MÉTODO PARA LA ADAPTACIÓN DE TRANSFORMACIONES M2M LEGADAS ANTE CAMBIOS EN LA LÓGICA DE MAPEO Y A EXTENSIONES DE METAMODELOS MEDIANTE PERFILES

JOSEBA ANDONI AGIRRE BASTEGIETA

Directora de Tesis:

GOIURIA SAGARDUI MENDIETA

Tesis dirigida a la obtención del título de
Doctor por Mondragón Unibertsitatea

Departamento de Electrónica e Informática
Mondragón Unibertsitatea

Mayo 2017

RESUMEN

El software se está convirtiendo en un elemento fundamental en los productos electrónicos. Cuando se desarrollan sistemas embebidos, los requisitos a cumplir no sólo se definen en términos de los aspectos funcionales del sistema, sino también en diferentes requisitos de diseño, como el tamaño, el consumo de energía, el tiempo de respuesta, la seguridad o la fiabilidad, usualmente denominados requisitos no funcionales (RNF). Los RNF son fundamentales en el desarrollo de sistemas embebidos. El diseño de un sistema embebido es un proceso complejo, que implica el modelado, la verificación y la validación de requisitos funcionales y no funcionales. La combinación de conceptos de desarrollo de software dirigido por modelos (DSDM) y de arquitecturas software se considera especialmente ventajosa para el desarrollo de sistemas complejos, tales como los sistemas embebidos.

El paradigma de desarrollo impulsado por modelo eleva el nivel de abstracción de las especificaciones del sistema y aumenta la automatización en el desarrollo del sistema. El DSDM utiliza los modelos como el principal artefacto del proceso de producción de software. Los modelos son transformados incrementalmente hasta generar el producto software. En el DSDM una transformación de modelo se especifica a través de un conjunto de reglas de transformación. Las tareas para definir, especificar y mantener las reglas de transformación son complejas y críticas.

Cuando un RNF no considerado en el desarrollo debe ser agregado, los metamodelos, los modelos y las transformaciones se ven afectados. En esta tesis se presenta la metodología denominada TRANSEVOL. TRANSEVOL es una solución para la adaptación de transformaciones modelo a modelo (M2M) legadas frente a cambios en los RNF que requieren (a) cambios en la lógica de mapeo; y (b) cambios en la lógica de mapeo al extender los metamodelos mediante perfiles. Mediante el uso de TRANSEVOL se reduce el tiempo de adaptación de las transformaciones M2M legadas. Para automatizar la deducción y localización de los cambios a realizar en una transformación M2M legada TRANSEVOL propone combinar la traza de ejecución de la transformación M2M legada con la especificación del nuevo requisito de mapeo expresado mediante modelos ejemplos. Combinando las diferencias con las trazas de ejecución se ubican los cambios a realizar en la transformación M2M legada. Para validar la propuesta se ha desarrollado un prototipo de herramienta.

ABSTRACT

Software is increasingly becoming an integral part of electronic-end-customer products. When developing embedded systems the requirements to fulfill are not only defined in terms of the functional aspects of the system, but also on different design requirements, such as size, power consumption, response time, security or reliability, usually called Non-Functional-Requirements (NFR). NFRs are critical in the development of embedded systems. Designing an embedded real-time system is a complex process, which involves modeling, verification and validation of functional and non functional requirements. The combination of model driven software development (MDD) and software architecture concepts is considered especially advantageous for developing complex systems, such as embedded systems.

The Model Driven development (MDD) paradigm raises the abstraction level of system specifications and increases automation in system development. MDD uses models as the primary artifact of the software production process, and the development steps consist of the application of transformation steps over these models. On MDD, a model transformation is specified through a set of transformation rules. Tasks for defining, specifying and maintaining transformation rules are usually complex and critical in MDD.

When a Non Functional Requirement (NFR) not considered in the development must be added metamodels, models and transformations are affected. This work presents the methodology TRANSEVOL. TRANSEVOL is a solution for the adaptation of legacy model-to-model (M2M) transformations to changes in NFR that requires (a) changes in the mapping logic; and (b) changes in the mapping logic due to the extension of metamodels using profiles. The use of TRANSEVOL reduces the adaptation time of legacy M2M transformations. To automate the derivation and localization of changes to be made in a legacy M2M transformation, TRANSEVOL proposes to combine the execution trace of the legacy model-to-model transformation with the specification of the new mapping requirement expressed by example models. Combining the model differences with the execution trace, the changes to be made in the legacy M2M transformation are deducted. To validate the proposal, a prototype of a tool has been develop.

LABURPENA

Softwarea produktu elektronikoen funtsezko atala bihurtu da. Sistema txertatu elektronikoen softwarea garatzean betebehar funtzionaletaz aparte baldintza ez funtzionalak ere bete behar dira. Beraz, sistema hauen diseinuan ezinbestekoak dira: softwarearen tamaina, energia-kontsumoa, erantzun denborak, segurtasuna edo fidagarritasuna. Baldintza hauei betebehar ez funtzionalak (BEZ) deritzote. BEZ-ak ezinbestekoak dira sistema txertatuen garapenean. Software arkitekturak eta ereduetan oinarritutako software garapena sistema txertatuen ekoizpenerako metodologia onuragarriak dira. Azken urteetan garapen eredu hauek sistema txertatuen industrian barneratzen hasi dira.

Ereduetan oinarritutako software garapenaren helburua sistemaren abstrakzio maila handitzea eta gauzapena automatizatzea da. Ereduetan oinarritutako software garapenean eredua da software ekoizpenaren elementu zentrala. Ekoizpen prozesuan ereduak etengabe eraldatzen dihoaz softwarearen kodea lortu arte. Ereduetan oinarritutako software garapenean ereduen eraldatzeak transformazio erregelen bitartez gauzatzen dira. Transformazio erregelen diseinua, gauzapena, aldaketa eta balidapena konplexua da.

BEZ berri bat agertzean metaereduak, ereduak eta transformazioak aldatu behar dira. Lan honetan laguntzeko TRANSEVOL izeneko metodologia aurkezten da. TRANSEVOL transformazioen aldaketa gauzatzeko metodologia da. TRANSEVOL transformazioak BEZ berrien eraginez aldatu behar direnean aplikatzeko soluzioa da. Bi egoera zehatzetan aplikatzen da TRANSEVOL: (a) metaereduak aldatu gabe eraldatze logika aldatu behar denean eta (b) metaereduei perfilak ezartzeagatik eraldatze logika aldatu behar denean. TRANSEVOLek transformazioren aldaketa prozesuaren denbora murrizten du. TRASNEVOLek egin beharreko aldaketak kalkulatu eta kokatzen ditu. Horretarako, transformazioen exekuzio aztarnak eta ereduen arteko desberdintasunak erabiltzen ditu. Lan honetan TRANSEVOL metodologia gauzatzen duen tresna bat garatu da metodologiaren baliogarritasuna egiaztatzeko.

AGRADECIMIENTOS

Hitz batzuk eskaini nahiko nizkieke lan hau egiten ari nintzen bitartean lagundu didaten pertsona guztiei. Lehendabizi Goiuriari eta Leireri, lanean aritu garen urte hauetan izan duten jarrera eskertzeko: bere ideiak, pazientzia eta laguntza. Nire familiari, bereziki Raquel, Irati eta Karleri beraien laguntza eta sostenguarengatik. Eta nire lankideak ere ez ditut ahazten: Axi, Roberto, Arenaza, Dani, Elkoro, Miren, Txema, Laki, Amallo eta Hazi taldeko kide guztiak. Eta, noski, informatika departamentuko eta unibertsitateko lankideak. Seguruenez norbait ahaztuko dut. Horrenbestez, aldez edo moldez lagundu didaten guztiei eskerrik beroenak.

DECLARACIÓN DE ORIGINALIDAD

Declaro a través de este documento que esta tesis, y el trabajo presentado en ella con sus resultados fueron hechos totalmente por mí, en el Departamento de Electrónica e Informática de la Escuela Politécnica Superior de Mondragón Unibertsitatea.

INDICE

LISTA DE FIGURAS

LISTA DE TABLAS

ACRONIMOS

ATL Atlas Transformation Language. 18, 19, 26, 27, 30-34, 44, 51-58, 61, 69, 72-74, 79, 80, 92-95, 98, 100, 104, 113, 114, 117, 118, 131, 135, 140-143

DSDM Desarrollo Software Dirigido por Modelos. 2, 3, 5-13, 15-19, 20, 23, 25, 28, 30, 32-36, 39, 40, 42, 44, 45, 48, 107, 113, 114, 116, 117, 119, 120, 126, 131, 135, 136, 140

DSL Domain Specific Language. 24, 25

EMF Eclipse Modeling Framework. 10, 18, 24, 26, 28, 34, 51, 62, 64-69, 72, 76-78, 80, 96, 107, 113, 114, 117, 128, 135, 140-142

M2M Transformación Modelo a Modelo. 3, 5, 10-13, 16-20, 23-36, 39-42, 44,45, 47-51, 55, 61-63, 67-69, 71, 72, 73, 75, 79, 80, 83, 90, 113-119, 121, 123, 127, 129, 131, 132, 135-142

M2T Transformación Modelo a Texto. 4, 13, 17, 28, 44, 116, 117, 125, 127

MARTE Modeling and Analysis of Real-time Embedded Systems. 10, 12, 18, 23, 44, 45, 116-119, 126

MOF MetaObject Facility. 4, 26, 28

MTBD Model Transformation By Demonstration. 33, 35, 36, 48

MTBE Model Transformation By Example. 32, 33, 35, 36, 48, 76, 78, 138

OMG Object Management Group 3, 25, 28, 30

PIM Platform Independent Model. 3-5, 9, 11-18, 28-30, 35, 36, 39, 40, 42, 44, 49, 50, 62, 63, 64, 80, 83, 90, 119, 138

PSM Platform Specific Model. 3-5, 10-14, 16, 17, 25, 26, 28-30, 35, 36, 39, 40, 42, 49, 50, 62, 63, 80

RNF Requisitos No Funcionales. 2, 3, 6-13, 15-20, 23-26, 35-36, 39-42, 44-45, 50, 61, 113-115, 118, 119, 126, 131, 135-138

SysML Systems Modeling Language. 2, 7, 10, 13, 24, 35, 40

UML Unified Modeling Language. 7, 10, 12, 13, 18, 19, 23-26, 28, 30, 35, 36, 40, 41, 44, 45, 107, 113, 114, 116-120, 126-129, 131, 136, 138, 141

XMI XML Metadata Interchange. 27, 67, 68, 107

XML eXtensible Markup Language. 28

1. Introducción

Un sistema embebido o empotrado (integrado, incrustado) es un sistema de computación (hardware y software) diseñado para realizar funciones dedicadas. Simplificando, cualquier sistema electrónico con capacidad de computación que no sea una estación de trabajo de propósito general, un ordenador de sobremesa o un ordenador portátil puede considerarse un sistema empotrado. Su nombre viene del hecho de estar embebido como parte de un dispositivo completo que a menudo incluye hardware y partes mecánicas. Los sistemas empotrados se basan en microcontroladores (MCU), microprocesadores (MPU) o chips diseñados a medida. Los sistemas embebidos se emplean en automóviles, aviones, trenes, vehículos espaciales, máquinas herramientas, cámaras, electrónica de consumo, electrodomésticos de oficina, aparatos de red, teléfonos celulares, navegación GPS, robots, juguetes, etc. Algunos ejemplos de sistemas embebidos podrían ser dispositivos como un taxímetro, un sistema de control de acceso, la electrónica que controla una máquina expendedora o el sistema de control de una fotocopiadora.

En la última década, el mercado mundial de los sistemas empotrados ha estado en torno a los 160 billones de euros, con un crecimiento anual del 9% [Eber09]. La figura 1.1 muestra el tamaño y volumen anual del desarrollo de los sistemas empotrados actuales, y el crecimiento en los últimos 50 años. Mientras que las estadísticas de volumen se pueden comparar con los sectores principales del desarrollo software, como por ejemplo Microsoft Windows, la complejidad del software empotrado, debido a las exigencias de tiempo real y restricciones en la interfaces de uso, es mucho mayor que en los sistemas de escritorio, o sistemas de información.

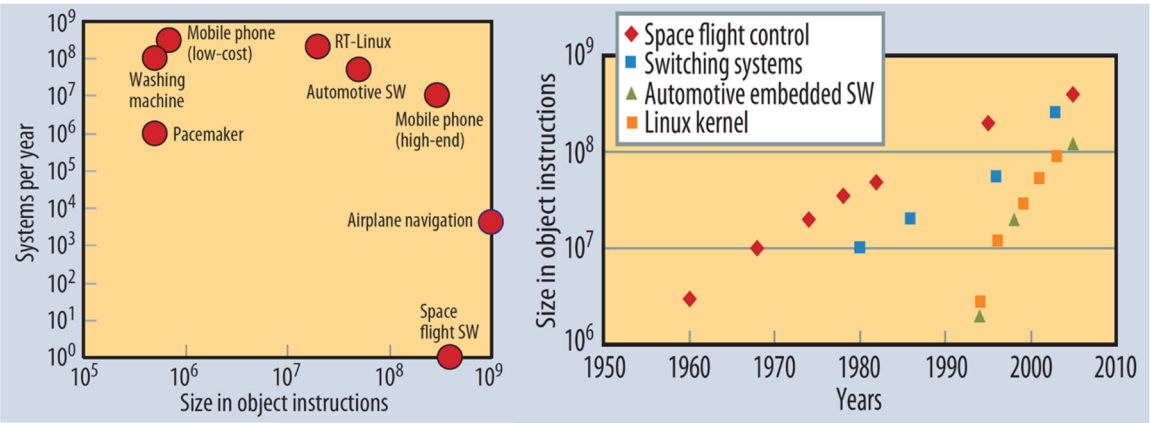


Figura 1.1 Tamaño de los sistemas empotrados actuales y la evolución de su crecimiento [Eber09]

Los sistemas embebidos se caracterizan por la heterogeneidad (hardware y software), distribución (en múltiples y heterogéneos recursos hardware), habilidad para reaccionar, criticidad, restricciones de consumo y tiempo real. Los sistemas embebidos se están convirtiendo extremadamente complejos y difíciles de diseñar y construir. Las particularidades del desarrollo software embebido son [Vol05]:

- *Recursos limitados*: Es necesario optimizar el desarrollo de software para gestionar los recursos limitados de memoria, de procesamiento, de consumo eléctrico, etc.
- *Requisitos de tiempo real*: La mayoría del software embebido interactúa con su entorno con limitaciones de tiempo.
- *Integración con el hardware*: La interacción con sensores y/o actuadores hace que durante el desarrollo de software embebido sea necesario tratar con aspectos hardware de bajo nivel.
- *Fiabilidad:* Los sistemas embebidos se utilizan ampliamente en sectores como aviónica, equipamiento médico, reactores nucleares y un fallo en este tipo de sistemas puede resultar perjudicial para la salud, por lo tanto, se requiere un alto grado de fiabilidad.
- *Coste*: Los sistemas embebidos, especialmente cuando se producen a gran escala, requieren minimizar los costes.

El software (SW) de los sistemas empotrados tienen que cumplir los RNF (Requisitos No Funcionales), tales como la portabilidad, fiabilidad, seguridad y mantenibilidad. Cada día aparecen nuevas plataformas de ejecución y frameworks con más prestaciones y costes más bajos. Las restricciones y los RNF, en la mayoría de los casos, requieren optimizar el software embebido en su contexto, lo cual supone que la mayor parte del software embebido se desarrolla ad-hoc, sin potenciar la reutilización. Esto, junto con la gran dependencia del software embebido en el hardware, aumenta la complejidad de su mantenimiento. Los cambios en los RNF afectan directamente a la implementación del SW, y la industria debe de responder a estos cambios en el menor tiempo posible para sobrevivir en el mercado actual. Por lo tanto, presentar una buena calidad en la gestión de los RNF, aumentar la reusabilidad y mejorar la mantenibilidad permitirá obtener un diferencial en el cambiante mercado actual.

En las últimas décadas se han desarrollado metodologías y herramientas para facilitar y mejorar el diseño, la reusabilidad y la evolución software. La combinación del Desarrollo Software Dirigido Por Modelos (DSDM) y las arquitecturas software está considerado especialmente ventajoso para el desarrollo de sistemas complejos, como los sistemas empotrados. Al diseñar la arquitectura del sistema se toman un conjunto de decisiones que afectan al cumplimiento de los RNF del producto final. Para poder realizar un desarrollo basado en arquitecturas software es fundamental el uso de lenguajes de modelado que permiten realizar diseños en diferentes niveles de abstracción, como por ejemplo SysML [Sys08] y AADL [Fei06]. El uso de lenguajes para describir y diseñar la arquitectura proporciona una base importante para la comunicación, el análisis y la implementación. En lugar de utilizar lenguajes de programación para la codificación, los desarrolladores modelan el sistema utilizando notación grafica o textual intuitiva, que ofrece un mayor grado de abstracción. En el DSDM los modelos que definen la arquitectura se transforman automáticamente para generar el código del producto SW. El DSDM está alcanzando un grado de estabilidad y herramientas que permiten el uso de dicha tecnología en la industria. En [Lie14] se realiza un estudio sobre el uso del DSDM en el dominio de los sistemas empotrados. En el estudio, las diferentes empresas de desarrollo software que tomaron parte, dejaban claro que la

principal razón de uso del DSDM se focalizaba en la reducción del tiempo de desarrollo, la reusabilidad y la mantenibilidad.

Este trabajo de investigación se sitúa en el dominio del DSDM aplicado a los sistemas empotrados. El desarrollo de los sistemas de DSDM requiere un ciclo de vida semejante al del desarrollo del software tradicional. La diferencia radica, en que ahora el sistema de DSDM es el que se debe de diseñar, validar, mantener, documentar e implementar. En los sistemas empotrados, debido a los cambios continuos en los RNF, la tarea de mantenimiento es fundamental. La problemática principal de este trabajo, es mejorar el proceso de mantenimiento y adaptación de los sistemas de DSDM frente a los cambios en los objetivos de los RNF de los sistemas empotrados. A continuación, se describen las características principales de un proceso DSDM para el desarrollo software de los sistemas empotrados, las necesidades de evolución de los RNF de los sistemas empotrados, y su impacto en los sistemas de DSDM. Una vez analizado el impacto de la evolución de los RNF de los sistemas empotrados en los sistemas DSDM se establece el contexto de este trabajo de investigación y las principales contribuciones.

1.1 Procesos DSDM para sistemas empotrados

En este apartado se detallan los puntos principales de un proceso de DSDM en el marco de los sistemas empotrados, donde los RNF son fundamentales. El objetivo es enumerar los principales "*problemas*" que surgen al incluir nuevos RNF de dominio en un sistema de DSDM enfocado a los sistemas empotrados.

1.1.1 Introducción al DSDM

El DSDM es un nuevo paradigma para el desarrollo de sistemas software, donde los modelos son el núcleo. La propuesta MDA (Model Driven Architecture) [Kle03] es una iniciativa de OMG (Object Managament Group) para la definición de una metodología estándar para el desarrollo dirigido por modelos. La idea principal del paradigma MDA es poder definir el software a desarrollar en diferentes niveles de abstracción permitiendo la derivación automática de un modelo de un nivel de abstracción superior, más cercano al dominio, a un modelo de un nivel de abstracción inferior, más cercano a la implementación. Mediante sucesivos refinamientos, se obtiene un modelo que se puede traducir a una implementación bajo una tecnología concreta partiendo de un modelo funcional independiente de la tecnología. Siendo los modelos el núcleo del proceso de desarrollo, la propuesta MDA asiste al desarrollador en los procesos de concepción, desarrollo, implementación y mantenimiento del software.

El enfoque MDA se basa en dos conceptos esenciales: el modelo independiente de plataforma (PIM) y el modelo especifico de plataforma (PSM). La funcionalidad del sistema se define en un modelo PIM mediante un lenguaje específico del dominio, y éste se transforma en un modelo PSM. La traducción de un PIM en un PSM se hace generalmente usando lenguajes de transformación modelo a modelo (M2M) [Sen03] que permiten automatizar el mapeo entre los

elementos de los diferentes niveles de abstracción. Una vez obtenido el PSM, éste se traduce/transforma en el código/texto de la aplicación diseñada. Para traducir los modelos PSM en código se utilizan lenguajes de transformación de modelo a texto (M2T) [Cza03], habitualmente basado en plantillas. La figura 1.2 representa un proceso MDA genérico. En la arquitectura MDA los lenguajes de modelado se denominan metamodelos. Los elementos de los metamodelo son utilizados para crear los modelos. Para definir los metamodelos, MDA define el lenguaje MOF. MOF es un meta-metamodelo utilizado para definir metamodelos. A MOF se le denomina como el lenguaje de modelado para definir lenguajes de modelado.

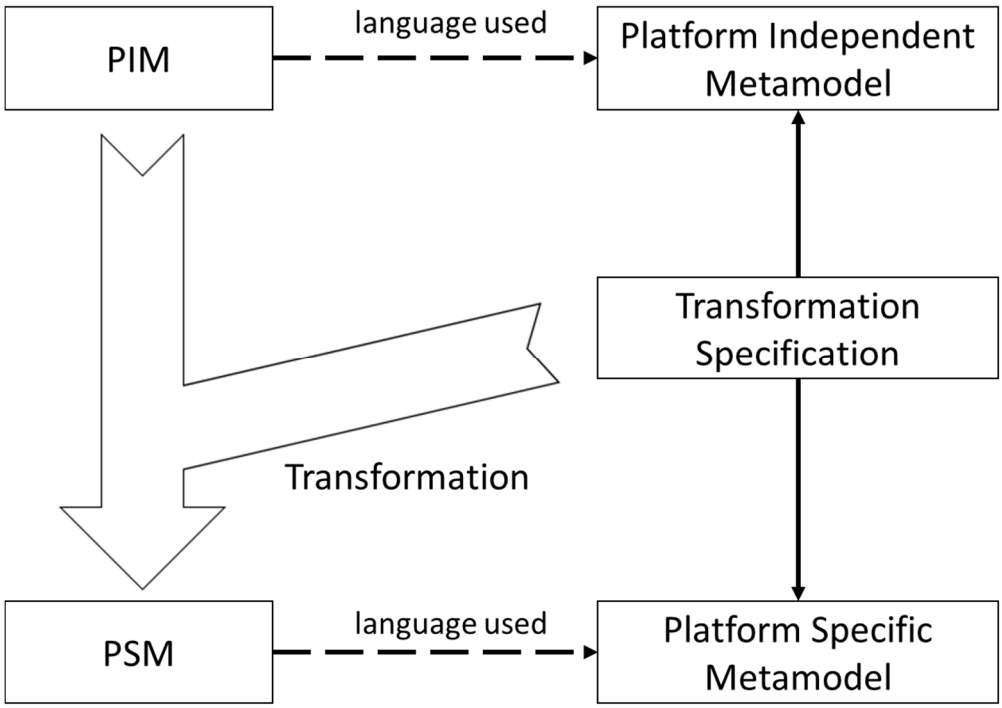


Figura 1.2 Proceso MDA genérico [Kle03]

El objetivo principal de MDA es separar el cambio en el negocio del cambio en la tecnología, facilitando que el diseño y la arquitectura puedan evolucionar fácilmente, potenciando la portabilidad, interoperabilidad y la reutilización. Las fases principales del ciclo de vida de MDA son:

- Definir el lenguaje de modelado (metamodelo) mediante el cual se definen los PIM.
- Definir el lenguaje de modelado (metamodelo) mediante el cual se definen los PSM de los cuales se generará el código.
- Una vez definidos los lenguajes de modelado se diseñan e implementan los modelos correspondientes a cada aplicación. Se suelen crear editores gráficos de modelos (a partir de los lenguajes de modelado anteriores) para poder diseñar las aplicaciones gráficamente.
- Definir transformaciones automáticas modelo a modelo para ir refinando sus diseños iniciales hasta obtener un modelo que permita la generación de texto/código.
- Definir transformaciones modelo-texto para generar automáticamente código, documentación, ficheros de configuración u otros artefactos finales, a partir de los modelos.

El objetivo de un sistema de DSDM basado en la arquitectura MDA, no tiene por qué ser obligatoriamente la generación de código. Los modelos se pueden utilizar también para el análisis del diseño, simulación o validación el sistema. Una vez realizado el diseño del sistema se generan los artefactos correspondientes: código, modelos de análisis, configuración de despliegue, elementos de validación, etc. Dependiendo del objetivo se utilizarán diferentes elementos de modelado. En el ámbito de los sistemas empotrados, los sistemas de DSDM principalmente se utilizan para la generación de código y la simulación. En la figura 1.3 se resume parte de los resultados obtenidos en el estudio [Mar14] que analiza el uso actual del DSDM en el dominio de los sistemas empotrados.

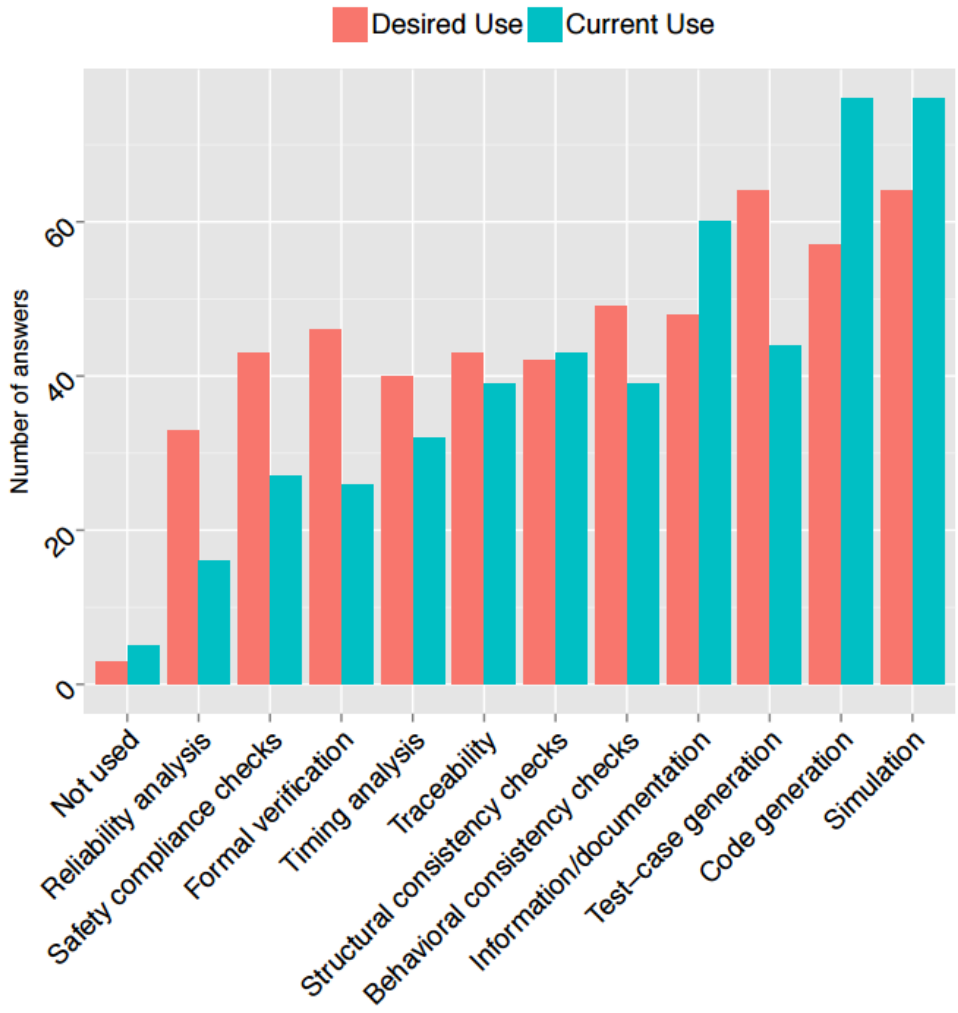


Figura 1.3 Uso de los sistemas de DSDM en el ámbito de los sistemas empotrados [Mar14]

Por lo tanto, el DSDM se puede entender como un proceso basado en la transformación de modelos, y es fundamental proporcionar herramientas adecuadas que den soporte al ingeniero en el proceso de desarrollo del producto tanto para el metamodelo y modelado, como para el desarrollo de las transformaciones. Este trabajo de investigación se focaliza en facilitar el desarrollo, el mantenimiento y la adaptación de las reglas de **transformaciones M2M** entre los modelos PIM y PSM en los sistemas de DSDM utilizados para la **generación de código** de sistemas empotrados.

1.1.2 DSDM en el dominio de los sistemas empotrados

Un sistema empotrado conoce su entorno mediante sensores y controla el entorno utilizando actuadores. Cuando se plantea desarrollar modernos sistemas empotrados, como por ejemplo los basados en sistemas SoC (System-On-a-chip), el hardware y el software no se pueden considerar de forma independiente. Las posibilidades de implementación son enormes, desde funcionalidades implementadas en bloques hardware IP a funcionalidades desarrolladas mediante software embebido en plataformas multiprocesador distribuidas. Además, a la hora de implementar el sistema empotrado también se tienen que tener en cuenta al mismo tiempo diferentes tipos de RNF: consumo de energía, memoria, limitaciones de energía, restricciones temporales, seguridad, etc. En este contexto el diseño de la arquitectura del sistema es fundamental. El diseño de las arquitecturas se enfoca en aspectos de diseño estructural del sistema con el fin de satisfacer ciertos RNF y facilitan la migración y despliegue en diferentes plataformas. En la figura 1.4 se resume el flujo general de diseño de los sistemas empotrados actuales.

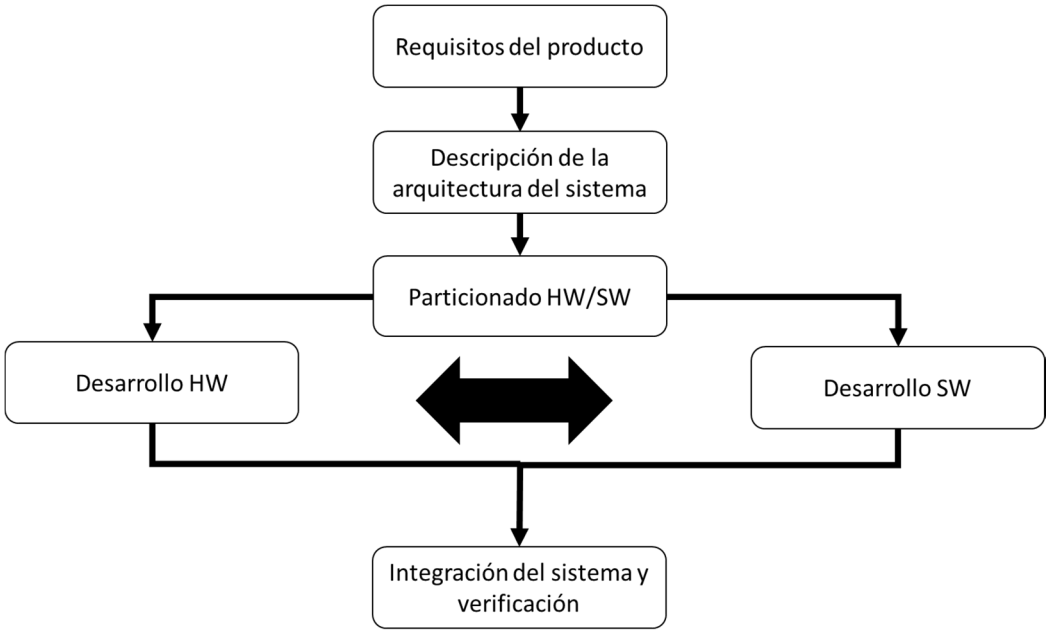


Figura 1.4 Flujo en el desarrollo de sistemas empotrados

Al desarrollar el software de los sistemas empotrados es necesario poder especificar en diferentes niveles tanto la arquitectura del sistema, los diferentes RNF, la plataforma de ejecución y la distribución de las funcionalidades en la plataforma de ejecución. Los lenguajes para la descripción de arquitecturas (ADL) han sido adoptados por los ingenieros software del dominio de los sistemas empotrados como núcleo para el proceso de desarrollo. Por ejemplo, el lenguaje AADL [Fei06] definido por la Sociedad de Ingenieros de Automoción (SAE), permite modelar tanto el hardware, software, los RNF y el comportamiento de los sistemas de automación. De esta forma, los diseños se utilizan para analizar si el diseño del software es capaz de responder a los RNF establecidos, y también como guía y plantilla en la implementación. Por lo tanto, para que un proceso de desarrollo SW sea adecuado en el dominio de los sistemas empotrados debe

permitir especificar, analizar y diseñar los requisitos funcionales, la arquitectura del sistema, el particionado HW/SW, la plataforma de ejecución y la distribución de los recursos.

La adopción del DSDM como metodología para el desarrollo de los sistemas empotrados empieza a ser una realidad hoy en día. Para que una solución de DSDM sea adecuada en el contexto de los sistemas empotrados debe de proporcionar mecanismos para:

1. Especificar los diferentes requisitos del sistema: Funcionales, de rendimiento, de consumo, de seguridad, etc.
2. Describir la arquitectura del sistema.
3. Especificar los RNF en la arquitectura del sistema.
4. Especificar la plataforma hardware de ejecución.
5. Debe permitir especificar el particionado hardware/software.
6. Realizar análisis para verificar el cumplimiento de los diferentes RNF, como por ejemplos análisis temporales de rendimiento.
7. La generación de código para la plataforma seleccionada.
8. Realizar simulaciones y co-simulaciones.
9. Ofrecer trazabilidad de los diferentes requisitos.
10. Generación de documentación.

Actualmente existen herramientas y métodos basados en modelos que permiten integrar en el proceso de desarrollo el diseño de la arquitectura del sistema, el análisis de los requisitos del sistema, el análisis funcional del sistema y la generación automática de diferentes artefactos [Ger02] [ATT10]. En [Lie14] se realiza una revisión del actual uso del desarrollo basado en modelos en el dominio de los sistemas empotrados. En el estudio se analizan los entornos de modelado usados, los lenguajes de modelado, los tipos de notaciones y el objetivo de uso de los modelos, además de comparar el porcentaje de tareas del proceso de desarrollo dirigidas por modelos con las no dirigidas por modelos. En la figura 1.5, se representa el porcentaje de uso de los diferentes lenguajes de modelado en los sistemas DSDM recogido en el estudio citado previamente. En dicho estudio se determina que, actualmente, UML y SysML son los lenguajes de modelado principales, junto con los diferentes estándares ADL.

Un ejemplo del uso del DSDM en el dominio de la automoción y los sistemas empotrados es el proyecto ATTEST2 basado en el lenguaje de modelado de arquitecturas EAST-ADL [Cue10], implementado como una extensión de UML. El proyecto ATTEST2 es una solución de DSDM donde el objetivo es lograr una estructura de información adecuada para toda la información de ingeniería involucrada en el desarrollo del SW del dominio de la automoción. En la figura 1.6 se resumen los diferentes niveles de abstracción y los diferentes modelos utilizados en la solución de DSDM ATTEST2. Mediante el perfil EAST-ADL inicialmente se realiza un análisis del sistema mediante la especificación de los requisitos en el nivel denominado Vehículo, y se realiza una especificación de la arquitectura funcional en el nivel de análisis. Una vez realizado el análisis, se diseña la arquitectura funcional junto con la arquitectura hardware (HW) y la abstracción de plataforma en el nivel de diseño. Del diseño se genera el modelo de la arquitectura SW AUTOSAR. Finalmente se obtiene el código final a empotrar en el vehículo.

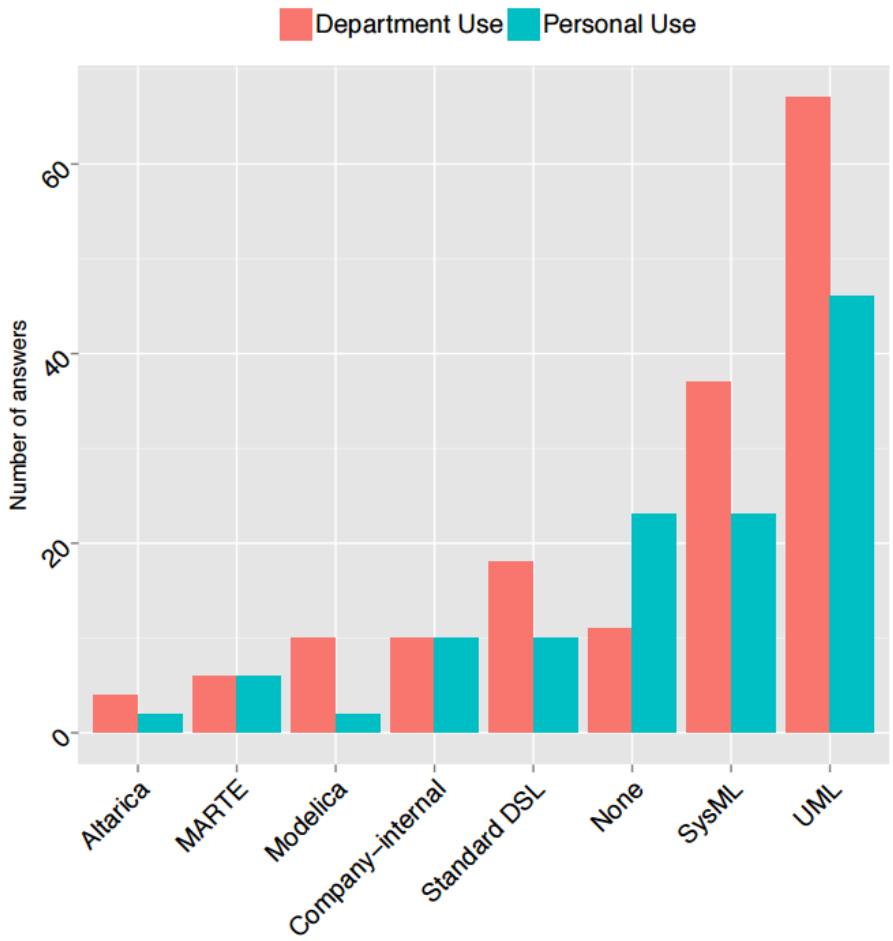


Figura 1.5 Uso de lenguajes de modelado en el desarrollo de sistema empotrados [Mar14]

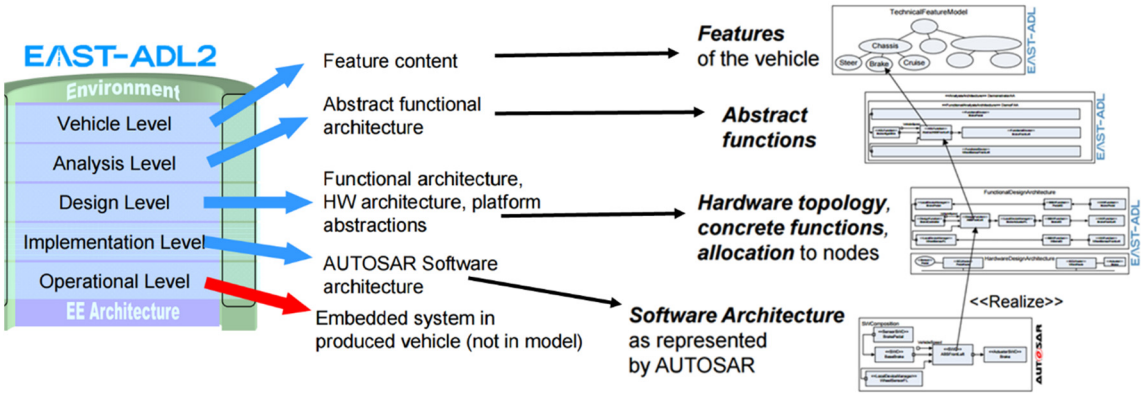


Figura 1.6 Visión global de la solución de DSDM ATTEST2 basado en el lenguaje de modelado EAST-ADL [Cue10]

1.1.3 RNF de los sistemas empotrados en el DSDM

Al desarrollar sistemas empotrados los requisitos a cumplir no son únicamente de carácter funcional, otros aspectos como el tamaño del binario, consumo energético, respuesta temporal, seguridad o fiabilidad deben de contemplarse con el objetivo de satisfacer al cliente. Los RNF son fundamentales, y estos tienen un impacto importante en el software final. Los RNF definen las cualidades o atributos generales del sistema resultante, imponen restricciones al producto que

se está desarrollando, y al propio proceso de desarrollo, y especifican las restricciones externas que el producto debe cumplir. Ejemplos de RNF puede ser características relacionadas con la seguridad, usabilidad, fiabilidad y con propiedades de rendimiento. Basándonos en la definición de atributos de calidad del ISO/EIC 25000 [ISO07], los diferentes RNF se han agrupado en tres grupos: 1) Requisitos no funcionales de dominio 2) Requisitos de plataforma y portabilidad 3) Requisitos del modelo de negocio. En la siguiente lista se enumeran los diferentes tipos de atributos de calidad en los tres grupos citados previamente:

1) RNF del dominio
    a. Eficiencia de desempeño
    b. Fiabilidad
    c. Seguridad
    d. Usabilidad
2) RNF del entorno de ejecución
    a. Portabilidad
    b. Compatibilidad
3) RNF relacionados con la políticas que definen el ciclo de vida de desarrollo
    a. Mantenibilidad
    b. Reglas de negocio
    c. Certificaciones

En [Eck16] se analiza el rol y el uso de los RNF en las soluciones de DSDM. Actualmente el grado de integración de los RNF en el DSDM no es alto, a pesar de su importancia. En [Ame10] se analiza dónde y cómo se representan los RNF en los sistemas de DSDM. En la figura 1.7 se resumen las diferentes soluciones de DSDM que permiten el uso de RNF. En dicho trabajo se remarcan las ventajas e importancia de tratar los RNF en el PIM. En estos casos el tiempo requerido en la fase de diseño y modelado es superior, debido a que hay que integrar en los modelos de diseño los conceptos relacionados con los RNF. Pero al mismo tiempo, el integrar la información de los RNF en el PIM permite realizar análisis útiles del diseño, generar el producto final de forma semiautomática, y ofrecer la posibilidad de generar diferentes arquitecturas SW dependiendo de las necesidades de cada producto. Las transformaciones entre modelos son las encargadas de analizar los RNF expresados en el nivel de abstracción correspondiente y generar los artefactos relacionados con las características no funcionales en el nivel de abstracción inferior. En este trabajo se abordaran aquellos sistemas donde los RNF son expresados en los modelos PIM, por ejemplo las soluciones d) y e) de la figura 1.7.

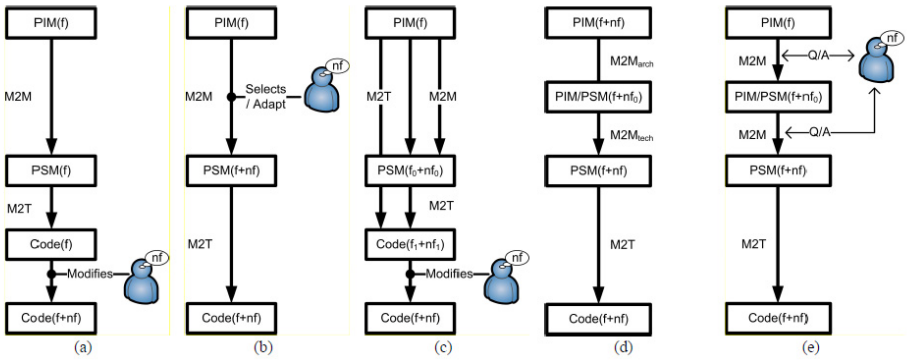


Figura 1.7 Arquitectura DSDM para sistemas empotrados [Ame10]

Para trabajar con RNF en el DSDM es fundamental expresar dichos requisitos en el modelo para después usarlos en el proceso DSDM. Los sistemas de DSDM que contemplan RNF mayoritariamente utilizan extensiones de metamodelos, o lenguajes específicos para representar los RNF [Ame10] [Eck16]. En el ámbito de los sistemas empotrados, la gran parte de los trabajos focalizados en los RNF extienden los lenguajes de diseño SysML/UML mediante perfiles. Un ejemplo es el perfil EAST-ADL previamente citado que extiende UML. Otro ejemplo es la extensión MARTE de UML [MAR11]. MARTE permite expresar tanto la plataforma de ejecución y la distribución de los recursos, como características temporales necesarios en los sistemas deterministas de tiempo real. En las soluciones de DSDM basadas en perfiles UML la información relacionada con los RNF se expresa mediante marcas o estereotipos. Los perfiles de UML proveen un mecanismo de extensión de metamodelos para personalizar los modelos UML para dominios y plataformas concretas. Los mecanismos de extensión permiten refinar la semántica del metamodelo, estrictamente, de forma aditiva, sin contaminar el metamodelo origen. Los perfiles se definen mediante estereotipos y restricciones, que se aplican a los elementos de los modelos. Una vez aplicados los estereotipos a los modelos de diseño, la transformación M2M es la encargada de generar e integrar en los modelos PSM los elementos software que permiten cumplir con los RNF especificados. La figura 1.8 representa este tipo de transformaciones. Las transformaciones M2M de este tipo son el objetivo a analizar en este proyecto de investigación. En [Lan12] se presenta una solución equivalente a los perfiles UML que permiten extender mediante perfiles metamodelos basados en ECore de EMF (Eclipse Modeling Framework).

