de reglas de transformación que permiten la implementación completa de las extensiones identificadas en un perfil UML generado de forma completamente automática.

4. **La definición de un proceso completo que apoye a propuestas MDE industriales en la generación de perfiles UML correctos.** El proceso completo es la implementación del proceso genérico presentado en esta tesis. Este proceso completo ha sido diseñado para apoyar las necesidades de propuestas MDE industriales, como son la facilidad para incorporar mejoras y cambios en los constructores conceptuales, así como la posibilidad de validar adecuadamente los resultados obtenidos en cada etapa del proceso, y de esta manera obtener un perfil UML correcto.

5. **La definición de un mecanismo para permitir la integración entre modelos DSML y UML.** El proceso completo además de generar un perfil UML correcto, permite integrar los modelos UML definidos con el perfil UML generado y modelos definidos utilizando el DSML asociado. Esta integración es posible en primer lugar porque el perfil UML provee una equivalencia completa con el DSML, y en segundo lugar porque el proceso de generación del Metamodelo de Integración y el proceso de generación automática del perfil UML, proveen la información de mapeo necesaria para poder automatizar el intercambio entre metamodelos mediante herramientas como la presentada por Abouzahra et al. en [1].

Con estas contribuciones se ha logrado obtener una propuesta completa, destinada a generar perfiles UML asociados a propuestas MDE, que da soporte a los objetivos específicos propuestos, ya que:

- Facilita la incorporación de los cambios que sufra la propuesta MDE.
- Permite extender UML con toda la expresividad semántica requerida por una propuesta MDE.
- No altera la semántica original de UML.
- Opera bajo el estándar XMI que permite su interpretación por diferentes herramientas de modelado.
- Permite el intercambio entre modelos UML y modelos DSML.

## *7.2. Participación en Proyectos Industriales*

La definición de objetivos y contribuciones obtenidas en esta tesis master, están fuertemente influenciadas por la experiencia adquirida en el desarrollo de proyectos MDE industriales dentro del contexto del grupo de investigación OO-Method, en conjunto con la empresa CARE-Technologies [6]. Entre los proyectos realizados es posible destacar:

**Proyecto Method Engineering Toolset (MET):** Este proyecto está destinado a la generación de herramientas que permitan la definición y aplicación de metodologías de desarrollo, que son construidas a partir de fragmentos de metodologías almacenados en un repositorio común. Este proyecto implementa el metamodelo definido en la ISO 24774 y se ha realizado en conjunto con Brian Henderson-Sellers y Cesar Gonzalez, los autores de de esta norma ISO.

**Proyectos XMI Importer y XMI Exporter [28]:** Estos proyectos están destinados a la generación de herramientas de intercambio entre modelos conceptuales OO-Method y UML, utilizando la Superestructura de UML [46] y el formato de intercambio de modelos XMI [49].

**Proyecto Function Point Counter (FPC) [16]:** Este proyecto está orientado construir una herramienta MDE que permite calcular automáticamente el tamaño funcional de las aplicaciones, a partir de los modelos conceptuales OO-Method que las describen. Esta herramienta se ha construido mediante reingeniería de la herramienta construida originalmente para realizar esta medición. En el proceso de reingeniería, además de recambiar la plataforma de ejecución, se han definido mecanismos para reducir los tiempos de medición, y se han extendido los constructores conceptuales considerados en el proceso de medición para dar soporte a la nueva versión del modelo conceptual OO-Method. Las decisiones de diseño consideradas en la construcción de esta herramienta de medición, son la base para definición de la estructura que posee el proceso completo presentado en esta tesis. Ya que en la elaboración de esta herramienta, se establecen los aspectos que deben ser considerados para elaborar herramientas que apoyen procesos MDE industriales, así como la forma de aplicar estos aspectos en soluciones concretas. Por ejemplo, la definición de mecanismos que permitan a las herramientas construidas adaptarse a la evolución de los modelos conceptuales de las propuestas MDE.

**Proyecto Olivanova-ZENOS:** Este proyecto está orientado a construir una solución para la generación de aplicaciones de gestión conectadas al ERP SAP, utilizando la tecnología de compilación de modelos OO-Method y la herramienta ZENOS para el intercambio de información con SAP. Este proyecto se realizó en conjunto con las empresas Integranova [24] y Actum [2], esta última desarrolladora de la herramienta ZENOS.