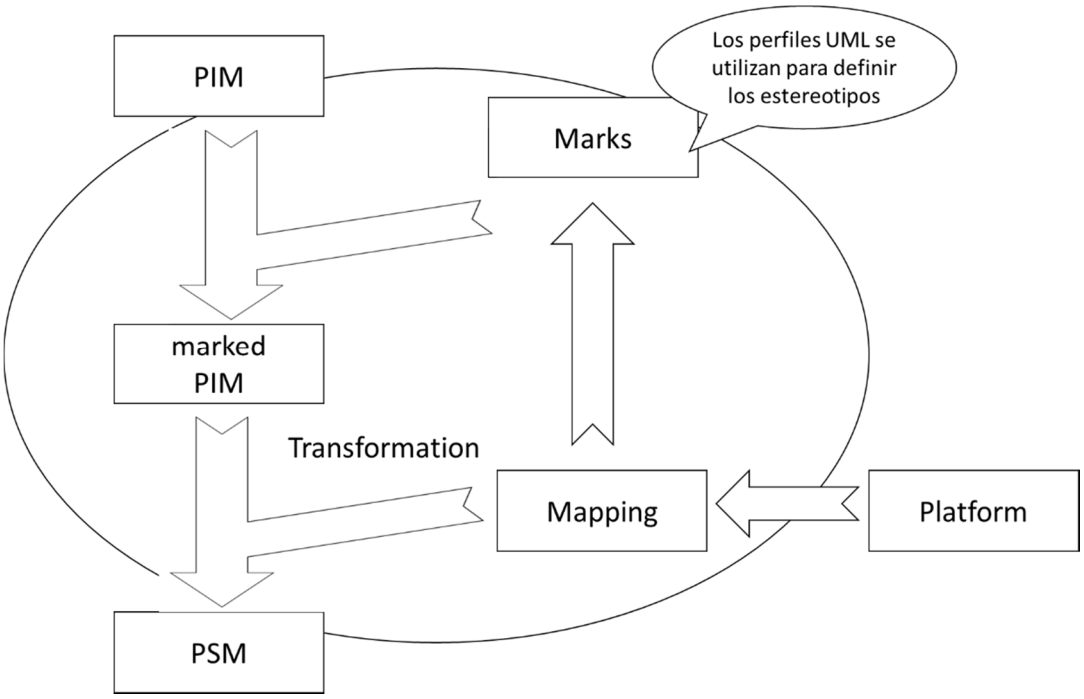


Figura 1.8 Transformaciones M2M dirigidas por RNF expresadas mediante perfiles [Mil03]

1.1.4 Impacto del cambio de los RNF de los sistemas empotrados en el DSDM

Una característica en el desarrollo de sistemas empotrados es la evolución de los RNF. Los desarrolladores deben enfrentarse a impredecibles y emergentes cambios en los requisitos [Har93]. Los requisitos cambian principalmente debido a errores en el análisis de los requisitos y debido a cambios en el entorno del sistema [Saw01]. Habitualmente la necesidad de cambio en los requisitos es detectada una vez el sistema ha sido implementado. Los cambios en los RNF requieren adaptar el sistema de DSDM. La principal diferencia del proceso de adaptación de los sistemas de desarrollo tradicionales frente a sistemas de DSDM son los artefactos afectados. Para poder aplicar el nuevo RNF en una arquitectura MDA, y así poder implementar el RNF en todos los productos SW a generar, además de poder expresar el RNF en el PIM y la nueva arquitectura SW tanto en el PIM como en el PSM, la transformación M2M debe de reimplementarse para poder así aplicar de forma automatizada el nuevo requisito en todas las aplicaciones SW generadas y a generar. Los artefactos de un sistema de DSDM que pueden ser afectados son los siguientes:

1) Lenguajes de modelado
2) Modelos
3) Transformaciones modelo a modelo
4) Transformaciones modelo a texto
5) Herramientas de modelado
6) El ciclo de vida del desarrollo

En los sistemas empotrados las evoluciones de los RNF están relacionados con nuevos requisitos emergentes o cambios no contemplados en los valores objetivos, normalmente relacionados con el dominio y el entorno de ejecución. Cuando se quieren integrar nuevos RNF sus

especificaciones se deben de integrar en el proceso de DSDM. Por ejemplo los modelos PIM deben de expresar los nuevos RNF. Para ello, el metamodelo del PIM se debe de extender. Lo más habitual es utilizar perfiles para la extensión del metamodelo, de esta forma el metamodelo original no se contamina. Al aplicar un perfil que permite expresar RNF los elementos del PIM se anotan y las anotaciones se contemplan en las transformaciones M2M para generar los PSM de forma adecuada. En algunos trabajos, como por ejemplo [Ell15], el metamodelo PIM no se extiende, en su lugar se crean propiedades en los elementos del modelo PIM que permiten a las transformaciones obtener información de RNF. Esta solución es equivalente a la anotación mediante estereotipos de perfiles, pero no se utiliza ninguna extensión de los metamodelos. Otra opción, es agregar un nuevo metamodelo que permita crear modelos donde se especifican los nuevos RNF. Después se relacionan las diferentes características con los elementos del diseño PIM [Mol09] [Gon14]. En cualquiera de estas tres situaciones las transformaciones M2M deben de adaptarse para contemplar los nuevos conceptos relacionados con los RNF, y también para modificar las reglas de transformación y adaptarlas a las nuevas lógicas de mapeo.

Para realizar los cambios en las transformaciones M2M entre el PIM y el PSM, primero se debe de localizar el cambio en la transformación y a continuación implementar el cambio. Debido al porcentaje de uso de UML, SysML y perfiles UML en los sistemas de DSDM en el dominio de los sistemas empotrados, este trabajo se focaliza en las transformaciones de este tipo de soluciones. El escenario de adaptación de transformaciones M2M contemplado en este trabajo es aquella donde se parte de una transformación M2M legada y se quiere adaptar su lógica de mapeo para responder a nuevos RNF. Al no conocer en detalle la implementación de la transformación M2M este trabajo de adaptación y mantenimiento tiene costes importantes.

El objetivo de este trabajo de investigación es facilitar el proceso de localización e implementación de los cambios a realizar en las transformaciones M2M al aplicar un nuevo perfil relacionado con nuevos RNF, y al cambiar la lógica de mapeo debido a cambios en los RNF. A continuación, se analiza el impacto sobre un sistema de generación de código de DSDM en ambas situaciones. El objetivo es determinar qué elementos del DSDM son los afectados y que tareas de adaptación son necesarias. Para validar los resultados de este trabajo de investigación, se ha utilizado un sistema de DSDM ejemplo para analizar las diferentes situaciones. El sistema de DSDM ejemplo utilizado en la validación genera código C de un diseño UML de componentes que contempla características de los sistemas empotrado mediante el perfil MARTE. La arquitectura de esta plataforma ejemplo de DSDM es una arquitectura tipo MDA, ver figura 1.9, para sistemas empotrados, donde los PIM se transforman en PSM y finalmente los PSM se traducen a código. Tanto en [Agi14], como en el anexo de este documento, se presenta en mayor detalle el sistema de DSDM ejemplo.

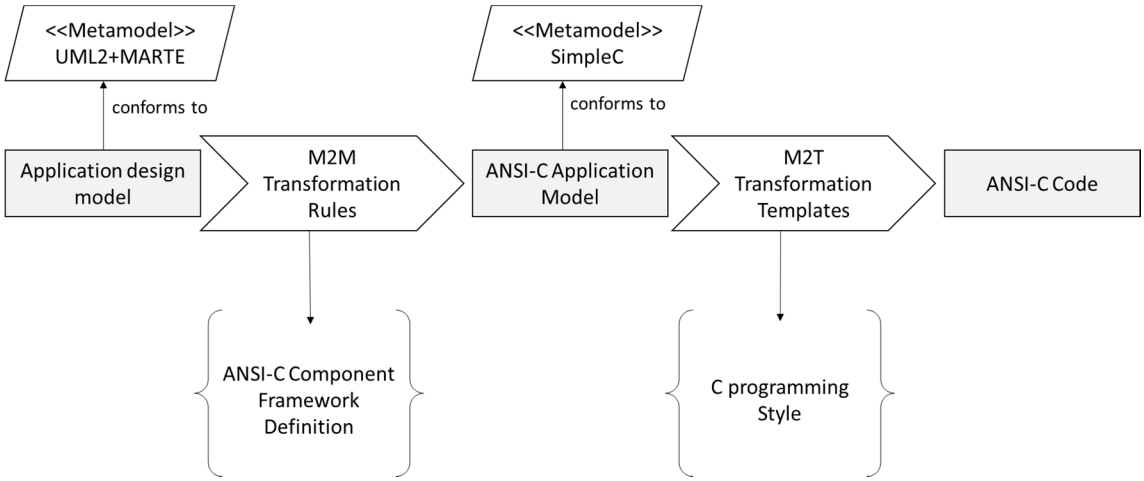


Figura 1.9 Solución DSDM para la generación de código ANSI-C a partir de componentes UML para sistemas empotrados [Agi14]

1.1.4.1 Integración de nuevos conceptos de abstracción mediante perfiles

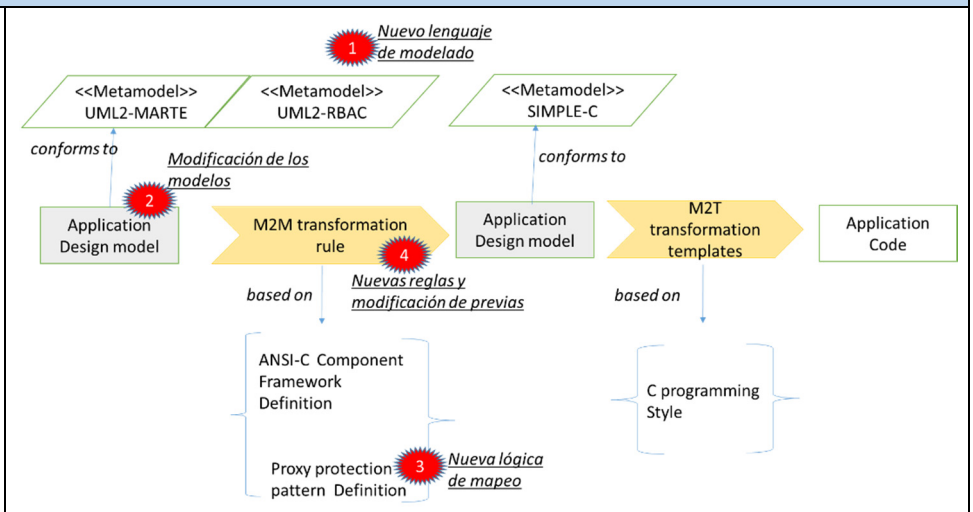
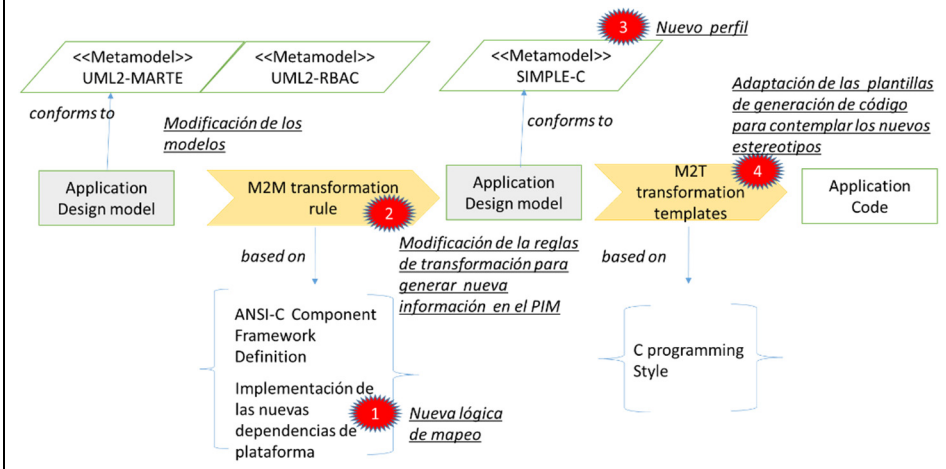
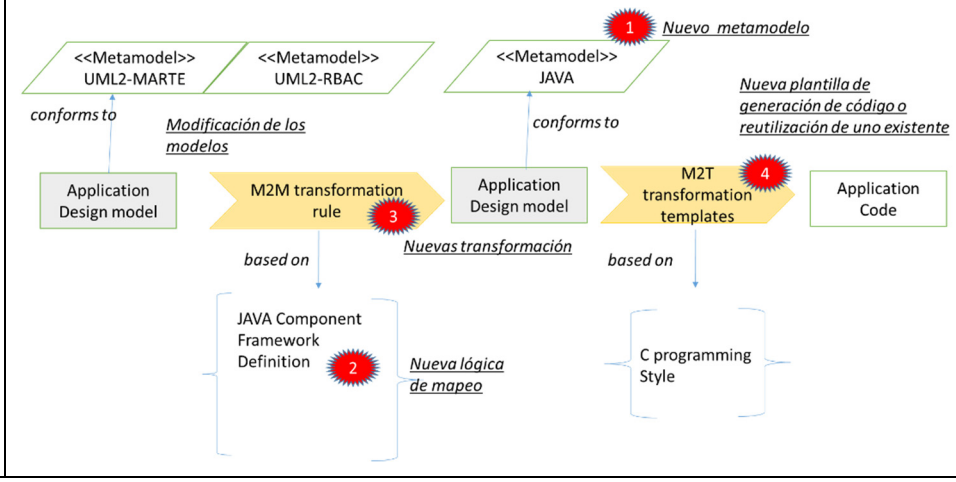
En un entorno de DSDM una situación cotidiana puede ser modificar una transformación M2M desarrollada por otra organización, y agregar en la transformación características no contempladas previamente y requeridas por la empresa, como por ejemplo requisitos de eficiencia de desempeño, fiabilidad, aspectos de seguridad y usabilidad. En estas situaciones, como mínimo el metamodelo del PIM debe de extenderse. También puede ocurrir que el PSM se tenga que extender. La técnica más habitual en los sistemas de DSDM para sistemas empotrados es extender el metamodelo mediante un perfil. En estas situaciones se puede seleccionar un perfil ya existente o crear un nuevo perfil para expresar los nuevos conceptos, esto dependerá de la existencia de dicho perfil para el metamodelo utilizado en el PIM. Para sistemas de DSDM basados en lenguajes de modelado de propósito general, como UML y SysML, existe una gran variedad de perfiles asociados a diferentes RNF. En este tipo de sistemas lo más habitual es agregar al sistema un perfil ya existente. En estas situaciones, el metamodelo se extiende mediante perfiles, y los PSM deberán de readaptarse para presentar una arquitectura SW que implemente el nuevo concepto en la plataforma seleccionada. Por lo tanto, los modelos PIM se deben de modificar manualmente y la transformación M2M se debe de modificar para que los modelos PSM respondan al nuevo requisito, y así el código generado sea correcto. Este tipo de escenarios de evolución en el DSDM se llaman evoluciones de abstracción [Van07]. En este tipo de evoluciones la tarea más complicada es adaptar las transformaciones M2M para integrar los nuevos conceptos de dominio.

También puede ocurrir, que el metamodelo de plataforma tuviera que extenderse. Estos escenarios además de requerir adaptar la transformación M2M también requieren modificar la transformación M2T [Gar14]. El caso más complicado ocurre cuando el lenguaje de modelado específico de plataforma se debe sustituir, ya que tanto la transformación M2M y M2T requieren grandes cambios. Otra situación parecida al cambio de lenguaje pero que solo afecta a la transformación M2T podría ser la adaptación a un nuevo estilo de codificación.

En la tabla 1.1 se resumen las tareas de adaptación a realizar en tres escenarios que requieren aplicar nuevos perfiles para contemplar nuevos RNF. En el primer escenario el nuevo

perfil se aplica al metamodelo PIM. En el segundo escenario el metamodelo del PSM es extendido mediante un perfil. En el tercer escenario el metamodelo del PSM es sustituido por un nuevo metamodelo. Este tercer escenario queda fuera del ámbito de este trabajo de investigación.

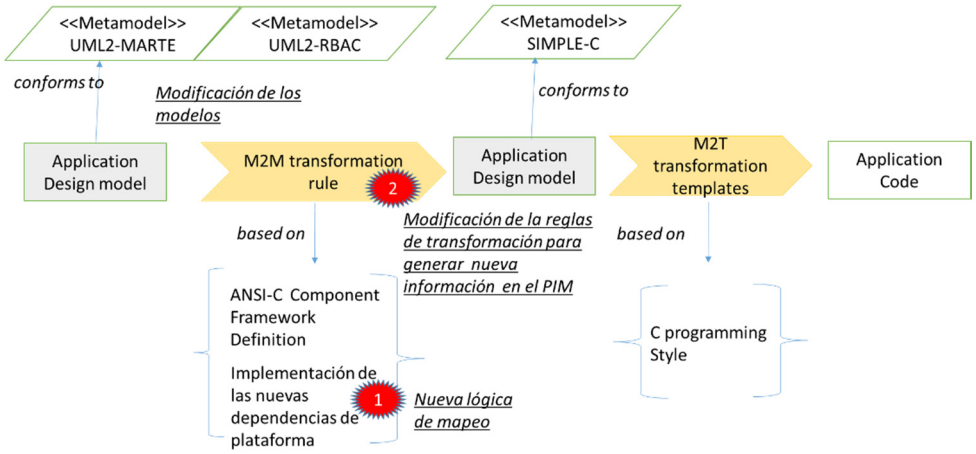
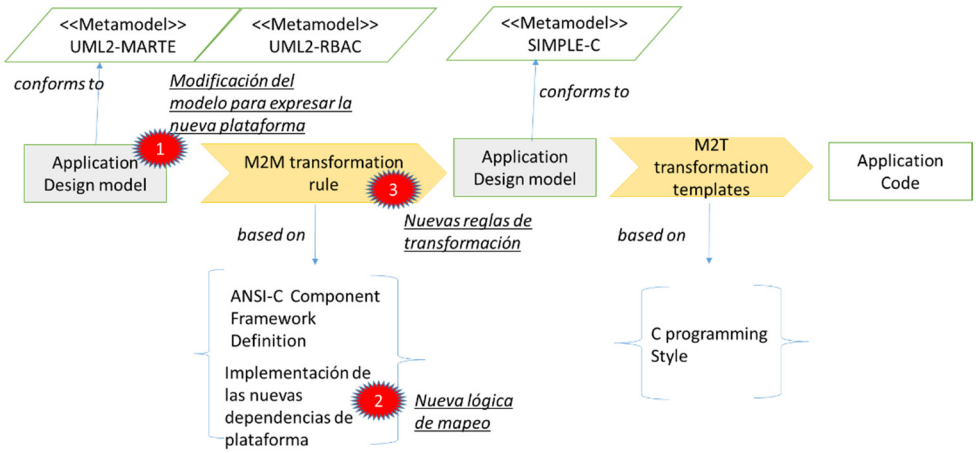
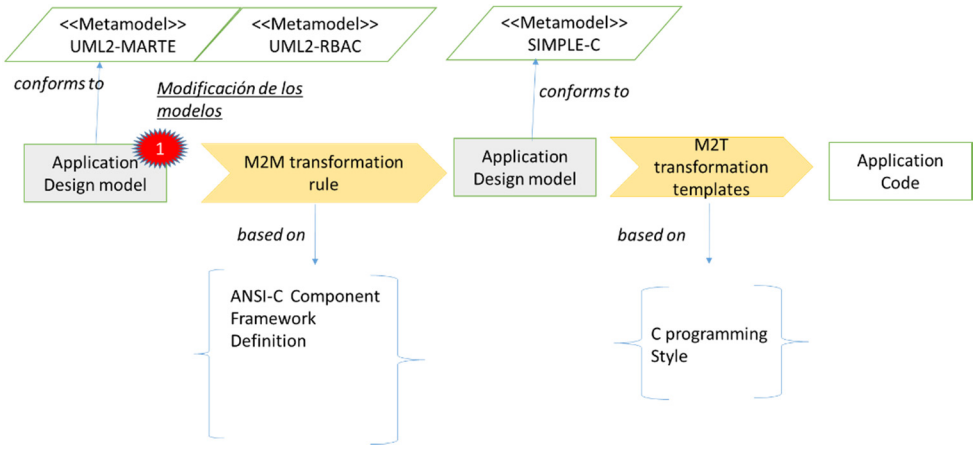
Tabla 1.1 Impacto de la integración de nuevas abstracciones mediante perfiles en una transformación de PIM a PSM

Escenarios de integración de nuevas abstracciones	
Evolución de abstracción en el PIM mediante perfiles	
Evolución de abstracción en el PSM mediante perfiles	
Sustitución del metamodelo PSM	

1.1.4.2 *Cambios en la lógica de mapeo*

En algunas situaciones los cambios en los objetivos de los RNF solamente requieren cambios en la lógica de mapeo, sin tener que modificar o extender ningún metamodelo. En la tabla 1.2 se resumen las tareas de adaptación a realizar en los tres escenarios de evolución de RNF donde no se requiere ni modificar ni extender ningún metamodelo.

Tabla 1.2 Impacto de escenarios de evolución de objetivos de RNF en una solución DSDM que no requiere modificaciones en los metamodelos.

Escenarios de modificación de parámetros de RNF	
Cambio en los objetivos de los RNF que requieren cambios en la lógica de mapeo	
Especificación de los nuevos RNF mediante cambios en los modelos del PIM	
Evoluciones regulares	

Por ejemplo, en [Ell15] se agregan conceptos de seguridad relacionados con el control de acceso sin extender el metamodelo AADL, simplemente se utilizan atributos concretos en los modelos para especificar la información de los RNF. En estas situaciones los nuevos objetivos no funcionales, especificados mediante atributos, se deben mapear al PSM, y por lo tanto los modelos PSM deben modificarse. En estas situaciones los modelos del PIM se modifican para agregar los nuevos atributos. La transformación M2M debe modificarse para generar los modelos PSM contemplando los nuevos requisitos. También puede ocurrir que los objetivos de los RNF varíen, y que para responder a los nuevos objetivos los modelos del PSM deben de modificarse, mientras los modelos del PIM se mantienen intactos. En estos escenarios la transformación M2M debe de adaptarse a la nueva lógica de mapeo. Este tipo de adaptaciones presenta una alta complejidad, sobre todo cuando se ha heredado la transformación M2M. Existen situaciones más sencillas donde el nuevo RNF se especifica modificando los modelos del PIM, y la transformación M2M es capaz de generar los modelos PSM adecuados, sin tener que modificar las reglas de transformación. Estas situaciones se denominan evoluciones regulares en el DSDM, y sus costes de mantenimiento son mínimos. Este tipo de evoluciones regulares no se contemplan en este trabajo.

1.2 Motivación

La fase de mantenimiento consume aproximadamente el 50% del coste de desarrollo [Ern11]. Bajo el nuevo paradigma de DSDM, el adaptar una aplicación a una evolución de RNF requiere ejecutar las tareas de mantenimiento sobre los artefactos del DSDM, y no directamente sobre el producto SW a comercializar. Para la consecución de una plataforma de DSDM que facilite el proceso de adaptación frente a evoluciones de RNF, además de contemplar las arquitecturas SW y los RNF, también se deben de ofrecer mecanismos para facilitar el mantenimiento de los metamodelos, modelos y las reglas de transformación. El desarrollo de las transformaciones es un punto clave en la adaptación de los sistemas de DSDM [Sen03]. Las transformaciones M2M básicas pueden estar compuestas por 30-90 reglas de transformación y 400-7000 líneas de código para metamodelos con un número no demasiado alto de metaclases [Van07] [Van11]. El desarrollo y mantenimiento de las reglas de transformación es una tarea compleja, que requiere un gran conocimiento de los lenguajes de modelado de dominio y de las técnicas utilizadas en el desarrollo de DSDM.

El proceso de desarrollo y mantenimiento de las transformaciones M2M presenta características similares al del desarrollo tradicional: documentación escasa y genérica, cambio de personal desarrollador continuo, personal de desarrollo no formado adecuadamente, necesidad de trabajar con código legado y complejidad de los sistemas de DSDM que requieren conocer muchos conceptos del desarrollo (metamodelos, dominios de aplicación, arquitecturas SW, lenguajes de transformación,…). Además, será muy habitual, que la adaptación de la transformación M2M tenga que realizarla un desarrollador diferente a la persona que diseño los metamodelos e

implementó la transformación. Todas estas características convierten el proceso de adaptación de las transformaciones M2M una tarea compleja.

El poder ofrecer un método ágil y eficaz para la adaptación de las transformaciones M2M legadas es un punto crítico. El escenario a abordar en este trabajo se acerca a la situación donde se deba de adaptar una transformación M2M legada, con los metamodelos de entrada y salida definidos, donde se deben de incorporar nuevos conceptos de modelado relacionados con RNF. En estas situaciones, los conceptos relacionados con los nuevos RNF se integran mediante perfiles que extienden los metamodelos de entrada y salida, sin que estos sean modificados y contaminados. En estos casos el proceso de adaptación de las transformaciones M2M presenta un coste temporal alto. Este trabajo se centra en facilitar y automatizar la adaptación de transformaciones entre PIM y PSM legadas frente a evoluciones de RNF de sistemas empotrados. En algunas situaciones las transformaciones M2M no requieren modificaciones, y simplemente modificando los modelos PIM se obtienen los nuevos PSM de forma correcta (evoluciones regulares). En otras situaciones la adaptación es más complicada, y requiere una adaptación de la transformación de M2M sin modificar los metamodelos de diseño, este tipo de adaptación supone un coste de mantenimiento alto. También puede ocurrir que una evolución de abstracción sea necesaria para poder expresar los conceptos relacionados con los nuevos requisitos. En estos casos, además de extender mediante perfiles los metamodelos y modificar los modelos de diseño, la transformación M2M debe de modificarse. Por último, las evoluciones de RNF en los sistemas empotrados también pueden obligar a modificar el metamodelo de plataforma y la transformación M2T. Las evoluciones de RNF que impactan en las transformaciones M2T no son objetivo de estudio en este trabajo. El objetivo de este trabajo de investigación es facilitar la adaptación de las transformaciones M2M legadas antes cambios en los RNF que requieren modificar la lógica de mapeo, tanto cuando se requiere extender los metamodelos mediante perfiles, como cuando no. Los escenarios a los que debe de responder este trabajo son:

1) Cambios en la lógica de mapeo en la transformación M2M cuando los metamodelos no son extendidos.
2) Cambios en la lógica de mapeo en la transformación M2M al extender los metamodelos mediante perfiles.

Este trabajo define un método para reducir el tiempo de adaptación de las transformaciones M2M legadas de sistemas de DSDM frente a cambios en los RNF de dominio. Los objetivos del método desarrollado son:

1) Facilidad para la especificación del nuevo requisito de mapeo relacionado con la evolución del RNF.
    a. Uso de la especificación en el proceso de validación.
2) Deducción automática de la operación de adaptación en la transformación M2M.
3) Localización automática en la transformación M2M legada de la operación de adaptación.
4) Expresar la operación de adaptación mediante un modelo que permita automatizar la implementación del cambio.

Las evoluciones de meta-modelos y sustituciones de lenguajes de modelado no han sido contempladas en este trabajo.

1.3 Contribuciones

En esta tesis se presenta la metodología y herramienta denominada TRANSEVOL. TRANSEVOL es una solución para la adaptación de transformaciones M2M legadas frente a cambios en los RNF que requieren (a) cambios en la lógica de mapeo; y (b) cambios en la lógica de mapeo al extender los metamodelos mediante perfiles. Mediante el uso de TRANSEVOL se reduce el tiempo de adaptación de las transformaciones M2M legadas.

Para automatizar la deducción y localización de los cambios a realizar en una transformación M2M legada TRANSEVOL propone combinar la traza de ejecución de la transformación M2M legada con la especificación del nuevo requisito de mapeo expresado mediante modelos ejemplos. Los modelos ejemplos expresan las nuevas necesidades de mapeo de una forma ágil. Los modelos ejemplo son incrementos de los modelos previos. Analizando los incrementos se deducen los cambios a realizar. Combinando las diferencias con las trazas de ejecución se ubican los cambios a realizar en la transformación M2M legada. Los modelos ejemplos se utilizan también en una primera validación de la implementación de los cambios.

Se ha desarrollado un prototipo de herramienta que permite aplicar y validar la metodología propuesta en esta tesis. La herramienta se basa en el framework EMF de eclipse [Ste08], y esta implementada mediante el lenguaje de programación JAVA. La herramienta analiza trazas de ejecución expresados mediante el metamodelo Trace [Jou05]. Para expresar la operaciones de adaptación a implementar se ha desarrollado el metamodelo *MMAdaptationGoal*. El prototipo actualmente permite analizar transformaciones M2M implementadas bajo el lenguaje de transformación ATL [Jou08].

Además, el prototipo desarrollado se ha evaluado ante dos escenarios de adaptación en un sistema de generación de código para sistemas empotrados. El sistema de DSDM seleccionado como caso de uso es una transformación M2M que genera código ANSI-C de diseños de arquitectura basados en componentes UML [Agi14] [Agi12]. Para expresar conceptos del dominio de los sistemas empotrados se utiliza el perfil MARTE. La transformación M2M consta de 8 ficheros y 70 reglas de transformación. Para validar la metodología TRANSEVOL se ha aplicado el prototipo ante dos escenarios de adaptación. El primer escenario requiere adaptar la lógica de mapeo de la transformación M2M para migrar de API de concurrencia. En este caso se requiere adaptar la transformación M2M sin realizar ningún cambio en los metamodelos. En el segundo escenario se requiere adaptar la transformación M2M al extender el metamodelo PIM para integrar control de acceso en las interfaces expuestos por los componentes UML. En este segundo escenario la extensión del metamodelo se lleva a cabo aplicando un nuevo perfil UML que permite expresar el patrón RBAC de control de acceso. El prototipo ha deducido correctamente los cambios a realizar en la transformación M2M en ambas situaciones.

1.3.1 Contribución técnica

Esta tesis presenta las siguientes contribuciones técnicas:

- Un método para la adaptación y desarrollo de transformaciones basadas en demostraciones aplicable a transformaciones M2M legadas exógenas y endógenas.
- El diseño de la metodología TRANSEVOL y la implementación de una herramienta prototipo.
- El diseño y la implementación del metamodelo *MMAdaptationGoal* para especificar las operaciones de adaptación a realizar en transformaciones M2M legadas.
- La validación de la metodología TRANSEVOL frente a evoluciones de RNF que requieren:
    - Cambios en la lógica de mapeo cuando no cambian los metamodelos.
    - Cambios en lógica de mapeo cuando se aplican nuevos perfiles UML.

1.3.2 Artículos

En el transcurso de este trabajo de investigación se han realizado las siguientes publicaciones:

Capítulos de libros

- AGIRRE, Joseba A.; SAGARADUI, Goiuria; ETXEBERRIA, Leire. Model Transformation by Example Driven ATL Transformation Rules Development Using Model Differences. En *Software Technologies: 9th International Joint Conference, ICSOFT 2014, Vienna, Austria, August 29-31, 2014, Revised Selected Papers*, pp.113-130. Springer, 2015.
- AGIRRE, Joseba A; SAGARDUI, Goiuria; ETXEBERRIA, Leire. 2014. Generación de código ANSI-C de modelos de componentes UML para sistemas empotrados. En MOLINA, Jesús García, et al.(eds). *Desarrollo de software dirigido por modelos: conceptos, métodos y herramientas*. Ra-Ma, pp.391-423. ISBN 9788499642154.

Conferencias internacionales

- AGIRRE, Joseba A.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Evolving legacy model transformations to aggregate non functional requirements of the domain. En *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE, 2015. p. 437-448.
- AGIRRE, Joseba A.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Transevol: A tool to evolve legacy Model Transformations By Example. En *Software Engineering and Applications (ICSOFT-EA), 2014 9th International Conference on*. IEEE, 2014. p. 234-245.

Talleres internacionales

- AGIRRE, Joseba Andoni; ETXEBERRIA, Leire; SAGARDUI, Goiuria. Automatic Impact Analysis of Software Architecture Migration on Model Driven Software Development. En *AMT@ MoDELS*. 2013.

Otras publicaciones

- AGIRRE, Joseba Andoni; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Plataforma DSDM para la Generación de Software Basado en Componentes en Entornos Empotrados. En *JISBD*. 2010. p. 7-15.
- AGIRRE, J.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. A flexible model driven software development process for component based embedded control systems. *III Jornadas de Computación Empotradas JCE, SARTECO*, 2012.

1.4 Estructura del documento

Este documento está estructurado en 7 capítulos. El **capítulo 1** realiza una introducción al DSDM en el ámbito de los sistemas empotrados y expone la problemática a la que responde este trabajo de investigación. También se resumen las contribuciones obtenidas mediante este trabajo de investigación.

El **capítulo 2** introduce el contexto de esta tesis analizando los conceptos del DSDM relacionado con este trabajo. Primero el tratamiento de RNF en el DSDM es introducido. A continuación diferentes mecanismos utilizados en el desarrollo, adaptación y evolución de transformaciones M2M son analizados.

El **capítulo 3** presenta la metodología de investigación seguida en este trabajo. Los objetivos y las hipótesis de esta tesis son presentados. También se presentan los casos de estudio utilizados en este trabajo.

El **capítulo 4** describe los cimientos de la metodología propuesta en este trabajo. Primero se presenta la arquitectura utilizada en la metodología TRANSEVOL. A continuación se presenta el metamodelo *MMAdaptationGoal*, utilizado para describir las operaciones de adaptación. Después se resumen los escenarios de adaptación contemplados por TRANSEVOL y se describe sucintamente el algoritmo para la deducción y localización de cambios. Finalmente se explica cómo se utiliza el prototipo desarrollado mediante un sencillo ejemplo.

En el **capítulo 5** se describe en detalle el algoritmo para la deducción y localización de las operaciones de adaptación para cada escenario de cambio de lógica de mapeo soportado por TRANSEVOL.

En el **capítulo 6** se presenta la validación realizada de la metodología TRANSEVOL. Primero se presenta la transformación M2M utilizada en la validación. A continuación, se recogen los resultados de aplicar el prototipo para adaptar la transformación M2M frente a un cambio de lógica de mapeo que no requiere modificar ningún metamodelo. Esta adaptación es debida a un requisito de migración del API de concurrencia. La segunda validación se realiza frente a un escenario de adaptación que requiere adaptar la transformación M2M al aplicar un nuevo perfil al metamodelo de diseño.

Con el **capítulo 7** se finaliza este documento. En este capítulo se presentan las conclusiones sobre el trabajo realizado. También se describen posibles futuros trabajos a realizar.

PRIMERA PARTE
Estado del arte

2. Fundamentos y contexto

El objetivo de este trabajo es reducir el tiempo de adaptación de las transformaciones M2M frente a modificaciones en los RNF del dominio de los sistemas empotrados. Concretamente, el objetivo de este trabajo de investigación es facilitar y mejorar el proceso de adaptación de las transformaciones M2M cuando la lógica de mapeo cambia y los metamodelos se extienden mediante perfiles. En este capítulo se ubica y justifica el trabajo de investigación TRANSEVOL en el ámbito del desarrollo y adaptación de las transformaciones M2M de los sistemas de DSDM. Primero, se resumen los trabajos más recientes sobre la representación de los RNF en los sistemas de DSDM y su uso en las transformaciones M2M. A continuación se analiza el estado actual de las soluciones focalizadas en la mejora del desarrollo y adaptación de las transformaciones M2M. Finalmente se realiza un análisis crítico justificando la necesidad de este trabajo de investigación.

2.1 RNF en el DSDM

En este apartado se realiza un resumen de los trabajos actuales sobre la integración de los RNF en el DSDM. Primero se recopilan diferentes trabajos donde el objetivo es representar los RNF en los modelos. Una vez analizado como se representan los RNF en el DSDM, se analiza cómo se trata la información relacionada con los RNF en las transformaciones M2M.

2.1.1 Notación de los RNF en el DSDM

A la hora de expresar RNF en los modelos, los trabajos actuales se dividen en dos técnicas principalmente:

- Extensiones de metamodelos mediante perfiles.
- Uso de lenguajes específicos para la especificación de RNF.

El uso de perfiles para extender metamodelos con el objetivo de añadir semántica para expresar RNF es habitual. El enfoque presentado en [Wad07], utiliza un perfil UML, llamado UP-SRNF, para apoyar la especificación de RNF dentro de modelos UML para sistemas SOA (Service Oriented Architecture). Los perfiles UML utilizados en [Zhu07] son utilizados para especificar la decisión arquitectónica y los RNF. Lo interesante de esta propuesta es que ambos perfiles se pueden utilizar juntos. Esto es importante porque, como se afirma en el mismo documento "*La razón fundamental detrás de cada decisión de arquitectura está el lograr ciertos RNF*".

Por ejemplo, el perfil MARTE [Mar11] estandarizado por OMG, permite extender el lenguaje UML para diseñar sistemas de tiempo real y empotrados. Entre sus características principales están:

- Mediante anotaciones permite expresar propiedades no funcionales.

- Añade conceptos de tiempo y de recursos al lenguaje UML convencional.
- Ofrece la definición de conceptos tanto de plataforma SW como de plataforma HW.
- Permite definir anotaciones cuantitativas de rendimiento [Gon17], como las demandas de recursos realizadas por los diferentes módulos software en ejecución, los requisitos de rendimiento, etc.
- Permite anotar el diseño de la arquitectura SW, no solo con el objetivo de realizar análisis de rendimiento, sino también para refinar y optimizar la generación de código [Amm16], co-diseños HW-SW [Kha16] [Peñ10] [Ebe15] [Her14] y generación de descripciones VHDL [Lei14].

Otro ejemplo de perfil UML aplicado en los sistemas empotrados es UML4IoT [Thr16]. El perfil UML4IOT es un perfil UML creado para modelar sistemas ciber-físicos. El perfil UML4IoT permite al desarrollador generar automáticamente componentes ciber-físicos compatibles con la interfaz LWM2M [She15] de los sistemas IoT (Internet of Things). En [Apr13] se presenta una extensión de SysML en modo de perfil, creado con el objetivo de ofrecer un entorno de desarrollo donde los expertos en seguridad puedan colaborar con los ingenieros de sistema en todas las fases del desarrollo del producto software ciber-físico.

Tanto la herramienta Papyrus [Lan12], como TOPCASED [Far06], ambas basadas en el proyecto EMF de eclipse, permiten crear perfiles UML y han sido utilizadas en la creación de perfiles UML para la especificación de RNF. El proyecto EMF-Profiles [Lan12] ha sido creado para aplicar el concepto de perfiles UML a los DSL (Domain Specific Language) basados en EMF.

Otra alternativa para especificar RNF es el uso de lenguajes específicos de dominio. En [Gon12] se define una solución multi-modelo que partiendo de un modelo de variabilidad, el diseño de la arquitectura de una línea de productos, la definición de los RNF y la configuración del producto a generar se deriva el diseño AADL del producto. En esta solución los RNF se expresan en un modelo independiente, y a continuación se establecen las relaciones entre los RNF, las diferentes características de la línea de productos y los diferentes elementos del diseño arquitectónico. Para establecer relaciones entre los diferentes modelos, y así realizar una composición de modelos, se utiliza el Weaving [Del06] entre modelos. En [Mol09] se presenta una metodología que, partiendo de un modelo de análisis de requisitos plantea extender el metamodelo de diseño agregando nuevos metaelementos para así poder especificar los RNF derivados del proceso de análisis de requisitos. En [Gal14] se define una solución para el desarrollo de servicios web con requisitos de seguridad. Esta es una solución híbrida donde se utilizan tanto perfiles UML, weaving entre modelos y lenguajes específicos de dominio. Partiendo de un análisis de requisitos realizado a nivel empresarial, se utilizan tres modelos para diseñar la arquitectura junto con los RNF. Para especificar los RNF se utilizan dos perfiles de UML, y se genera un modelo para especificar la seguridad y otro modelo para especificar el control de acceso. Finalmente, los RNF se relacionan con el diseño funcional mediante weaving, y utilizando esta información se genera un modelo, basado en el metamodelo iMM, que contiene toda la información para generar

un modelo PSM. En este caso el modelo iMM es un DSL para la definición de servicios web seguros. En la tabla 2.1 se resumen los trabajos de modelado de RNF analizados en este estado del arte.

Tabla 2.1 Clasificación de trabajos de modelado de RNF en el DSDM en el dominio de los sistemas empotrados

Trabajo	Técnica			Conceptos expresados		
	Perfil UML	DSL	Modificando el metamodelo	RNF	Plataforma HW	Arquitectura y plataforma SW
[Wad07]	X			X		
[Wad07]	X			X	X	X
[Lei14]	X				X	X
[Amm16]	X				X	X
[Kha16]	X			X	X	X
[Gon17]	X			X	X	X
[She15]	X					
[Thr16]	X					X
[Gon12]		X		X		X
[Mol09]			X	X		
[Gal14]	X	X		X		

2.1.2 Transformaciones dirigidas por RNF

En el DSDM, las transformaciones son utilizadas para superar la brecha semántica entre las abstracciones del dominio del problema y los artefactos software. Por lo general, las transformaciones asignan requisitos funcionales a elementos más concretos. Trabajos recientes comienzan a diseñar estas transformaciones teniendo en cuenta también los RNF. Estos enfoques de DSDM utilizan los modelos funcionales extendidos con RNF como entrada. Las transformaciones del DSDM usan los RNF para seleccionar entre diferentes diseños o implementaciones, normalmente relacionadas con patrones de arquitectura software.

En [Ste08] se presenta una metodología que permite crear transformaciones M2M personalizables que trabajan con modelos de entrada que contienen información de RNF. La transformación M2M utiliza la información de los RNF para determinar la arquitectura en base a diferentes patrones de diseño. La transformación M2M genera como salida modelos expresados mediante AADL. Los modelos de salida presentan el diseño de la arquitectura que mejor se adapta a los RNF especificados. Las reglas de transformación utilizan filtros para tomar estas decisiones. En [Gon14] se presenta un enfoque multimodelo para representar las relaciones entre la variabilidad, representada mediante un modelo de características, y la variabilidad arquitectónica, representada mediante un modelo CVL (Common Variability Language) [Hau12]. Utilizando la información que relaciona las diferentes características con las diferentes arquitecturas se consigue automatizar la derivación de la arquitectura AADL de un producto SW. En [Pan08] y [Ins08], la información relacionada con la usabilidad se utiliza para guiar las transformaciones. En [Aba12] se presenta un método basado en el refinamiento, paso a paso, de los modelos que integran la especificación de los RNF con el objetivo de obtener un modelo PSM que cumpla con los RNF espe-

cificados. Partiendo de modelos abstractos, las transformaciones ejecutan las decisiones de diseño de forma que se va generando un diseño concreto que cumple con los RNF. Es un proceso de mapeo entre una representación de alto nivel de abstracción y sus sucesivos diseños. Los desarrolladores pueden hacer predicciones de los RNF en las primeras etapas del proceso de desarrollo, y también pueden comprobar si los RNF son satisfechos por el producto final o no. Para ello, antes de cada refinamiento, se realiza un análisis de los RNF al diseño de la arquitectura del nivel correspondiente, con el objetivo de tomar decisiones sobre el diseño. El enfoque es extensible a todas las propiedades no funcionales cuantitativas y cuantitativas.

2.2 Transformaciones de modelos

Las trasformaciones de modelo se pueden definir como "*un programa que muta un modelo en otro modelo*" [Tra05] [Mil03]. Para implementar una transformación de modelos, que teniendo uno o varios modelos de entrada genere uno o más modelos de salida, hace falta un conocimiento claro sobre la sintaxis abstracta y la semántica utilizada para definir los modelos de entrada y salida. En la figura 2.1 se representa la base de una transformación de modelo. Los modelos se definen mediante lenguajes de modelado. Por ejemplo, los modelos basados en UML se representan mediante el lenguaje de modelado de propósito general UML [Uml04]. Para definir los metamodelos se utilizan meta-metamodelos como MOF [Mof06] o Ecore de EMF [Ste08]. Una transformación presenta modelos de entrada que se transforman en modelos de salida. La definición de la transformación de modelo se realiza mediante lenguajes específicos para la transformación de modelos, como por ejemplo ATL [Jou08]. La definición de la transformación indica qué elementos del metamodelo de entrada se mapea en qué elementos del metamodelo de salida. El motor de transformaciones ejecuta la transformación definida. Al ejecutar la transformación se indica cual es el modelo de entrada y automáticamente se genera su correspondiente modelo de salida.

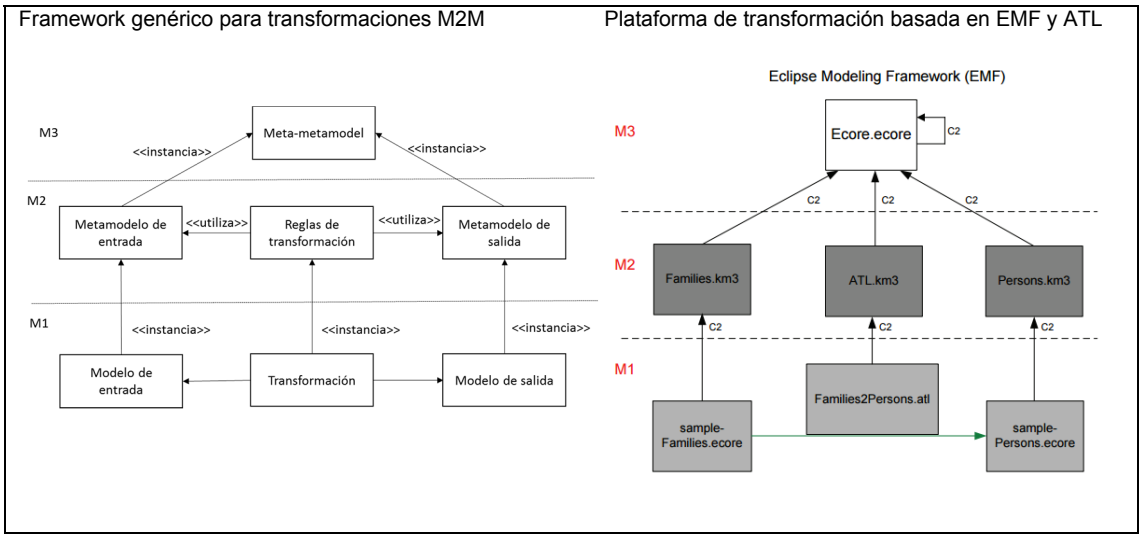


Figura 2.1 Patrón básico de una transformación modelo a modelo

La definición de una transformación de modelo está compuesta por una serie de reglas de transformación. Las reglas de transformación indican en qué elemento del modelo de salida se convierten las instancias de un tipo del modelo de entrada. En cada regla de transformación se especifica el patrón de entrada y el patrón de salida en el que se convertirá. Por cada instancia del modelo de entrada que cumpla con el patrón de entrada, se genera un elemento del modelo de salida que cumple con el patrón de salida. Los patrones de las reglas de transformación se especifican en base a los metaelementos (meta-clases) de los metamodelos. En la figura 2.2 se representa la transformación M2M "Families2Person", un ejemplo introductorio utilizado para el aprendizaje del lenguaje de transformación ATL. El objetivo de la transformación es generar instancias de personas partiendo de definiciones de familias. El metamodelo utilizado para definir familias contiene familias y miembros. Las familias tienen un apellido y cada miembro de la familia un nombre. Los miembros de una familia son: un padre, una madre, hijos e hijas. Por otro lado, el metamodelo para definir personas lo constituye una metaclase abstracta base, denominada "*Person*", y dos subclases, "*Male*" y "*Female*", que heredan de la clase base y son utilizadas para diferenciar el sexo. La metaclase "*Person*" solo tiene una propiedad, "*Fullname*", que representa el nombre completo de una persona, por ejemplo "*Mr. Alex Chilton*". La transformación convierte las instancias miembros de una familia en personas. Utilizando el apellido de la familia crea personas con sus nombres completos. Los modelos en este caso se representan mediante ficheros XMI con extensión ".*ecore*". En la figura 2.2 se representa la transformación, los metamodelos y los modelos de la transformación "Families2Person". Las dos reglas de transformación, "*Member2Female*" y "*Member2Male*" escritas mediante el lenguaje ATL, son las encargadas de realizar la transformación.

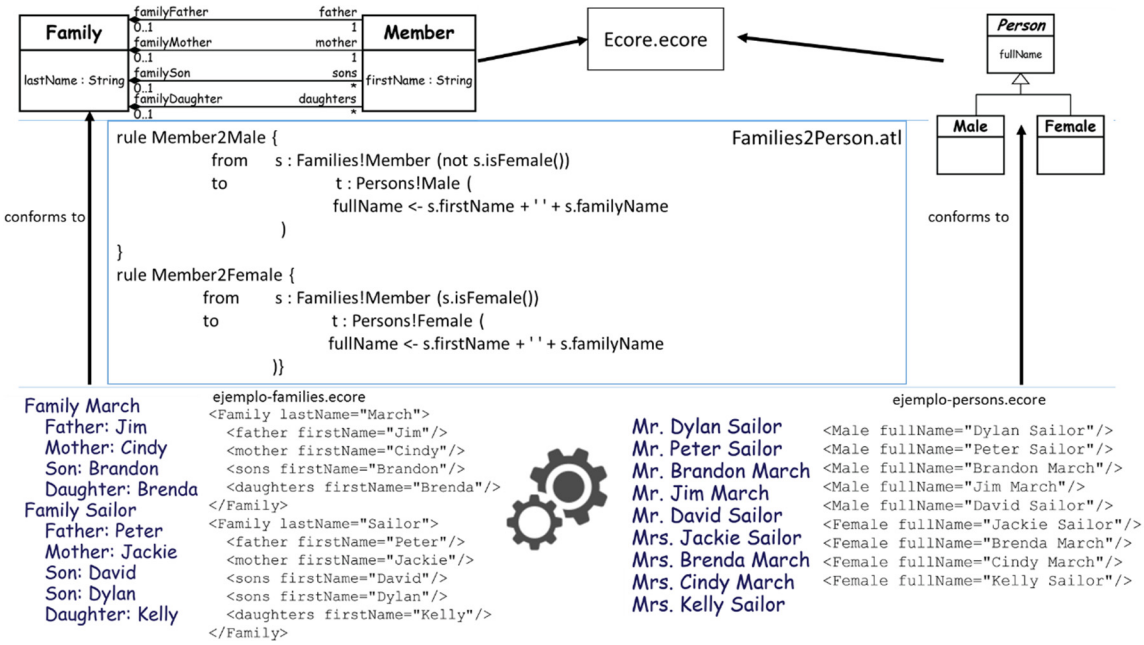


Figura 2.2 Transformación M2M "*Families2Person*"

Además de las transformaciones de modelos a modelos (M2M), los modelos también se transforman en texto. Las transformaciones modelos a texto (M2T) se utilizan normalmente para generar los artefactos textuales finales, como pueden ser el código, la documentación, ficheros de configuración, etc. Este tipo de transformaciones normalmente se desarrollan mediante lenguajes basados en plantillas. Las plantillas permiten describir fragmentos de texto a generar. Un meta-programa interpreta el modelo de entrada, y utilizando las plantillas genera el texto de salida correspondiente. Existen una gran variedad de lenguajes basados en plantillas para la implementación de transformaciones M2T. Bajo el paraguas de EMF Eclipse [Ste08] existen diferentes proyectos relacionados con las transformaciones M2T. XLST [Kay08] es un lenguaje que permite transformar documentos XML en otros documentos XML, en páginas HTML, texto plano u objetos que utilizan el formato XSL, que a su vez se pueden convertir en PDF o PNG. XSLT está definido por el consorcio W3C. Existe una gran cantidad de herramientas para definir y aplicar transformaciones XSLT, como por ejemplo Xalan de Apache Software Foundation. El algoritmo de transformación es muy eficiente, pero las transformaciones complejas necesitan documentos XSLT muy grandes. El proyecto Apache Velocity [Gra03] también se puede utilizar como lenguaje de transformación modelo a texto. El grupo OMG definió una especificación denominada MOF Model to Text Language (MTL) [Omg08] para los lenguajes de transformación modelo a texto. El lenguaje MOFScript [Old06] fue la propuesta inicial a la especificación MOF MTL. El proyecto Acceleo [Mus06] es una implementación pragmática del estándar MOF MTL. En la imagen 2.3 se puede ver parte de una plantilla ACCELEO que convierte instancias del tipo "*Class*" de modelos UML en código Java. Xpand [Has07] es un lenguaje especializado en la generación de código de modelos EMF basado también en plantillas. Xpand ofrece además soporte integrado para expresiones AOP (Aspect Oriented Programming). El lenguaje de generación del proyecto Epsilon EGL (Epsilon Generation Language) [Ros08a] también es otra solución basada en plantillas para la transformación de modelos a texto. EGL permite el uso de regiones protegidas y permite integrar y gestionar texto generado con código escrito manualmente. En este proyecto de investigación solo se abordarán las transformaciones M2M. Las transformaciones M2T no son objeto de estudio en este proyecto.

2.2.1 Tipos de transformaciones modelo a modelo

En el DSDM las transformaciones de modelo son cruciales. Las transformaciones de modelo son las responsables de relacionar los modelos PIM con los modelos PSM. Mediante las transformaciones es posible automatizar el proceso de mapeo entre elementos de diferentes niveles de abstracción. Las transformaciones partiendo de modelos de entrada basados en metamodelos se encargan de:

- Refinar modelos: El proceso de desarrollo se puede definir como una combinación de pasos donde los requisitos se van convirtiendo en la implementación incrementalmente.
- Generar diferentes vistas: La generación de diferentes vistas permite al desarrollador del producto concentrarse en los elementos del diseño relacionados

con sus responsabilidades. Esto también permite utilizar un conjunto del diseño con objetivos diferentes, por ejemplo, simulación, análisis de rendimiento, etc.

- Aplicar patrones arquitectónicos y de software: Aplicar patrones de diseño software es fundamental en las diferentes fases del desarrollo software. Este tipo de transformaciones se pueden entender como refinamientos.
- Refactorización de modelos: Con objeto de mejorar la calidad de los modelos, mejorar el mantenimiento y para reducir la complejidad, se necesitan diversos trabajos de refactorización. La refactorización de modelos es una tarea que las transformaciones M2M permiten automatizar y simplificar.

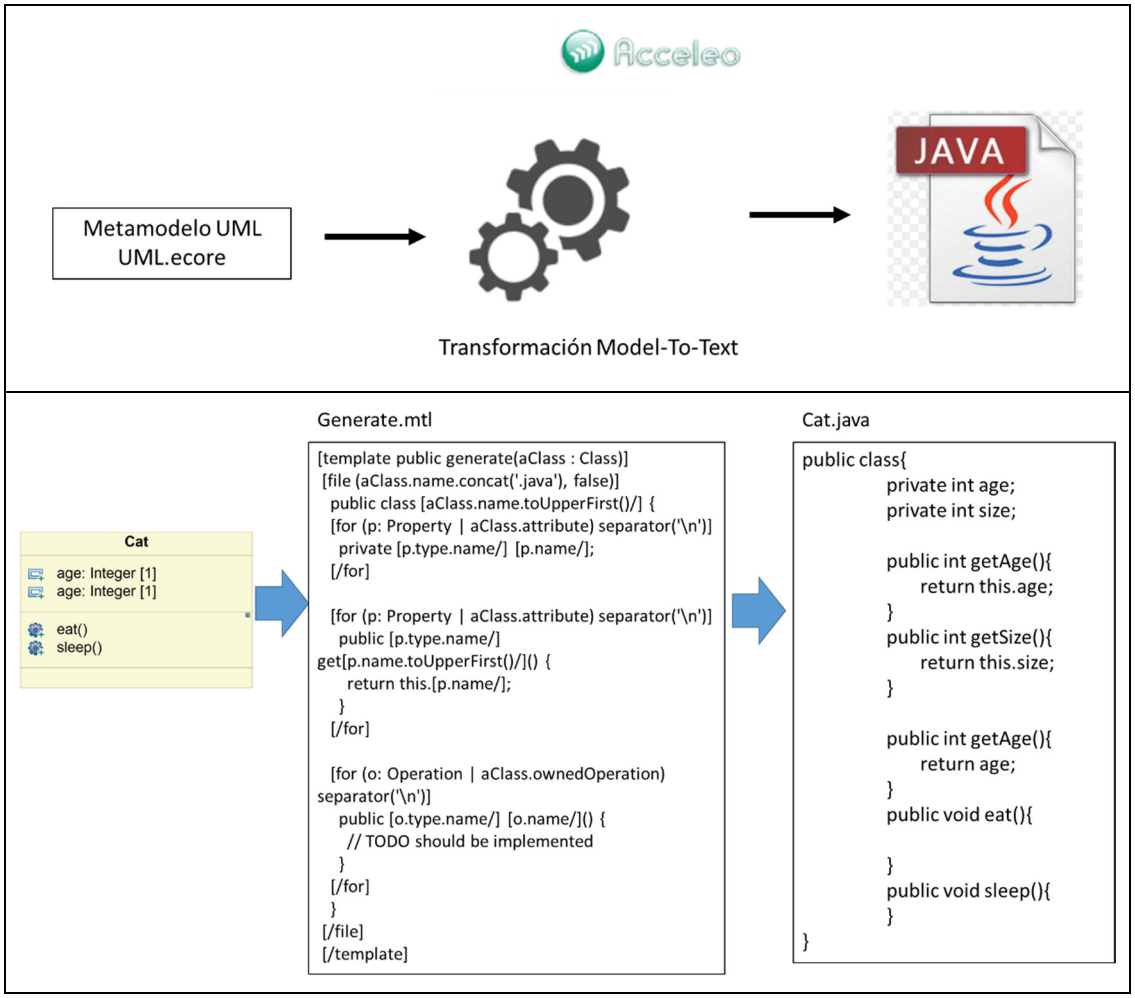


Figura 2.3 Ejemplo de transformación modelo a texto mediante Acceleo

Las transformaciones M2M se pueden clasificar como verticales y horizontales. Las transformaciones horizontales representan transformaciones entre modelos que residen en el mismo nivel de abstracción. Las transformaciones verticales son aquellas que convierten modelos de un nivel de abstracción a modelos de otro nivel de abstracción, por ejemplo, una transformación de modelos de requisitos a diseños, o modelos de diseño PIM a modelos PSM. Las transformaciones también se clasifican en base a si el metamodelo de entrada y el de salida son iguales, ver figura 2.4. Una transformación se denomina endógena si los modelos de entrada y

de salida se especifican mediante el mismo metamodelo. En las transformaciones exógenas los modelos de entrada y salida son expresados utilizando diferentes metamodelos.

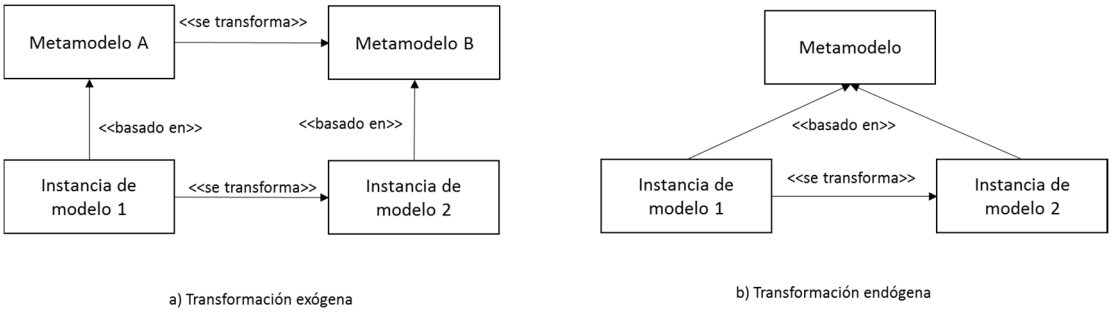


Figura 2.4 Transformaciones exógenas y endógenas

Por ejemplo, una transformación entre un modelo UML y un modelo entidad-relación se puede considerar una transformación exógena y horizontal. Una transformación modelo a modelo para mejorar la calidad del modelo se considera una transformación endógena y horizontal. Las transformaciones M2M utilizadas para refinar modelos y expresar nuevos conceptos se consideran transformaciones endógenas y verticales. Las transformaciones entre modelos PIM y PSM son verticales y exógenas.

2.2.2 Lenguajes de transformación M2M

Para poder implementar las transformaciones de las soluciones de DSDM son fundamentales las herramientas y los lenguajes de transformación. En los últimos años han aparecido un gran número de lenguajes de transformación. QVT Relations (QVT-R) [Omg08b] es el lenguaje estándar propuesto por OMG para las transformaciones de modelos.

La primera clasificación de lenguajes de transformación se realiza en base a si son lenguajes visuales o textuales. En la tabla 2.2 se clasifican los diferentes lenguajes de transformación modelo a modelo en base a si son visuales o no.

Tabla 2.2 Lenguajes de transformación modelo a modelo

Lenguajes textuales	Lenguajes Visuales
ATL	GReAT
Kermeta	VIATRA2
Tefkat	ATOM3
QVT Operational	Fujaba
QVT Relational	QVT Relational
Epsilon Transsformation Language (ETL)	

Además, los lenguajes de transformación modelo a modelo pueden ser declarativos o imperativos. En la tabla 2.3 se agrupan los diferentes lenguajes de transformación M2M en imperativos y declarativos. Existen algunos lenguajes que utilizan, tanto características imperativas como operacionales, por ejemplo ATL [Jou08] y Epsilon Transformation Language (ETL) [Kol08].

Tabla 2.3 Clasificación de lenguajes de transformación declarativos e imperativos

Lenguajes declarativos	Lenguajes imperativos
ATL	ATL
Tefkat	Kermeta
QVT Relational	QVT Operational
VIAGRA2	ETL
GReAT	
ATOM3	
Fujaba	
Epsilon Object Languaje (EOL)	
ETL	

Uno de los lenguajes de transformación más utilizado es ATL. En este trabajo de investigación el estudio se centra en las transformaciones M2M implementadas mediante ATL, aunque la solución desarrollada se puede extender a otros lenguajes declarativos como QVT Relational o híbridos como ETL.

2.2.3 Desarrollo y adaptación de transformaciones M2M

Las transformaciones M2M se han convertido en artefactos software a desarrollar. El desarrollo de las transformaciones M2M no es una tarea trivial. El tamaño y la complejidad de las transformaciones de modelo crecen a medida que su uso se extiende en la industria. Trabajos que ofrecen procesos sistemáticos para el desarrollo de transformaciones de modelos de alta calidad y altamente confiables son fundamentales. Por lo tanto, el desarrollo de transformaciones M2M requiere un ciclo de vida de desarrollo equivalente al desarrollo tradicional de software [Sil14]. Para mejorar el proceso de desarrollo de transformaciones M2M se han realizado trabajos relacionados con la definición de metodologías de desarrollo, especificación de requisitos, definición de patrones y validación.

Un desarrollo tradicional de desarrollo de transformaciones M2M, con el objetivo de establecer las relaciones entre los elementos del metamodelo de entrada y el de salida, podría ser aquel donde el desarrollo se basa en un proceso iterativo de refinamientos entre los requisitos de mapeo y la implementación. En [Gue12] se define un método para el desarrollo de transformaciones que contemplan todas las fases de un proceso de desarrollo de SW tradicional basado en el lenguaje TransML. Primero, las transformaciones M2M requieren una fase de especificación de requisitos de mapeo. Para poder especificar los requisitos de mapeo se deben de establecer y especificar las correspondencias entre los elementos de los metamodelos de entrada y salida. Partiendo del modelo de requisitos de mapeo se van generando diferentes modelos con la información de transformación a implementar. En cada nivel la información es más concreta y en el último refinamiento se establecen relaciones entre los elementos de los metamodelos de entrada y salida para representar la transformación M2M. Mediante un metamodelo de matching creado para el diseño de transformaciones se genera un modelo de relaciones, del cual se ge-

nera un modelo que representa la transformación. La transformación M2M se especifica mediante el metamodelo transML. Finalmente mediante un HOT (Higher Order Transformation) [Tis09] se genera la transformación en el lenguaje de transformación necesario: ATL, QVT u otro.

Con el objetivo de mejorar el proceso de desarrollo de transformaciones la especificación de los requisitos de transformación se ha convertido en tema de investigación relevante. En [Luc16] se define un metamodelo para la definición de la intención de las transformaciones. La definición de la intención puede ser utilizada para validar la transformación, para documentarla, para adaptarla en el proceso de mantenimiento e incluso para reutilizar la transformación.

El uso de patrones en el desarrollo de las transformaciones M2M facilita el desarrollo y la reutilización. [Erg16] define un lenguaje semi-formal para la especificación de patrones de transformaciones de modelo. Mediante un metamodelo independiente de los lenguajes de transformación se permite expresar un catálogo de 15 patrones de transformación.

La refactorización es un mecanismo utilizado en el desarrollo software para mejorar la calidad, y así facilitar el mantenimiento del software. En [Wim12] se presenta un catálogo de patrones de refactorización con el objetivo de ofrecer mecanismos para integrar la refactorización en el desarrollo de transformaciones M2M. En [Sen12] el objetivo es mejorar la reusabilidad de las transformaciones cuando el metamodelo de entrada es modificado. En este caso, el objetivo es encontrar similitudes entre el nuevo metamodelo y el anterior para detectar las transformaciones que se pueden reutilizar. Esta solución está enfocada para ser aplicada cuando se quiere desarrollar una nueva transformación M2M similar a una previa.

Muchas veces las transformaciones se crean manualmente. Hoy en día, el propio desarrollo basado en modelos está siendo utilizando para generar de forma semi-automática las transformaciones M2M, y así facilitar y mejorar su proceso de desarrollo [Tis09]. Los procesos semi-automáticos bien definidos, conllevan diferentes ventajas: mejorar el tiempo de desarrollo, disminuyen la posibilidad de introducir errores que ocurren en la codificación manual, y aumenta la calidad de la transformación al aplicar patrones [Del09]. Existen soluciones que se basan en establecer y descubrir relaciones semánticas entre los metamodelos de entrada y salida [Fal08] [Gue10]. Existe otra corriente, denominada Model Transformation By Example (MTBE) [Var06], basada en deducir las reglas de transformación en base a parejas de modelos ejemplo.

Otra tarea importante es el mantenimiento de las transformaciones. En este ámbito la adaptación de las transformaciones al nuevo requisito es importantísima. Por ejemplo, existen diferentes trabajos sobre la co-evolución entre metamodelos y transformaciones que son fundamentales para integrar los cambios de las nuevas versiones de los metamodelos en las transformaciones M2M.

2.2.3.1 Desarrollo basado de transformaciones M2M dirigidas por modelos ejemplo

Las tareas para especificar y definir las transformaciones M2M son complejas y críticas en el DSDM. El desarrollo de transformaciones dirigidas por ejemplos, MTBE, parte del principio de que es más fácil expresar la transformación en base a los modelos, y no utilizando conceptos de los metamodelos. Las soluciones MTBE permiten utilizar modelos para especificar las transformaciones, lo cual es más intuitivo. El paradigma MTBE se fundamenta en que el uso de una

sintaxis concreta (modelos) es más fácil de utilizar por el usuario, que una sintaxis abstracta (metamodelos) [Kap12]. Las soluciones basadas en ejemplo utilizan parejas de modelos ejemplo (un modelo ejemplo de entrada y un modelo ejemplo de salida) para definir las transformaciones, y partiendo de esas parejas de modelos se deducen las reglas de transformación. Debido a que las reglas de transformaciones se generan semi-automáticamente, se facilita el proceso de desarrollo. Los modelos ejemplo pueden ser utilizados también en la validación de las implementaciones de las transformaciones M2M. Las soluciones MTBE se clasifican en dos grupos: (I) Basadas en correspondencias y (II) Basadas en demostraciones (Model Transformation By Demonstration, MTBD) [Sun09].

La mayoría de las soluciones MTBE utilizan modelos de mapeo para deducir las transformaciones [Bal09]. [Str08] utiliza modelos de correspondencias entre los modelos ejemplo de entrada y salida para generar transformaciones ATL. En [Gar09] se utilizan anotaciones extra con la información del metamodelo de entrada y el de salida para derivar las transformaciones ATL. En [Del09] se establecen las relaciones creando enlaces de weaving entre los modelos ejemplos. Mediante estos enlaces se derivan los elementos de los metamodelos relacionados y se semi-automatiza la generación de las reglas de transformación. En las soluciones MTBE la deducción de reglas de transformación, además de con modelos de correspondencia, también se realiza en base a similitudes entre el modelo ejemplo de entrada y el de salida. Algoritmos heurísticos que analizan las similitudes entre modelos han sido utilizados para generar transformaciones M2M [Kes12]. En [Fau13] se presenta un algoritmo genético implementado mediante JESS, que analizando similitudes entre los modelos ejemplo genera las transformaciones. Esta solución no requiere información extra de mapeo, pero debido a la naturaleza del algoritmo de búsqueda utilizado no se puede aplicar en transformaciones M2M legadas.

En las transformaciones dirigidas por demostraciones la especificación de los requisitos de transformación se realiza modificando ejemplos modelos. En el MTBD la información de las modificaciones realizadas es la utilizada para deducir la lógica de mapeo. La mayoría de las soluciones MTBD son utilizadas en transformaciones endógenas, por ejemplo [Sun13]. En [Lan10] se especifica una solución MTBD aplicable a transformaciones exógenas. Utilizando una solución incremental, en cada etapa se demuestra una pequeña parte de la transformación, y poco a poco internamente se van generando pequeños fragmentos de la transformación. Una vez se han realizado todos los pasos, los diferentes fragmentos se unen para formar la transformación ATL.

2.2.3.2 Co-evolución de metamodelos y transformaciones

La co-evolución es el proceso de evolucionar varios artefactos simultáneamente cuando existe alguna interdependencia. Cuando se debe de realizar algún cambio en los metamodelos de un sistema DSDM los modelos, las transformaciones, los editores de modelos y las diferentes herramientas se ven afectadas [Iov12] [Dir12] [Ros10a]. En la actualidad existen trabajos de co-evolución tanto de migración de modelos y adaptación de transformaciones para situaciones de evolución del metamodelo. Por ejemplo, el objetivo de la herramienta Epsilon Flock [Ros14] es migrar automáticamente los modelos existentes al nuevo metamodelo. COPE [Her09] es una

solución basada en operadores predefinidos para la co-evolución de metamodelos y modelos basada en el framework EMF de eclipse.

Cuando los metamodelos cambian las transformaciones también se deben de adaptar. Las inconsistencias con los metamodelos se manifiestan en la ejecución de la transformación y su validación. Diferentes trabajos abordan la co-evolución entre los metamodelos y las transformaciones. Existen propuestas tanto para el análisis del impacto, como para la adaptación del proceso de DSDM. En [Ros08b] se utiliza la ontología TS (Technological Space Ontology) para describir los cambios ocurridos en los metamodelos. A cada versión del metamodelo se le asocia una serie de diferencias aplicando la ontología, mediante esta información se deducen los cambios a realizar en las transformaciones. En [Gar12] se utilizan las diferencias entre diferentes versiones de metamodelos para derivar las adaptaciones a realizar en las transformaciones ATL. En [Dir13] se utilizan modelos de weaving entre los metamodelos y las transformaciones para analizar el impacto de la evolución de los metamodelos de entrada. En [Kus15] se propone una colección de cambios atómicos que afectan a los metamodelos que permiten describir evoluciones arbitrarias de los metamodelos. Cada tipo de cambios presenta una serie de acciones, relacionadas con los modelos y las transformaciones, que aseguran una consistente co-evolución entre los diferentes artefactos. En [Kru11] se presentan una serie de operadores para facilitar la tarea de co-evolución entre metamodelos y transformaciones. Los operadores son aplicados a los metamodelos y permite la co-evolución semiautomática de las transformaciones.

2.3 Análisis crítico

El desarrollo de las transformaciones M2M no es una tarea trivial. La definición e implementación de las transformaciones de modelos es una tarea compleja y de un alto coste temporal debido a que el desarrollador necesita conocer conceptos de meta-modelado, los metamodelos implicados en la transformación, las relaciones entre los elementos de los diferentes metamodelos y los lenguajes de transformación. Según el DSDM se va introduciendo en el ámbito industrial, será habitual tener que adaptar transformaciones M2M de terceros a nuestras necesidades. Por ejemplo, el aplicar perfiles a los metamodelos de una transformación M2M legada, y adaptar la transformación se convertirá en una tarea habitual. El tener que modificar una transformación M2M legada para adaptarla a las nuevas necesidades de negocio sin modificar los metamodelos también será una tarea a realizar. Al igual que en el desarrollo software actual, en el desarrollo de transformaciones M2M, los desarrolladores también se moverán de un proyecto a otro, según las necesidades de la empresa, y muchas veces ocurrirá que nuevas personas deberán de corregir errores y adaptar transformaciones M2M desarrolladas por otros. Debido a esto, el objetivo de este trabajo es ofrecer una solución, ágil y fácil de usar, para adaptar transformaciones M2M legadas frente a cambios en la lógica de mapeo y extensiones de metamodelos mediante perfiles.

En el ámbito de las transformaciones M2M existen diferentes soluciones para facilitar el ciclo de vida de desarrollo. Los trabajos de co-evolución entre metamodelos y transformaciones

permiten adaptar las transformaciones M2M frente a pequeñas variaciones en los metamodelos. Este tipo de soluciones, no permiten deducir los cambios en la lógica de mapeo si no existen cambios en los metamodelos. Por ello, este tipo de soluciones no se pueden aplicar a los escenarios de este trabajo.

El especificar los nuevos requisitos de mapeo de una forma ágil y sencilla es fundamental cuando el desarrollador de transformaciones M2M se enfrenta un sistema legado. El uso de sintaxis concreta en lugar de sintaxis abstracta, facilita el proceso de especificación de requisitos de mapeo para un desarrollador que no conoce los detalles del sistema. Además, los cambios en los modelos ejemplo tienen una relación directa con los nuevos requisito de mapeo y no dependen de que existan variaciones en los metamodelos. Por ello, el MTBE es adecuado para adaptar transformaciones M2M debido a cambios en la lógica de mapeo. Existen diferentes soluciones basadas en el MTBE para el desarrollo de transformaciones M2M. La mayoría de las soluciones MTBE no se pueden aplicar a transformaciones M2M legadas. Por ejemplo, la propuesta [Lan10] requiere que las reglas de transformación se desarrollen desde el inicio bajo su metodología. Debido a esta característica su aplicación a transformaciones M2M legadas no es directa. Las soluciones MTBE basadas en la búsqueda de similitudes, como [Kes12] [Fau13] no han sido aplicadas a transformaciones M2M legadas debido a que, actualmente, en estas propuestas no diferencian que fragmentos han sido previamente mapeados por la transformación legada. En [Lev13] proponen una solución para evolucionar transformaciones M2M legadas, pero de forma manual. En el trabajo [Sun13], basado en MTBD, se ofrece una solución para automatizar la integración de RNF de dominio en la lógica de transformación. Esta solución solo se puede aplicar a transformaciones M2M endógenas. En el dominio de los DSDM para sistemas empotrados el tipo de transformaciones M2M a adaptar deben ser tanto endógenas (refinamiento y refactorizaciones) como exógenas (de PIM a PSM). Además, la mayoría de las soluciones basadas en ejemplos son constructivas, utilizan adiciones de elementos para expresar la información de mapeo. La eliminación de elementos también debe de ser contemplados en este trabajo.

El contexto de este trabajo de investigación es mejorar el proceso de adaptación de transformaciones M2M cuando los requisitos de mapeo cambian debido a los RNF de dominio. Concretamente, las situaciones a resolver son dos: (1) aquellas situaciones donde la lógica de mapeo cambia pero los metamodelos no sufren cambio alguno y (2) cuando los metamodelos son extendidos mediante perfiles y se requiere adaptar las transformaciones a las nuevas anotaciones. Este trabajo de investigación se ubica en el dominio de los sistemas empotrados. Los sistemas de DSDM del dominio de los sistemas empotrados actualmente utilizan en su mayoría lenguajes de diseño basados en UML y SysML, y utilizan perfiles UML para especificar conceptos relacionados con los RNF. En esta tesis los casos de estudio utilizados en la validación de la solución son transformaciones M2M que trabajan con modelos UML y perfiles UML. En la tabla 2.4 se analizan los diferentes trabajos relacionados con el desarrollo y adaptación de transformaciones M2M. En la tabla se analiza a qué tipo de transformaciones M2M se pueden aplicar

las diferentes propuestas: 1) Transformaciones M2M legadas, 2) transformaciones M2M exógenas y 3) transformaciones M2M endógenas. También se clasifican los trabajos en base a si la adaptación de la lógica de mapeo está relacionadas con: 1) Cambios en los metamodelos, 2) con aplicaciones de perfiles a los metamodelos o 3) con situaciones donde no existen cambios en los metamodelos. Las soluciones se clasifican en dos grupos: 1) Co-evolución metamodelos y transformaciones y 2) MTBE.

Tabla 2.4 Análisis de soluciones para la adaptación de transformaciones M2M.

Métodos para la adaptación de transformaciones M2M		**Tipo de transformaciones M2M**			**Escenario de adaptación de transformaciones M2M debidos a la evolución de RNF**		
Tipo de solución	**Propuesta**	**Legadas**	**Exógenas**	**Endógenas**	**Sin cambios en los metamodelos**	**Extensión de metamodelos mediante perfiles**	**Cambios en los metamodelos**
Diferencias metamodelos	[Gar12]	X	X	X			X
Weaving	[Dir13]	X	X	X			X
Operaciones de diferencias	[Kus15]	X	X	X			X
MTBE	[Sun13].	X		X	X	X	X
	[Kes12]		X	X	X	X	
	[Fau13]		X	X	X	X	
	[Agi15]	**X**	**X**	**X**	**X**	**X**	

La propuesta de este trabajo se fundamenta en el paradigma MTBD, una variación del MTBE. La solución se basa en el uso de modelos ejemplo para adaptar las transformaciones M2M frente a cambios en la lógica de mapeo. Concretamente, se modifican modelos previos para demostrar la nueva transformación. A continuación, se combinan las diferencias creadas en los modelos ejemplo junto con las trazas de ejecución previas. Mediante el uso de los modelos ejemplos se quiere facilitar el uso y poder derivar las operaciones de adaptación aunque no existan cambios en los metamodelos. La propuesta desarrollada, denominada TRANSEVOL, permite analizar transformaciones legadas M2M, tanto endógenas como endógenas, debido a la utilización de las trazas de ejecución. Además, se ha validado su uso en aplicaciones de perfiles UML relacionados con RNF del dominio de los sistemas empotrados en soluciones de DSDM reales. En los siguientes capítulos se describe en detalle la solución desarrollada, sus necesidades y sus detalles de implementación. La propuesta se ha validado en la transformación M2M entre el PIM y el PSM de un sistema de generación de código ANSI-C partiendo de diseños de arquitectura SW basados en componentes UML.

SEGUNDA PARTE
Contribución

3. Método teórico

Los sistemas de generación de código suelen están basados en una arquitectura MDA donde el PIM se trasforma en PSM y éste en código. El desarrollo y mantenimiento de las transformaciones modelo a modelo entre el PIM y el PSM es un punto crítico. Los expertos de dominio definen los lenguajes de modelado y crean los modelos, mientras que la implementación de los lenguajes de modelado y el desarrollo de las reglas de transformación es realizada por los expertos en arquitecturas dirigidas por modelos.

Las reglas de transformación deben de implementar complejos algoritmos de mapeo y su tamaño aumenta según aumenta la expresividad de los lenguajes de modelado. El mantenimiento y adaptación de las transformaciones es una tarea crítica, crucial y compleja. Un escenario típico de adaptación/mantenimiento de transformaciones ocurre cuando una vez implementado el sistema de DSDM se requieren cambios en los requisitos no funcionales de dominio que implican cambios en la lógica de mapeo. Estos escenarios de adaptación de transformaciones de modelo a modelo son aún más complejos cuando la transformación es legada y el responsable de adaptar el sistema de DSDM no conoce ni el diseño ni la implementación.

Al integrar los nuevos RNF la lógica de mapeo puedo variar, y por lo tanto es necesario adaptar las transformaciones. En los sistemas empotrados, los RNF habitualmente se expresan mediante perfiles, sin contaminar los metamodelos. En este trabajo de investigación se han abordado dos escenarios. En el primer escenario se parte de una transformación M2M legada, y el metamodelo de entrada se extiende mediante un perfil con el objetivo de agregar algún RNF de dominio no contemplado previamente. El objetivo es facilitar la implementación de los cambios a realizar en la transformación M2M debido a los nuevos requisitos en la lógica de mapeo. En el segundo escenario, partiendo de una transformación legada se quiere modificar la lógica de mapeo, por ejemplo debido a cambios en los RNF. La investigación, por lo tanto, se focaliza en desarrollar un método y una herramienta que ayude a obtener los cambios a realizar en las reglas de transformación legadas de un sistema de DSDM. De esta forma se consigue reducir el coste y el tiempo de adaptación de los sistemas de DSDM frente a evoluciones de arquitectura software relacionados con los RNF.

A continuación se detallan los objetivos y las hipótesis, la metodología de investigación utilizada y el caso de estudio utilizado en este trabajo de investigación.

3.1 Objetivos

El objetivo principal de este trabajo es desarrollar un método que permita automatizar la adaptación de transformaciones M2M legados frente a evoluciones de RNF de los sistemas empotrados que implican extender los metamodelos mediante perfiles y adaptar las transformacio-

nes para obtener modelos de salida que cumplan con los nuevos RNF. El sistema debe de permitir adaptar las transformaciones M2M de un sistema de MDA tanto si los metamodelos son extendidos como si no (únicamente se requieren cambios en el algoritmo de mapeo). El uso de UML es muy habitual en los diseños de los sistemas empotrados, por lo tanto, este trabajo debe de poder ser aplicado en sistemas de DSDM basados en UML, SysML y perfiles UML.

La tarea de adaptación de las reglas de transformación M2M es parecida al proceso de adaptación utilizado en el desarrollo de SW tradicional pero extrapoladas al nuevo paradigma de desarrollo:

1. Identificar el nuevo requisito "de mapeo" relacionado con la evolución del RNF de dominio.
2. Entender y analizar el cambio para localizar el impacto en la transformación M2M del sistema DSDM legado.
    a) En caso de ser necesario extender los metamodelos PIM/PSM.
    b) Deducir la operación de adaptación a realizar.
    c) Localizar la adaptación a implementar en la transformación M2M.
3. Adaptar las reglas de transformación.
4. Validar la implementación.

Debido a que la localización y la adaptación de las reglas de transformación son las tareas más costosas, es deseable automatizar tanto la deducción de la operación de adaptación, como la localización del cambio en la transformación M2M legada. La localización y búsqueda de la operación de adaptación debe recibir como entrada la especificación del nuevo requisito de mapeo relacionado con la evolución del RNF. Esta especificación debe de poder realizarse sin conocer en detalle la implementación de la transformación legada. La especificación del requisito debe de poder ser utilizado en la validación de la implementación para ofrecer un método ágil de desarrollo. Para poder automatizar o semi-automatizar la implementación de la operación de adaptación deducida, ésta debe poder ser expresada mediante un metamodelo adecuado. El método para adaptar las transformaciones M2M desarrollado en este trabajo se fundamenta en las siguientes necesidades:

a) Facilidad para la especificación del nuevo requisito de mapeo relacionado con la evolución del RNF.
    a. Uso de la especificación en el proceso de validación.
b) Deducción automática de las operaciones de adaptación para transformaciones M2M.
c) Localización automática en transformaciones legadas de la operación de adaptación.
d) Expresar la operación de adaptación mediante un metamodelo.
e) Debe poder ser aplicada a extensiones de metamodelos mediante perfiles.
f) Debe de poder analizar situaciones donde los metamodelos no cambian pero la lógica de mapeo requiere adaptaciones.

Por lo tanto, el objetivo de este trabajo es desarrollar **un método para automatizar la adaptación de transformaciones M2M legadas frente a cambios en la lógica de mapeo**. Para alcanzar el objetivo se debe de:

1. Desarrollar un metamodelo para especificar las operaciones de adaptación a implementar en las transformaciones M2M.
2. Desarrollar un algoritmo para la deducción y localización de las operaciones de adaptación.
3. Implementar una herramienta prototipo que implementa el método definido que permita demostrar la validez del método desarrollado.
4. Validar el método mediante la aplicación de la herramienta prototipo en la adaptación de transformaciones M2M legadas frente a:
    a. Evoluciones de RNF que requieren cambios en la lógica de mapeo.
    b. Evoluciones de RNF que requieren cambios en la lógica de mapeo debido a la aplicación de perfiles UML.

3.2 Hipótesis

La integración e implementación de los requisitos no funcionales requiere la gestión de los requisitos en las diferentes fases del desarrollo SW tradicional: fase de análisis, diseño software, codificación y validación. En el trabajo [Jää04], ver figura 3.1, se resumen y ubican las actividades dentro del ciclo de vida SW que permiten una correcta y efectiva implementación de los RNF. Las tareas para la correcta gestión de los RNF son: el uso de la trazabilidad para facilitar su evolución, revisiones y análisis de cobertura, verificación de los atributos, análisis del impacto en el diseño SW y la implementación en el código final. Este trabajo de investigación se centra en los sistemas de generación de código basados en modelos, y por ello las actividades que impactan en la codificación de los RNF en el código son fundamentales. La gestión de la trazabilidad y análisis/especificación de los requisitos están fuera del ámbito de la presente investigación.

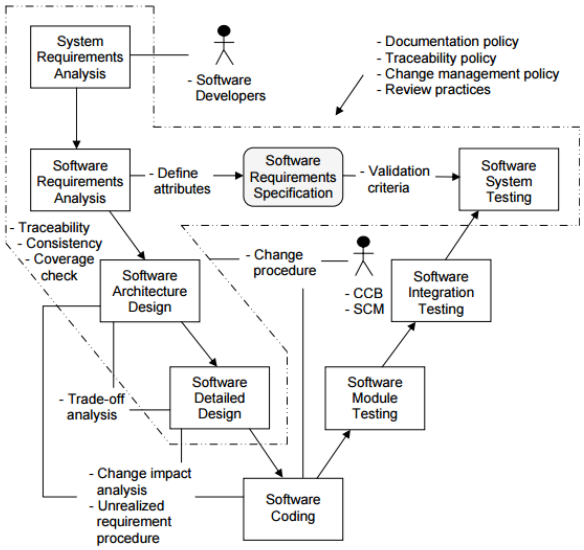


Figura 3.1 Implementación de los requisitos en el ciclo de vida SW [Jää04].

Cuando evoluciona un requisito el código final se debe de modificar. El proceso para la implementación del cambio, una vez realizada la especificación y el análisis del nuevo requisito, requiere analizar el impacto en el diseño de la arquitectura, a continuación en el diseño detallado del software y finalmente implementar los cambios tanto en el diseño como en el código. Los elementos utilizados para analizar el impacto en el código de un nuevo requisito son el diseño del software y su código. Extrapolando estas actividades a un sistema de DSDM, los elementos que se utilizarían para realizar el análisis del impacto en la arquitectura de DSDM serían los modelos PIM (diseño de arquitectura) y PSM (diseño detallado del software) que se modifican para contemplar correctamente el nuevo requisito. Después, las diferencias de estos nuevos modelos se comparan con los previos y las diferencias se utilizan en la localización e implementación de los cambios a realizar en la transformación M2M. El punto de partida y las hipótesis principales de este proyecto se basan en modificar modelos utilizados previamente y utilizando las diferencias entre los modelos deducir los cambios a realizar en la transformación M2M legada. Las hipótesis de este proyecto son:

- **Hipótesis A***: Utilizando las diferencias entre los modelos de entrada y las diferencias entre los modelos de salida es posible detectar los cambios a implementar en la lógica de mapeo de una transformación M2M debido a cambios en los RNF de dominio cuando los metamodelos no requieren cambios.*
- **Hipótesis B:** *Utilizando las diferencias entre los modelos los modelos de entrada y las diferencias entre los modelos de salida es posible detectar los cambios a implementar en una transformación M2M al extender los metamodelos mediante perfiles para integrar nuevos RNF de dominio.*
- **Hipótesis C:** *Es posible localizar las ubicaciones de las modificaciones a realizar en las transformaciones M2M legadas combinando datos de trazabilidad de ejecución de las reglas de transformación con las diferencias de los modelos de entrada y con las diferencias de los modelos de salida.*

3.3 Metodología

Existen muchas clasificaciones de metodologías de investigación [Bar05] [Mye97]. Una de las clasificaciones más comunes es la que divide los métodos en lógicos, hipotético-deductivos y empíricos. Los métodos lógicos hacen referencia a las metodologías que utilizan el pensamiento o deducciones para llegar al conocimiento. El método hipotético-deductivo se basa en la propuesta de una hipótesis como consecuencia de procedimientos inductivos. Se llega a conclusiones particulares a partir de las hipótesis que luego se comprueban experimentalmente.