**Proyecto Krypton:** Este proyecto está orientado al desarrollo de una suite de modelado para OO-Method basada en Eclipse GMF [10]. El proyecto Krypton, que está actualmente en desarrollo, se alinea perfectamente con la propuesta de esta tesis ya que requiere la definición en Eclipse EMF [9] del metamodelo EMOF de OO-Method. De acuerdo al proceso completo presentado, este metamodelo sirve como punto de partida para la generación del perfil UML (primer paso del proceso completo presentado).

## 7.3. Trabajos Actuales

El proceso completo presentado en esta tesis ha sido implementado en un prototipo basado en *Eclipse UML2* [12], el proyecto Eclipse *ATL* [8] y las herramientas *XMI Importer y XMI Exporter* [28]. Este prototipo se ha evaluado en el contexto de la propuesta industrial de OO-Method, y es la base de un desarrollo que actualmente se encuentra en proceso, que consiste en Implementar el proceso completo dentro del contexto del proyecto industrial Krypton presentado en la sección 7.2. Este desarrollo implica la construcción de herramientas industriales que puedan ser utilizadas tanto por la propuesta OO-method, así como por otras propuestas MDE que busquen su integración con UML y comprende las siguientes actividades:

1. **La generación del Metamodelo de Integración para OO-Method.** Este proceso de generación está en su primera etapa, la que consiste en definir el metamodelo del DSML de OO-Method. Este metamodelo está siendo definido mediante la herramienta Eclipse UML2 [12] y se convertirá en el primer metamodelo OO-Method completo especificado mediante el estándar de metamodelado MOF [35].
2. **La implementación de una herramienta que facilita la generación del Metamodelo de Integración.** Esta herramienta automatiza la identificación de problemas de integración, de acuerdo a las reglas definidas para un Metamodelo de Integración presentadas en la sección 4.2, y apoyará la propuesta sistemática presentada en la sección 4.3. Además de generar el Metamodelo de Integración, esta herramienta generará automáticamente el mapeo entre el Metamodelo de Integración y el metamodelo del DSML original, necesario para permitir el intercambio de modelos.

3. **La implementación de una herramienta que automatice la generación automática de perfiles a partir de un Metamodelo de Integración.** Esta herramienta se está desarrollando como una solución de código abierto y para implementar la reglas de transformación se utiliza el proyecto Eclipse ATL [8]. Además del perfil UML, esta herramienta generará automáticamente el mapeo entre el perfil UML y el Metamodelo de Integración, de esta manera, permitir la integración de tecnologías UML y DSML.

## 7.4. Trabajos Futuros

A partir de esta Tesis de Master se plantean una serie de trabajos futuros, entre estos trabajos es posible destacar:

1. **La implementación de una herramienta de intercambio entre modelos UML y DSML.** Esta herramienta será desarrollada utilizando la propuesta de Abouzahra et al. [1], que está implementada en ATL [8]. Dado que la herramienta de generación automática de perfiles UML se está definiendo en la misma tecnología, es posible obtener una mejor integración entre las implementaciones de ambas propuestas.
2. **Obtener una correcta representación de la sintaxis concreta del DSML mediante el perfil UML generado.** Con el trabajo presentado en esta Tesis de Master, se ha dado solución para integrar correctamente la semántica descrita mediante el metamodelo asociado al DSML de una propuesta MDE. Sin embargo, tal como se ha señalado en este documento, esta semántica sólo representa la sintaxis abstracta del DSML y no la sintaxis concreta necesaria para una correcta representación. Los perfiles UML presentan algunos mecanismos para definir esta sintaxis concreta, como son la definición de una notación particular para cada constructor conceptual. Sin embargo, ya se ha visto que la semántica de un constructor del

DSML puede estar dada por la combinación de distintos constructores UML, con lo que las posibilidades de representación que proveen los perfiles UML no son suficientes. Con un mecanismo que de soporte adecuado a la sintaxis concreta de un DSML, se podrá utilizar de forma completamente transparente cualquiera de las dos soluciones de modelo (perfiles UML y DSMLs), ya que se obtendría una integración completa de ambas tecnologías, tanto a nivel abstracto como concreto.

## *7.5. Publicaciones*

Las siguientes publicaciones están ligadas al desarrollo del proceso completo presentado en esta Tesis de Master, definiendo las bases que sustentan este proceso, así como las soluciones concebidas para alcanzar los objetivos propuestos:

1. Marín, B., **Giachetti, G.**, Pastor, O.: Intercambio de Modelos UML y OO-Method. X Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software (IDEAS). 2007.
2. **Giachetti, G.**, Marín, B., Condori-Fernández, N., Molina, J.C.: Updating OO-Method Function Points. 6th IEEE International Conference on the Quality of Information and Communications Technology (QUATIC). 2007.
3. Marín, B., **Giachetti, G.**, Pastor, O.: Una Herramienta Industrial para la Medición del Tamaño Funcional de Aplicaciones Desarrolladas en Entornos MDA. XI Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software (IDEAS). 2008.
4. Marín, B., **Giachetti, G.**, Pastor, O.: Automating the Measurement of Functional Size of Conceptual Models in an MDA Environment. Product-Focused Software Process Improvement (PROFES). LNCS Springer, 2008.