Por último, los métodos empíricos hacen referencia a un conjunto de metodologías donde se aproxima al conocimiento del objeto y directamente se hace uso de la experiencia. Dentro de los métodos empíricos, los métodos se pueden clasificar en cuantitativos y cualitativos. Los métodos cuantitativos son los que se desarrollan en las ciencias naturales o en las ciencias del

estudio de fenómenos naturales. Algunos ejemplos de métodos cuantitativos incluyen las encuestas, experimentos de laboratorio, métodos formales (p.e.: econométricas) y métodos numéricos tales como el modelado matemático. Por el contrario, los métodos cualitativos se desarrollaron en las ciencias sociales y son investigaciones del estudio social y de los fenómenos culturales.

En el presente caso, el método de investigación que se ha seguido durante la realización de la tesis ha sido el de "caso de estudio", que se puede clasificar como un método empírico y cualitativo. El método "caso de estudio" está particularmente bien adaptado a la investigación en la Ingeniería del Software y los Sistemas de Información [Mye97].

"*Un caso de estudio es una indagación empírica que investiga un fenómeno contemporáneo en el contexto de su vida real, especialmente cuando las fronteras entre el fenómeno y el contexto no son evidentes*" [Yin03].

Investigación de "caso de estudio", son aquellos casos de estudios únicos o múltiples y pueden incluir evidencias cuantitativas, confiar en múltiples fuentes de evidencia y se benefician de desarrollos de proposiciones teóricas anteriores. En vez de utilizar grandes muestras y seguir un protocolo rígido para examinar un número de variables limitados, los métodos "caso de estudio" realizan un examen longitudinal y en profundidad de una sola instancia o evento: un caso. Proporcionan una forma sistemática de mirar eventos, recoger datos, analizar información y realizar informes de resultados. Como resultado, el investigador puede lograr un conocimiento más profundo de por qué la instancia se comportó así, y qué puede ser importante para mirar más extensamente en investigaciones futuras. Los casos de estudio permiten ambas cosas: generar y testear hipótesis. El método de investigación "caso de estudio" puede basarse en una mezcla de evidencias cuantitativas y cualitativas.

Este tipo de método ha funcionado bien con la investigación realizada, ya que ha permitido probar las hipótesis y la validez del método desarrollado. En el presente caso, se ha utilizado un caso de estudio que ha permitido validar las hipótesis.

Pero objetivos de investigación similares pueden ser llevados a cabo de diferentes maneras dependiendo del contexto de investigación: infraestructuras disponibles, accesibilidad y proximidad a expertos, sinergias con proyectos de investigación, etc. Tomando en consideración el contexto de la presente tesis, se ha definido una estrategia de investigación basada en las siguientes actividades y en el método de "caso de estudio":

1. Actualizar el conocimiento revisando publicaciones recientes y relativas al estado del arte de la temática de la presente tesis, además de atender congresos de referencia.
2. Diseñar y desarrollar los diferentes elementos del modelo y el proceso, aumentando el enfoque de manera gradual mediante un proceso iterativo.
3. Experimentación y evaluación de los prototipos incrementales.
4. Asistir a congresos y talleres para presentar resultados parciales con el fin de validar el trabajo realizado, y para seguir analizando los progresos del estado del arte en la temática.

5. Establecer contactos con expertos en congresos, reuniones y mediante correo electrónico.
6. Rediseñar el modelo y la metodología con el feedback obtenido en la actividad anterior.
7. Desarrollar los prototipos finales con el fin de realizar una evaluación integral de los mismos.
8. Diseminar el conocimiento y experiencias adquiridas con la comunidad investigadora.

En la figura 3.2 se resume de manera gráfica la metodología empleada para la consecución de la tesis.

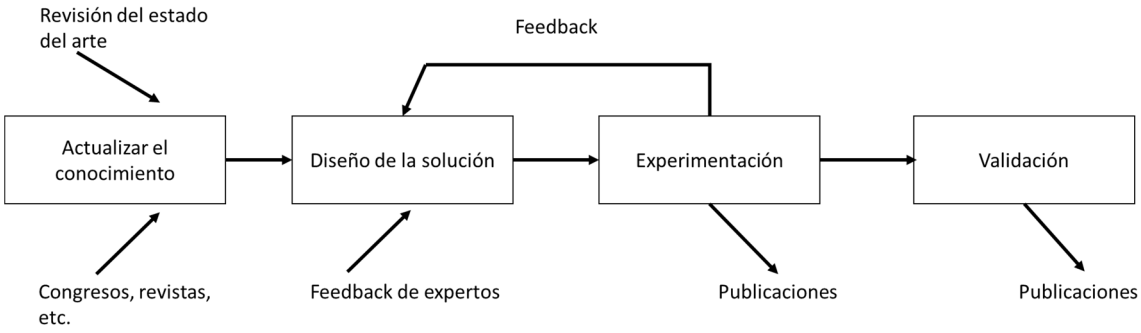


Figura 3.2 Metodolgía de investigación utilizada.

3.4 Caso de estudio

Para evaluar y validar el método desarrollado para la adaptación de transformaciones M2M legadas se ha seleccionado el sistema de DSDM UML2SimpleC [Agi12], ver figura 3.3. El sistema de DSDM UML2SimpleC presenta una arquitectura MDA de dos etapas para la generación de código ANSI-C partiendo de diseños basados en componentes UML. Los modelos independientes de plataforma (PIM) que representan los diseños de las aplicaciones se definen mediante el metamodelo UML y su extensión MARTE, que permite modelar RNF del sistema empotrado. A continuación, el diseño de componentes UML+MARTE se transforma en modelos que representan aplicaciones ANSI-C. Para definir las aplicaciones se utiliza un lenguaje de modelado denominado SimpleC, que representa un subconjunto de ANSI-C. Los modelos de componentes se transforman en modelos SimpleC mediante una transformación M2M implementada mediante el lenguaje ATL. Por último, los modelos SimpleC se transforman en código ANSI-C mediante una transformación M2T implementada con XPAND2. Este proceso DSDM tiene las siguientes características:

- El realizar el proceso en dos etapas permite reutilizar las transformaciones M2T (SimpleC-> ANSI-C) incluso si cambian los metamodelos superiores.

- El utilizar MARTE permite tener la información de la plataforma separada de la información funcional. Esto permite reutilizar las transformaciones relacionadas con la funcionalidad incluso si cambia la plataforma.

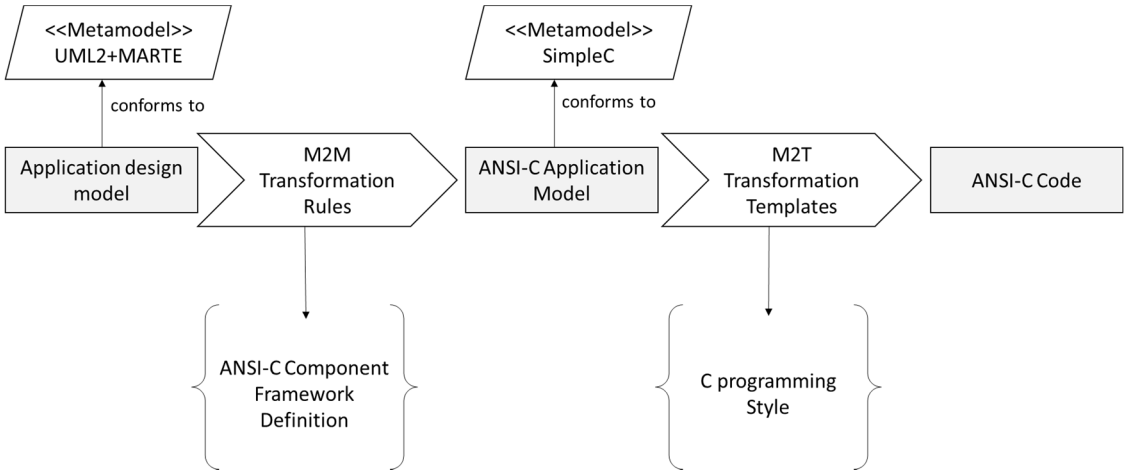


Figura 3.3 Sistema de DSDM **UML2SimpleC** para la generación de código ANSI-C de diseños UML-MARTE.

La transformación M2M del sistema UML2SimpleC se ha utilizado para validar el método TRANSEVOL. Para validar el método desarrollado la transformación M2M se ha adaptado frente a dos escenarios de cambios de RNF utilizando el método TRANSEVOL. El primer escenario utilizado para valorar y evaluar el método desarrollado es una evolución de plataforma que requiere cambios en la lógica de mapeo de la transformación M2M sin que existan cambios en los metamdoelos [Agi14b]. En este primer escenario de evolución se requiere cambiar de API de concurrencia en el software generado. En este caso los modelos de salida de la transformación M2M tienen que ser modificados y para ello las reglas de transformación deben de adaptarse.

El segundo de los escenarios de evolución de RNF utilizados para validar el metodo TRANSEVOL requiere integrar nuevos conceptos relacionados con RNF, y por consiguiente, es necesario adaptar las reglas de transformación. En esta caso se deben de integrar en el sistema de DSDM conceptos de seguridad relacionados con el control de acceso. Para poder expresar los nuevos conceptos de seguridad se agrega al sistema un perfil UML para expresar el control de acceso. Una vez aplicado el nuevo perfil y modificados los modelos se aplica la metodología TRANSEVOL para adaptar las reglas de transformación. Este escenario fue presentado en [Agi15]. Ambos escenarios son descritos en detalle en el capítulo 6.

4. Fundamentos y diseño de TRANSEVOL

El presente capítulo detalla la metodología TRANSEVOL, desarrollada para facilitar la adaptación de transformaciones M2M legadas frente a evoluciones en los requisitos de mapeo. Los escenarios de adaptación de transformaciones M2M objetivo de TRANSEVOL son dos: 1) Cambios en la lógica de transformación sin modificaciones en los metamodelos y 2) Cambios en la lógica de mapeo debido a la aplicación de un perfil para extender alguno de los metamodelos.

En el primer apartado de este capítulo se explica la visión general del método TRANSEVOL, basado en el uso de modelos ejemplo para especificar los nuevos requisitos de mapeo. A continuación, se describe el metamodelo MM*AdaptationGoal*. El metamodelo MM*AdaptationGoal* permite especificar las diferentes operaciones de adaptación contempladas en TRANSEVOL. En el segundo apartado, se describen todas las operaciones de adaptación contempladas por TRANSEVOL. Una vez analizadas las operaciones de adaptación, se describen los diferentes escenarios de evolución de lógica de mapeo contemplados en este trabajo, y se relacionan las diferentes operaciones de adaptación con cada escenario de evolución. Para deducir el tipo de escenario de evolución se utilizan las diferencias entre los modelos previos, utilizados por la transformación M2M legada, y los modelos ejemplo, creados para especificar la nueva lógica de mapeo. TRANSEVOL define un algoritmo basado en las diferencias entre modelos para detectar los diferentes escenarios de evolución, y así deducir las operaciones de adaptación necesarias. En el tercer apartado se detalla el núcleo y los conceptos utilizados por el algoritmo de deducción de operaciones de adaptación. Para poder ubicar las diferentes operaciones de adaptación en la transformación M2M es necesario relacionar las diferencias de los modelos con las reglas de transformación. En el último apartado de este capítulo se explica cómo se combinan la traza de ejecución de la transformación M2M legada y las diferencias de los modelos para recolectar la información necesaria para especificar y localizar las operaciones de adaptación en la transformación M2M legada.

4.1 Visión general del método TRANSEVOL

En este apartado se describe la visión general del método TRANSEVOL. Primero se describen los pasos a seguir para aplicar el método TRANSEVOL en la adaptación de transformaciones M2M legadas. A continuación se describe un sencillo ejemplo para ilustrar el método de adaptación TRANSEVOL. Y finalmente, se concluye enumerando los elementos técnicos necesarios para desarrollar el método TRANSEVOL.

4.1.1 Proceso para la adaptación de transformaciones M2M legadas mediante modelos ejemplo

El proceso de adaptación de un sistema SW presenta los siguientes pasos: a) Especificar el nuevo requisito b) Deducir la modificación a realizar c) Localizar la modificación a realizar d) Implementar la modificación y e) Validar la modificación. Extrapolando el proceso de adaptación SW a la evolución de las transformaciones M2M los pasos a seguir para realizar la adaptación son:

1. Identificar y expresar el nuevo requisito de mapeo
2. Entender y analizar el impacto del cambio en el sistema DSDM
    a. Deducir la modificación a realizar en la lógica de mapeo
    b. Localizar la modificación a realizar en la transformación M2M
3. Implementar los cambios en la transformación M2M
4. Validar la implementación

La idea principal de TRANSEVOL es obtener los cambios a realizar en una transformación M2M legada partiendo de la especificación de los nuevos requisitos de mapeo expresados mediante modelos ejemplo. El método desarrollado se basa en la generación automática de transformaciones dirigidas por modelos ejemplo MTBE. Dentro del MTBE, el método TRANSEVOL se puede catalogar como un método para el desarrollo y adaptación de transformaciones basadas en demostraciones MTBD. La razón de utilizar modelos ejemplos en la adaptación de transformaciones es debido a que los escenarios que se quieren solucionar en este trabajo son aquellos donde evoluciona la lógica de mapeo pero no existen cambios en los metamodelos.

Para demostrar los nuevos requisitos de transformación se modifican una pareja de modelos utilizada y generada previamente por la transformación M2M legada. Utilizando las diferencias generadas entre los nuevos modelos y los modelos previos se deducen y localizan los cambios a realizar en la transformación M2M legada. La localización de los cambios se realiza combinando la información de las diferencias generadas con la traza de ejecución de la transformación legada. Una vez localizados los cambios se implementan las nuevas reglas de transformación. Tras implementar los cambios, los modelos ejemplo utilizados para especificar los nuevos requisitos de transformación se utilizan para validar la nueva implementación. La figura 4.1 muestra la arquitectura global y los artefactos requeridos por TRANSEVOL. El proceso para el análisis de las transformaciones es el siguiente:

1) Definir el nuevo requisito de transformación y definir los modelos ejemplo de entrada y salida que demuestran el nuevo requisito de transformación.
    a. Crear el modelo ejemplo de entrada modificando un modelo previo sin el nuevo requisito de transformación.
    b. Crear el modelo ejemplo de salida modificando un modelo previo sin el nuevo requisito de transformación.
2) Obtener el modelo de diferencias entre los modelos de entrada.
3) Obtener el modelo de diferencias entre los modelos de salida.

4) Obtener la traza de ejecución de la transformación para los modelo de entrada y salida previos.
5) Localizar la ubicación de las operaciones de adaptación deducidas.
6) Deducir los cambios a realizar en la transformación modelo a modelo y expresarlos mediante un modelo.
7) Modificar las reglas de transformación para responder al nuevo requisito. Para automatizar la implementación se ejecuta un HOT que tiene como entrada la transformación M2M legada y el modelo de operaciones de adaptación generado en el paso anterior.
8) Validar la nueva implementación utilizando los modelos ejemplo.
9) Realizar una validación regresiva de la nueva implementación con los modelos de entrada y salida previos.

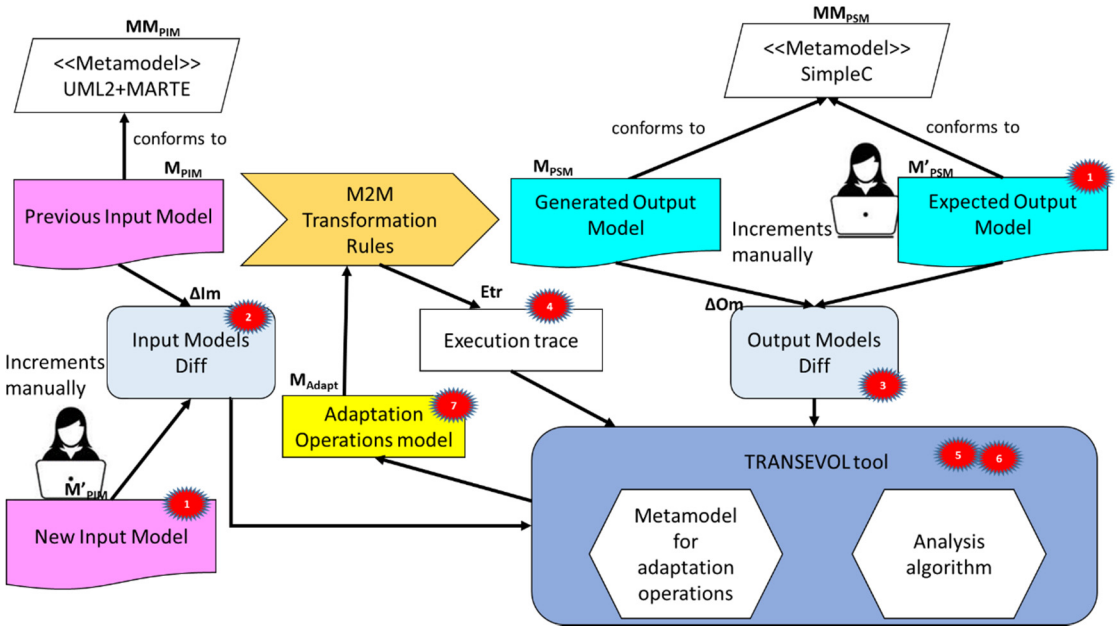


Figura 4.1 Visión general del método TRASNEVOL para la adaptación de transformaciones M2M legadas

En la siguiente lista se recogen todos los metamodelo y modelos que se deben de tener para realizar el análisis y deducir las modificaciones:

a) Metamodelo de entrada (MM_{PIM})
b) Modelo de entrada previo (M_{PIM})
c) Nuevo modelo de entrada ejemplo (M'_{PIM})
d) Modelo de diferencias entre los modelos de entrada (Δ**Im**)
e) Metamodelo de salida (MM_{PSM})
f) Modelo de salida previo (M_{PSM})
g) Nuevo modelo de salida ejemplo (M'_{PSM})
h) Modelo de diferencias entre los modelos de salida (Δ**Om**)
i) Modelo de la traza de ejecución de la trasformación legada con los modelos de entrada y salida previos (**Etr**)
j) Modelo con las operaciones de adaptación a implementar en la transformación legada (M_{Adapt})

4.1.2 Ejemplo de una adaptación de una transformación M2M mediante TRANSEVOL

A continuación se va a ilustrar el proceso de adaptación con un ejemplo concreto y sencillo. El objetivo del ejemplo es describir de una forma sencilla y genérica la base del método desarrollado. La transformación M2M legada transforma los elementos del tipo A del metamodelo MM_{PIM} (MM_{PIM}!A) en dos elementos del tipo B del metamodelo MM_{PSM} (MM_{PSM}!B). Debido a un cambio en los objetivos de un RNF se decide que por cada MM_{PIM}!A se deben de generar tres elementos MM_{PSM}!B. Por lo tanto el nuevo requisito de mapeo es que por cada MM_{PIM}!A se generan tres elementos del tipo MM_{PSM}!B. Para comenzar el proceso de adaptación primero se selecciona un modelo PIM (M_{PIM}), previamente diseñado, y su correspondiente modelo PSM (M_{PSM}) generado por la transformación M2M legada. El siguiente paso es reflejar el nuevo requisito de mapeo debido a cambios en los RNF de dominio. En este caso, se modifica manualmente el modelo M_{PSM} para reflejar las nuevas necesidades. Tras incrementar el modelo de salida M_{PSM} se obtiene el nuevo modelo M'_{PSM}. A continuación se calcula la diferencia entre el nuevo modelo y el previo y se obtienen las modificaciones realizadas. En este caso se ha agregado una instancia del tipo MM_{PSM}!B. El nuevo requisito de mapeo requiere cambiar la regla de transformación de la figura 4.2, para que genere tres elementos del tipo MM_{PSM}!B en lugar de dos. Para que esto se pueda realizar de forma automatizada partiendo de los elementos diferencia, se debe de obtener la regla de transformación a modificar. Para esto se debe de conocer cuál fue la regla de transformación que generó los elementos del tipo MM_{PSM}!B. Para automatizar esta localización es necesario conocer la traza de ejecución de la transformación legada con los modelos M_{PIM} y M_{PSM}. Una vez obtenida la traza de ejecución previa se puede obtener la regla de transformación a modificar localizando la regla de transformación que generó las otras instancias del tipo MM_{PSM}!B, en este caso la regla *from_A_2_B*. La regla de transformación se debe de modificar agregando un nuevo patrón de salida. La figura 4.2 resume la adaptación de la transformación M2M ejemplo.

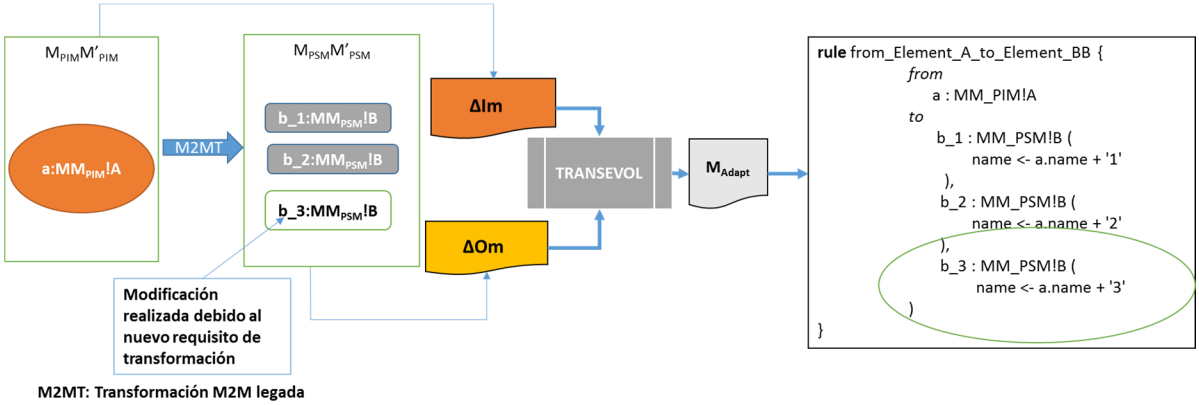


Figura 4.2 Ejemplo de adaptación de transformación M2M legada

4.1.3 Conclusión

Para poder desarrollar el método propuesto se han desarrollado los siguientes elementos:

1. Algoritmo para la deducción de operaciones de adaptación en base a diferencias entre modelos. El algoritmo utiliza la herramienta EMFCompare y el metamodelo EMFDiff [Tou06] para obtener y expresar las diferencias entre modelos.
2. Algoritmo para la localización de las operaciones de adaptación. Para ello es necesario obtener las trazas de ejecución de la transformación legada M2M. En el caso de TRANSEVOL las trazas de ejecución se especifican mediante el metamodelo Tracer [Jou05].
3. El metamodelo *MMAdaptationGoal*. Este metamodelo permite expresar las operaciones de adaptación.

El objetivo de este trabajo es generar automáticamente el modelo de operaciones de adaptación partiendo de un modelo de diferencias y una traza de ejecución para reducir así el tiempo de modificación. La propuesta TRANSEVOL utiliza conceptos del lenguaje ATL para expresar las operaciones de adaptación. Debido a esto el método TRANSEVOL actualmente solo se ha aplicado a transformaciones M2M implementadas mediante ATL. En los siguientes apartados se describen en detalle cada uno de los elementos que constituyen el método desarrollado.

4.2 Metamodelo MMAdaptationGoal

El método TRANSEVOL define y utiliza el metamodelo denominado MM*AdaptationGoal* para expresar los cambios a implementar en las reglas de transformación M2M. En la figura 4.3 se puede ver el diagrama del metamodelo *MMAdaptationGoal*. El expresar las operaciones mediante un metamodelo permite automatizar mediante HOTs la adaptación de las transformaciones M2M. Las reglas de transformación están sujetas a las siguientes operaciones de adaptación en los diferentes escenarios de evolución: *addRule* (agregación de una regla de transformación), *splitRule* (dividir una regla de transformación en dos), *deleteRule* (eliminación de una regla de transformación), *extractSuperRule* (generalización de reglas de transformación) *deleteOutputPatternElement* (eliminación de un patrón de mapeo), *removeBinding* (eliminación de una asignación), *addInputPatternElement* (agregación de un elemento de entrada de una regla de transformación), *addOutputPatternElement* (agregación de un elemento de salida de una regla de transformación), *addBinding* (agregación de una asignación), *moveOutputPatternElement* (mover patrón de mapeo de salida de una regla de transformación), *moveBinding* (cambio de ubicación de una asignación), *updateBinding* (modificación de una asignación) y *UpdateFilter* (modificación del filtrado de una regla de transformación).

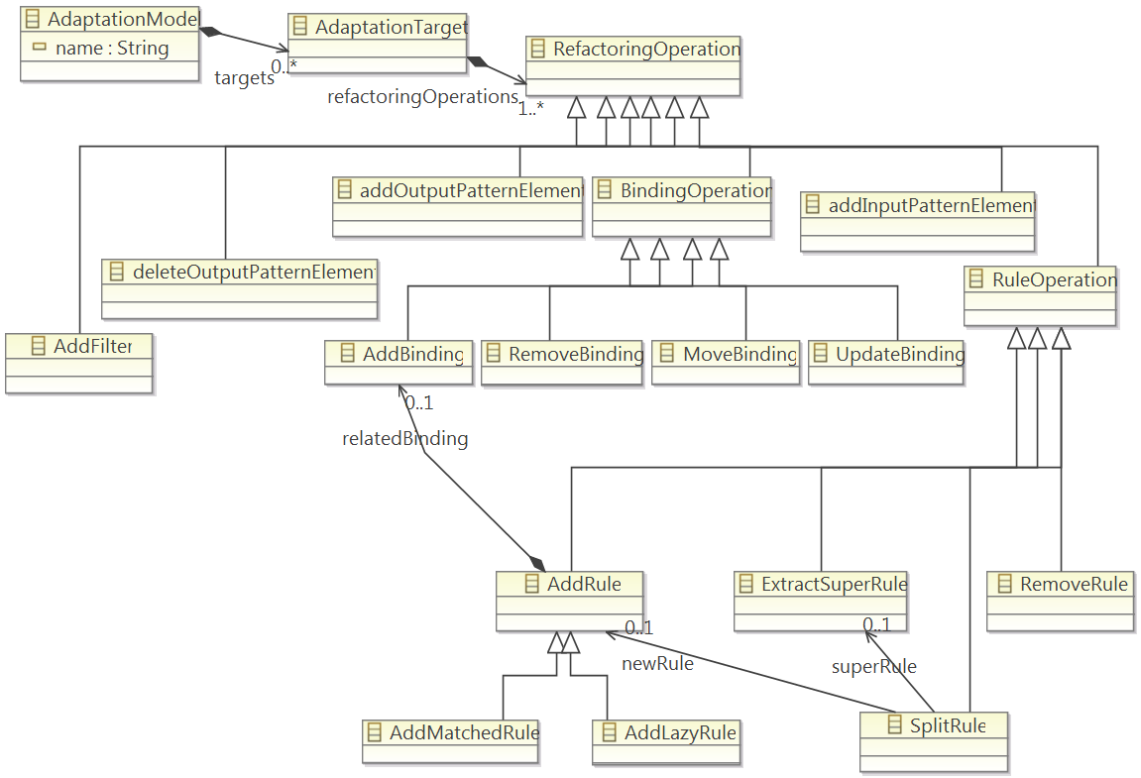


Figura 4.3 Metamodelo MMAdaptationGoal

Tras las diferencias generadas por los modelos ejemplo TRANSEVOL genera un modelo especificando las adaptaciones necesarias en la transformación legada. Cualquier operación de adaptación se define como un elemento del tipo *MMAdaptationGoal!AdaptationTarget*. Cada *MMAdaptationGoal!AdaptationTarget* está compuesto por un conjunto de operaciones de adaptación *MMAdaptationGoal!RefactoringOperation*. Cada operación de adaptación requiere diferente información para expresar la modificación a realizar. Para poder expresar esta información el metamodelo *MMAdaptationGoal* incorpora conceptos del metamodelo del lenguaje ATL. Los elementos del metamodelo ATL se utilizan para definir las diferentes operaciones de adaptación. En la tabla 4.1 se resume la información requerida por cada operación de adaptación. En los siguientes apartados se describen en mayor detalle cada una de las operaciones de adaptación del metamodelo *MMAdaptationGoal.*

Tabla 4.1 Atributos de las operaciones de adaptación del metamodelo MMAdaptationGoal

Operación de adaptación	Propiedades de la operación de adaptación
Add Rule	newRule*: ATL!Rule* relatedBinding: *MMAdaptationGoal!AddBinding*
Add MatchedRule (extends AddRule)	newRule*: ATL!Rule* relatedBinding: *MMAdaptationGoal!AddBinding*
Add LazyRule (extends AddRule)	newRule*: ATL!Rule* relatedBinding : *MMAdaptationGoal!AddBinding*
Remove Rule	affectedRule:ATL!Rule relatedBinding: *MMAdaptationGoal!RelatedBinding*
Split Rule	affectedRule: *ATL!Rule* newRule: *MMAdaptationGoal!AddRule* *superRule: MMAdaptationGoal!ExtractSuperRule*
Add Binding (extends BindingOperation)	affectedRule: *ATL!Rule* newBinding : *ATL!Binding*
Remove Binding (extends BindingOperation)	affectedRule: *ATL!Rule* affectedBinding: *ATL!Binding*
Update Binding (extends BindingOperation)	affectedRule: *ATL!Rule* affectedBinding: *ATL!Binding* newValue:OCL!OclExpression
Move Binding (extends BindingOperation)	affectedRule: *ATL!Rule* toRule: *ATL!Rule* binding: *ATL!Binding*
Add filter to input pattern	newFilter: *OCL!OclExpression* affectedRule: *ATL!Rule*
Add input pattern element	affectedRule: *ATL!Rule* *newInput:ATL!InputPatternElement*
Add output pattern elemen	affectedRule: *ATL!Rule* *outputPattern: ATL!OutputPatternElement*
Delete out pattern element	affectedRule: *ATL!Rule* *outputPattern: ATL!OutputPatternElement*
Extract super rule	newRule: *ATL!Rule*

4.2.1 AddBinding

La necesidad de agregar una nueva asignación, denominado *binding* en ATL, se da cuando una nueva relación entre instancias de salida es necesaria. El objetivo de agregar una asignación es crear una nueva relación o referencia entre dos objetos. Este tipo de modificaciones se compone por una operación de asignación en un patrón de elemento de salida de una regla, denominada regla afectada. La figura 4.4 presenta la metaclase *MMAdaptationGoal!AddBinding*. En la figura 4.5 se representa un modelo *MMAdaptationGoal* que especifica la necesidad de agregar una referencia entre los elementos del tipo *MMB!B y MMB!C*. En el código de la figura 4.5 la regla afectada por esta nueva asignación es la regla *B2B*, que genera elementos del tipo MMB!B.

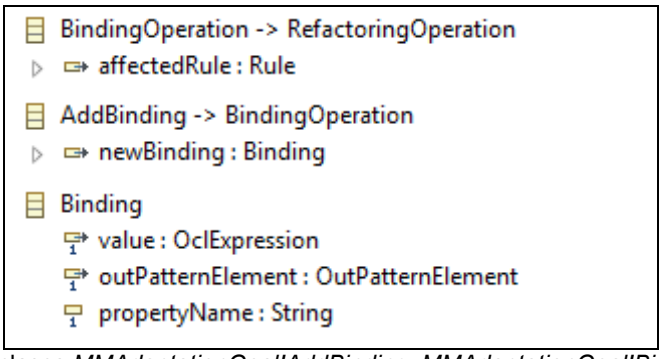


Figura 4.4 Metaclases *MMAdaptationGoal!AddBinding*, *MMAdaptationGoal!BindingOperation* y *ATL!Binding*.

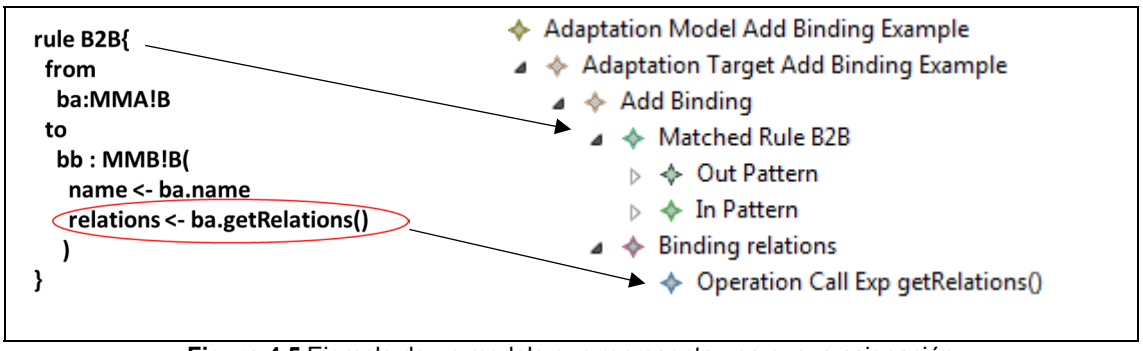


Figura 4.5 Ejemplo de un modelo que representa una nueva asignación

4.2.2 RemoveBinding

La decisión de borrar una asignación previa de una regla de transformación puede venir dada por modificaciones en los metamodelos o por modificaciones en la lógica de mapeo. La operación de borrado de asignación se define mediante una regla afectada y la asignación a eliminar, ver figura 4.6.

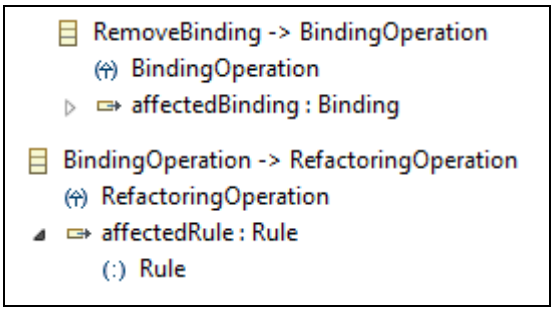


Figura 4.6 Metaclase *MMAdaptationGoal!RemoveBinding*

4.2.3 UpdateBinding

La modificación de una asignación requiere modificar la expresión de la asignación. Este tipo de operaciones son similares a la eliminación de asignaciones, exceptuando que la expresión de asignación es modificada. La imagen 4.7 representa la metaclase *MMAdaptationGoal!UpdateBinding*.

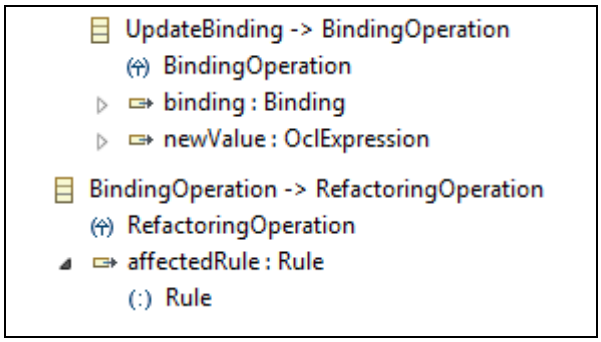


Figura 4.7 Metaclase *MMAdaptationGoal!UpdateBinding*.

4.2.4 MoveBinding

Debido a cambios en la lógica de mapeo puede ocurrir que una asignación se deba reubicar. Para especificar una reubicación de asignación se debe de especificar la regla de la cual se va a extraer la asignación y la regla destino. En la figura 4.8 se define la metaclase *MMAdaptationGoal!MoveBinding*.

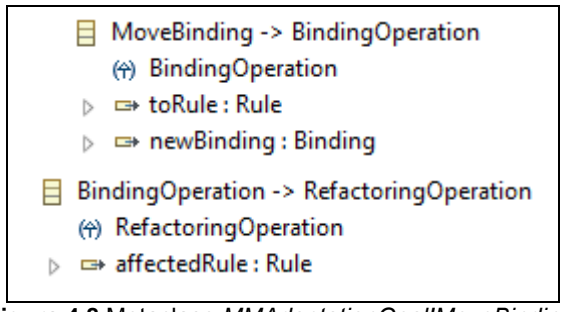


Figura 4.8 Metaclase *MMAdaptationGoal!MoveBinding*.

4.2.5 AddRule

La metaclase *MMAdaptationGoal!AddRule* representa la operación de adaptación que requiere la agregación de una nueva regla de transformación. Este tipo de situaciones pueden ocurrir cuando se quiere incluir en la transformación M2M nuevos tipos de metaclases previamente no contemplados. Cuando se quiere incluir una nueva regla de transformación se debe de crear la nueva regla de transformación y además hay que incluir los nuevos elementos de salida generados en los contenedores correspondientes. Para incluir los elementos generados en los elementos contenedores de salida se debe de realizar una nueva operación de asignación. Esta operación en el lenguaje de mapeo ATL se denomina *binding*. Por lo tanto, para agregar una nueva regla son necesarios: 1) una nueva regla de transformación y 2) un binding. En la figura 4.9 se puede ver la definición de la metaclase *MMAdaptationGoal!AddRule*. Una operación del tipo *AddRule* es del tipo *RuleOperation* y está compuesta por un elemento del tipo *ATL!*MatchedRule, denominado *newRule*, y por una operación *MMAdaptationGoal!AddBinding*. En el lenguaje ATL se contemplan dos tipos de reglas: 1) MatchedRules y 2) LazyRules. Las metaclase *MMAdaptationGoal!AddMacthedRule* y *MMAdaptationGoal!AddLazyRule* son especializaciones de la clase *MMAdaptationGoal!AddRule* utilizadas para diferenciar entre las reglas del tipo *matched* y *lazy*. La única diferencia entre ambas es que al agregar un *lazyRule* en la operación de asignación se

le llama imperativamente a la regla de transformación. Al agregar un *MatchedRule* en la asignación se relacionan las instancias creadas por la regla con sus contenedores directamente.

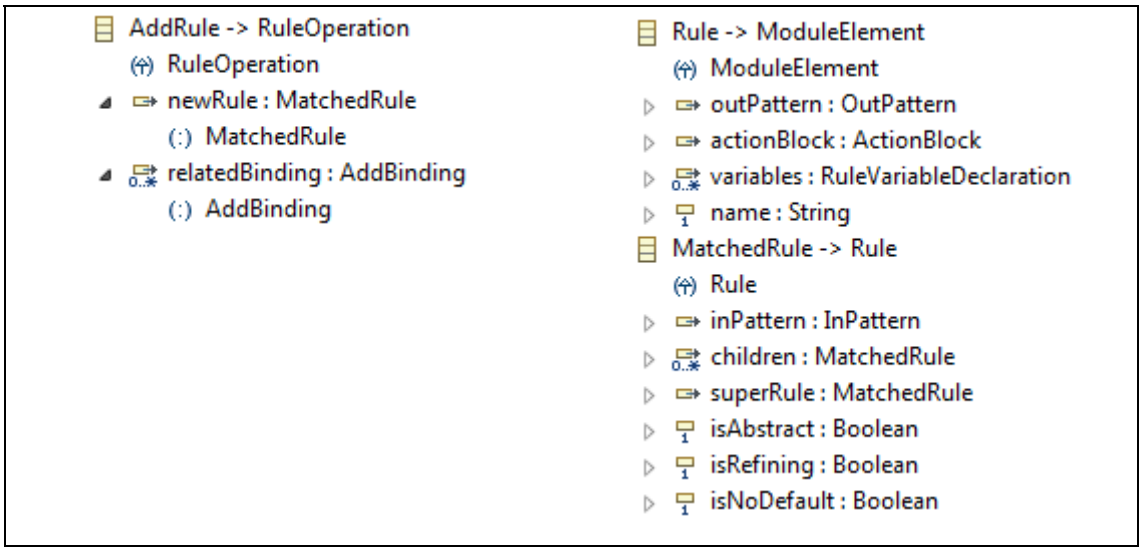


Figura 4.9 Metaclase *MMAdaptationGoal!*AddRule, metaclase *ATL!Rule* y la metaclase *ATL!MatchedRule*.

Para ilustrar como se realiza la definición de una operación de creación de una nueva regla de transformación se utilizará un pequeño ejemplo. El punto de partida del ejemplo es una simple regla de transformación que convierte elementos de entrada del tipo *MMA!A* en elementos de salida del tipo *MMB!A*, ver figura 4.10. El escenario de evolución requiere que los elementos de entrada del tipo *MMA!B* se conviertan en elementos del tipo MMB!B. Estos nuevos elementos serán contenidos por los elementos del tipo MMB!A.

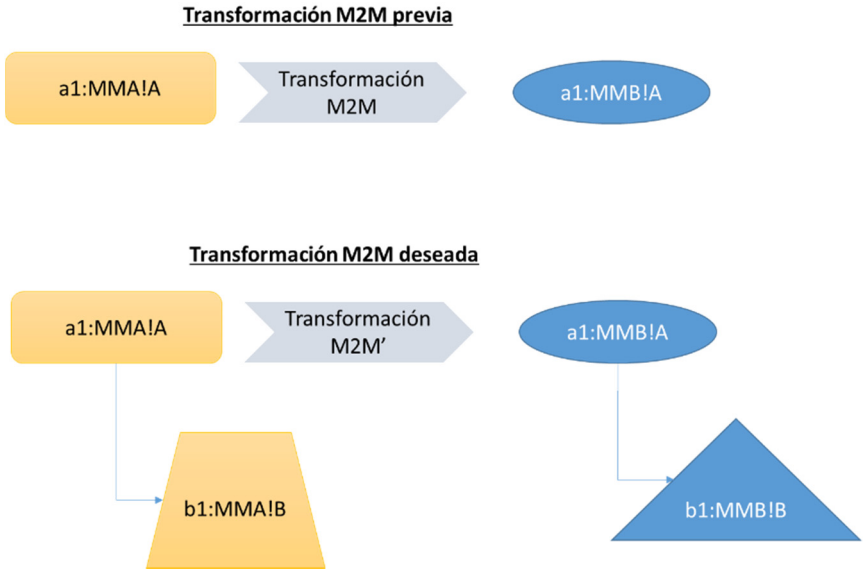
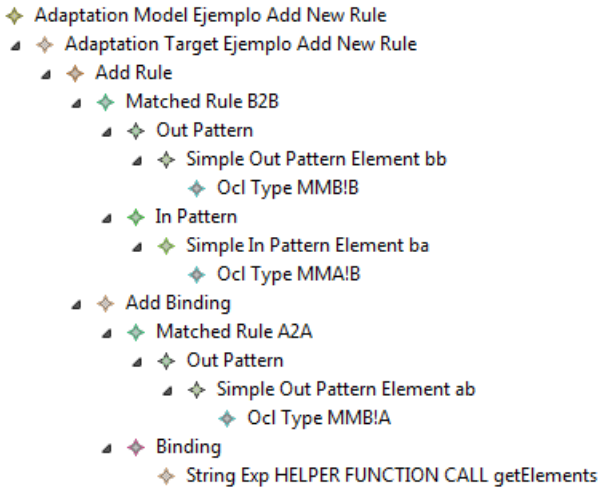


Figura 4.10 Ejemplo de un escenario de evolución que requiere una nueva regla de transformación

Para implementar el nuevo requisito es necesario crear una nueva regla de transformación que transforme elementos del tipo *MMA!B* en elementos del tipo *MMB!B* y además estos elementos sean asignados a sus contenedores *MMB!A* correspondientes. Esta operación de

adaptación se refleja en el modelo *"addRuleExample.adaptationgoal"* de la tabla 4.2 y corresponde al código ATL de la misma.

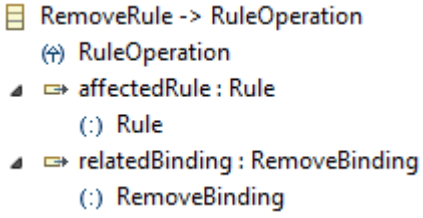
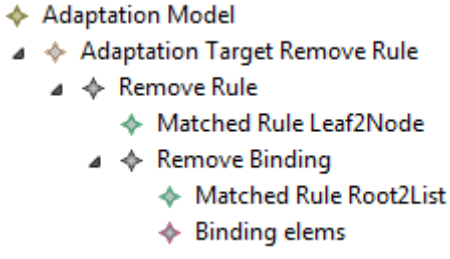
Tabla 4.2 Ejemplo de un modelo del tipo MMAdaptationGoal que especifica la necesidad de una nueva regla

Modelo para una nueva regla de transformación	Código ATL ejemplo
	**rule A2A{** **from** aa:MMA!A **to** ab : MMB!A(name <- aa.name, *new binding* elements <- aa.getElements()) } *new rule* **rule B2B{** **from** ba:MMA!B **to** bb : MMB!B(name <- ba.name --TODO) }

4.2.6 Remove Rule

Existen situaciones donde se debe de eliminar algún mapeo entre elementos. Esto normalmente, puede ocurrir porque se quiere dejar de mapear ciertos elementos de entrada. La eliminación de una regla se expresa mediante la metaclase *MMRuleAdaptation!RemoveRule*. Una eliminación de regla requiere expresar la información representada en la tabla 4.3. Además de definir la regla a eliminar también se expresa la asignación a eliminar.

Tabla 4.3 Metaclase *MMAdaptationGoal!Remove*Rule y un modelo ejemplo que especifica la eliminación de una regla.

Metaclase *MMAdaptationGoal*!RemoveRule	Modelo que representa la eliminación de una regla
	

4.2.7 AddFilter

En muchas situaciones se requiere aplicar un mapeo solo a ciertas instancias de un tipo de metaclases. Para conseguir este tipo de mapeos se debe de añadir un filtro al patrón de elementos de entrada de una regla de transformación. El código de la imagen 4.11 representa una regla de

transformación con filtro. Para especificar estas situaciones se define la regla afectada y se especifica la expresión de filtrado. En la figura 4.12 se puede ver la metaclase *MMAdaptationGoal!AddFilter.*

```
rule A2A {
from
  aa:MMA!A (aa.onlySelected())
to
  ab : MMB!A(
     name <- aa.name,
     elements <- aa.getElements()
  )
}
```

Figura 4.11 Regla de transformación con filtro en el patrón de entrada.

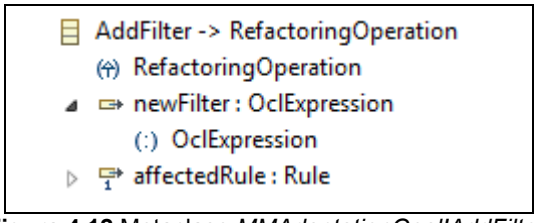


Figura 4.12 Metaclase *MMAdaptationGoal!AddFilter*.

4.2.8 AddOutputElement

Existen situaciones donde es necesario generar dos elementos de salida de un tipo por cada elemento de entrada de un tipo. Si la regla de transformación inicialmente solo genera una instancia por cada elemento es necesario modificar la regla de transformación para que genere dos elementos de salida. Para adaptar la regla a las nuevas necesidades se debe de agregar un patrón de elemento de salida. Esta situación también puede ser necesaria cuando en una regla de transformación se requiere generar por cada elemento de entrada un nuevo elemento de salida de un tipo previamente no contemplado. La metaclase que especifica la adición de un nuevo patrón de elemento de salida a una regla de transformación se representa en la tabla 4.4.

Tabla 4.4 Metaclase *MMAdaptationGoal!AddOutputPatternElement*.

Metaclase MMAdaptationGoal!AddOutputPatternElement	Adición de un patrón de salida a una regla de transformación ATL
addOutputPatternElement -> RefactoringOperation RefactoringOperation affectedRule : Rule (:) Rule newBinding : AddBinding (:) AddBinding	**rule A2_A_C {** **from** aa:MMA!A **to** ab : MMB!A(name <- aa.name, elements <- aa.getElements()), cb : MMB!C(name <- aa.name,) } *New output pattern element*

4.2.9 AddInputPatternElement

También puede ocurrir que para generar un elemento de salida se requieran dos elementos de entrada. En la imagen 4.13 se especifica la información necesaria para agregar un nuevo tipo de elemento de entrada a una regla de transformación. Esta metaclase es similar a *MMAdaptation-Goal!AddOutputPatternElement.*

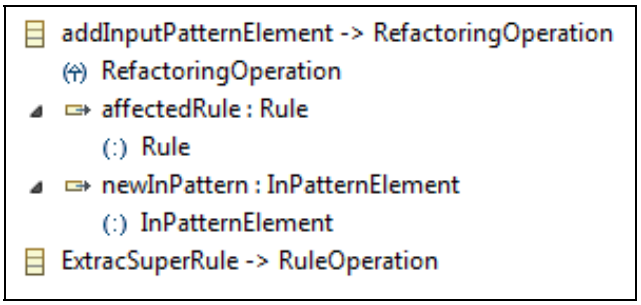


Figura 4.13 Metaclase *MMAdaptationGoal!AddInputPatternElement*

4.2.10 DeleteOutputPatternElement

Existen circunstancias donde se requiere eliminar un patrón de elemento de salida. La especificación de este tipo de operaciones es muy similar a la adición de patrones de salida. En la imagen 4.14 se representa la metaclase *MMAdaptationGoal!DeleteOutputPatternElement.*

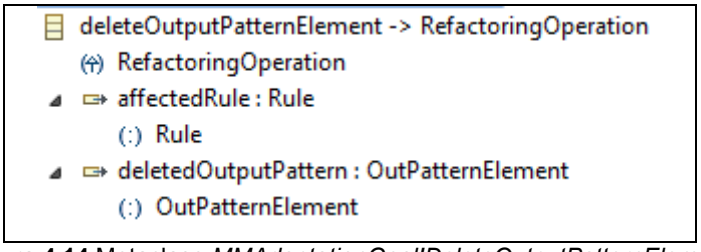


Figura 4.14 Metaclase *MMAdaptationGoal!DeleteOutputPatternElement*.

4.2.11 SplitRule

Esta operación de adaptación representa aquellas situaciones donde partiendo de una regla de transformación uno a uno, se quiere pasar a una transformación donde dependiendo de alguna característica del elemento de entrada se quiere generar diferentes tipos de elementos/instancias de salida. Por ejemplo, los elementos del tipo *MMA!A* que contienen un valor concreto de un atributo se seguirán convirtiendo en elementos del tipo *MMB!A* pero aquellos elementos del tipo *MMA!A* que no presenten ese valor se transformarán en elementos del tipo *MMB!C*. Este tipo de operaciones de adaptación requieren crear un filtrado en los patrones de los elementos de entrada de la regla a dividir y generar una nueva regla con el mismo patrón de entrada pero con el filtro invertido. El código de la figura 4.15 representa un ejemplo de código donde se divide una regla de transformación en dos. En la tabla 4.5 se presenta tanto la metaclase que representa una división de regla y un modelo ejemplo que especifica la división de una regla de transformación.

Regla inicial

```
rule A2A{
from
  aa:MMA!A
to
  ab : MMB!A(
     name <- aa.name,
     elements <- aa.getElements()
   )
}
```

Regla dividida

```
rule A2A{
 from
  aa:MMA!A (not aa.isSpecial())
 to
  ab : MMB!A(
   name <- aa.name,
   elements <- aa.getElements()
  )
}
rule A2C {
 from
   aa:MMA!A (aa.isSpecial() )
 to
  ab : MMB!A(
   name <- aa.name,
   elements <- aa.getElements()
  ),
  cb : MMB!C(
    name <- aa.name + 'special',
  )
}
```

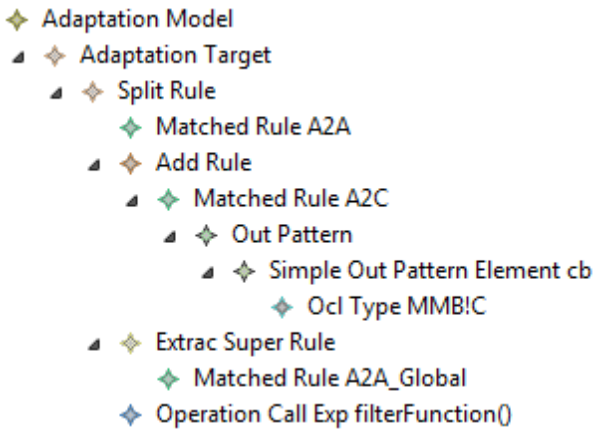
Figura 4.15 Ejemplo de una operación de adaptación que divide una regla en dos.

Tabla 4.5 Metaclase MMAdaptationGoal!SplitRule

Metaclase *MMAdaptationGoal*!SplitRule	**Modelo ejemplo de una división de regla**
SplitRule -> RuleOperation RuleOperation affectedRule : Rule newRule : AddRule superRule : ExtracSuperRule filter : OclExpression ExtracSuperRule -> RuleOperation RuleOperation superRule : Rule	Adaptation Model Adaptation Target Split Rule Matched Rule A2A Add Rule Matched Rule A2C Out Pattern Simple Out Pattern Element cb Ocl Type MMB!C Operation Call Exp filterFunction()

En estas situaciones, con el objetivo de mejorar la calidad del código resultante, se aplica una operación de extracción de super regla en el caso de que las dos reglas de transformación contengan parte del mapeo común. En la tabla 4.6 se puede ver el efecto de aplicar una extracción de super regla en una operación de división de regla.

Tabla 4.6 Ejemplo de división de regla con extracción de súper regla.

Código ATL división y extracción de regla	Modelo de división de regla con extracción de super regla
abstract rule A2A_Global{ **from** aa:MMA!A **to** ab : MMB!A(name <- aa.name, elements <- aa.getElements()) } **rule A2A extends A2A_Global{** **from** aa:MMA!A (**not aa.isSpecial()**) } **rule A2C extends A2A_Global {** **from** aa:MMA!A (aa.isSpecial()) **to** ab : MMB!A(name <- aa.name, elements <- aa.getElements()), cb : MMB!C(name <- aa.name + 'special',) }	

4.3 Deducción de operaciones de adaptación a implementar en las transformaciones M2M

En este apartado se realiza una introducción al algoritmo implementado para la deducción de las operaciones de adaptación. En el capítulo 5 se realiza una descripción detallada del algoritmo. El objetivo de este trabajo es deducir automáticamente las operaciones de adaptación que se deben de implementar en una transformación M2M legada para responder a nuevas necesidades de mapeo. El primer paso para comenzar el proceso de adaptación es especificar el nuevo requisito de mapeo. En el método definido, la especificación del requisito de mapeo se realiza mediante una pareja de modelos ejemplos de entrada/salida que representa el nuevo requisito de transformación. La razón principal para el uso de modelos ejemplo en la especificación se basa en el principio de que utilizar conceptos de modelos es más fácil que utilizar conceptos de metamodelos. Además los escenarios de adaptación a resolver en este trabajo son aquellos donde no existen cambios en los metamodelos, con la excepción de la aplicación de perfiles. El desarrollador de la transformación M2M utiliza modelos ejemplos para especificar el nuevo requisito de transformación relacionado con la evolución de algún RNF de dominio. Una vez se han definido los modelos ejemplo la herramienta debe de deducir las operaciones de adaptación necesarias para que la regla de transformación responda correctamente al nuevo requisito de transformación, ver figura 4.16. En este apartado se describe el algoritmo desarrollado para la deducción automática de las operaciones de adaptación utilizando modelos ejemplo.

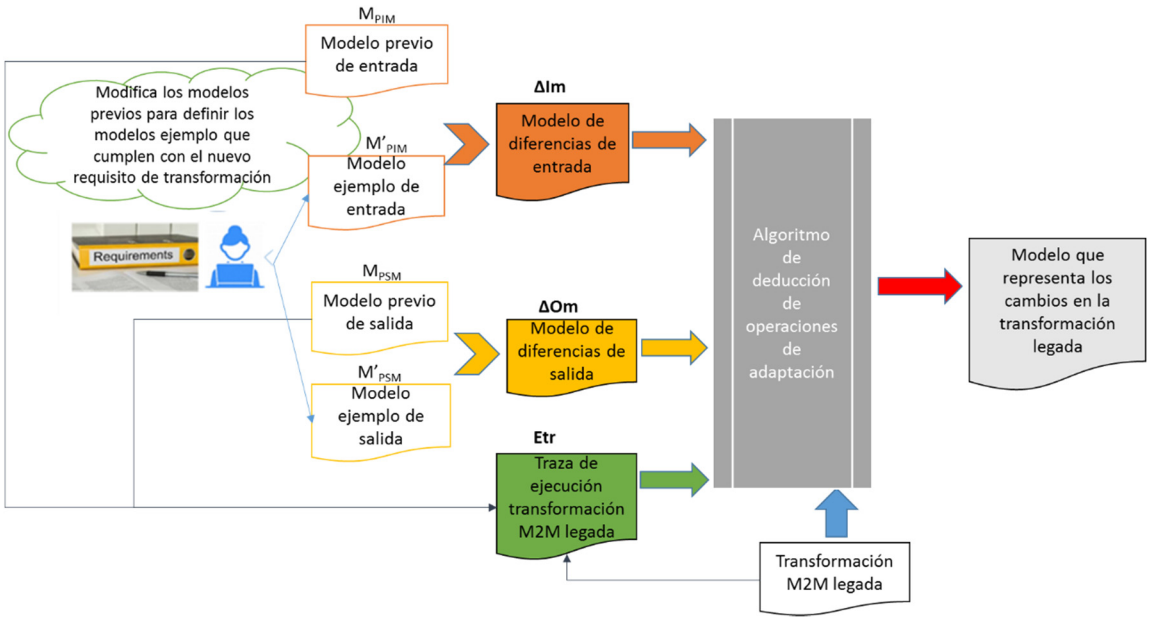


Figura 4.16 Perspectiva global del método TRANSEVOL.

Para definir los modelos ejemplo el desarrollador modifica manualmente un modelo de entrada y su correspondiente modelo de salida utilizados previamente por la transformación M2M legada. La herramienta utiliza las diferencias entre los modelos ejemplos y los modelos previos para determinar las operaciones de adaptación a implementar en la transformación M2M. Los modelos de diferencias son generados mediante la herramienta *EMFCompare* [Bru08].

EMFCompare utiliza el metamodelo *EMFDiff* [Tou06] para expresar las diferencias. El modelo de diferencias entre el modelo de salida ejemplo (M'_{PSM}) y el modelo de salida generado por la transformación legada (M_{PIM}) se denomina **ΔOm** ($M'_{PSM} - M_{PSM}$). El modelo de diferencias entre el modelo de entrada ejemplo (M'_{PIM}) y el modelo de entrada inicial (M_{PIM}) se denomina **ΔIm** ($M'_{PIM} - M_{PIM}$).

Los modelos ejemplo utilizados son incrementos, positivos y negativos, que demuestran el nuevo requisito de transformación basándose en modelos previos utilizados por la transformación M2M legada. El algoritmo de deducción utiliza los modelos de diferencias **ΔIm** y **ΔOm** para deducir las operaciones de adaptación necesarias a implementar. Para poder determinar correctamente el escenario de mapeo al que corresponden las diferencias creadas por los modelos ejemplo, es necesario utilizar también la cobertura de clases [Fle09].

La cobertura de clases indica las metaclases utilizadas por un modelo. El algoritmo utiliza tanto la cobertura de clases del metamodelo de entrada y del metamodelo de salida. El concepto de cobertura de clases se utiliza para saber el número de tipos de instancias que existen en un modelo de diferencias. Además, utilizando la cobertura de clases también se puede saber si en los modelos ejemplo de entrada hay instancias de un nuevo tipo previamente no contemplado. Utilizando la cobertura de clases se determina si se requiere una nueva regla, o si se requieren más de un patrón de salida en una regla. Por un lado, se utiliza el incremento de la cobertura de clases entre los modelos previos y los modelos ejemplo. El incremento de la cobertura de clases en el metamodelo de entrada debido al modelo ejemplo de entrada se denomina **ΔImc,** y se

expresa como **∆Imc =** CC(M'$_{PIM}$) – CC(M'$_{PIM}$), siendo CC() la función que calcula la cobertura de clase de un modelo. El incremento de la cobertura de clases en el metamodelo de salida debido al modelo ejemplo de salida se denomina **∆Omc**. El algoritmo también utiliza la cobertura de clases de los modelos de diferencias, tanto de entrada como salida. En la tabla 4.7 se representan los diferentes conceptos relacionados con la cobertura de clases utilizados por TRANSEVOL.

Tabla 4.7 Conceptos relacionados con la cobertura de clases utilizados en TRANSEVOL.

Concepto	Descripción
∆Imc = CC(M'$_{PIM}$) – CC(M$_{PIM}$)	Incremento de la cobertura de clases respecto al metamodelo de entrada
∆Omc = CC(M'$_{PSM}$) – CC(M$_{PSM}$)	Incremento de la cobertura de clases respecto al metamodelo de salida
CC(∆Im) = CC(M'$_{PIM}$ - M$_{PIM}$)	Cobertura de clases en el modelo de diferencias de entrada
CC(∆Om) = CC(M'$_{PSM}$ - M$_{PSM}$)	Cobertura de clases en el modelo de diferencias de salida

En TRANSEVOL los siguientes modelos son necesarios para analizar y deducir automáticamente las necesidades de implementación de un nuevo requisito de transformación: **∆Im, ∆Om, CC (∆Im)**, **CC (∆Om)**, **∆Imc** y **∆Omc.** Una vez se han obtenido los modelos de diferencias y los incrementos de cobertura de clases se determina cual es el escenario de adaptación. En la siguiente lista se enumeran los diferentes tipos de escenarios de adaptación de transformaciones contempladas por TRANSEVOL:

1. Nuevo mapeo uno a varios
2. Eliminación de patrones de salida y reglas de mapeo
3. Evolución de un mapeo uno a uno a uno a varios
4. Modificación de referencias
5. Cambios en los atributos
6. Evoluciones en mapeos varios a varios
7. Escenarios de filtrado
8. Aplicación de estereotipos y filtrado

Tras detectar el escenario de adaptación el algoritmo establece las operaciones de adaptación. El algoritmo determina las operaciones de adaptación dependiendo del escenario detectado. Tras decidir las operaciones de adaptación se deben de rellenar correctamente los datos necesarios para especificar la operación de adaptación. Utilizando únicamente las diferencias generadas y las coberturas de clase el algoritmo no puede obtener toda la información necesaria para definir completamente las operaciones de adaptación. Por ejemplo, en una operación de *MMAdaptationGoal!NewBinding*, hacen falta más datos para determinar cuál es la regla afectada. Para poder localizar las reglas afectadas el algoritmo utiliza el modelo que especifica la traza de ejecución de la transformación M2M legada con los modelos previos (M$_{PIM}$ y M$_{PSM}$), denominado **Etr**. Para determinar y especificar correctamente todos los datos relacionados con una operación de adaptación el algoritmo de deducción necesita los siguientes elementos:

- **∆Im** (Modelo de diferencias entre el modelo ejemplo de entrada y el modelo previo de entrada).

- **ΔOm** (Modelo de diferencias entre el modelo ejemplo de salida y el modelo previo de salida).
- **ΔImc** (Incremento de la cobertura de clases del metamodelo de entrada).
- **ΔOmc** (Incremento de la cobertura de clases del metamodelo de entrada).
- **CC (ΔIm)** (Cobertura de clases en el modelo de diferencias de entrada).
- **CC (ΔOm)** (Cobertura de clases en el modelo de diferencias de salida).
- **Etr** (Traza de ejecución de la transformación legada con M_{PIM} como modelo de entrada y M_{PIM} como modelo de salida generado).

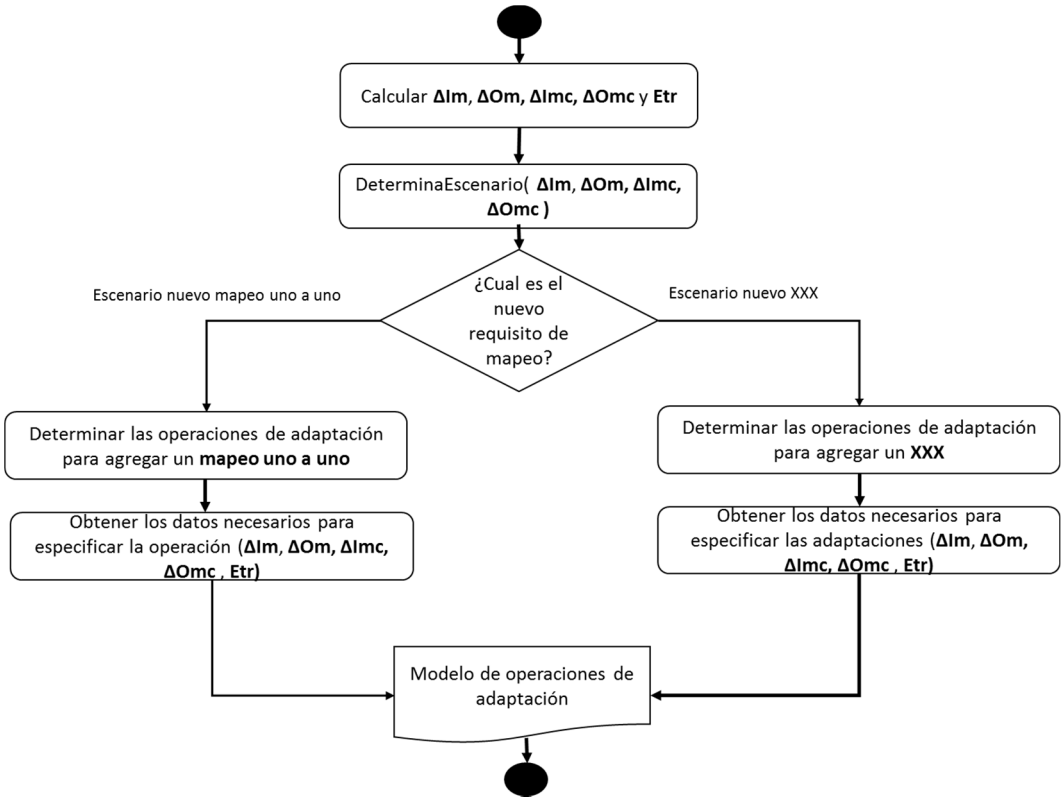


Figura 4.17 Algoritmo para la deducción de operaciones de adaptación

En la figura 4.17 se ilustra el algoritmo de deducción de operaciones de adaptación. Para comprender en detalle cómo funciona el algoritmo en los siguientes apartados de este capítulo se describe el metamodelo EMFDiff y la relación de los diferentes tipos de diferencias EMFDiff con las diferentes operaciones de adaptación. También se describe como se realiza la localización de las reglas afectadas por las operaciones de adaptación mediante las trazas de ejecución. En el capítulo 5 se detalla el algoritmo y la detección de los diferentes escenarios de evolución de transformación contemplados en este trabajo, junto con la deducción de las operaciones de adaptación para cada uno de los escenarios contemplados por TRANSEVOL.

4.3.1 Metamodelo EMFDiff

El algoritmo relaciona tipos de diferencias definidos mediante el metamodelo *EMFDiff* con operaciones de adaptación a aplicar sobre las reglas de transformación. En la figura 4.18 se representa parte del metamodelo *EMFDiff*. El metamodelo *EMFDiff* clasifica los cambios en base al

elemento afectado. Las metaclases del tipo *ModelElement* hacen referencia a cambios que afectan a los elementos de los modelos, sin contemplar cambios en sus atributos. Las diferencias del tipo *ModelElement* engloban adiciones de elementos no presentes en el modelo inicial, eliminaciones de algunos de los elementos existentes del modelo inicial y reubicaciones de los elementos del modelo inicial. La metaclase *ModelElementChangeLeftTarger* representa la adición de un elemento, mientras que la metaclase *ModelElementChangeRightTarget* especifica la eliminación de un elemento. La reubicación de un elemento se define mediante el tipo *MoveModelElement*. Los cambios en los atributos de los elementos se agrupan bajo la metaclase *AttributeChange*. Existen 4 tipos de diferencias respecto a los atributos. La adición de atributo se denomina *AttributeChangeLeftTarget*. La eliminación de un atributo se denomina *AttributeChangeRightTarget*. La modificación de valores de atributo se representa mediante el tipo *UpdateAttribute*. También se detecta el cambio del orden en los atributos mediante la metaclase *AttributeOrderChange*. Los cambios en las referencias entre los elementos no se clasifican como cambios de atributos y se agrupan bajo el tipo *ReferenceChange*. Para las referencias también existen adiciones (*ReferenceChangeLeftTarget*), eliminaciones (*ReferenceChangeRightTarget*), modificaciones (*UpdateReference*) y cambios en el orden (*ReferenceOrderChange*).

En cada tipo de diferencia *EMFDiff* las diferentes propiedades se definen mediante atributos del tipo *EObject* y *EJavaObject* del metamodelo EMF. Con la información de estos atributos se puede obtener las instancias afectadas por el cambio y sus tipos. En la tabla 4.8 se presentan las clases del metamodelo *EMFDiff* para la adición de un elemento, la eliminación de un elemento, el cambio de una propiedad y el cambio de una referencia. El algoritmo de deducción desarrollado relaciona las diferencias entre los modelos con operaciones de adaptación a implementar.

En la tabla 4.9 se relacionan los diferentes tipos de diferencias del metamodelo EMFDiff con posibles operaciones de adaptación del metamodelo *MMAdaptationGoal*. Esta relación está realizada en base a las diferencias del modelo Δ**Om**. Como se puede apreciar en la tabla para ciertos tipos de diferencias la operación de adaptación no es siempre la misma, puede variar ligeramente. Para decidir qué operación de adaptación es la correcta para cada diferencia se debe de determinar el escenario de mapeo correctamente. Para ello, como se ha dicho previamente, es necesario combinar la información de los modelos Δ**Om, ΔIm, CC (ΔIm) , CC (ΔOm) ΔOmc** y Δ**Imc.** También puede ocurrir que al crear los modelos ejemplo realmente se esté especificando una evolución regular de modelos que no requiere ninguna modificación en la transformación. Estas situaciones se tienen que detectar para no crear operaciones de adaptación cuando estas no son requeridas, o para detectar una pareja de modelos ejemplo que no especifican ninguna modificación en la lógica de mapeo.

Figura 4.18 Diagrama metamodelo EMFDiff.

Tabla 4.8 Metaclases del metamodelo EMFDiff para expresar la adición de un elemento, la eliminación de un elemento, el cambio de una propiedad y el cambio de una referencia.

Metaclase del metamodelo EMFDiff
ModelElementChangeLeftTarget -> ModelElementChange GenModel rightParent : EObject leftElement : EObject
ModelElementChangeRightTarget -> ModelElementChange GenModel leftParent : EObject rightElement : EObject
AttributeChangeLeftTarget -> AttributeChange GenModel leftTarget : EJavaObject
UpdateReference -> ReferenceChange GenModel leftTarget : EObject rightTarget : EObject

Tabla 4.9 Relación entre metaclases del metamodelo EMFDiff y operaciones de adaptación del metamodelo MMAdaptationGoal.

Tipo de diferencia EMFDiff	Descripción de la diferencia EMFDiff	Operación de adaptación
ModelElement ChangeLeftTarget	Agregación de un elemento	Nueva regla
		Nuevo patrón de salida
		División de regla
		Nueva regla con filtro
ModelElement ChangeRightTarget	Eliminación de un elemento	Filtro en patrón de entrada
		Eliminación de regla
		Eliminación de patrón de salida
MoveModelElement	Modificación del contenedor de un elemento	Cambio de asignación
		Filtro en el patrón de entrada
ReferenceChange LeftTarget	Agregación de una nueva referencia	Nueva asignación
		Filtro en el patrón de entrada
UpdateReference	Modificación de una referencia	Modificación de asignación
		Filtro en el patrón de entrada
AttributeChange LeftTarget	Agregación de un atributo	Nueva asignación
		Filtro en el patrón de entrada
UpdateAttribute	Modification of an attribute value	Modificación de asignación
		Filtro en el patrón de entrada
		Nuevo patrón de salida

En la tabla 4.10 se presentan los modelos de diferencias de un escenario ejemplo de evolución de transformación M2M. En la tabla se representan los modelos de entrada y salida previos, utilizados por la transformación M2M legada. También se pueden ver los modelos ejemplo utilizados para especificar un nuevo requisito de transformación. En la última línea de la tabla se puede ver el resultado de aplicar la herramienta *EMFCompare* tanto a los modelos de entrada como a los de salida. En este caso la evolución de la transformación implica mapear los elementos del tipo *MMTree!Leaf* de los modelos de entrada en elementos del salida del tipo *MMElementList!Elements*, es decir una nueva regla de transformación es requerida. Para especificar este nuevo requisito de transformación incrementando los modelos previos se realizan agregaciones de nuevas instancias de dichos tipos. Debido a esta nueva necesidad, en el nuevo modelo de entrada aparecerán nuevas instancias del tipo *MMTree!Leaf* y en el nuevo modelo de salida aparecerán nuevas instancias del tipo *MMElementList!Elements*. En este caso concreto se han agregado tres nuevos elementos del tipo *MMElementList!Elements*. Esta circunstancia se expresa mediante la existencia de tres tipos de diferencia del tipo *MMDiff!ModelElementChangeLeftTarget*. En la figura 4.19 se puede ver el fichero XMI correspondiente al modelo de diferencias de salida de este ejemplo. En este tipo de diferencias se especifica cual es el nuevo elemento agregado mediante la propiedad del tipo *EObject* denominada *leftElement*. El elemento contenedor donde se ha agregado la nueva instancia se especifica mediante la propiedad *rightParent* del

tipo EObject. Mediante estas dos propiedades podemos conocer cuál es el patrón de salida de la nueva regla a implementar y podemos conocer la instancia del elemento contenedor. Si se conoce la instancia del elemento contenedor y se dispone de la traza de ejecución de la transformación M2M legada relacionada con estos modelos previos de entrada y salida, también se puede localizar la regla afectada por la nueva operación de. Este proceso de localización se explica en detalle en el apartado 4.3.2. Mediante las diferencias *MMDiff!ModelElementChangeLeftTarget* del modelo **ΔIm** se obtiene el patrón del elemento de entrada para la nueva regla. Con este pequeño ejemplo se ha demostrado como se puede obtener la información de las nuevas operaciones de adaptación de los modelos de diferencias.

```
<diff>
  <ownedElements xsi:type="diff:DiffGroup">
    <subDiffElements xsi:type="diff:DiffGroup">
      <subDiffElements xsi:type="diff:ModelElementChangeLeftTarget">
        <rightParent
          href="platform:/One_to_One_mapping/it1.mmelementlist#/"/>
        <leftElement
          href="platform:/One_to_One_mapping/It2.mmelementlist#//@elements.0"/>
      </subDiffElements>
      <subDiffElements xsi:type="diff:ModelElementChangeLeftTarget">
      <subDiffElements xsi:type="diff:ModelElementChangeLeftTarget">
      <rightParent href="platform:/One_to_One_mapping/it1.mmelementlist#/"/>
    </subDiffElements>
  </ownedElements>
  <leftRoots href="platform:/One_to_One_mapping/It2.mmelementlist#/"/>
  <rightRoots href="platform:/One_to_One_mapping/it1.mmelementlist#/"/>
</diff>
```

Figura 4.19 Ejemplo de un fichero XMI de un modelo de diferencias ΔOm.

Tabla 4.10 Ejemplo de los modelos previos, los modelos ejemplo y los modelos de diferencias de un escenario de evolución de transformación M2M.

	Modelos de entrada	**Modelos de salida**
Modelos previos	Node A Node A_1 Node A_2	Root Element A
Modelos ejemplo	Node A Node A_1 Leaf B_1 Node A_2 Leaf B_2 Leaf B_3	Root Element A Common Element B_1 Common Element B_2 Common Element B_3
Modelos de diferencias	3 change(s) in model 3 change(s) in Node A 1 change(s) in Node A_1 Leaf B_1 has been added 2 change(s) in Node A_2 Leaf B_2 has been added Leaf B_3 has been added Node A / Node A_1 / Leaf B_1 / Node A_2 / Leaf B_2 / Leaf B_3 Node A / Node A_1 / Node A_2	3 change(s) in model 3 change(s) in Root Element A Common Element B_1 has been added Common Element B_2 has been added Common Element B_3 has been added Root Element A / Common Element B_1 / Common Element B_2 / Common Element B_3 Root Element A

4.3.2 Localización de las operaciones de modificación en la transformación M2M legada

En este apartado se describe como se utilizan las trazas de ejecución de las transformaciones legadas para localizar las reglas de transformación afectadas por los nuevos requisitos de transformación. Para obtener la información de las reglas de transformación afectadas se combina la información de los modelos de diferencia **ΔOm** con la traza de ejecución de la transformación M2M, denominado **Etr**. La traza **Etr** relaciona cada instancia de salida, con su instancia correspondiente del modelo de entrada y con la regla de transformación que realizó el mapeo. El objetivo del modelo **Etr** es relacionar las reglas de transformación con las instancias del modelo de entrada previo ($\mathbf{M_{PIM}}$) y con las instancias del modelo de salida previo ($\mathbf{M_{PSM}}$). Para obtener el modelo **Etr** el HOT ATL2Trace [Jou05] es aplicado a la transformación legada. El resultado de aplicar el HOT es un refinamiento de las reglas de transformación ATL. Al ejecutar estas nuevas transformaciones refinadas con el modelo de entrada $\mathbf{M_{PIM}}$, además de generar su correspondiente modelo de salida, $\mathbf{M_{PSM}}$, también se genera el modelo de traza de ejecución **Etr,** ver figura 4.20.

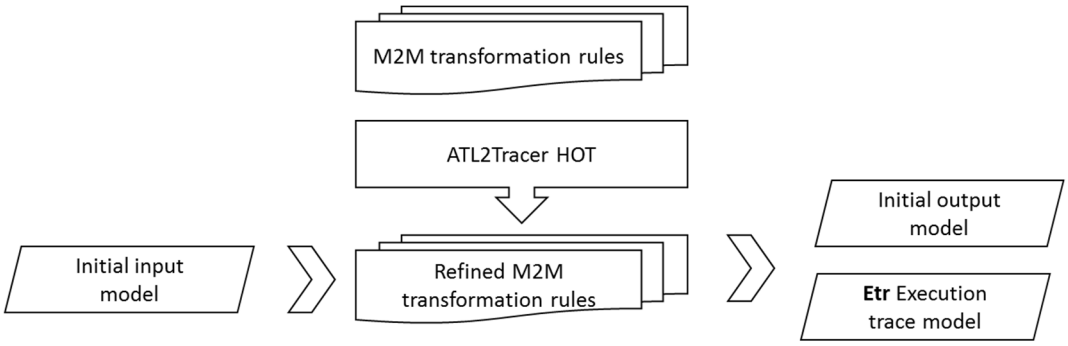


Figura 4.20 Proceso para obtener la traza de ejecución de una transformación ATL [Jou05].

El modelo **Etr** se representa conforme al metamodelo *Trace,* ver figura 4.21. El metamodelo *Trace* permite crear elementos del tipo *Trace!TraceLink* que permite guardar y relacionar el nombre de las instancias de salida generados, su correspondiente elemento del modelo de entrada y la regla que generó el mapeo. El algoritmo utiliza elementos *Trace!TraceLink* del modelo **Etr** para localizar las reglas que generaron una instancia concreta del modelo de salida relacionado con alguna de las diferencias *EMFDiff* del modelo **ΔOm**. La información del **Etr** también es utilizada para relacionar diferencias del **ΔIm** con las reglas de transformación legadas.

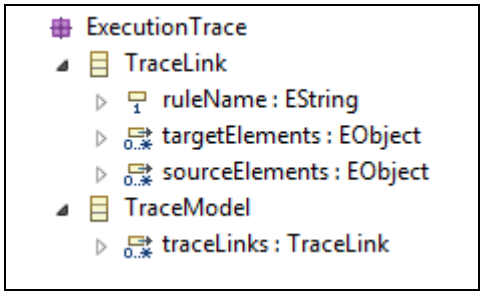


Figura 4.21 Metamodelo Trace [Jou05].

Para explicar el funcionamiento del algoritmo de localización se utiliza el ejemplo presentado en la tabla 4.10. En el ejemplo se requiere una nueva regla de transformación que mapee los elementos del tipo *MMTree!Leaf* de los modelos de entrada en elementos del salida del tipo *MMElementList!Elements*. Para definir una operación del tipo *MMAdaptationGoal!NewRule* se debe de crear una nueva regla y una nueva asignación en una regla existente de la transformación legada. En este caso es necesaria la identificación y localización de la regla afectada por la nueva asignación. En el escenario del ejemplo se selecciona una de las diferencias del tipo *MMDiff!ModelElementChangeLeftTarget* del modelo **ΔOm**. Con una de las tres diferencias es suficiente, debido a que las tres están relacionadas con la misma operación de adaptación. Para poder obtener la regla donde se debe de localizar la nueva asignación es necesario conocer la instancia contenedor del nuevo elemento de salida que ha generado la diferencia *MMDiff!ModelElementChangeLeftTarget*. Para ello se analiza el elemento *EObject* de la propiedad rightParent de la diferencia *MMDiff!ModelElementChangeLeftTarget*. Una vez conocida la instancia contenedor se busca en el modelo **Etr** el *Trace!TraceLink* que indica cual es la regla que generó dicha instancia. En la figura 4.22 se resume como se realiza la búsqueda de una regla afectada partiendo de una diferencia del tipo diferencia *MMDiff!ModelElementChangeLeftTarget*. El algoritmo de localización de las reglas legadas es prácticamente el mismo para todos los escenarios de adaptación. En algunos escenarios de adaptación es la propia instancia de salida el punto inicial para comenzar la búsqueda, mientras que en otros escenarios es el elemento contenedor el punto de partida. Una vez identificada la regla afectada el siguiente paso es rellenar todos los campos de del elemento *MMAdaptationGoal!AddRule* con la información necesaria: patrón de entrada de la regla, patrón de salida de la regla, la expresión de asignación, el mapeo de los valores de los atributos y el nombre de la regla afectada por el *binding.* En el siguiente apartado se analiza cómo se recolectan los datos necesarios para la especificación de las diferentes operaciones de adaptación para los diferentes escenarios de evolución contemplados en este trabajo.

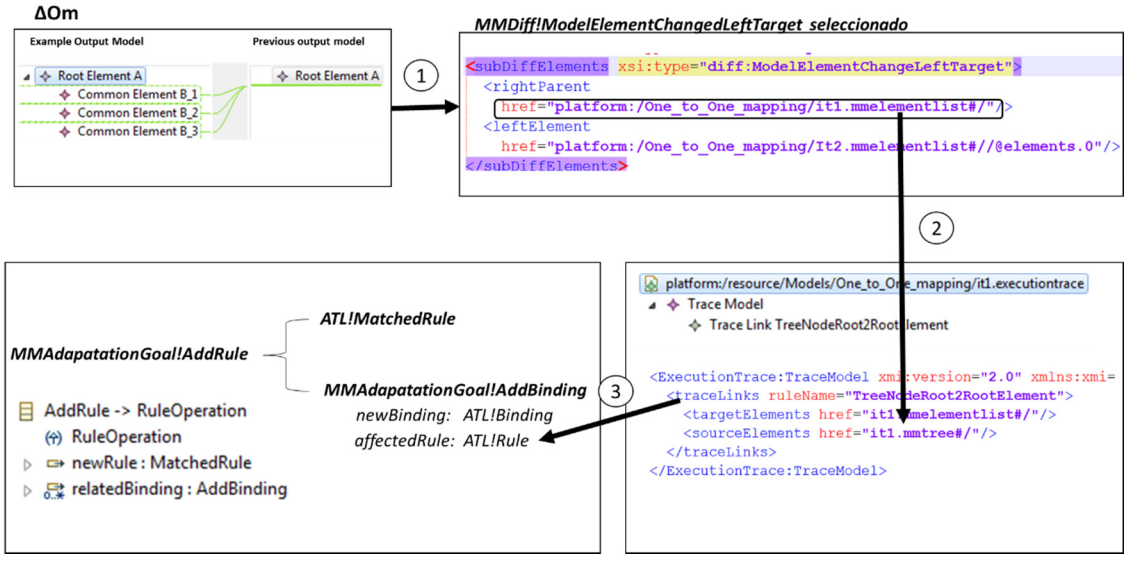


Figura 4.22 Búsqueda de una regla afectada por una nueva asignación (Binding).

4.4 Escenarios de adaptación de transformaciones M2M

En este apartado se listan los escenarios de adaptación de transformaciones M2M contemplados por TRANSEVOL. En el capítulo 5 se describe en detalle la detección de cada escenario de adaptación y la generación de las operaciones de adaptación para cada una de ellas. El método TRANSEVOL deduce las operaciones de adaptación para los siguientes escenarios de evolución de requisitos de mapeo:

1. Nuevo mapeo uno a uno.
2. Nuevo mapeo uno a varios.
3. Evolución de un mapeo uno a uno a un mapeo uno a varios.
4. Eliminación de patrones de salida
5. Eliminación de regla de mapeo.
6. Escenario de filtrado mediante eliminación de instancias.
7. Filtrado mediante agregación de instancias en el modelo de entrada.
8. Nueva regla de transformación con filtrado mediante agregación de instancias.
9. Modificación de contenedor.
10. Modificación de asignaciones debido a cambios en los atributos de salida.
11. Evolución de un mapeo 1 a N a 2 a N.
12. Detección de un nuevo mapeo N a 1.
13. Modificación de asignaciones debido al uso de estereotipos.
14. División de una regla en dos debido a la aplicación de estereotipos y la modificación de atributos de salida.
15. Evolución de filtrado de mapeo debido a la aplicación de estereotipos.

4.5 Evoluciones regulares

Las evoluciones regulares son aquellas situaciones donde se modifica el modelo de entrada y sin modificar la transformación M2M se genera correctamente el modelo de salida. Al adaptar una transformación M2M legada puede ocurrir que el desarrollador especifique una pareja de modelos ejemplo que realmente no requiere que se haga ninguna modificación en la transformación. El método TRANSEVOL no presenta ningún modelo para la detección de estas situaciones. Para evitar los falsos positivos antes de ejecutar el método TRANSEVOL se debe de ejecutar la transformación legada con el modelo ejemplo de entrada y comprobar si el modelo de salida generado coincide con el modelo de salida ejemplo. En caso de que sean iguales quiere decir que la pareja de modelos ejemplo está especificando una evolución regular y por lo tanto no se requiere adaptar la transformación.

4.6 Ejemplo de uso de la herramienta TRANSEVOL

Para validar el método desarrollado se ha implementado un prototipo de herramienta. Actualmente existe una primera versión de la herramienta TRANSEVOL implementada en JAVA utilizando el framework EMF. La herramienta utiliza la versión de EMFDiff 2.1 para el cálculo de diferencias y se basa en el modelo ATL 3 para la definición del metamodelo *MMAdaptationGoal*, que permite expresar las operaciones de adaptación a realizar. En este apartado se explica el uso de la metodología TRANSEVOL mediante la aplicación de la herramienta prototipo en un pequeño y sencillo ejemplo. La transformación modelo a modelo que vamos a utilizar como ejemplo es la transformación ATL Tree2Node((*http://www.eclipse.org/atl/documentation/basicExamples_Patterns/*)). El objetivo del ejemplo es conocer el proceso y flujo de datos utilizado por la herramienta cuando queremos adaptar una transformación legada a un nuevo requisito de transformación. Antes de comenzar con el proceso de adaptación de la transformación legada se explica de forma global el ejemplo *Tree2Node*. Para ello, primero se describen los metamodelos origen y destino de la transformación y se explica el objetivo global de la transformación. El objetivo de la transformación se descompone en dos requisitos. El primer requisito lo implementa la transformación legada y el segundo requisito será el que hay que añadir a la transformación legada, es decir, el segundo requisito representa el escenario de evolución. La implementación del segundo requisito de transformación es el utilizado para describir el funcionamiento y la base de la metodología desarrollada. Para adaptar la transformación legada M2M primero se crean los modelos ejemplo que representan el nuevo requisito de transformación. Después se obtienen los modelos de diferencias y la traza de ejecución. Y finalmente se obtienen las operaciones de modificación mediante la herramienta. En este ejemplo los nuevos requisitos no están relacionados con requisitos no funcionales de dominio con el objetivo de simplificar la descripción. El objetivo de esta capitulo es explicar el uso del método de una manera sencilla, y por ello se ha seleccionado un ejemplo lo más sencillo posible para entender el flujo y funcionamiento de la herramienta desarrollada.

4.6.1 Transformación Tree2Node

El objetivo de esta transformación es convertir modelos de árboles en listas. La figura 4.23 representa un ejemplo de una transformación de un árbol a una lista.

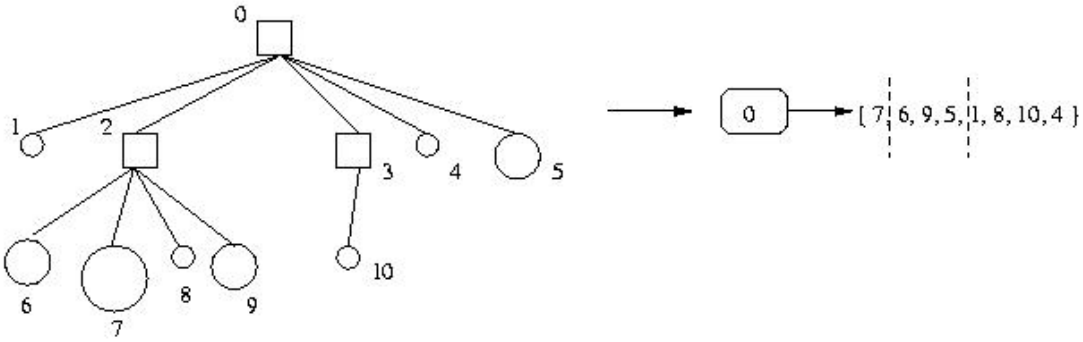


Figura 4.23 Ejemplo de un modelo de árbol y su correspondiente modelo de lista.

En la figura 4.24 se pueden apreciar los metamodelos que definen los árboles y listas de este ejemplo. El metamodelo de árboles se denomina *MMTree* y el de la lista se denomina *MMElementList* Los modelos de árbol se definen mediante nodos y hojas. Y las hojas pueden ser de tres tipos: pequeños, medianos y grandes. Las listas están compuestas por un elemento raíz que contiene elementos comunes.