5. **Giachetti, G.**, Valverde, F., Pastor, O.: Improving Automatic UML2 Profile Generation for MDA Industrial Development. 4th International Workshop on Foundations and Practices of UML (FP-UML) – ER Workshop. LNCS Springer, 2008. *Pendiente de publicación*.

6. **Giachetti, G.**, Marín, B., Pastor, O.: Perfiles UML y Desarrollo Dirigido por Modelos: Desafíos y Soluciones para Utilizar UML como Lenguaje de Modelado Específico de Dominio. V Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM) – Taller JISDB. 2008. *Pendiente de publicación*.

# *Referencias*


1. Abouzahra, A., Bézivin, J., Fabro, M.D.D., Jouault, F.: A Practical Approach to Bridging Domain Specific Languages with UML profiles. In: Proceedings of Best Practices for Model Driven Software Development (OOPSLA'05) (2005)
2. ACTUM: Web site, http://www.actum.de/
3. Albert, M., Pelechano, V., Fons, J., Ruiz, M., Pastor, O.: Implementing UML Association, Aggregation, and Composition. A Particular Interpretation Based on a Multidimensional Framework. In: Proceedings of Conference on Advanced Information Systems Engineering (CAISE'03), pp. 143–158 (2003)
4. Atkinson, C., Kuhne, T.: Model-Driven Development: A Metamodeling Foundation. In: IEEE Software, vol. 20 nº 5, 36–41 (2003)
5. Bruck, J., Hussey, K.: Customizing UML: Which Technique is Right for You? IBM (2007)
6. CARE-Technologies: Web site, http://www.care-t.com/
7. Diaz, I., Sanchez, J., Pastor, O.: Metamorfosis: Un marco para el análisis de requisitos funcionales. In: Proceedings of Workshop on Requirements Engineering (WER), pp. 233–244 (2005)
8. Eclipse: ATL Project, http://www.eclipse.org/m2m/atl/
9. Eclipse: Eclipse Modeling Framework Project, http://www.eclipse.org/emf/
10. Eclipse: Graphical Modeling Framework Project, http://www.eclipse.org/gmf/
11. Eclipse: Graphical Modeling Framework Project.
12. Eclipse: UML2 Project, http://www.eclipse.org/uml2/

13. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional (2003)

14. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-driven development using uml 2.0: Promises and pitfalls. In: IEEE Computer, vol. 39 nº 2, 59–66 (2006)

15. Fuentes-Fernández, L., Vallecillo, A.: An Introduction to UML Profiles. In: The European Journal for the Informatics Professional (UPGRADE), vol. 5 nº 2, 5–13 (2004)

16. Giachetti, G., Marín, B., Condori-Fernández, N., Molina, J.C.: Updating OO-Method Function Points. In: Proceedings of 6th IEEE International Conference on the Quality of Information and Communications Technology (QUATIC 2007), pp. 55–64. (2007)

17. Gómez, J., Insfrán, E., Pelechano, V., Pastor, O.: The Execution Model: a component-based architecture to generate software components from conceptual models. In: Proceedings of Workshop on Component-based Information Systems Engineering (1998)

18. Graham, I., Bischof, J., Henderson-Sellers, B.: Associations considered a bad thing. Journal of Object-oriented Programming, vol. 9 nº 9, 1–48 (1997)

19. Guéhéneuc, Y., Albin-Amiot, H.: Recovering binary class relationships: Putting icing on the UML cake. In: Proceedings of Best Practices for Model Driven Software Development (OOPSLA'04), pp. 301–314 (2004)

20. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? In: IEEE Computer, vol. 37 nº 10, 64-72 (2004)

21. Henderson-Sellers, B.: On the Challenges of Correctly Using Metamodels in Software Engineering. In: Proceedings of 6th Conference on Software Methodologies, Tools, and Techniques (SoMeT), pp. 3–35 (2007)

22. IEEE: Standard Glossary of Software Engineering Terminology. Standard 729-1983. IEEE Computer Society (1983)

23. Insfrán, E., Pastor, O., Wieringa, R.: Requirements Engineering-Based Conceptual Modelling. In: Journal Requirements Engineering (RE) Springer-Verlag, 61–72 (2002)

24. INTEGRANOVA: Web site, http://www.integranova.com/

25. Kent, S.: Model Driven Engineering. In: Proceedings of Integrated Formal Methods (IFM), pp. 286-298. Springer (2002)

26. Lagarde, F., Espinoza, H., Terrier, F., Gérard, S.: Improving UML Profile Design Practices by Leveraging Conceptual Domain Models. In: Proceedings of 22th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 445–448 (2007)

27. Luoma, J., Kelly, S., Tolvanen, J.-P.: Defining Domain-Specific Modeling Languages: Collected Experiences. In: Proceedings of 4th OOPSLA Workshop on Domain-Specific Modeling (DSM'04) (2004)

28. Marín, B., Giachetti, G., Pastor, O.: Intercambio de Modelos UML y OO-Method. In: Proceedings of X Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software (IDEAS), pp. 283–296 (2007)

29. Mellor, S., Clark, A., Futagami, T.: Guest Editors' Introduction: Model-Driven Development. In: IEEE Software, vol. 20 nº 5, 14–18 (2003)

30. Moreno, N., Fraternali, P., Vallecillo, A.: WebML Modeling in UML. In: IET Software, vol. 1 nº 3, 67–80 (2007)

31. OMG: Catalog of UML Profile Specifications

32. OMG: MDA Guide Version 1.0.1

33. OMG: MDA Products and Companies, http://www.omg.org/mda/committed-products.htm

34. OMG: MDA Web site, http://www.omg.org/mda/

35. OMG: MOF 2.0 Core Specification

36. OMG: MOF Web site, http://www.omg.org/mof/

37. OMG: Object Constraint Language 2.0 Specification

38. OMG: Object Management Group Web site, http://www.omg.org/

39. OMG: A Proposal for an MDA Foundation Model

40. OMG: QVT 1.0 Specification

41. OMG: SysML Final Adopted Specification v1.0

42. OMG: SysML Web site, http://www.omgsysml.org/

43. OMG: UML 1.4.2 Specification

44. OMG: UML 2.0 Infrastructure Specification

45. OMG: UML 2.1.1 Infrastructure Specification

46. OMG: UML 2.1.1 Superstructure Specification

47. OMG: UML 2.1.2 Infrastructure Specification

48. OMG: UML 2.1.2 Superstructure Specification

49. OMG: XMI 2.1 Specification

50. OMG: XMI 2.1.1 Specification

51. OO-Method: RETO Tool Web Site, http://reto.dsic.upv.es/reto/

52. Opdahl, A.L., Henderson-Sellers, B., Barbier, F.: Ontological analysis of whole-part relationships in OO-models. Information and Software Technology, vol. 43, 387–399 (2001)

53. Pastor, O., Gómez, J., Insfrán, E., Pelechano, V.: The OO-Method Approach for Information Systems Modelling: From Object-Oriented Conceptual Modeling to Automated Programming. In: Information Systems. Elsevier Science, vol. 26 nº 7, 507–534 (2001)

54. Pastor, O., Hayes, F., Bear, S.: OASIS: An Object-Oriented Specification Language. In: Proceedings of International

Conference on Advanced Information Systems Engineering (CAiSE), pp. 348–363 (1992)

55. Pastor, O., Molina, J.C.: Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer (2007)

56. Pastor, O., Molina, J.C., Iborra, E.: Automated production of fully functional applications with OlivaNova Model Execution. ERCIM News nº 57 (2004)

57. Pohjonen, R., Kelly, S.: Domain-Specific Modeling. Dr. Dobb's Journal (2002)

58. Selic, B.: The Pragmatics of Model-Driven Development. In: IEEE Software, vol. 20 nº 5, 19–25 (2003)

59. Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 2–9 (2007)

60. Snoeck, M., Dedene, G.: Core modeling concepts to define aggregation. L'objet, vol. 7 nº 3, 281–306 (2001)

61. Staron, M., Wohlin, C.: An Industrial Case Study on the Choice Between Language Customization Mechanisms. In: Proceedings of Product-Focused Software Process Improvement (PROFES). LNCS, pp. 177-191. Springer (2006)

62. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2007)

63. Wimmer, M., Schauerhuber, A., Strommer, M., Schwinger, W., Kappel, G.: A Semi-automatic Approach for Bridging DSLs with UML. In: Proceedings of 7th OOPSLA Workshop on Domain-Specific Modeling (DSM), pp. 97–104 (2007)