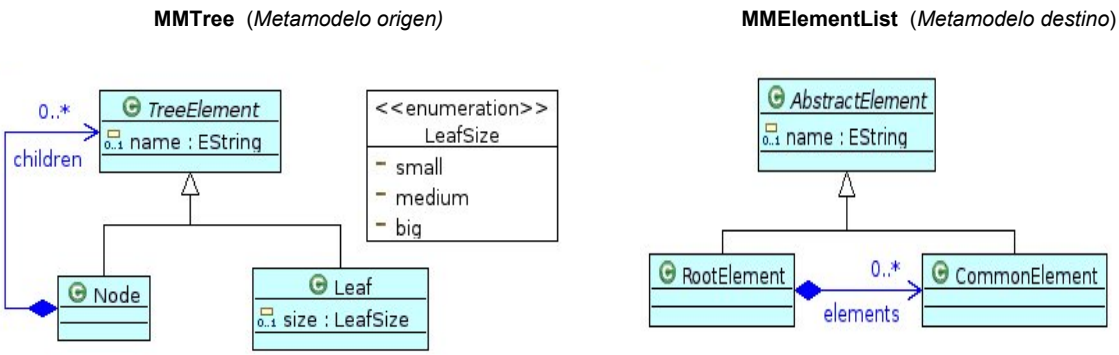


Figura 4.24 Metamodelo para la definición de árboles *MMTree* y metamodelo para la definición de listas *MMElementList*.

Los requisitos de mapeo de la transformación son los siguientes:

1. El nodo raíz del árbol, del tipo *MMtree!Node*, debe de ser transformado en el elemento raíz de la lista de elementos del tipo *MMElementList!RootElement*.
2. Un elemento del tipo hoja, *MMTree!Leaf*, debe de transformarse en un elemento común del tipo *MMElementList!CommonElement*.
3. La lista de elementos deberá estar ordenada de la siguiente forma
    a. Las hojas grandes deberán de ubicarse en el inicio de la lista.
    b. Después deberán de ubicarse los de tamaño medio y finalmente los pequeños.

La regla de trasformación ATL de la figura 4.25 implementa el primer requisito de mapeo. Esta regla de transformación convierte el nodo raíz del árbol en el elemento raíz de la lista. Y los nodos que no son raíz no se mapean a ningún elemento de salida, se filtran. En este ejemplo esta regla de transformación conforma la transformación legada M2M y el segundo requisito de mapeo ejemplifica el requisito de evolución a implementar sobre la transformación legada.

```
rule TreeNodeRoot2RootElement {
  from
    rt : MMTree!Node (rt.isTreeNodeRoot())
  to
    lstRt : MMElementList!RootElement (
    name <- rt.name
  )
}
```

Figura 4.25 Regla de transformación ATL que convierte el nodo raíz de un árbol en el elemento raíz de una lista.

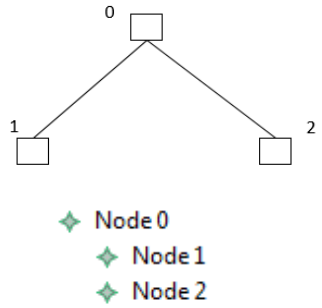
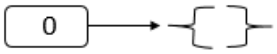
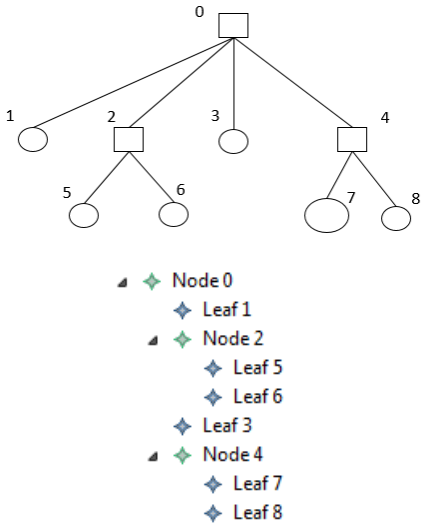
Para validar esta transformación se han definido dos modelos de entrada diferentes, ver tabla 4.11. En el primer modelo de entrada solo tenemos nodos, no existen hojas. Mientras que en el segundo modelo tenemos nodos y hojas. Al aplicar la transformación legada actual se obtiene el mismo modelo de salida, donde solo existe el elemento raíz de la lista. Esto es debido a que aún no se ha implementado el siguiente requisito.

Una vez el primer requisito esta implementado se considera la transformación legada. El nuevo requisito de mapeo requiere convertir una hoja del árbol en un elemento de la lista. Este nuevo requisito de transformación representa el escenario de evolución. El nuevo requisito de transformación es un mapeo de uno a uno, las instancias del tipo hojas, *MMTree!Leaf*, de los modelos de árbol se deben de convertir en instancias del tipo *MMElementList!CommonElement* de los modelos lista. Para obtener la nueva transformación se deben de implementar las siguientes adaptaciones en la transformación legada:

1. Implementar la regla de transformación que convierte *MMTree!Leaf* en *MMElementList!CommonElement*. La nueva regla de transformación se denomina *Leaf2CommonElement* y su código se puede ver en la figura 4.26.
2. Relacionar los elementos comunes de la lista creados mediante la regla *Leaf2CommonElement* con el elemento raíz generada mediante la regla de transformación *TreeNodeRoot2RootElement*. Para ello se debe de agregar una asignación a la propiedad *elements* del elemento *MMElementList!RootElement* en la regla de transformación *TreeNodeRoot2RootElement*.

La metodología y herramienta TRANSEVOL desarrollado en este trabajo permite automatizar este trabajo de mantenimiento de las reglas de transformación. Para ello, la herramienta necesita como entradas los modelos ejemplo que representan el requisito de transformación y la traza de ejecución de la transformación legada. A continuación, se detalla el proceso de adaptación para implementar la nueva regla de transformación en el ejemplo *Tree2Node* mediante la herramienta TRANSEVOL. Al aplicar la herramienta TRANSEVOL se deducen automáticamente los cambios a implementar en la regla de transformación legada. En este ejemplo como resultado se deberán de obtener la especificación de una nueva regla de transformación y una nueva asignación.

Tabla 4.11 Dos modelos de árboles y su correspondiente modelo de salida tras aplicar la regla de transformación de la figura 4.26.

Modelos de árbol de entrada	Modelos lista tras aplicar la transformación
	
	

```
rule TreeNodeRoot2RootElement {
  from
    rt : MMTree!Node (rt.isTreeNodeRoot() )
  to
    lstRt : MMElementList!RootElement (
      name <- rt.name,
      elements <- rt.getAllChildren()
    )
}
rule Leaf2CommonElement {
  from
    s : MMTree!Leaf
  to
    t : MMElementList!CommonElement(
      name <- s.name
    )
}
```

Figura 4.26 Nueva regla de transformación y asignación que convierte *MMTree!Leaf* en *MMElementList!CommonElement*.

4.6.2 Definición del requisito de transformación mediante modelos ejemplo

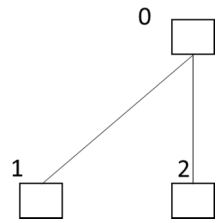
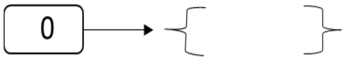
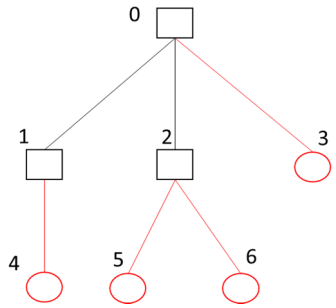
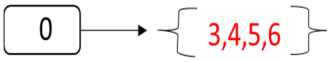
En este trabajo se ha optado por expresar los requisitos de las transformaciones M2M mediante pares de modelos ejemplo de entrada y salida, basándonos en los principios del MTBE. Para crear los modelos ejemplos se realizan pequeños incrementos a los modelos previos ya utilizados en la validación de la transformación legada, demostrando así la nueva transformación. En este ejemplo el nuevo requisito de transformación a implementar es un mapeo uno a uno entre los tipos *MMTree!Leaf* y *MMElementList!CommonElement*. Para especificar el mapeo se parte de un modelo de entrada que contiene únicamente los elementos que contemplaba previamente la transformación legada. En este caso los modelos de partida a incrementar y modificar para especificar el nuevo requisito son la primera pareja de modelos de entrada y salida de la tabla 4.12. Para poder especificar el nuevo requisito de transformación se han agregado 4 instancias del tipo *MMTree!Leaf* en el modelo de entrada y 4 instancias de tipo *MMElementList!CommonElement* en el modelo de salida. El modelo de entrada previo se denomina **it1.mmtree** y el de salida **it1.mmelementlist**. El par de modelos ejemplo de salida y entrada se denominan **it2.mmtree** y **it2.mmelementlist**. Todos los modelos están definidos mediante EMF. Una vez definidos los modelos ejemplo el siguiente paso es obtener las diferencias entre los modelos previos y los modelos ejemplo.

4.6.3 Creación de los modelos diferencia

TRANSEVOL utiliza las diferencias entre los modelos previos y los modelos ejemplos para deducir las operaciones de modificación a implementar en la transformación. Por lo tanto, la herramienta requiere dos modelos de diferencias: 1) el modelo de diferencias entre los modelos de entrada, Δ**Im,** y 2) el modelo de diferencias entre los modelos de salida, Δ**Om**. En la herramienta prototipo los metamodelos y modelos se basan en EMF y para obtener las diferencias entre modelos se ha utilizado la herramienta EMFCompare que se basa en el metamodelo EMFDiff. Aplicando la herramienta EMFCompare a los modelos previos y a los modelos ejemplos de la tabla 4.12 se obtienen las siguientes diferencias:

- Diferencias entre los modelos de entrada (*it1_it2_tree.emfdiff*): 4 diferencias de agregación. En cada una de ellas un elemento del tipo *MMTree!Leaf* ha sido añadido, ver figura 4.27.
- Diferencias entre los modelos de salida (*it1_it2_elementlist.emfdiff*): 4 diferencias de agregación. En cada una de ellas un elemento del tipo *MMElementList!CommonElement* ha sido añadido, ver figura 4.28.

Tabla 4.12 Pareja de modelos previos y pareja de modelos ejemplo utilizados para especificar la nueva regla de transformación *Leaf2CommonElement*.

Cada MMTree!Leaf se convierte en un MMElementList!CommonElement	**Modelo de entrada**	**Modelo de salida**
Modelos previos	**it1.mmtree** 	**it1.mmelementlist** 
Nuevos modelo ejemplo	**it2.mmtree** 	**it2.mmelementlist** 

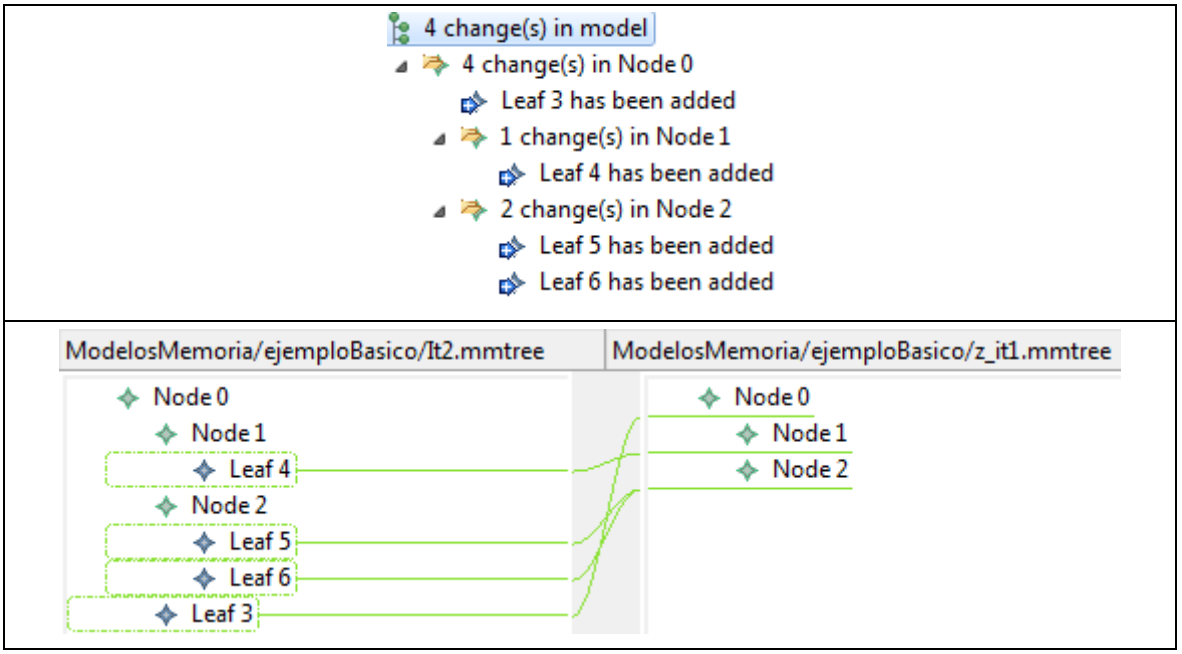


Figura 4.27 Modelo de diferencias *it1_it2_tree.emfdiff*.

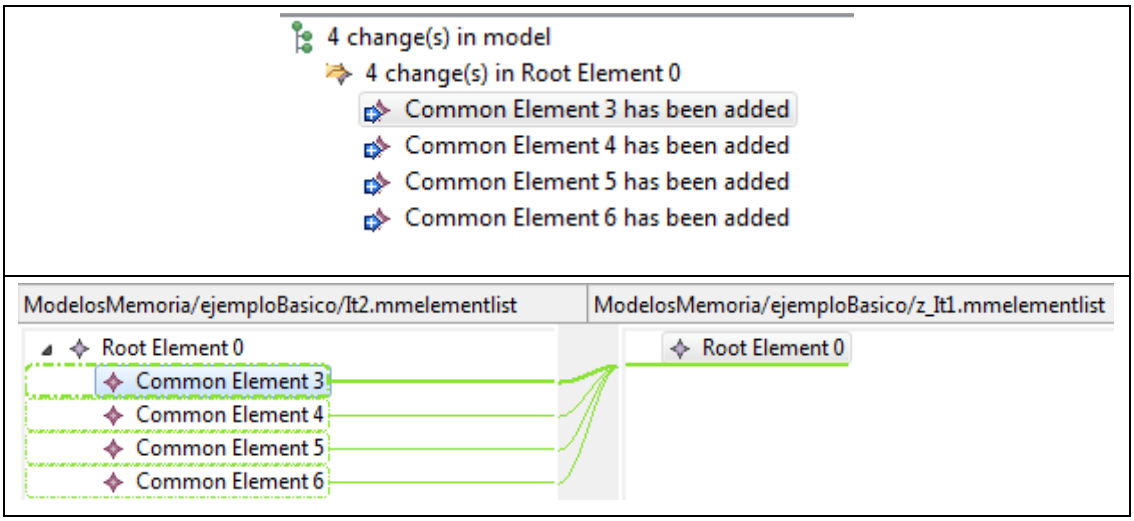


Figura 4.28 Modelo de diferencias *it1_it2_elementlist.emfdiff*.

Analizando las diferencias se puede deducir que por cada elemento del modelo de entrada del tipo hoja *MMTree!Leaf* se genera en el modelo de salida un elemento común de la lista *MMElementList!CommonElement*. Además los nuevos elementos de lista se han agregado al elemento raíz de la lista RootElement *0*. TRANSEVOL detecta que las diferencias generadas tanto en Δ**Im** y Δ**Om** son agregaciones de elementos, siendo el número de diferencias en ambos igual. Además Δ**Imc=1** y CC(Δ**Omc**)=1. En este caso los datos del escenario especificado coinciden con los datos de la tabla 4.12, que indica que es un nuevo mapeo uno a uno. Por lo tanto la adaptación de la transformación requiere una nueva regla y una nueva asignación.

Para implementar la nueva regla se necesita saber el tipo del elemento origen y el tipo del elemento destino. La información que contiene el modelo de diferencias es suficiente para conocer el elemento origen y destino de la nueva regla de transformación. Además de la nueva regla de transformación también se requiere agregar una operación de asignación en la regla de transformación legada. Para cumplir con el nuevo requisito de transformación se deberá asociar a la propiedad *elements* del *MMElementList!RootElement* los *MMElementList!CommonElement* creados partir de las hojas. Mediante el modelo de diferencias se puede obtener el elemento de salida afectado, en este caso el *RootElement*, pero no se puede saber la regla afectada por el binding. Para conocer que regla de transformación afectada por la asignación es necesario conocer la traza de ejecución de la transformación legada. Por lo tanto el siguiente paso es conocer la traza de ejecución de la transformación legada.

4.6.4 Obtención de la traza de ejecución de la transformación legada

Una vez obtenido los modelos de diferencia TRANSEVOL ha detectado un nuevo mapeo uno a uno y por consiguiente la necesidad de crear una nueva regla de transformación y una asignación. El nuevo binding es necesario para relacionar los nuevos elementos de la lista con el elemento raíz de la lista. Para ello se debe de ubicar el binding en la regla de transformación que género previamente el elemento raíz de la lista. Para obtener esta información es necesario tener

la traza entre los elementos de entrada, los elementos de salida generados y las reglas de transformación. El lenguaje de transformación ATL y su máquina virtual actualmente no ofrecen como posible salida los datos relacionados con la ejecución. Por ello, para conocer la traza de ejecución de las transformaciones ATL necesitamos instrumentar automáticamente las reglas de transformación. Para realizar esta instrumentación se ha utilizado el HOT ATL2Tracer [Jou05] que permite instrumentar las transformaciones ATL para que la transformación, además de generar el modelo de salida, genere un modelo que representa la traza de ejecución. De esta manera al ejecutar la transformación además de generar el modelo de salida también se genera un modelo donde cada elemento de salida generado es asociado a su correspondiente elemento de entrada y la regla de transformación responsable de dicho mapeo.

En la figura de 4.29 se puede ver la traza que especifica que regla generó la instancia *RootElement0*. La instancia de salida *RootElement0* es el elemento afectado por la nueva asignación. La traza indica que la regla que lo generó es la regla *TreeNode2RootElement*. Por lo tanto, esta es la regla afectada por la nueva asignación. TRANSEVOL encuentra esta información utilizando su algoritmo de búsqueda, la cual combina la traza de ejecución con las diferencias.

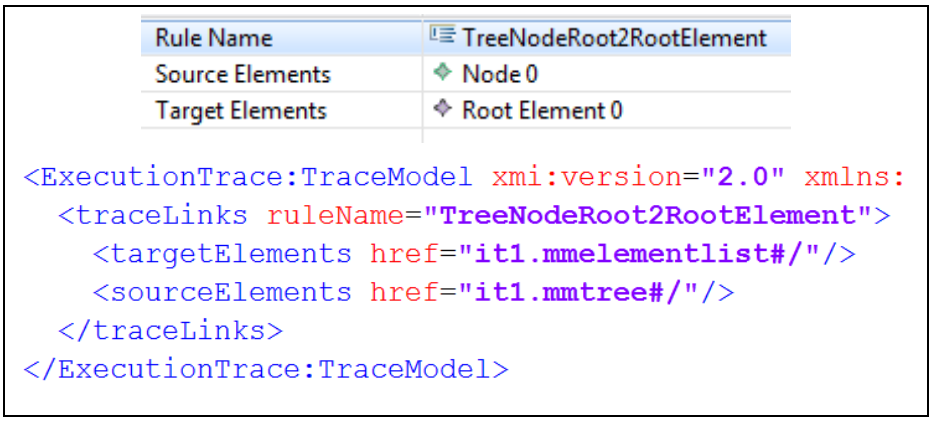


Figura 4.29 Traza de ejecución de la transformación Tree2Node entre el modelo previo de entrada it1.mmtree, el modelo previo de salida it1_out.mmelementlits y la regla de transformación M2M legada.

4.6.5 Modelo de operaciones de adaptación

Cuando se ejecuta la herramienta TRANSEVOL esta genera como salida un modelo de operaciones de adaptación donde se especifican los cambios a realizar en la transformación M2M legada. En la figura 4.30 está el modelo de operaciones de adaptación generado por la herramienta TRANSEVOL para el escenario de evolución del ejemplo *Tree2Node* presentando en este capítulo. En el modelo se puede ver como la herramienta ha deducido que hay que agregar una nueva regla para implementar el nuevo mapeo entre hojas del árbol y elementos comunes de la lista. Para agregar correctamente el nuevo mapeo hace falta generar una nueva regla denominada *Leaf2MMElementListCommonElement* y una asignación en la regla de transformación legada *TreeNodeRoot2RootElement*, como habíamos adelantado previamente. En la especificación de la nueva regla se define el patrón de entrada, instancias del tipo *MMTree!Leaf*, y el patrón de salida, instancias del tipo *MMElementList!CommonElement*. La nueva asignación se ubica en la regla *TreeNodeRoot2RootElement* y afecta a la propiedad *commonElement* del elemento raíz

de la lista. Utilizando esta información se pueden implementar los cambios necesarios en la transformación M2M legada del ejemplo para responder al nuevo requisito de evolución, que en este caso es convertir los elementos de tipo hoja, MMTree!Leaf, en elementos comunes del tipo *MMElementList!CommonElement*.

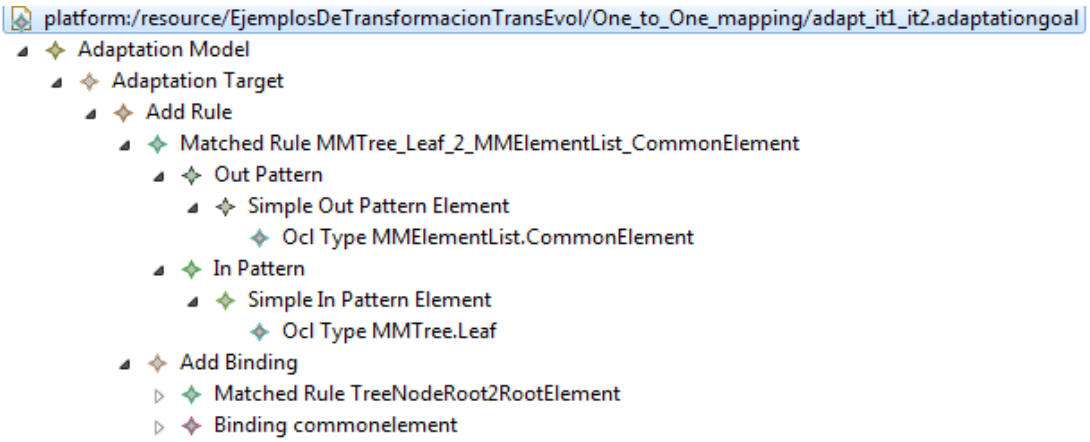


Figura 4.30 Modelo de operaciones de adaptación para el ejemplo Tree2Node.

4.6.6 Resumen

La herramienta TRANSEVOL requiere una serie de metamodelos y modelos para su ejecución. Primero requiere la especificación del nuevo requisito de transformación mediante una pareja de modelos de entrada/salida ejemplo. Para que la herramienta deduzca el tipo de mapeo especificado necesita los modelos diferencias entre los modelos ejemplos y modelos de entrada/salida previos, utilizados por la transformación legada. Por último, se requiere la traza de ejecución de la transformación legada con los modelos previos. Combinando la información de los modelos de diferencias y la traza de ejecución la herramienta genera un modelo donde se especifican los cambios a realizar sobre la transformación legada. En la siguiente lista se enumeran los modelos y metamodelos necesarios para que la herramienta se ejecute correctamente. Todos los modelos con excepción del último son modelos de entrada de la herramienta.

1. Metamodelo de entrada (*input metamodel,* $\mathbf{MM_{PIM}}$).
2. Modelo de entrada previo (*input init model,* $\mathbf{M_{PIM}}$).
3. Nuevo modelo de entrada ejemplo (*input extensión,* $\mathbf{M'_{PIM}}$).
4. Modelo de diferencias entre los modelos de entrada (*input diff model ,* $\Delta\mathbf{Im}$).
5. Metamodelo de salida (*output metamodel,* $\mathbf{MM_{PSM}}$).
6. Modelo de salida previo (*output init model,* $\mathbf{M_{PSM}}$).
7. Nuevo modelo de salida ejemplo (*output example model,* $\mathbf{M'_{PSM}}$).
8. Modelo de diferencias entre los modelos de salida (*output diff model,* $\Delta\mathbf{Om}$).
9. Modelo de la traza de ejecución de la trasformación legada con los modelos de entrada y salida previos (*execution trace*, **Etr**).
10. Modelo de la transformación ATL legada en formato EMF (*transformation model*).

11. Modelo con las operaciones de modificación sobre la transformación legada (*MMAdaptation Model,* $\mathbf{M}_{\mathbf{Adapt}}$). Esta es la salida de la herramienta.

Los nombres en ingles de la lista previa son los utilizados por la herramienta para identificar cada modelo y metamodelo de entrada. La herramienta TRANSEVOL lee de un fichero de configuración el nombre de los modelos y metamodelos a utilizar en su ejecución. La figura 4.31 representa el fichero de configuración de la herramienta TRANSEVOL para el ejemplo *Tree2Node*. El resultado se guarda en el fichero *adapt_it1_it2.adaptationgoal*.

```
input_extension=mmtree
output_extension=mmelementlist

input_metamodel= MMTree.MMTreePackage
output_metamodel= MMElementList.MMElementListPackage

transformation_model=models/One_to_One_mapping/Tree2Node.atl.ecore
execution_trace= models/One_to_One_mapping/it1.tracer
input_init_model= models/One_to_One_mapping/it1.mmtree
input_example_model= models/One_to_One_mapping/it2.mmtree
output_init_model= models/One_to_One_mapping/it1.mmelementlist
output_example_model= models/One_to_One_mapping/it2.mmelementlist
input_diffs_model= models/One_to_One_mapping/it1_it2_in.emfdiff
output_diffs_model= models/One_to_One_mapping/it1_it2_out.emfdiff
adaptation_model= models/One_to_One_mapping/adapt_it1_it2.adaptationgoal
```

Figura 4.31 Fichero de configuración **transevol.properties** de la herramienta prototipo TRANSEVOL para el ejemplo ***Tree2Node***.

5. Detección de los escenarios de adaptación de las transformaciones M2M

En este capítulo se presentan los diferentes escenarios de adaptación de transformaciones M2M que detecta el método TRANSEVOL. Se describe cómo se detecta cada escenario de adaptación y se detallan las operaciones de adaptación requeridas en cada uno de esos escenarios de evolución junto con su especificación. Estos escenarios de evolución hacen referencia a cambios en la lógica de mapeo de la regla de transformación legada y a la necesidad de agregar nuevos conceptos de dominio en los metamodelos.

5.1 Algoritmo de deducción de escenarios de adaptación

El método TRANSEVOL se basa en la información del modelo de diferencias de entrada, **∆Im,** y en el modelo de diferencias de salida, **∆Im,** para determinar el escenario de adaptación especificado y deducir las operaciones de adaptación a implementar en las reglas de transformación. Para determinar y especificar correctamente todos los datos relacionados con una operación de adaptación el algoritmo de deducción necesita los siguientes elementos:

- **∆Im** (Modelo de diferencias entre el modelo ejemplo de entrada y el modelo previo de entrada).
- **∆Om** (Modelo de diferencias entre el modelo ejemplo de salida y el modelo previo de salida).
- **∆Imc** (Incremento de la cobertura de clases del metamodelo de entrada).
- **∆Omc** (Incremento de la cobertura de clases del metamodelo de entrada).
- **CC (∆Im)** (Cobertura de clases en el modelo de diferencias de entrada).
- **CC (∆Om)** (Cobertura de clases en el modelo de diferencias de salida).
- **Etr** (Traza de ejecución de la transformación legada con M_{PIM} como modelo de entrada y M_{PIM} como modelo de salida generado).

Con el objetivo de facilitar la compresión del algoritmo a continuación se listan sucintamente el significado de los valores de diferencias principales utilizados por TRANSEVOL

- **∆Imc=1** significa que una nueva metaclase de entrada se está contemplando.
- **CC (∆Im)=1** significa que las diferencias se aplican solo a un tipo de metaclase de entrada. Por lo tanto, una nueva regla o un nuevo patrón de entrada será necesario.
- **CC (∆Im)=N** significa que las diferencias se relacionan con N tipo de metaclases de entrada.
- **∆Omc=1** significa que una nueva metaclase de salida se está contemplando.
- **CC (∆Om)=1** significa que las diferencias se aplican solo a un tipo de metaclase de salida.

- **CC (∆Om)=N** significa que las diferencias se relacionan con N tipo de metaclases de salida.

TRANSEVOL clasifica los escenario de adaptación en base al tipo de diferencias creados en el modelo de diferencias de salida **∆Om**. En la siguiente lista se enumeran y clasifican los diferentes escenarios de adaptación en base al tipo de diferencias creadas en los modelos de salida:

a) Escenarios relacionados con **agregaciones** en los modelos de salida
    1. Nueva regla 1 a 1
    2. Nueva regla 1 a N
    3. Agregación de patrones de salida
    4. Nueva regla con filtrado
    5. División de regla
b) Escenarios relacionados con **modificaciones de atributos** en los modelos de salida
    1. Modificación de asignaciones
    2. División de regla con modificación de asignaciones
c) Escenarios relacionados con **modificación de contenedores** en los modelos de salida
    1. Cambio de contenedor
    2. División de regla con modificación de asignaciones
d) Escenarios relacionados con **eliminaciones** en los modelos de salida
    1. Eliminación de un patrón de salida
    2. Filtrado de un patrón de entrada
e) Escenarios donde **∆Om = 0**
    1. Filtrado de patrón de entrada
f) Escenarios de aplicación de estereotipos
    1. Modificación de asignaciones debido al uso de estereotipos
    2. División de una regla en dos debido a la aplicación de estereotipos y la modificación de atributos de salida
    3. Evolución de filtrado de mapeo debido a la aplicación de estereotipos

La lógica del algoritmo de detección varía ligeramente si se han aplicado estereotipos de un perfil en los modelos de entrada. Debido a esto lo primero que realiza el algoritmo es determinar si las diferencias del modelo **∆Im** son debidas a la aplicación de estereotipos. En caso de que las diferencias de **∆Im** no estén relacionadas TRANSEVOL analiza el tipo de diferencias creadas en los modelos de diferencias para determinar el tipo de escenario de adaptación. Actualmente TRANSEVOL solo permite especificar en cada iteración un único tipo de diferencias en **∆Om**. En la figura 5.1 se presenta como se realiza la clasificación de escenarios de adaptación.

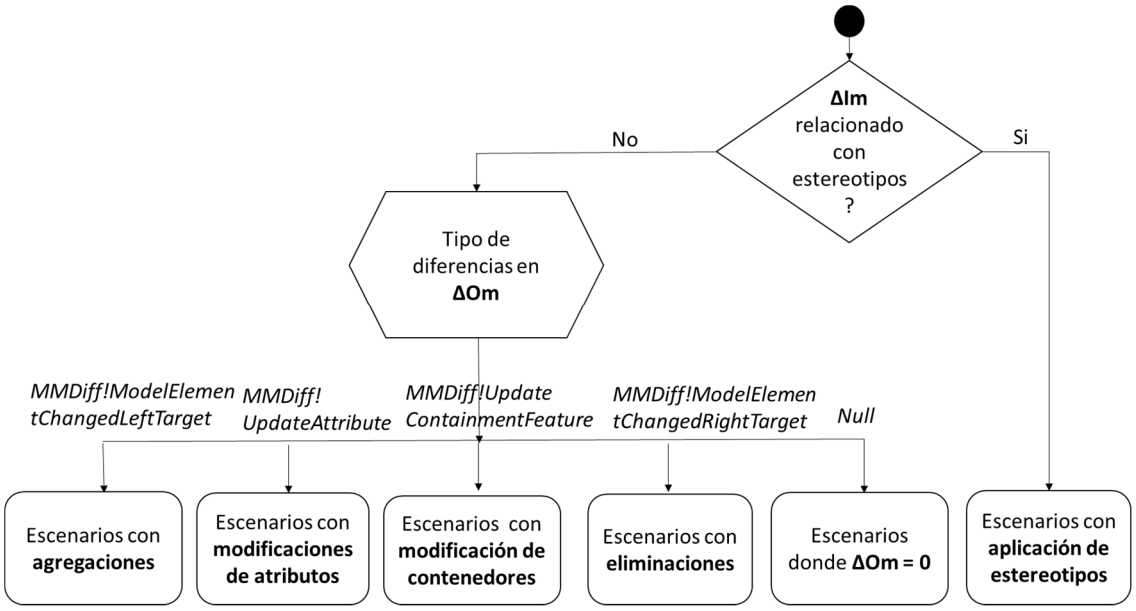


Figura 5.1 Algoritmo TRANSEVOL para la clasificación de escenarios.

En caso de que no se hayan aplicado estereotipos y las diferencias en el modelo de salida son agregaciones, *MMDiff!ModelElementChangeLeftTarget*, el algoritmo debe deducir si se trata de una nueva regla de transformación, de un nuevo patrón de salida en una regla existente o una división de regla (nueva regla con filtrado). Los escenarios de nueva regla con filtro y división de regla debido a agregaciones se detectan mediante la misma condición. Para diferenciarlos se analiza si la regla afectada existe previamente. Si no existe el escenario es una nueva regla, si la regla ya existe previamente el escenario es una división. En la figura 5.2 se resume la parte del algoritmo relacionada con los escenarios de agregación. En la imagen 5.2 también se resumen las condiciones de detección de escenarios. Estas condiciones se describen en detalle por cada escenario en el siguiente apartado.

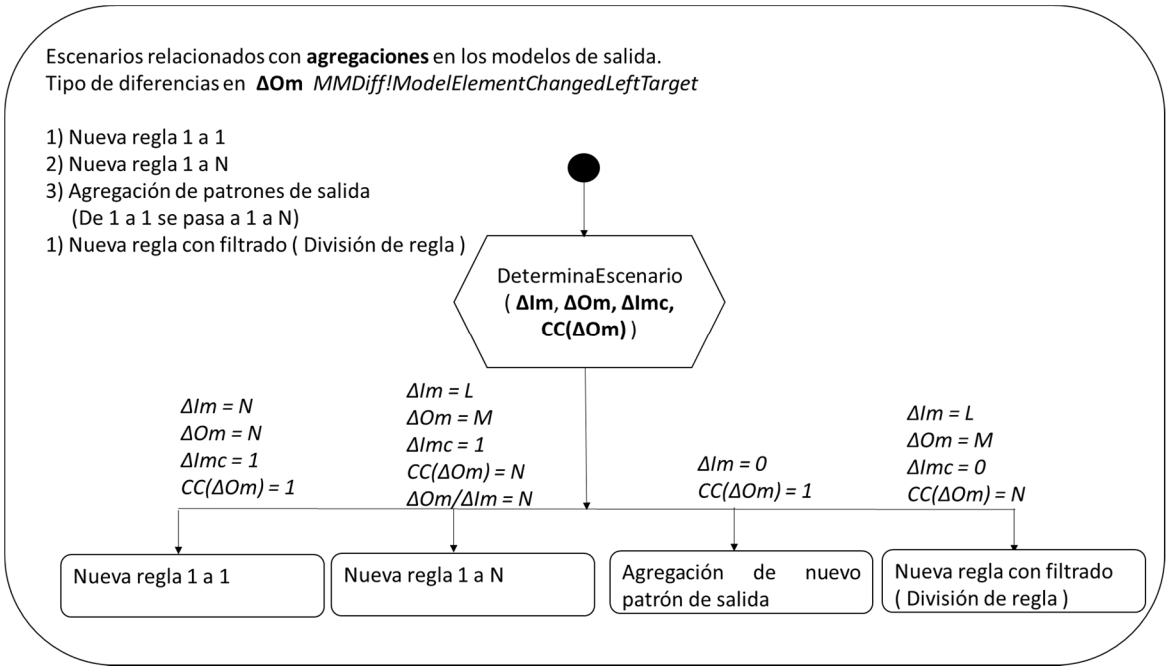


Figura 5.2 Algoritmo para la detección de escenarios de adaptación con diferencias de agregación de instancias en ∆**Om**.

Cuando las diferencias en ∆**Om** son modificaciones, *MMDiff!UpdateAttribute*, se requieren cambios en la asignaciones de la reglas. Estos cambios en las asignaciones pueden ser debidas a cambios en el mapeo de los atributos, o a que se utiliza un nuevo patrón de entrada en la regla. También puede ocurrir que la modificación de los atributos de elementos de salida solo afecte a algunas instancias de un patón de salida de una regla. En esto casos, se debe de crear una nueva regla con filtro en los patrones de entrada, de forma que se puedan generar las instancias de salida de forma diferente. Este tipo de situaciones ocurren cuando se quiere mapear de forma diferente instancias de un tipo dependiendo de los valores de sus atributos. Este tipo de escenario es similar a aquellos escenarios donde se han aplicado estereotipos en los modelos de entrada.

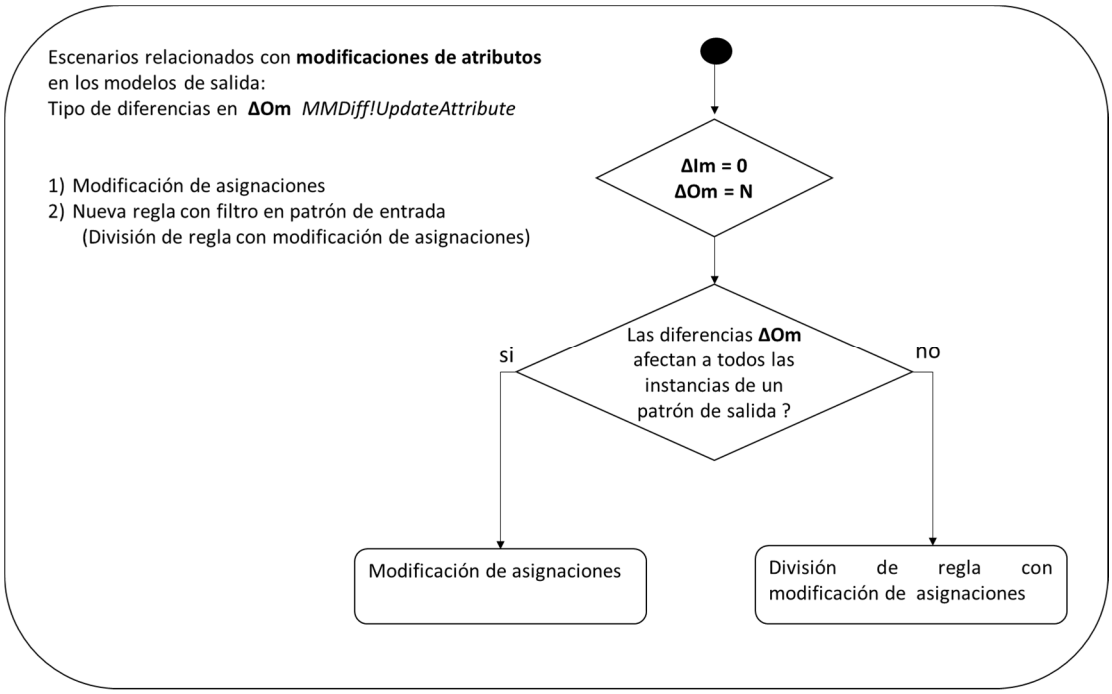


Figura 5.3 Algoritmo y condiciones para la detección de escenarios con modificación de atributos en ∆**Om**.

El siguiente grupo de escenarios está relacionado con el cambio de contenedor de las instancias de salida. En estas situaciones puede ocurrir que el cambio de contenedor afecta a una regla existente. Esto ocurre cuando todas las instancias de salida relacionadas con un patrón de salida de una regla son las afectadas. Cuando solo una parte de estas instancias son afectadas indica que se debe de crear una nueva regla con filtro, es decir una división de regla. En la figura 5.4 se resumen el algoritmo y las condiciones de los escenarios relacionados con modificación de contenedores.

Las eliminaciones de instancias en los modelos de salida están relacionados con la eliminación de un patrón de salida de una regla de transformación. Se contemplan dos situaciones de eliminación de patrón de salida en TRANSEVOL. En la primera todas las instancias generadas por unas reglas y relacionadas con un patrón de salida son eliminadas, y por lo tanto se debe de eliminar el patrón de salida de una regla existente, si al eliminar el patrón de salida la regla queda sin patrones de salida se elimina la regla. En la segunda situación no todas las instancias relacionadas con un patrón de salida son eliminadas y por lo tanto una nueva regla con filtro es necesaria. En la figura 5.5 se representa este algoritmo con sus condiciones.

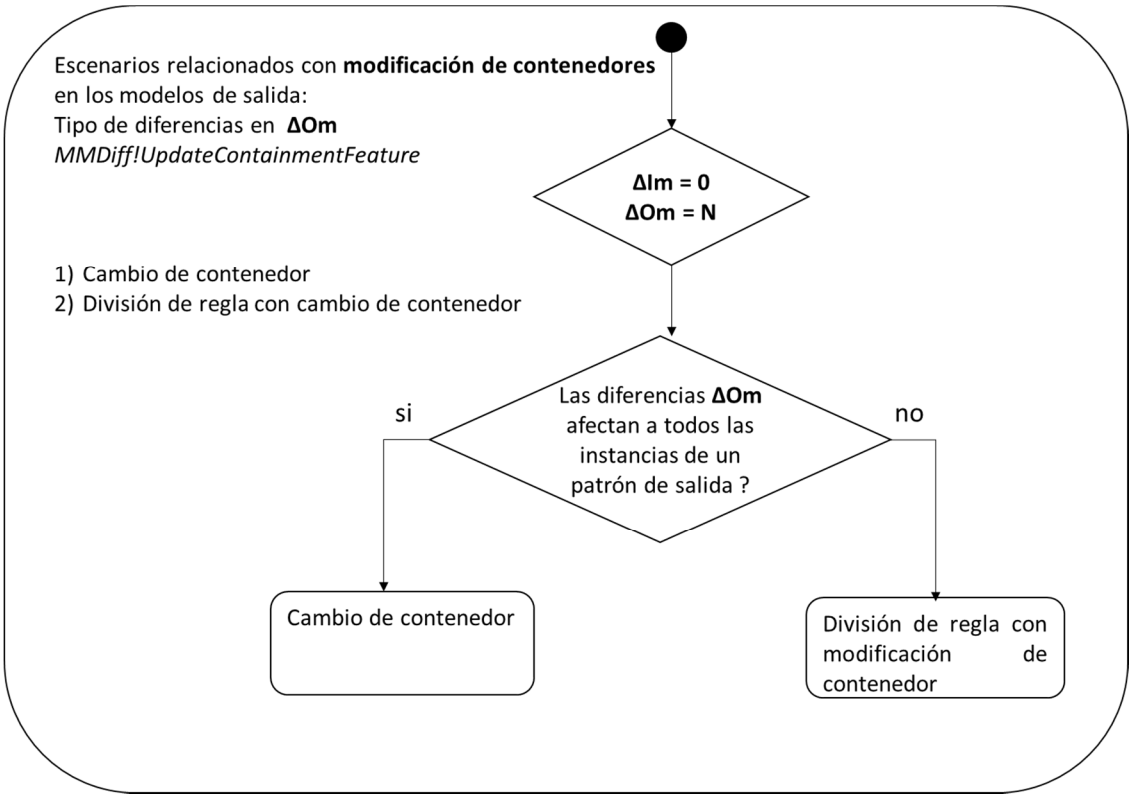


Figura 5.4 Algoritmo para la detección de escenarios de cambio de contenedor en los modelos de salida.

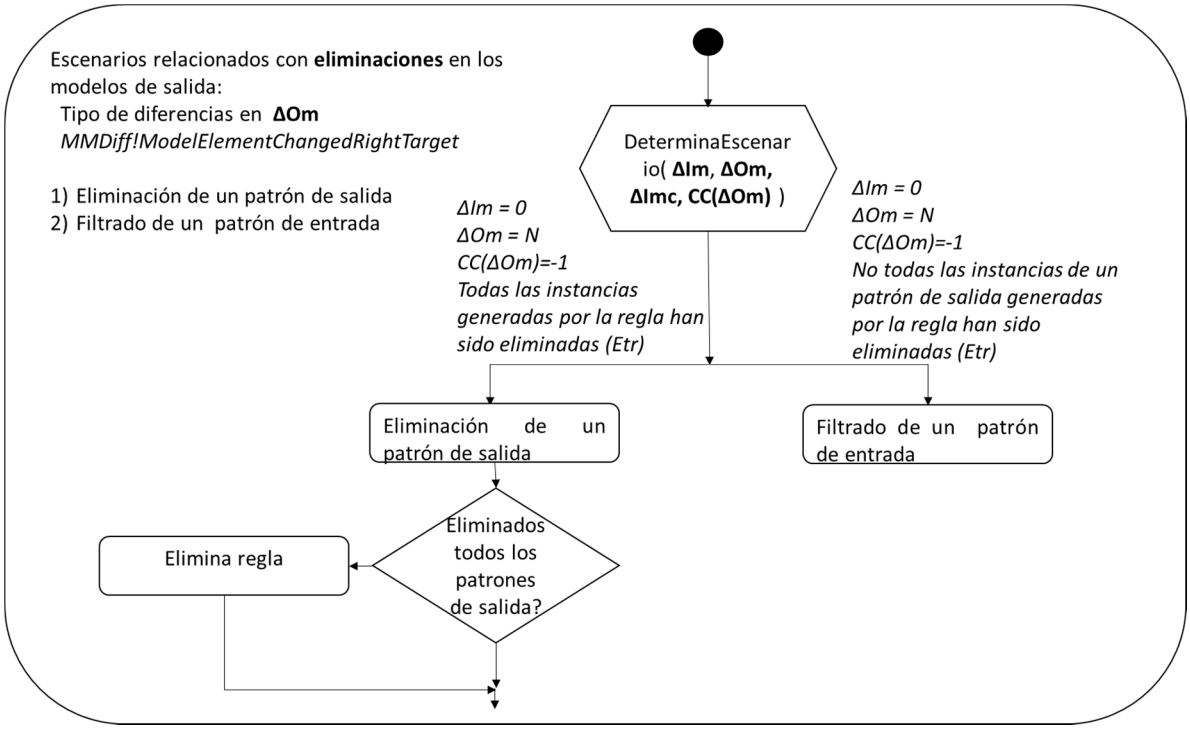


Figura 5.5 Algoritmo y condiciones para escenarios de eliminación de instancias de salida.

Existe la posibilidad de especificar un filtro en un patrón de entrada de una regla existente siendo **∆Om=0.** Esto ocurre cuando se agregan instancias en los modelos de entrada de un tipo de clase para las cuales ya existe una regla de transformación y no se realiza ninguna modificación en los nuevos modelos de salida. Estas situaciones requieren mapear solo ciertas instancias

de un tipo y no todas las instancias. Por lo tanto, se debe de aplicar un filtro a la regla que transforma ese tipo de instancias. En la figura se resume el algoritmo para escenarios de filtrado siendo ∆**Om=0.**

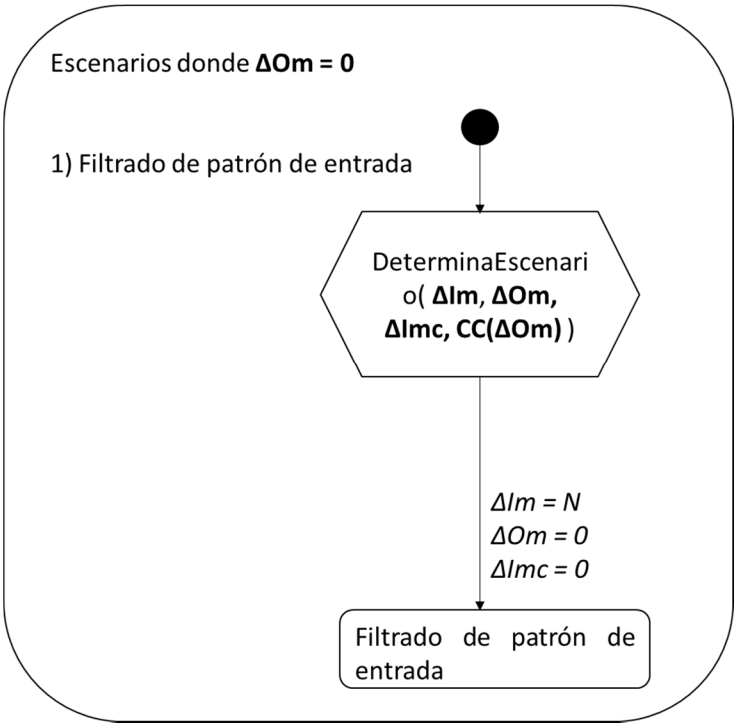


Figura 5.6 Condición de escenario de filtrado siendo ∆**Om=0**.

El último grupo de escenarios tratados por el algoritmo del método TRANSEVOL son aquellos escenarios donde se han aplicado estereotipos en los modelos de entrada. Existen situaciones donde sólo se modifican las instancias de salida relacionadas con las instancias de entrada estereotipada. En estas situaciones TRANSEVOL entiende que el desarrollador quiere especificar que el mapeo de las instancias estereotipadas se realiza de forma diferente al de las no estereotipadas. Y por ello en esas situaciones es necesario la aplicación de filtros. Si el objetivo de la aplicación de los estereotipos es ofrecer más datos para el cálculo de los atributos de las instancias de salida no es necesario la aplicación filtros. En estos casos las diferencias de entrada ∆**Im** relacionadas con estereotipos no se contemplan en la deducción del escenario de adaptación y el algoritmo de análisis es exactamente igual al de los escenarios de cambio de lógica.

Cuando el filtrado es necesario debido al uso de estereotipos tres escenarios son detectados por TRANSEVOL: 1) División de regla con filtro debido a modificaciones 2) División de regla con filtro debido a agregaciones de instancias en el modelo de salida y 3) Aplicación de filtro en una regla existente debido a eliminaciones. Estos escenarios se identifican mediante las condiciones de la figura 5.7.

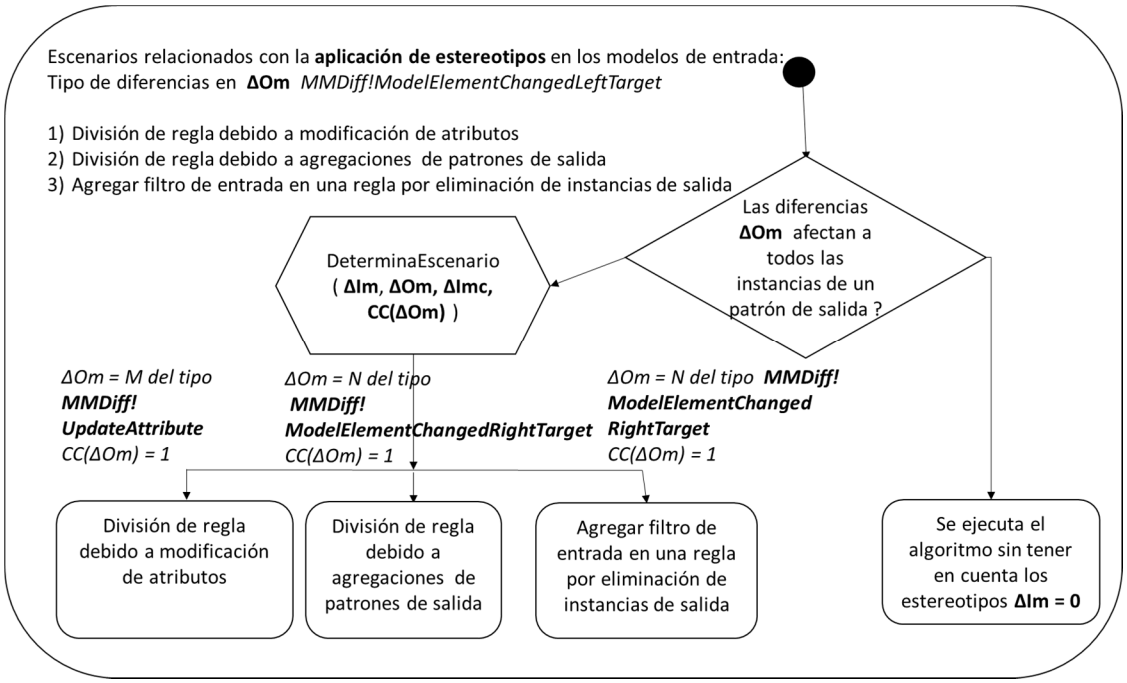


Figura 5.7 Algoritmo para escenarios con aplicación de estereotipos.

5.2 Escenarios de adaptación contemplados en TRANSEVOL

En este apartado se analizan uno por uno todos los escenarios de adaptación de transformaciones M2M contemplados por TRANSEVOL. Para cada uno de los escenarios se especifican las condiciones de detección y las operaciones de adaptación a implementar. También se explica que datos son utilizados para la correcta especificación de las operaciones de adaptación. En cada una de los escenarios se utiliza un pequeño ejemplo para facilitar la compresión.

5.2.1 Escenario nuevo mapeo uno a uno

Este tipo de escenarios se dan cuando en la transformación M2M se quiere contemplar una metaclase del metamodelo de entrada que previamente no se mapeaba. Este tipo de escenarios también son el resultado de agregar una nueva metaclase al metamodelo de entrada (PIM). Este tipo de escenario requiere agregar una nueva regla de transformación. Por lo tanto ante estos escenarios se debe de generar un modelo MMAdaptationGoal con un elemento del tipo *MMAdaptationGoal!AddMatchedRule*.

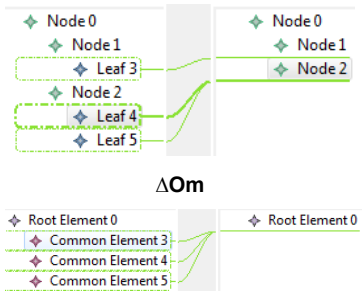
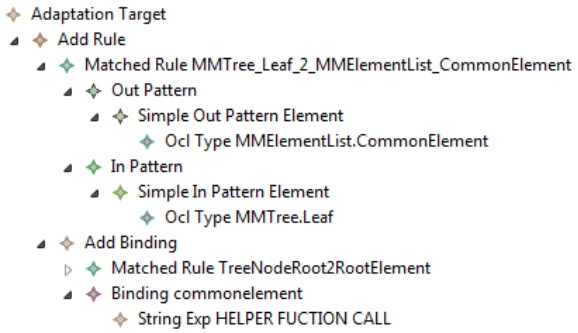
Para determinar si el escenario de evolución corresponde a un mapeo de uno a uno se utilizan los modelos **ΔOm, ΔIm, ΔOmc** y **ΔImc**. Al contemplar un nuevo tipo de elemento en el modelo de entrada el valor de **ΔImc** es igual a 1 (**ΔImc** = 1)**.** Lo cual indica que se quiere mapear una nueva metaclase no mapeada previamente. **ΔOmc** puede ser 0 o 1, dependiendo de si se requiere generar instancias de un nuevo tipo del metamodelo de salida. En estas situaciones. **ΔOmc = 1.** En los mapeos 1 a 1 solo se generan instancias de salida de un solo tipo, y por ello CC(**ΔOm) = 1.** Los modelos de diferencias **ΔOm** y **ΔIm** contendrán un numero de diferencias superior a 0 del tipo *MMDiff!ModelElementChangeLeftTarget* (**ΔOm > 0** y **ΔIm > 0**) y **ΔOm = ΔIm**.

Si **ΔOm>ΔIm** el número de patrones de salida por patrón de entrada es **ΔOm/ΔIm.** Si **ΔOm<ΔIm** se determina como escenario de mapeo con filtrado. Los filtrados en escenarios de mapeo uno a uno se describen en un apartado posterior. Las condiciones de detección de un escenario de nueva regla de mapeo uno a uno se recogen en la tabla 5.1. En la tabla 5.2 se representan un escenario concreto que requiere una nuevo mapeo uno a uno y se muestran los valores concretos de **ΔOm, ΔIm, ΔOmc** y **ΔImc**.

Tabla 5.1 Condición para la detección de un nuevo mapeo uno a uno.

Condición para la detección del escenario			**Tipo de diferencia**
ΔImc = 1	**0<=ΔOmc<=1** **CC (ΔOm) = 1**	**ΔOm = ΔIm = N** **N > 0**	*MMDiff!ModelElementChangeLeftTarget*

Tabla 5.2 Ejemplo de modelos de diferencias ΔOm y ΔIm en un escenario de adaptación de un nuevo mapeo uno a uno y el modelo de operaciones de adaptación resultante.

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condición	**Modelos de diferencias**	
ΔIm=3 **ΔOm= 3** **ΔImc=1** **ΔOmc=1**	**ΔIm**  **ΔOm** 	
Nota: Las diferencias de **ΔIm** y **ΔOm** son del tipo *MMDiff!ModelElementChangeLeftTarget*		

Una vez detectado el escenario de evolución se debe de generar la operación de adaptación con todos los datos necesarios. Para definir correctamente una operación del tipo *MMAdaptationGoal!AddMatchedRule* se debe de obtener: el patrón del elemento de entrada de la nueva regla, el patrón del elemento de salida de la nueva regla, la regla afectada por el nuevo *binding* y la expresión del nuevo binding. Para conocer el patrón de los elementos de entrada y el patrón de salida de la nueva regla de transformación se utiliza el EObject del campo *leftElement* de alguna las diferencias *MMDiff!ModelElementChangeLeftTarget*. La propiedad *leftTarget* indica el elemento agregado. Utilizando alguna de las diferencias *MMDiff!ModelElementChangeLeftTarget* del **ΔIm** se obtiene el tipo del patrón de entrada de la nueva regla de transformación**.** Para obtener el patrón de salida se utiliza el mismo proceso utilizando una de las diferencias *MMDiff!ModelElementChangeLeftTarget* del modelo **ΔOm**. Para detectar la regla afectada por el nuevo binding se utiliza el algoritmo basado en el modelo **Etr** y presentando en el apartado 4.3.2 Para localizar la nueva asignación se parte de la propiedad *rightParent* de alguna de las diferencias *MMDiff!ModelElementChangeLeftTarget* **ΔOm**. La propiedad *rightParent* indica cual es el objeto contenedor de la nueva instancia agregada. Después se busca en el modelo **Etr** la regla

que generó previamente la instancia del contenedor, siendo esta regla la afectada. El código de la asignación se implementa actualmente como una llamada a una función *Helper* de ATL. TRANSEVOL actualmente no deduce el código de la función *Helper*. Esta función actualmente la debe de implementar manualmente el desarrollador, la cabecera de la función y la llamada a ella si se describe en el modelo *MMAdaptationGoal* de salida. El *mapping* de las propiedades de la nueva regla actualmente también hay que definirlo manualmente en el modelo *MMAdaptation-Goal* de salida. En la figura 5.8 se describe el proceso implementado para obtener los datos de una nueva regla de transformación.

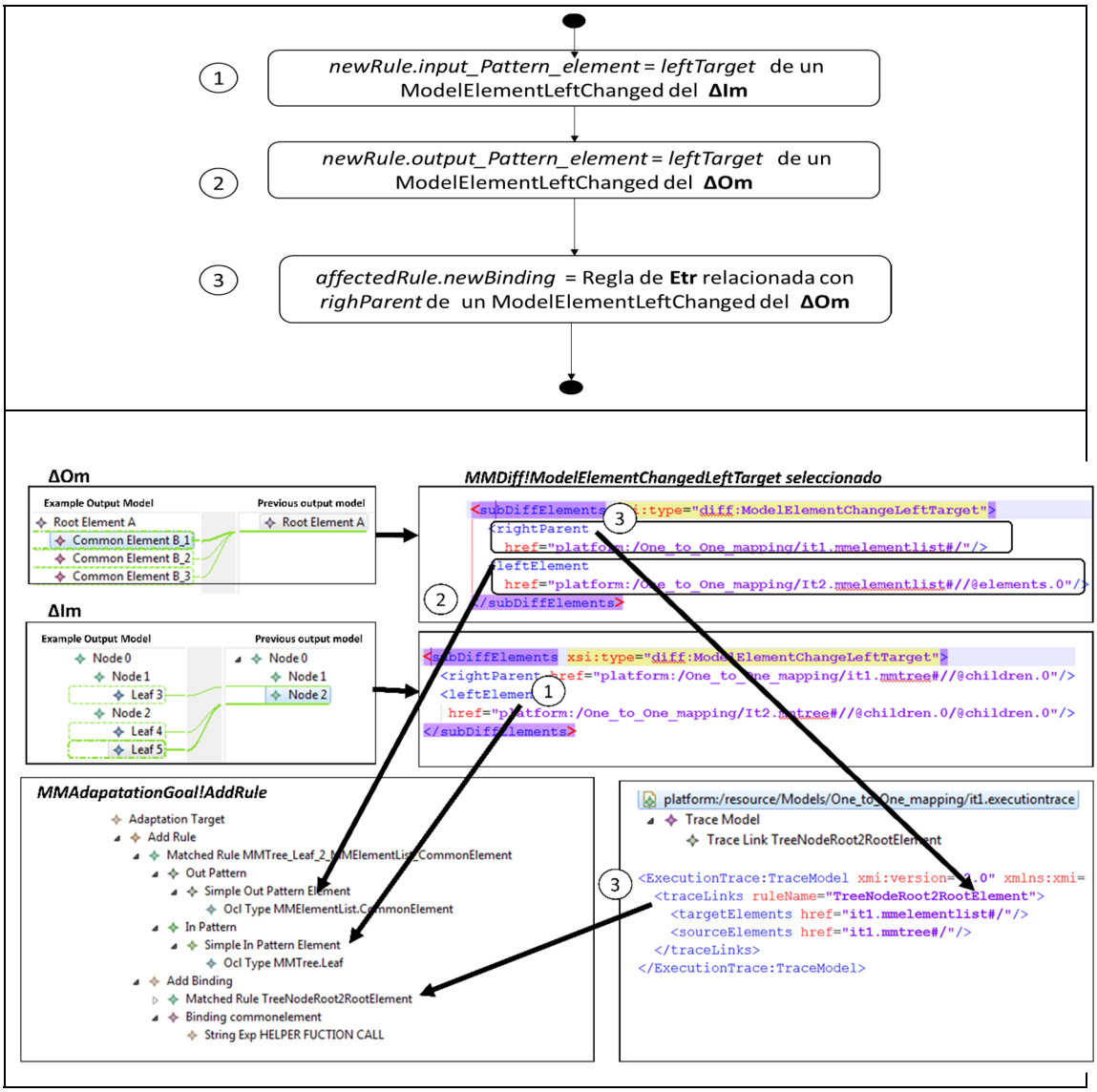


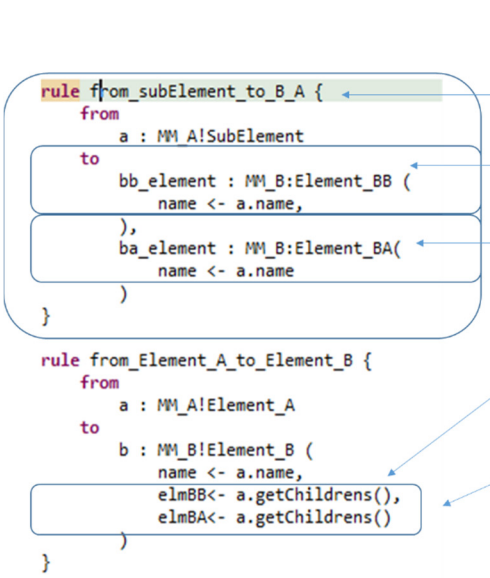
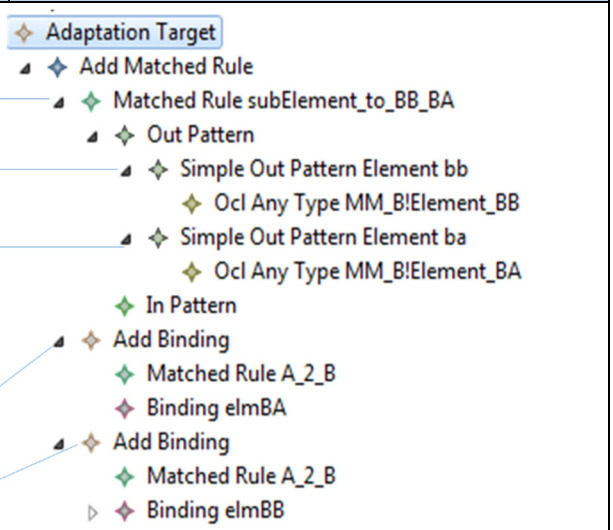
Figura 5.8 Proceso para la obtención de datos de una operación de adaptación *MMAdaptationGoal!Add-MatchedRule*.

Puede ocurrir que el nuevo binding ya exista. Por ejemplo, siendo **ΔOmc=1,** pueda darse la circunstancia de que la nueva metaclase herede de una clase genérica de la cual ya se ha realizado un binding para elementos de ese tipo. Actualmente no se ha implementado la detección de estas situaciones. Al generar la nueva regla se genera un mensaje de aviso indicando esta problemática, de forma que el desarrollador de transformaciones pueda comprobar si el nuevo binding es realmente necesario.

5.2.2 Escenario de un nuevo mapeo uno a varios

Este tipo de escenario es una extensión de los escenarios de mapeo uno a uno. En estas situaciones en lugar de generar un único elemento de salida por elemento de entrada se generan M instancias de N tipos. La operación de adaptación que se requiere es una nueva regla con N nuevos patrones de salida y N operaciones de *binding*. En la tabla 5.3 se presenta un ejemplo de código ATL que agrega un mapeo uno a varios y su correspondiente modelo de adaptaciones.

Tabla 5.3 Ejemplo de una regla de transformación uno a varios.

Código ATL de un mapeo uno a varios	Modelo MMAdaptationGoal que especifica una nueva regla uno a varios
	

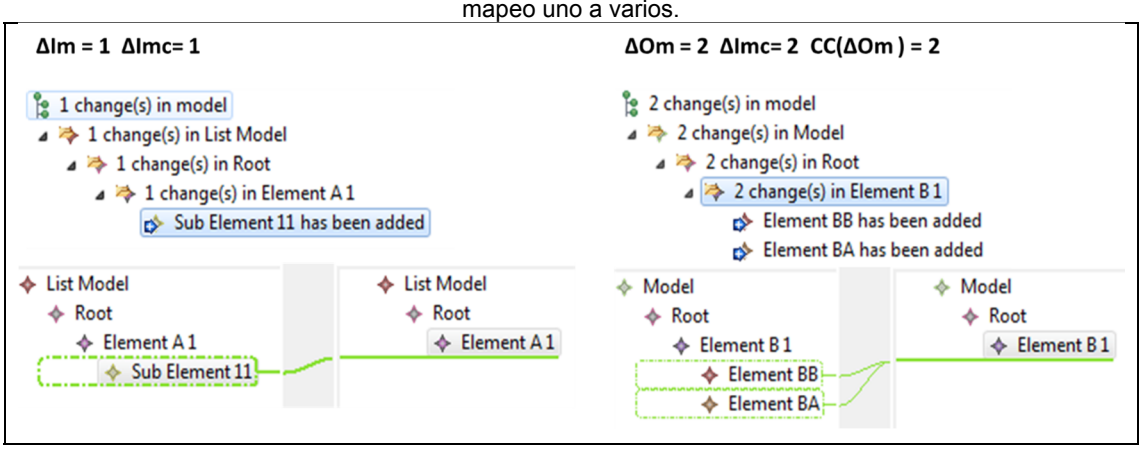
Las condiciones de detección son parecidas a los escenarios de mapeo uno a uno. El tipo de diferencias que se crean al crear los modelos ejemplos de entrada y salida que representan un escenario de un nuevo mapeo uno a varios son del tipo *MMDiff!ModelElementChangeLeftTarget.* **CC**(∆**Omc**)**=N** siendo **N > 1** y ∆**Om>**∆**Im**. **CC**(∆**Omc**)**=N** indica que diferentes tipos de instancias se están generando en el modelo de salida**.** Debido a que **CC**(∆**Omc**) > 1 la detección de la regla afectada para las nuevas asignaciones no es idéntico al caso del escenario de mapeo uno a uno. En el mapeo uno a uno el valor de **CC**(∆**Omc**) = 1. Siendo **CC**(∆**Omc**) = 1 mediante la propiedad *rightParent* de alguna de las diferencias del tipo *MMDiff!ModelElementChangeLeftTarget* de ∆**Om** se obtiene directamente el elemento contenedor, y combinando este dato con **Etr** se obtiene la regla afectada para el nuevo binding, como se explicó previamente. Cuando **CC**(∆**Omc**) > 1 puede ocurrir que las nuevas instancias de salida de diferentes tipos se ubiquen en diferentes tipos de elementos contenedores. Para contemplar la posibilidad de que existen diferentes tipos de contenedores para cada tipo de salida se deben de agrupar los diferentes *MMDiff!ModelElementChangeLeftTarget* en base al tipo de contenedores. Mediante esta agrupación se obtiene la relación de nueva instancia de salida, contenedor y regla de transformación afectada. En la tabla 5.4 se define la condición de detección del escenario de un nuevo mapeo

uno a varios. En la figura 5.9 se representa un ejemplo de modelos de diferencias **ΔIm** y **ΔIm** que corresponden a un nuevo mapeo uno a varios.

Tabla 5.4 Condición para la detección de un escenario de un nuevo mapeo uno a varios.

Condición para la detección del escenario			Tipo de diferencia
ΔImc = 1	**CC (ΔOm) = N** **N > 1**	**ΔOm > ΔIm**	*MMDiff!ModelElementChangeLeftTarget*

Figura 5.9 Ejemplo de **m**odelos de diferencias ΔOm y ΔIm en un escenario de adaptación de un nuevo mapeo uno a varios.



5.2.3 Evolución de un mapeo “uno a uno” a “uno a varios”

En estos escenarios de evolución el requisito es agregar un nuevo patrón de salida a una regla existente. Estos escenarios se caracterizan por agregar diferencias del tipo *MMDiff!ModelElementChangeLeftTarget* únicamente al modelo **ΔOm.** La situación más habitual que define la necesidad de agregar un nuevo patrón de salida es aquella donde sin ocurrir modificaciones en los modelos de entrada se requiere crear un nuevo patrón de salida en una regla de transformación. En estas situaciones no existe ninguna modificación en el modelo de entrada, **ΔIm=0**. Ante estos escenarios se propone agregar el nuevo patrón de salida en la regla de transformación del contenedor de los elementos a generar. Habitualmente el nuevo patrón de elemento de salida será de un tipo diferente a los ya existentes. También puede ocurrir que el nuevo patrón de salida sea del mismo tipo que algún patrón existente. La figura 5.10 representa los cambios a realizar en una regla de transformación ATL para pasar de ser una transformación una a uno a uno a dos. La operación de adaptación a agregar en estos escenarios es un *MMAdaptationGoal!AddOutputPatternElement.* La operación de adaptación *MMAdaptationGoal!AddOutputPatternElement* consta de un nuevo patrón de salida, la regla afectada donde se debe de agregar el nuevo patrón de salida y el nuevo binding para asociar los elementos generados a su contenedor.

```
rule from_Element_A_to_Element_B {
from
    a : MM_A!Element_A
to
   b : MM_B!Element_B (
     name <- a.name,          nueva asignación
     element <- bb_element
    ),                        Nuevo patrón de salida
    bb_element : MM_B:Element_BB(
    name <- a.name
  )
}
```

Figura 5.10 Regla ATL afectada por un nuevo patrón de salida.

Este tipo de situaciones se caracterizan por **∆Im = 0, ∆Imc = 0**, **CC(∆Om)=1 y ∆Om =M** siendo **M > 0.** Este escenario indica que sin variar los modelos de entrada se deben generar nuevas instancias de un tipo en los modelos de salida. Por lo tanto, se requiere agregar un nuevo patrón de salida a una regla de mapeo. Cuando la regla afectada no es la regla que genera el elemento contenedor de los nuevos patrones, el algoritmo actual no es capaz de detectar la regla afectada debido a que **∆Im = 0**. Para solucionar este tipo de situaciones se debe de completar la información que ofrecen las diferencias con información extra. Una posibilidad para localizar las reglas afectadas seria indicar en un modelo ejemplo, mediante estereotipos, que instancias de entrada son las que se relacionan con los nuevos patrones de salida. De esta forma cuando se consigue que **∆Im!= 0** es posible conocer las instancias de entrada relacionadas con los nuevos elementos de salida, y por consiguiente mediante **Etr** es posible obtener la regla afectada. En la tabla 5.5 se resume la condición para la detección de este tipo de situaciones. Por otro lado, en la tabla 5.6 se representan los modelos **∆Om** y **∆Im** que especifican un escenario de adaptación de este tipo. También se representa un modelo *MMAdaptationGoal* que especifica la operación de adaptación a implementar.

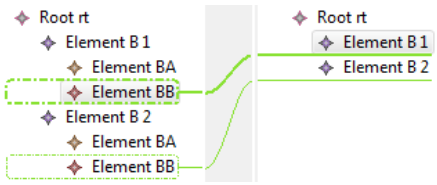
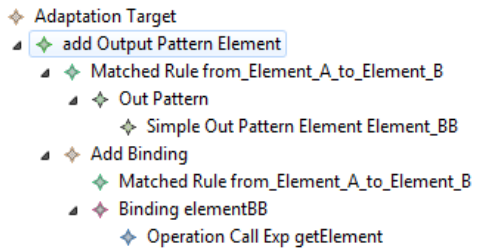
Tabla 5.5 Condición para la detección de una evolución de un mapeo uno a uno a un mapeo uno a varios.

Condición para la detección del escenario			**Tipo de diferencia**
∆Im = 0 **∆Imc** = 0	**∆Om = M** **M > 0**	**CC(∆Om)= 1**	*MMDiff!ModelElementChangeLeftTarget*

La información del nuevo patrón de salida se obtiene de las diferencias *MMDiff!ModelElementChangeLeftTarget* **∆Om,** al igual que en los escenarios de mapeo uno a uno. La regla afectada por el nuevo patrón de salida y por el nuevo binding se encuentra mediante la propiedad *rightParent* y el **Etr**. Puede ocurrir que el nuevo *binding* ya exista. Ante estas situaciones el desarrollador debe de detectar la duplicidad en el código y corregir el modelo de operaciones de adaptación. También puede ocurrir que al crear los modelos ejemplo **CC(∆Om) = N** , siendo **N > 1.** Este tipo de situaciones requiere crear N nuevos patrones. La herramienta desarrollada detecta este tipo de situaciones correctamente. De todas maneras se recomienda dividir este requisito de transformación en varios de forma que en cada iteración cada *MMAdaptationGoal!AddOutputPatternElement* corresponda a una metaclase, siendo **CC(∆Om)=1.**

Tabla 5.6 Ejemplo de modelos de diferencias que especifican un escenario de adaptación de un mapeo uno a uno a uno a dos y su correspondiente modelo de adaptación de operaciones.

<table>
<tr><th colspan="2">Escenarios de evolución mapeo uno a uno</th><th rowspan="2">Operación de adaptación</th></tr>
<tr><th>Condición</th><th>Modelos de diferencias</th></tr>
<tr><td>ΔIm=0
ΔOm= 2
ΔImc=0
ΔOmc=1</td><td>ΔIm
 

ΔOm
 
</td><td> 
</td></tr>
<tr><td colspan="3">Nota: Las diferencias de ΔIm y ΔOm son del tipo MMDiff!ModelElementChangeLeftTarget</td></tr>
</table>

5.2.4 Escenario de eliminación de patrones de salida

En este tipo de escenario las diferencias en los modelos de diferencias son del tipo *MMDiff!ModelElementChangeRightTarget*, *que* indican la eliminación de instancias. Cuando alguna de las transformaciones a eliminar presenta más de un patrón de entrada o más de un patrón de salida tres operaciones de adaptación son posibles: 1) la eliminación de un patrón de salida y 2) la eliminación de la regla. Para diferenciar estas situaciones se contabiliza el número de patrones de salida de la posible regla a eliminar. Si el número de patrones es mayor que el número de tipo de instancias eliminadas, entonces se elimina un patrón y no la regla. Para contabilizar el número de patrones de salida de una la regla se utiliza el modelo EMF de la transformación.

El escenario más sencillo de eliminación de mapeo es aquella donde se eliminan todas las instancias de salida relacionadas con una regla de transformación. En este tipo de escenario no tienen por qué existir diferencias en los modelos de entrada. Por otro lado deben de existir N eliminaciones en el modelo de salida, **ΔOm=M (**siendo **M>0).** En estas situaciones **ΔOmc <=0** (Normalmente **ΔOmc=-1**). Cuando el algoritmo detecta que se han eliminado elementos de salida, agrupa las instancias de salida eliminadas por tipos. Cada grupo representa un patrón de salida. Mediante el modelo **Etr** se verifica si los patrones de salida corresponden a una misma regla de transformación. Si las instancias eliminadas corresponden a una única regla, el algoritmo continúa con el análisis. En caso contrario, el algoritmo finaliza el análisis debido a que el escenario especificado se relaciona con más de una regla de transformación, y actualmente TRANSEVOL no contempla este tipo de escenarios. Lo siguiente que debe de hacer el algoritmo es decidir si es una eliminación o un filtrado. En los escenarios de eliminación se eliminan todas las instancias relacionadas con un patrón de salida de una regla de transformación. Si todas las instancias del mismo tipo y generados por la misma regla han sido eliminadas, el escenario de adaptación es de eliminación de una regla o de un patrón de salida. Si todas las instancias no

han sido eliminadas, entonces el escenario es de filtrado. La función *todasInstanciasDeReglaEliminada* realiza el chequeo que permite diferenciar entre una situación de eliminación de regla o filtrado. Para conocer si todas las instancias han sido eliminadas se utiliza el modelo **Etr**. Unas vez deducido que es un escenario de eliminación, se deduce si es una eliminación de una regla de transformación o de un patrón de salida. Finalmente se genera la operación de adaptación. En la tabla 5.7 se resumen las condiciones para detectar la eliminación de regla de transformación. En la tabla 5.8 se representa un ejemplo de eliminación de regla cuando **∆Im=0**.

Tabla 5.7 Condición para la eliminación de un mapeo.

Tipo de diferencia	**Condición para la detección del escenario**		
MMDiff!ModelElementChangeRightTarget	**∆Im = 0**	**∆Om = M*** **M > 0**	**CC(∆Om)=-1**
***Se eliminan todas las instancias relacionadas con un patrón de salida**			

Tabla 5.8 Ejemplo de escenario de eliminación de regla siendo ∆Im = 0.

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condición	**Modelos de diferencias**	
∆Im=0 **∆Imc=0** **∆Om= 2** **∆Omc=-1**	**∆Om** 2 change(s) in Model 2 change(s) in Root Element B 1 has been removed Element B 2 has been removed 	**rule from_Root_A_to_Root_B {** **from** a : MM_A!Root **to** b : MM_B!Root (name <- a.name, *binding a eliminar* elms<- a.getElements()) } *regla a eliminar* **rule from_Element_A_to_Element_B {** **from** a : MM_A!Element_A **to** b : MM_B!Element_B (name <- a.name,) }
Nota: Las diferencias de **∆Im** y **∆Om** son del tipo *MMDiff!ModelElementChangeRightTarget* y el número de instancias eliminadas por patrón de salida es igual que el número de instancias generadas por patrón de salida.		

Cuando existe alguna diferencia del tipo *MMDiff!ModelElementChangeRightTarget* se utiliza la instancia (EObject) de la propiedad *rightElement,* que indica cual fue el elemento eliminado para la localizar la regla afectada. Para determinar la regla a eliminar se localiza en el modelo **Etr** la regla que generó alguna de las instancias eliminadas. Para localizar el binding a eliminar se utiliza la instancia de la propiedad *leftTarget*, que indica cual es la instancia contenedora afectada por la eliminación de la instancia de salida. En el modelo **Etr** se obtiene la regla afectada localizando la regla que creó alguno de los contenedores afectados por la eliminación.

5.2.5 Escenarios de filtrado

El filtrado se utiliza cuando se quiere transformar de forma diferente instancias de un mismo tipo. En los escenarios de filtrado se transforman sólo aquellas instancias de una metaclase que presentan cierta característica. Normalmente se filtra en base al valor de un atributo o la aplicación de un estereotipo. La transformación ejemplo de ATL *The in/out ports* (*http://www.eclipse.org/atl/documentation/basicExamples_Patterns/*) presenta un escenario típico de filtrado. En el ejemplo *The in/out ports* las instancias de entrada del tipo *TypeA!PortA* se transforman de forma diferente dependiendo de la asociación con su contenedor del tipo *TypeA!BlockA*. Aquellas instancias de *TypeA!PortA* agrupadas en la propiedad *inports* de un elemento *TypeA!BlockA* se transforman en instancias del tipo *TypeB!InPortB*, y aquellas que pertenecen a la propiedad *outports* de un elemento *TypeA!BlockA* se transforman en instancias del tipo *TypeB!OutPortB*. En la figura 5.11 se presenta el código ATL de las reglas de transformación con filtro del ejemplo *The in/out ports*.

```
rule BlkA2BlkB {
        from  blkA : TypeA!BlockA
        to
              blkB : TypeB!BlockB (
                        inputPorts <- blkA.inputPorts,
                        outputPorts <- blkA.outputPorts
              )
}
rule PortA2InPortB {
        from  s : TypeA!PortA (
                  s.refImmediateComposite().inputPorts->includes(s))
        to
            t : TypeB!InPortB (
                  name <- s.name
              )
}
rule PortA2OutPortB {
        from  s : TypeA!PortA (
                  s.refImmediateComposite().outputPorts->includes(s))
        to
              t : TypeB!OutPortB (
                        name <- s.name
              )
}
```

Figura 5.11 Código ATL de la transformación *The in/out ports* (http://www.eclipse.org/atl/documentation/basicExamples_Patterns/).

El método desarrollado detecta 2 escenarios de filtrado:1) Agregación de una condición de filtrado al patrón de entrada y 2) Creación de una nueva regla de transformación con filtrado. Este tipo de escenarios se expresan mediante modelos ejemplos que agregan elementos a modelos previos o eliminando elementos de modelos previos. Para poder detectar este tipo de es-

cenarios se podrían utilizar tanto las diferencias de eliminación (*MMDiff!ModelElementChangeRightTarget*) como las agregaciones (*MMDiff!ModelElementChangeLeftTarget*.). Si se quiere representar este escenario mediante eliminaciones de instancias, previamente se ha tenido que expresar un mapeo uno a uno donde todas las instancias de un tipo se transforman sin condición alguna. Una vez generada la nueva regla, el siguiente paso es eliminar algunas de las instancias, no todas, que generó la nueva regla. Al crear los modelos ejemplos se deben de eliminar las instancias que no se deben de generar. Al eliminar las instancias se obtiene: **∆Im = 0, ∆Imc = 0, ∆Om = N** (del tipo *MMDiff!ModelElementChangeRightTarget*) y **∆Omc = 0.** Cuando el requisito de mapeo se especifica mediante la eliminación de elementos, tanto la especificación del escenario como el algoritmo de detección son muy similares a los del escenario de eliminación de una regla de transformación. En este caso la única diferencia es que el número de instancias de salida eliminadas por cada patrón de salida es menor al número de instancias totales por cada patrón de salida. En la tabla 5.9 se resumen las condiciones para la detección de un filtrado mediante eliminación de instancias. Cuando la regla de transformación afectada por el filtrado presenta N patrones de entrada el algoritmo actual no detecta correctamente el patrón de entrada sobre el que hay que aplicar el filtro. Esto ocurre debido a que **∆Im = 0** y no se tiene información sobre los elementos de entrada. Para solventar esta situación hay que indicar alguna de las instancias que represente el patrón de entrada afectado. Esto se puede realizar aplicando un estereotipo a una instancia de salida.

Tabla 5.9 Condiciones para la detección de un escenario de filtrado mediante eliminación de instancias.

Tipo de diferencia	**Condición para la detección del escenario**		
MMDiff!ModelElementChangeRightTarget	**∆Im = 0** **∆Imc** =0	**∆Om = M*** **M > 0**	**CC(∆Om)=-1**
*El número de instancias eliminadas por patrón de salida es menor que el número de instancias generadas por patrón de salida.			

Tabla 5.10 Ejemplo de escenario de filtrado de patrón de entrada mediante eliminación de instancias

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condición	**Modelos de diferencias**	
∆Im = 0 **∆Imc = 0** **∆Om = N** **∆Omc = 0**	**∆Om** 2 change(s) in model Root Element 1 has been removed Root Element 2 has been removed Root Element 0 Root Element 0 Root Element 1 Root Element 2	**rule TreeNodeRoot2RootElement {** **from** *filtro agregado* rt : MMTree!Node (rt.isTreeNodeRoot()) **to** lstRt : MMElementList!RootElement (name <- rt.name) }
Nota: Las diferencias de **∆Im** y **∆Om** son del tipo *MMDiff!ModelElementChangeRightTarget* y el número de instancias eliminadas por patrón de salida es menor que el número de instancias generadas por patrón de salida.		

Al representar filtrados mediante diferencias de eliminación puede ocurrir que los metamodelos contengan algunas restricciones que no permitan expresar correctamente el filtrado mediante eliminaciones. En el ejemplo de la tabla 5.10, el metamodelo de salida *MMList* permite tener más de una raíz, pero lo normal es que una lista solo contenga un elemento raíz, como en

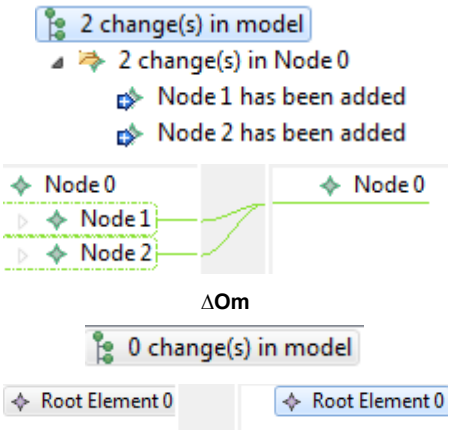
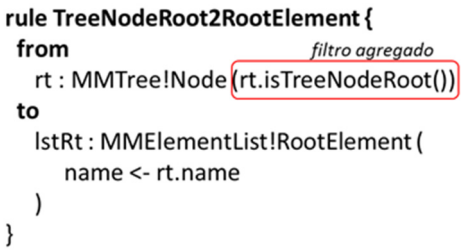
la transformación ATL ejemplo *Tree to List* (*http://www.eclipse.org/atl/documentation/basicExamples_Patterns/*). En ese caso no se puede generar el modelo de salida previo de tabla 5.10. Para llegar a la misma implementación de transformación se debe de especificar el requisito de transformación mediante agregación de instancias.

Al utilizar agregación de instancias se pueden especificar dos escenarios de filtrado: 1) Filtrado de un patrón de salida y 2) Nueva regla de mapeo con filtrado. El filtrado del patrón de entrada de una regla de transformación se detecta mediante **ΔIm = N, ΔImc = 0, ΔOmc = 0** y **ΔOm = 0.** Esta condición indica que nuevas instancias de una metaclase previamente transformada no requiere ningún mapeo. En la tabla 5.11 se resume la condición de detección, mientras que en la tabla 5.12 se presenta un ejemplo de este tipo de escenario.

Tabla 5.11 Condiciones para la detección de un escenario de filtrado mediante agregación de instancias en el modelo de entrada.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget	**ΔIm = N** **ΔImc** =0	**ΔOm = 0**

Tabla 5.12 Ejemplo de un escenario de filtrado de patrón de entrada mediante agregación de instancias en el modelo de entrada.

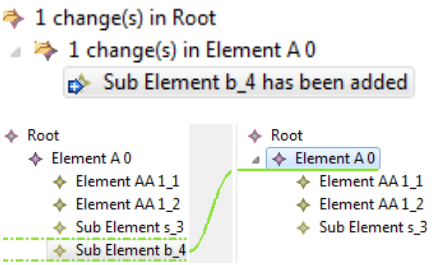
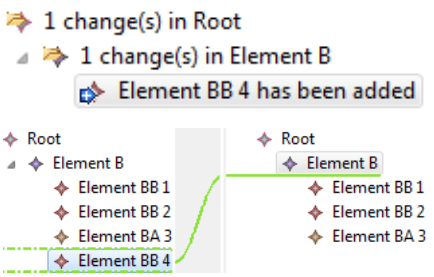
Escenarios de evolución mapeo uno a uno		Operación de adaptación
Condición	**Modelos de diferencias**	
ΔIm = N **ΔImc = 0** **ΔOmc =0** **ΔOm = 0**	**ΔIm** 	
Nota: Las diferencias de **ΔIm** y **ΔOm** son del tipo *MMDiff!ModelElementChangeLeftTarget*.		

El segundo de los escenarios de filtrado mediante agregación requiere especificar también una nueva regla de transformación. Este tipo de situaciones presenta las siguientes condiciones: **ΔIm = N, ΔImc = 0, ΔOm = N** y **ΔOmc >= 0**. En la tabla 5.13 se resume la condición de detección. En estas situaciones una nueva regla con un filtro se genera y su filtro opuesto se debe de establecer en la regla existente que transforma los elementos de entrada del mismo tipo. Si **ΔOmc = 0** se requiere una comprobación extra para verificar que realmente es un nuevo mapeo con filtrado. Podría ocurrir que el desarrollador ha creado un par de modelos ejemplos pero que no indican ninguna evolución y simplemente se han agregado N elementos de entrada que una regla previa ha convertido en N instancias de salida.

Tabla 5.13 Condiciones para la detección de un escenario de nueva regla de transformación con filtrado mediante agregación de instancias.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget	**ΔIm = N** **ΔImc** =0	**ΔOm = N*** **ΔOmc >= 0**
***Si ΔOmc=0 hay que comprobar en el modelo Etr si la nueva regla ya existe**		

Tabla 5.14 Ejemplo de un escenario de nueva regla de transformación con filtrado mediante agregación de instancias.

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condi-ción	**Modelos de diferencias**	
ΔIm = 1 **ΔImc = 0** **ΔOm = 1** **ΔOmc =0**	**ΔIm**  **ΔOm**  	**rule from_subElement_to_Element_BA {** *new filter* **from** a : MM_A!SubElement **(not a.isType_b())** **to** b : MM_B!Element_BA (name <- a.name) } *new rule* **rule from_subElement_to_Element_BB {** **from** a : MM_A!SubElement **(a.isType_b())** *new filter* **to** b : MM_B!Element_BA (name <- a.name) }
Nota: Las diferencias de **ΔIm** y **ΔOm** son del tipo *MMDiff!ModelElementChangeLeftTarget*.		

En esta situación también ocurre que **ΔIm = N, ΔImc = 0, ΔOm = N** y **ΔOmc = 0.** En estas situaciones hay que verificar si el mapeo ya existe previamente. La comprobación de si ya existe la nueva regla de transformación la realiza la función *comprobarExistenciaRegla*. Para poder diferenciar estas dos situaciones el sistema recoge el tipo de instancia de alguna de las agregaciones de **ΔIm** y el tipo de instancia de salida de alguna de las agregaciones de **ΔOm**. Para ello se utiliza la propiedad *leftElement* de las diferencias. A continuación se analiza en el **Etr** si existe una regla de transformación que tiene como entrada y salida las metaclases que se han agregado en el par de modelos de entrada. Si el mapeo no existe se considera una situación de evolución de nuevo mapeo con filtrado. Si el mapeo existe en el **Etr** se supone que realmente no es una situación de evolución, siendo un falso positivo. Otro escenario similar ocurre cuando se quiere tener dos reglas de transformación con las mismas metaclases de entrada y salida pero con diferentes asignaciones en las propiedades de las instancias de salida. Para poder diferenciar

esta situación, el desarrollador debe especificar que quiere un nuevo mapeo uno a uno con filtrado. Para ello actualmente se está trabajando en combinar el método TRANSEVOL con la especificación de alto nivel de la intención de mapeo [Luc16]. En la tabla 4.24 se presenta un ejemplo de escenario de filtrado con una nueva regla de transformación.

5.2.6 Modificación del contenedor de elementos de salida

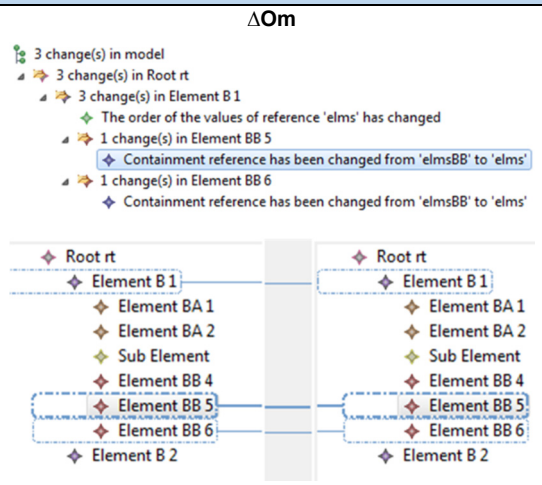
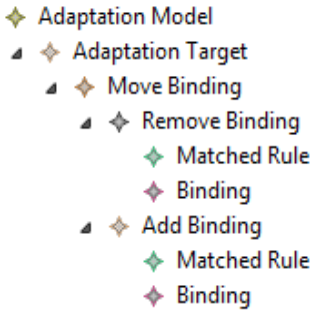
Este tipo de modificaciones normalmente son debidos a cambios en el diseño de los metamodelos y son solventados mediante co-evolución entre metamodelos y transformaciones. En aquellas situaciones donde la modificación de los contenedores de ciertas instancias es debido a cambios en la lógica de mapeo, la co-evolución con el metamodelo no se puede aplicar debido a que los metamodelos no presentan modificaciones. En estas situaciones, se debe eliminar el binding de la regla de transformación que genera los contendores previos, y crear una nueva asignación en la regla que genera los contenedores a donde se deben de mover las asignaciones. La operación de adaptación a implementar en estos escenarios en un *MMAdaptationGoal!MoveBinding*. Para detectar estas situaciones se utilizan diferencias del tipo *MMDiff!UpdateContainmentFeature*. En este tipo de evoluciones no hay cambios en los modelos de entrada (Δ**Im = 0**), mientras que si los hay en los modelos de salida, Δ**Om>0** Cuando se modifica el contenedor de un elemento también se generan modificaciones en el orden de las referencias. Este tipo de diferencias no se tienen en cuenta a la hora de realizar el análisis. En la tabla 5.15 se resume la condición de detección de escenario. Actualmente TRANSEVOL solo trabaja con escenarios que requieren mover una única asignación. En la tabla 5.16 se representa el diagrama de un modelo de diferencias con cambios de contenedor, la condición de detección y la operación de adaptación generada. En el modelo de diferencias se aprecia que dos elementos del tipo ElementB son movidos de la propiedad *elms* a *elmsBB*.

Para conocer el binding a eliminar y el nuevo binding a crear, el algoritmo utiliza las propiedades de alguna de las diferencias del tipo *MMDiff!UpdateContainmentFeature*. El algoritmo localiza el contenedor previo mediante la propiedad *rightTarget* de la diferencia *MMDiff!UpdateContainmentFeature*. Una vez conocido el contenedor previo el siguiente paso es conocer el binding a eliminar. Para ello, primero se localiza la regla que generó el contenedor en el modelo *Etr*. Y una vez localizada la regla donde se debe de eliminar el binding, se deduce el binding a eliminar utilizando la propiedad *leftElement*, que indica la referencia modificada. Para conocer el nuevo contenedor se utiliza el mismo proceso pero partiendo de la propiedad *leftTarget*, que indica cual es la nueva instancia contenedor. Y otra vez se utiliza el modelo *Etr* para localizar la regla a donde se debe de mover la asignación. Para conocer la propiedad donde se debe de realizar la asignación se utiliza la propiedad *leftElement*. En la figura 5.12 se representa la operación de adaptación a generar y su relación con las propiedades de la diferencias del tipo *MMDiff!UpdateContainmentFeature*.

Tabla 5.15 Condiciones para la detección de un escenario de modificación de contenedor.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!UpdateContainmentFeature	**ΔIm =** 0	**ΔOm = N**
Nota : Todas las diferencias deben de estar relacionadas con un solo binding		

Tabla 5.16 Ejemplo de escenario que especifica el cambio de contenedor junto con su operación de adaptación correspondiente.

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condición	**Modelos de diferencias**	
ΔIm = 0 **ΔOm = 2**	**ΔOm** 	
Nota: **ΔOm** son del tipo *MMDiff!UpdateContainmentFeature* y afectan a las mismas propiedades de un mismo tipo de contenedor.		

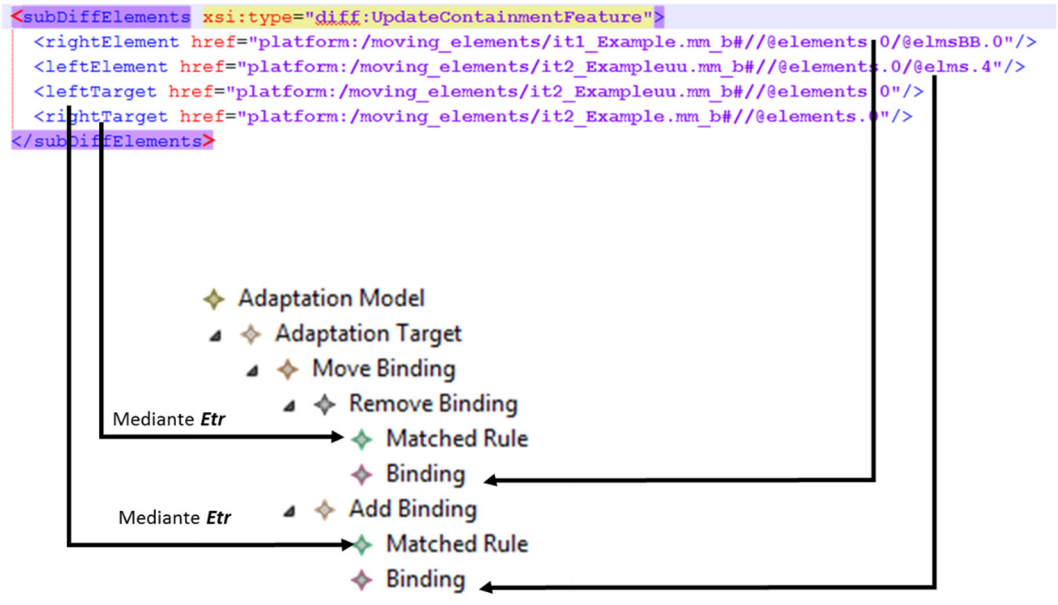


Figura 5.12 Relación entre la operación de cambio de asignación y la diferencia *MMDiff!UpdateContainmentFeature*.

5.2.7 Cambios en los atributos

Estos escenarios de evolución se dan cuando por decisiones de mapeo se decide cambiar el cálculo de los valores de los atributos de ciertas metaclases de salida. Para detectar este tipo de escenarios se utilizan diferencias del tipo *MMDiff!UpdateAttribute* en los modelos de salida. En estos escenarios no existen cambios en los modelos de entrada, los cambios solo se realizan en los modelos de salida. En la tabla 5.17 se presenta la condición de detección de escenario. En estos escenarios, la operación de adaptación corresponde a una modificación de una asignación, *MMAdaptationGoal!UpdateBinding*. TRANSEVOL no deduce el cálculo del valor de atributo, simplemente especifica que hay que modificar una asignación y genera una función *Helper* sin código que se utilizará para calcular los valores a asignar. La implementación de la función *Helper* queda en manos del desarrollador. En la tabla 5.18 se representa un escenario donde se especifica la modificación de un atributo. También se representa, mediante código ATL cual sería la modificación a realizar.

Tabla 5.17 Condiciones para la detección de cambios en los atributos.

Tipo de diferencia	**Condición para la detección del escenario**	
MMDiff!UpdateAttribute	**ΔIm =** 0	**ΔOm = N**
Nota : Todas las diferencias deben de estar relacionadas con un solo binding		

Tabla 5.18 Ejemplo de escenario que especifica el cambio de contenedor junto con su operación de adaptación correspondiente.

Escenarios de evolución mapeo uno a uno		**Operación de adaptación**
Condición	**Modelos de diferencias**	

ΔIm = 0
ΔOm = N
	ΔOm
	1 change(s) in Element B
	1 change(s) in Element BA 4_especial
	Attribute prueba : EString in Element BA 4_especial has changed
	Root / Element B / Element BB 1 / Element BB 2 / Element BA 3 / Element BA 4_especial
	Name 4_especial / Prueba 4_especial
	Root / Element B / Element BB 1 / Element BB 2 / Element BA 3 / Element BA 4_especial
	Name 4_especial / Prueba 4
		`rule from_subElement_to_Element_BA {`
		` from`
		`  a : MM_A!SubElement (not a.isType_b())`
		` to`
		`  b : MM_B!Element_BA (`
		`  name <- a.name`
		`  )`
		`}`
		`rule from_subElement_to_Element_BA_special {`
		` from`
		`  a : MM_A!SubElement (a.isType_b() )`
		` to`
		`  b : MM_B!Element_BA (` *asignación a modificar*
		`  name <- a.name + '_special'`
		`  )`
		`}`

Nota: **ΔOm** son del tipo *MMDiff!UpdateAttribute* y afectan a las mismas propiedades de un mismo tipo de contenedor.		

Para definir la operación de adaptación se requiere localizar en el ***Etr*** la regla que generó el elemento que requiere el cambio de atributo. El elemento sobre el cual se ha realizado el cambio se obtiene del atributo *leftElement* de una de las diferencias. El atributo a modificar se obtiene de la propiedad *attribute* de alguna de las diferencias de un *MMDiff!UpdateAttribute*.

5.2.8 Mapeos varios a varios

Respecto a los escenarios relacionados con reglas de transformación de varios a varios solo dos escenarios concretos son contemplados. Por un lado, se trata la evolución de una transformación de uno a varios a una transformación de varios a varios. En este tipo de escenario de evolución se agrega un nuevo patrón de elementos de entrada a una regla previa. El segundo tipo de escenario es la definición de una nueva regla de transformación varios a uno. A continuación se describen y analizan ambos escenarios.

El primero de los escenarios se determina cuando **∆Imc >= 1, ∆Im = N, ∆Om = N** y **∆Omc >=0,** siendo las diferencias de entrada del tipo *MMDiff!ModelElementChangeLeftTarget* y las diferencias de salida del tipo *MMDiff!UpdateAttribute*. Es decir, se realizan agregaciones en los modelos de entrada y modificaciones de atributos en los modelos de salida. Estas diferencias indican que un nuevo tipo de instancia previamente no contemplado genera cambios en instancias de salida generados por alguna regla de transformación. Debido a que **∆Imc >= 1,** se sabe que se están contemplado nuevos elementos de entrada, y como los **∆Om** son cambios en las asignaciones, se deduce que se ha agregado un nuevo patrón de entrada a una nueva regla, la cual es la afectada por los cambios en los atributos. La operación de adaptación requiere un nuevo patrón de entrada en la regla afectada y también define las asignaciones afectadas. Para deducir la regla y asignaciones afectadas se utilizan las diferencias de salida. De las modificaciones de atributo, *MMDiff!UpdateAttribute*, se utiliza el elemento afectado, representado por la propiedad *leftElement*, y se busca en el ***Etr*** la regla que lo generó. Esta regla es la afectada por el nuevo patrón de entrada. La modificación de la referencia nos da la información del *binding* relacionado con el elemento de entrada. Para buscar la regla afectada por el nuevo *binding* también se utiliza el ***Etr***. En este escenario todos los cambios de atributos se deben de dar en patrones de salida de una misma regla. Puede ocurrir que los cambios de atributo afectan a más de un tipo de patrón de salida. En la tabla 5.19 se resume la condición de detección de evolución de un mapeo 1 a N a 2 a N. En la tabla 5.20 se representa un escenario ejemplo de una evolución de un mapeo 1 a N a 2 a N.

Las condiciones para detectar un escenario de una nueva regla de mapeo de varios a uno es equivalente al de una nueva regla, con la excepción de que **∆Imc >= 1.** En este escenario, tanto las diferencias de entrada como de salida son agregaciones**.** En la tabla 5.21 se resume la condición de detección de escenario: **∆Imc >= 1, ∆Im = N, ∆Om = M** y **CC(∆Om)=1.** Si CC(**∆Im**)>**∆Imc** significa que existe algún filtrado en algunos de los patrones de entrada. Actualmente TRANSEVOL solo contempla CC(**∆Im**) = **∆Imc.** Este tipo de escenario es equivalente a un nuevo mapeo uno a uno con la diferencia de que **∆Imc >= 1.** Tanto el tipo de diferencias y las adaptaciones son equivalentes con la excepción de que por cada metaclase agregada se debe de generar un nuevo patrón de entrada. En el caso de que **∆Imc = 0** y CC(**∆Im**) > 1, la situación es la misma con la diferencia de que la situación es un mapeo varios a uno, pero con filtrado, debido a que los patrones de entrada ya se han utilizado previamente. De todas maneras este tipo de escenario se puede expresar mediante dos escenarios: 1) Una nueva regla 1 a 1 y 2) nuevo patrón de entrada.

Tabla 5.19 Condiciones para la detección de un escenario de evolución de un mapeo 1 a N a 2 a N.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget (Modelos de entrada) *MMDiff!UpdateAttribute* (Modelos de salida)	**ΔIm** = N	**ΔOm = M**
Nota : Todas las diferencias deben de estar relacionadas con un solo binding		

Tabla 5.20 Ejemplo de un escenario que especifica un escenario de evolución de un mapeo 1 a N a 2 a N.

Escenarios de evolución mapeo uno a uno		Operación de adaptación
Condición	**Modelos de diferencias**	
ΔImc=1 **ΔIm=1** **ΔOmc=0** **ΔOm=1**	**ΔIm** 1 change(s) in Root 1 change(s) in Element A Element AB 2 has been added List Model / Root / Element A / Element AA 1 / Element AB 2 — List Model / Root / Element A / Element AA 1 **ΔOm** 1 change(s) in Element B 0 1 change(s) in Element BA 1 Attribute prueba : EString in Element BA 1 has changed Model / Root / Element B 0 / Element BA 1 — Model / Root / Element B 0 / Element BA 1	Adaptation Model Adaptation Target NewInputPattern add Input Pattern Element Matched Rule fromAA2BA In Pattern
Nota: **ΔIm** son del tipo *MMDiff!ModelElementChangeLeftTarget* y **ΔOm** son del tipo *MMDiff!UpdateAttribute*.		

Tabla 5.21 Condiciones para la detección de un nuevo un mapeo N a 1.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget (Modelos de entrada) *MMDiff!UpdateAttribute* (Modelos de salida)	**ΔIm** = N **ΔImc >= 1**	**ΔOm = M** **CC(ΔOm)=1**
Nota : Todas las diferencias deben de estar relacionadas con un solo binding		

Cuando se quiere especificar en una solo iteración un mapeo varios a varios el método no diferencia si se quiere resolver mediante una regla de transformación de **N** a **N** o mediante **N** reglas de mapeo **1** a **1**. Para evitar esta ambigüedad el método TRANSEVOL recomienda especificar primero un mapeo N a 1 o 1 a N y en las siguientes especificaciones se definen modelos ejemplos que requieren agregar patrones de entrada o salida.

5.2.9 Agregando nuevos conceptos de dominio mediante perfiles

Una de las situaciones más habituales en las evoluciones de los sistemas empotrados es la necesidad de agregar nuevas abstracciones. En un sistema de DSDM para introducir nuevas abstracciones se agregan nuevas metaclases. Las nuevas metaclases se pueden agregar mediante una herencia o una asociación. Otra técnica muy utilizada es el extender el metamodelo mediante un perfil que permite aplicar diferentes estereotipos a las metaclases, de forma que se pueden expresar nuevos conceptos en los diseños sin modificar el metamodelo. Los perfiles UML, que permiten extender los metamodelos mediante el uso de estereotipos, han sido una de las claves del éxito de UML. El proyecto EMF Profiles [Lan12] reutiliza el concepto de los perfiles UML para extender metamodelos EMF. En este trabajo se analizan escenarios de evolución de abstracción donde se utilizan perfiles EMF y perfiles UML. En el caso de los perfiles UML la información de los estereotipos se guarda en el mismo fichero XMI que el diseño, ver figura 5.13. Al aplicar perfiles EMF los estereotipos aplicados se guardan en un segundo fichero XMI, ver figura 5.14.

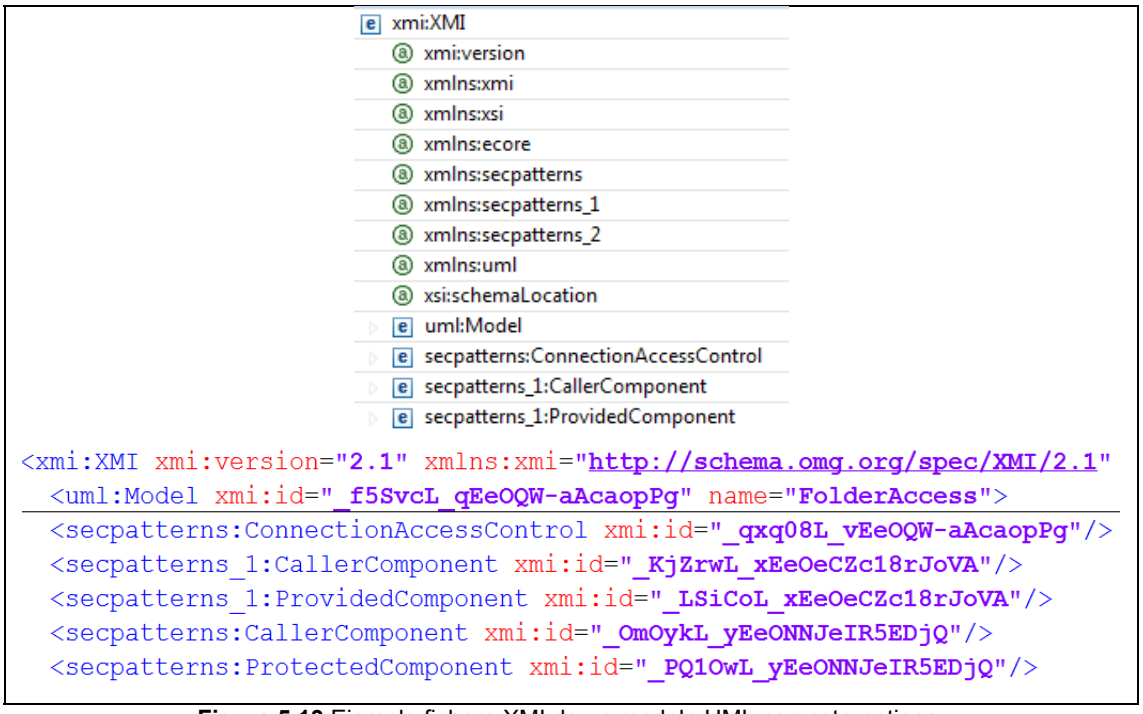


Figura 5.13 Ejemplo fichero XMI de un modelo UML con estereotipos.

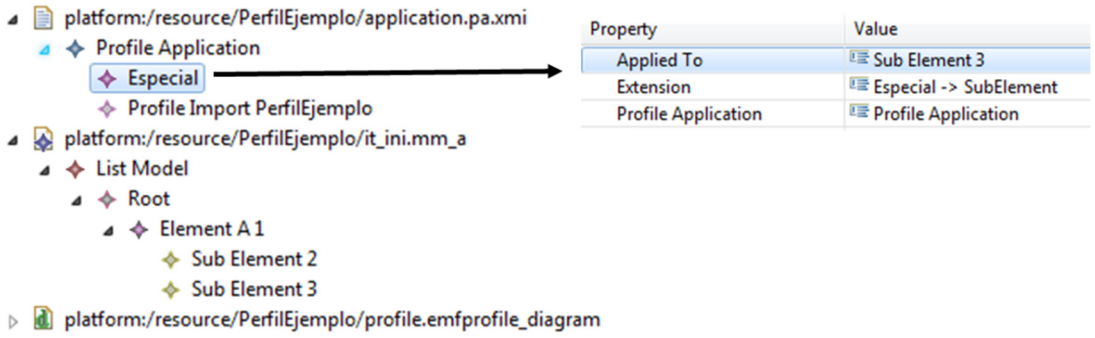


Figura 5.14 Ejemplo fichero XMI de un modelo EMF, *it_ini.mm_a*, y fichero XMI con la información de los estereotipos EMF aplicados sobre el modelo *it_ini.mm_a*.

El agregar una nueva metaclase está relacionado directamente con los escenarios de evolución de creación de reglas de mapeo (1 a 1, 1 a N, N a 1, N a N) y agregación de patrones de entrada y salida. En cambio, el empleo de estereotipos está relacionado con la necesidad de filtrado y modificación de atributos. El utilizar nuevos estereotipos en los modelos de diseño requiere adaptar la lógica de mapeo. La información extra ofrecida por los estereotipos se utiliza en las reglas de mapeo para asignar nuevos valores en los atributos de los elementos de salida. La información de los estereotipos también se puede utilizar para diferenciar diferentes instancias de una misma metaclase, y aplicar así diferentes reglas de transformación a instancias del mismo tipo de metaclases. Por lo tanto, el uso de estereotipos conlleva escenarios de evolución que requieren cambios en las asignaciones de atributos y referencias, y también aplicar filtrados en las reglas de transformación existentes. La detección de los escenarios de evolución debido a los estereotipos es exactamente equivalente a las detecciones de evolución de filtrado y de cambios de atributos analizadas previamente. La única diferencia con los escenarios de filtrado y cambios de atributos es que siempre existen diferencias en los modelos de entrada debido a la aplicación de los estereotipos, por lo tanto Δ**Imc>0**.

Este tipo de escenarios se caracteriza por tener diferencias en los modelos de entrada del tipo *MMDiff!ModelElementChangeLeftTarget,* los cuales están relacionados con la adición de estereotipos. El algoritmo TRANSEVOL cuando detecta diferencias de entrada del tipo *MMDiff!ModelElementChangeLeftTarget* analiza si las instancias agregadas son aplicaciones de estereotipos. Si las instancias agregadas no están relacionadas con estereotipos el algoritmo funciona como se ha descrito hasta ahora. En cambio si TRANSEVOL detecta que algunas de las instancias agregadas son aplicaciones de estereotipos funciona ligeramente diferente. Como el algoritmo de detección no requiere la información del estereotipo para el análisis, descarta dicha información en el análisis, dependiendo del escenario se utiliza la información de la diferencia creada por el estereotipo para conocer los patrones de entrada. TRANSEVOL solo permite en cada modelo ejemplo aplicar estereotipos sobre instancias de una única metaclase. Los escenarios con estereotipos contemplados por TRANSEVOL son los siguientes:

1. Modificar el cálculo de atributos utilizando los datos de los estereotipos.
2. División de una regla para mapear de forma diferente las instancias estereotipadas.
3. Filtrado de mapeo debido a la aplicación de estereotipos.

En el primero de los escenarios se aplican los estereotipos con el objetivo de dotar a las instancias de un tipo con más información en los modelos de entrada. Esta nueva información se utilizará para calcular los valores de los atributos de las instancias de salida generadas. En estas situaciones las reglas de mapeo se deben de modificar. Concretamente se deben de modificar las asignaciones en los patrones de salida que utilizan los nuevos datos. Este tipo de escenarios están relacionados con los escenarios de cambio de atributos. Los escenarios de cambio de atributo no presentan cambios en los modelos de entrada, Δ**Im = 0**, mientras que Δ**Om = N,** con diferencias del tipo *MMDiff!UpdateAttribute*. Al especificar este mismo escenario pero

con estereotipos ∆**Im = M**, donde las diferencias son del tipo *MMDiff!ModelElementChangeLeftTarget*, que son agregaciones de instancias que especifican la aplicación de estereotipos. En estos escenarios el algoritmo debe de detectar la situación y deducir la asignación a modificar en la regla afectada. La condición de detección es equivalente al escenario sin estereotipos, pero ∆**Im = M**. TRANSEVOL cuando detecta que en ∆**Om** las diferencias son del tipo *MMDiff!UpdateAttribute* y que todas las diferencias de entrada están relacionadas con la aplicación de estereotipos aplica el algoritmo para la búsqueda de la asignación afectada de la misma forma que si no hubiera estereotipos. Es decir, no utiliza las diferencias de ∆**Im,** ya que mediante **Etr** y las diferencias de salida es capaz de buscar la asignación afectada. Actualmente TRANSEVOL solo indica que asignación debe de modificarse, y no deduce la operación de asignación. Las diferencias creadas por los estereotipos se podrían utilizar para establecer el código de la asignación. En la tabla 5.22 se resume la condición de detección de este tipo de escenario. En este tipo de escenarios hay que modificar en los modelos ejemplos de salida todas las instancias relacionadas con el cambio de atributos.

Tabla 5.22 Modificación de asignaciones debido al uso de estereotipos.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget (Modelos de entrada) *MMDiff!UpdateAttribute* (Modelos de salida)	∆**Im** = N*	∆**Om = M****
* Las diferencias de entrada no se tienen en cuenta en el cálculo de la operación de adaptación. **Se modifican todas las instancias de salida relacionadas con la regla a modificar.		

Existen situaciones donde solo se modifican las instancias de salida relacionadas con las instancias de entrada estereotipada. En estas situaciones TRANSEVOL entiende que el desarrollador quiere especificar que el mapeo de las instancias estereotipadas se realiza de forma diferente al de las no estereotipadas. En estos casos la operación deducida es la división de una regla en dos, donde cada regla tiene un filtro contrario. En estos casos la regla que trata con las instancias estereotipadas queda intacta, mientras que la nueva puede requerir eliminación de patrones de salida, agregación de nuevos patrones de salida o modificaciones de asignaciones. En ∆**Om** las diferencias puede ser de tres tipos: eliminación de instancias, agregación de entradas o modificación de atributos. Para saber que se requiere una división de regla TRANSEVOL detecta que no todas las instancias de un patrón de salida de una regla han sido modificadas, solo parte de ellas. En la tabla 5.23 se resume la condición de este escenario. Las diferencias en ∆**Im** son aplicaciones de estereotipos**.** En estos casos las diferencias de entrada se utilizan para detectar el patrón de entrada sobre el cual se debe de aplicar el filtro.

Tabla 5.23 Condición para la división de una regla en dos debido a la aplicación de estereotipos y la modificación de atributos de salida.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget (Modelos de entrada) *MMDiff!UpdateAttribute* (Modelos de salida) *MMDiff!ModelElementChangeLeftTarget* (Modelos de salida) *MMDiff!UpdateAttribute* (Modelos de salida)	∆**Im** = N*	∆**Om = M****
* Las diferencias de entrada no se tienen en cuenta en el cálculo de la operación de adaptación. ** No se modifican todas las instancias de salida relacionadas con una regla, solo algunas.		

El último de los escenarios se utiliza para especificar la evolución de un mapeo N a N a un mapeo N a N con filtrado, debido a que no transforma todas las instancias de un tipo, sino solo aquellas estereotipadas. Es decir, se debe de aplicar un filtro a la regla para que deje de mapear las instancias de entrada no estereotipadas de un tipo. En este tipo de escenario se eliminan las instancias de salida que corresponden a los elementos de entrada que no se quieren mapear. En estos escenario las diferencias del modelo Δ**Im** son agregaciones de estereotipos que se representan mediante diferencias del tipo *MMDiff!ModelElementChangeLeftTarget*. Las diferencias del modelo Δ**Om** son eliminaciones de instancias *MMDiff!ModelElementChange-RightTarget*. En estos casos CC(Δ**Om**)>=1. La operación de adaptación en estos casos es la aplicación de un filtro a un patrón de entrada. El patrón de entrada en este caso está relacionado con la metaclase de las instancias a las cuales se ha aplicado el estereotipo. La tabla 5.24 resume la condición de detección de este tipo de escenarios.

Tabla 5.24 Condiciones para la detección de un escenario de evolución de filtrado de mapeo debido a la aplicación de estereotipos.

Tipo de diferencia	Condición para la detección del escenario	
MMDiff!ModelElementChangeLeftTarget (Modelos de entrada) *MMDiff!ModelElementChangeRightTarget* (Modelos de salida)	Δ**Im** = N*	Δ**Om = M** CC(Δ**Om**)>=1
* Las diferencias de entrada están relacionadas con la aplicación de estereotipos.		

TERCERA PARTE
Validación

6. Validación de la propuesta TRANSEVOL

En este capítulo se presenta el sistema de DSDM seleccionado como caso de estudio principal en este trabajo de investigación y las validaciones realizadas. Primero se describe el proceso de validación utilizado. A continuación se realiza un resumen de las validaciones iniciales realizadas sobre el método TRANSEVOL. Después se describe el caso de estudio real *fromUML2C,* con el cual se han realizado las validaciones frente a cambios debidos en los RNF. El sistema de generación de código, basado en el DSDM, *fromUML2C* presenta las características necesarias para validar los objetivos de este trabajo de investigación:1) es del ámbito de los sistemas empotrados 2) Utiliza UML y perfiles UML y 3) la transformación M2M presenta una complejidad alta. Una vez presentado el sistema de DSDM f*romUML2C* los siguientes dos apartados describen la adaptación realizada en la transformación M2M de f*romUML2C* utilizando TRANSEVOL frente a dos cambios en los RNF. Dos situaciones relacionadas con los cambios en los RNF de dominio se han validado. Primero se presenta la validación de la adaptación de transformaciones M2M frente a cambios en lógica de mapeo debido a cambios en los RNF de dominio. Y a continuación se describe la validación de la adaptación de las transformaciones frente a la aplicación de un nuevo perfil para representar RNF de seguridad. El objetivo es validar la correcta adaptación de las transformaciones M2M debido a los nuevos requisitos en la lógica de mapeo.

6.1 Proceso de validación

Para realizar la validación del método TRANSEVOL se ha implementado un prototipo de herramienta mediante JAVA y EMF. La herramienta se puede ejecutar fuera del IDE eclipse como aplicación *standalone*. Debido a que para obtener tanto las trazas de ejecución como los modelos de diferencias y ejecutar las transformaciones ATL es más flexible utilizar el IDE eclipse, las validaciones se han realizado siempre desde el propio IDE eclipse. Concretamente el prototipo de herramienta se ha ejecutado siempre en el bundle de modelado eclipse versión MARS. El metamodelo *MMAdaptationGoal* se basa en el metamodelo ATL versión 3.3. Para expresar las diferencias se ha utilizado el metamodelo EMFDiff 2.1. Para generar los modelos de diferencias se ha utilizado el plugin EMFCompare 1.3 del bundle de modelado de la versión Galileo de eclipse. Para realizar las diferentes validaciones del método TRANSEVOL se ha definido el siguiente proceso de validación:

1. Selección de una transformación M2M.
2. Definición el escenario de adaptación de forma semántica en un documento.
3. Selección de un modelo de entrada.
4. Generación del modelo de salida ejecutando la transformación legada.
5. Obtención de la traza de ejecución.

6. Modificación de los modelos de entrada y salida para demostrar el nuevo requisito de transformación.
7. Ejecutar la transformación con el modelo ejemplo de entrada y comprobar si es una evolución regular. Si es una evolución regular no ejecutar TRANSEVOL.
8. Generación de los modelos de diferencias.
9. Ejecución de la herramienta TRANSEVOL.
10. Implementación de las operaciones de adaptación deducidas por TRANSEVOL.
11. Validar la nueva transformación M2M con los modelos ejemplos.
    a. Ejecutar la nueva transformación M2M con el modelo ejemplo de entrada.
    b. Comparar la salida generada automáticamente con el modelo de salida ejemplo.
12. En caso de que sea posible, cuando no hay eliminación de reglas de transformación, eliminación de patrones o algunos filtrados, se realiza una validación regresiva.
13. Se generan parejas de modelos de entrada y salida para realizar una mayor cobertura de validación.

Se han realizado validaciones tanto a transformaciones M2M endógenas como endógenas. La herramienta TRANSEVOL solo se ha validado frente a transformaciones ATL.

6.2 Validaciones iniciales

Para validar el correcto funcionamiento de la propuesta, diferentes validaciones se han llevado a cabo. Todas las validaciones se han realizado sobre transformaciones M2M implementadas bajo el lenguaje de transformación ATL, y utilizando el proceso descrito previamente. Mientras se desarrollaba la herramienta prototipo sencillos escenarios de validación se diseñaban para cada una de los escenarios de adaptación. Una vez desarrollada la herramienta, con el objetivo de realizar una primera fase de validación, y antes de validar el sistema frente a un sistema real de DSDM frente a evoluciones de RNF de dominio, se escogieron 8 transformaciones M2M ejemplo de la página de documentación de ATL (*http://www.eclipse.org/atl/documentation/*). Dos de las transformaciones escogidas eran endógenas, mientras que las demás eran exógenas. Las transformaciones endógenas seleccionadas eran refinamientos de modelos: la aplicación del patrón “Bridge” en diagramas UML [Lan05] y la transformación de una máquina de estados jerárquica a plana. La tabla 6.1 resume los resultados obtenido en esta primera fase de validación, para cada caso de estudio. En cada caso de estudio, se define la dimensión inicial de la transformación M2M legada (número de reglas de transformación y funciones HELPERS). En algunos de los ejemplos utilizados la transformación M2M se desarrolló desde cero, en estos casos la dimensión inicial es 0. En la tabla también se determina el número de iteraciones utilizadas para derivar las operaciones de adaptación y finalizar la transformación. En cada iteración se utiliza una pareja de modelos ejemplo. Los metamodelos utilizados en los casos de estudio de la tabla 6.1, están definidos utilizando el meta-metamodelo Ecore de EMF.

Tabla 6.1 Resultados de las primeras validaciones.

Transformación M2M	Tipo	Dimension	Operaciones de adaptación deducidas	Número de iteraciones
StateMachine to flattened	Endógena	7 matched rules	2 agregaciones de filtro en patrones de entrada. 1 nueva regla con filtrado 2 actualizaciones de asignaciones 2 nuevas reglas 1 a 1 junto con dos nuevas asignaciones	4
ListMetamodel Refactoring	Exógena	0	2 nuevas reglas de mapeo 1 a 1 con sus asignaciones correspondientes	2
Bridge pattern	Endógena	8 matched rules	1 nueva regla con filtrado	1
Families to person	Exógena	0	1 nueva regla de mapeo 1 a 1 1 nueva regla con filtrado	3
Tree to Node	Exógena	0	2 nuevas reglas de mapeo 1 a 1 1 Agregación de filtrado	2
TreeToList	Exógena	0	2 nuevas reglas de mapeo 1 add filter to input patter	3
Port example	Exógena	0	2 add matched rules 1 split rule (with the filters)	3
Side effect example	Exógena	0	Evolución de un mapeo 1 a 1 a 1 a N 3 reglas de mapeo	4

Las validaciones de las adaptaciones de las transformaciones M2M se realizaron utilizando el proceso de validación presentado previamente. En los 8 casos de estudio de la tabla 6.1 las transformaciones generadas fueron correctas. Las transformaciones de la tabla 6.1 no contemplan los escenarios de eliminación de mapeo, modificación de atributos y las relacionadas con mapeos N a N. La validación de este tipo de escenarios se ha realizado creando pequeños ejemplos utilizados en el periodo de desarrollo de la herramienta prototipo.

6.3 Validación ante evoluciones de RNF en una transformación M2M legada del ámbito de los sistemas empotrados

Durante el desarrollo de la herramienta TRANSEVOL una serie de sencillos modelos ejemplo se han utilizado para validar la detección y generación de las diferentes operaciones de adaptación. Los escenarios utilizados se pueden considerar ejemplos de juguete, por ello una validación en un escenario real del ámbito de los sistemas empotrados y en una transformación M2M legada de mayor dimensión es necesaria. Para una primera validación en el contexto de los sistemas

empotrados la transformación *fromUML2SimpleC* [Agi14a] ha sido seleccionada. La transformación *fromUML2SimpleC* transforma modelos de diseño de aplicaciones para sistema empotrados en aplicaciones ANSI-C. En el diseño de las aplicaciones se utiliza UML junto con el perfil MARTE para expresar conceptos de los sistemas empotrados. El objetivo de este apartado es validar el método TRANSEVOL ante dos cambios en los RNF. Esto cambios en los RNF requieren adaptar la transformación M2M ante cambios en la lógica de mapeo y por la aplicación de estereotipos. Estos dos escenarios de adaptación a validar son el objetivo principal de este trabajo de investigación.

6.3.1 Sistema de generación de código DSDM fromUML2SimpleC

El sistema de generación de código *fromUML2SimpleC* ha sido utilizado para realizar la validación de TRANSEVOL frente a evoluciones de RNF en un escenario real e industrial, complejo y de una dimensión considerable. *fromUML2SimpleC* es un sistema de DSDM para la generación de código ANSI-C de modelos de componentes UML para sistemas empotrados. Este sistema fue presentado en [Agi12]. En el anexo se presente una versión reducida del capítulo 19 del libro [Agi14a], donde se describe un ejemplo de uso completo de dicho sistema de generación de código. El sistema de generación de código combina los paradigmas de DSDM y CBD (Component Based Development) de modo que permite especificar la arquitectura software de sistemas de control concurrentes basado en componentes y generar automáticamente el código en ANSI-C desplegable en plataformas empotradas. El diseño se realiza mediante el lenguaje de modelado UML. Para poder utilizar UML con una semántica adecuada que permita especificar plataformas de ejecución se ha extendido mediante perfiles UML, concretamente se ha utilizado el perfil MARTE [Mar11]. Los diseños UML de las aplicaciones se convierten en código ANSI-C mediante una transformación M2M y otra transformación M2T. Estas transformaciones implementan las buenas prácticas y patrones de diseño necesarios para poder tratar con conceptos de componentes software en C [Sche93]. Estas transformaciones se pueden aplicar a la generación de código de múltiples productos. En este apartado se describe la arquitectura del sistema de DSDM y la generación de código con el objetivo de dimensionar la complejidad del sistema de DSDM utilizado como caso de uso en la validación final del método TRANSEVOL.

El proceso de generación de código se ha dividido en dos etapas, ver figura 6.1. Primero se utiliza una transformación M2M, donde los modelos de componentes, basados en el metamodelo *UML2+MARTE*, se transforman en modelos *SimpleC*, basados en el metamodelo *SimpleC*. *SimpleC* es un metamodelo que permite expresar un subconjunto de ANSI-C. En una segunda etapa una plantilla, implementa en XPAND2, es aplicada a los modelos *SimpleC* para generar el código ANSI-C. El objetivo de esta separación en etapas es doble, por un lado, simplifica el proceso de generación de código, y por otro, las transformaciones M2T pueden reutilizarse en otros proyectos.

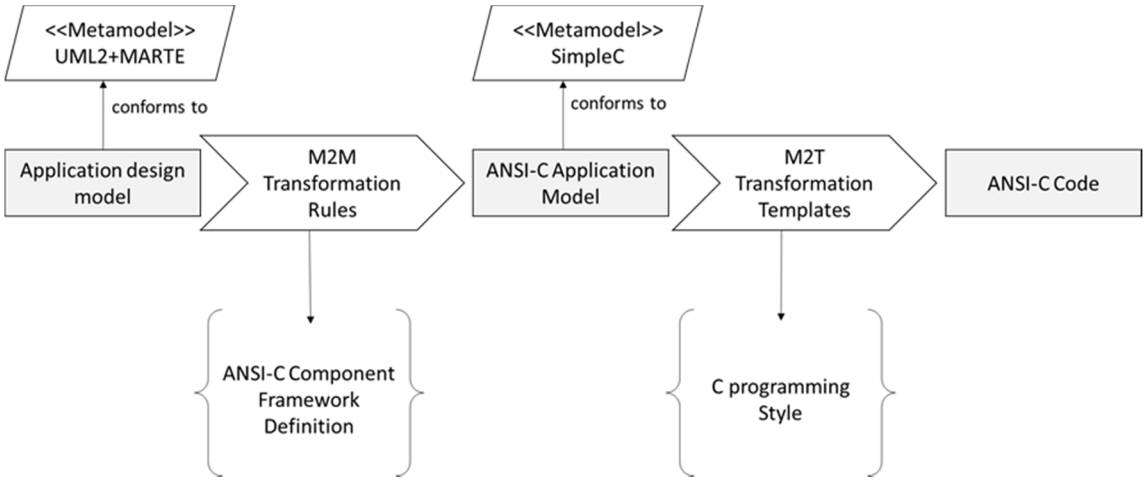


Figura 6.1 Proceso de generación de código *fromUML2C*.

A continuación se enumeran las herramientas y tecnologías utilizadas para la implementación del sistema de generación de código. Para el metamodelado se ha utilizado EMF. Para el metamodelo UML se ha utilizado la versión 2.3 basada en EMF. El metamodelo *SimpleC* ha sido definido e implementado en un proyecto EMF. Para el modelado UML de la arquitectura software se ha utilizado la herramienta Papyrus 1.12. Esta herramienta se puede utilizar tanto como plugin de eclipse o como una adaptación de eclipse. Este plugin además de permitir un diseño visual de diagramas UML, presenta una implementación del perfil UML-MARTE, permitiendo estereotipar diseños UML mediante el perfil MARTE. Para las transformaciones M2M se ha utilizado ATL. Para las transformaciones M2T se ha utilizado XPAND2 de OpenArchitectureWare. El sistema de DSDM consta de cuatro proyectos:

a) *fromUML2SimpleC*: Proyecto ATL donde se implementa la transformación M2M. Esta transformación convierte modelos de componentes UML estereotipados y diseñados conjuntamente con el perfil MARTE en modelos SimpleC. La transformación M2M esta implementada mediante el lenguaje ATL y utiliza el mecanismo de superimposición de ATL para implementar y ejecutar de una forma incremental la transformación.
b) *MMSimpleC*: Proyecto EMF donde se define el metamodelo *SimpleC*, el cual define un subconjunto del lenguaje de programación ANSI-C.
c) *SimpleC2Code*: Proyecto XPAND2 que permite generar código ANSI-C teniendo como entrada modelos SimpleC.
d) *KOBRAProfile*: Debido a que UML no diferencia entre instancia de componente y tipo de componente ha sido necesario crear un proyecto UML-Profile de Papyrus para la definición de los estereotipos para el uso de ambos conceptos. El diseño de los componentes UML se ha realizado utilizando la metodología KOBRA [Atk00], donde es fundamental diferenciar entre instancia y tipo de componente. A pesar del nombre del proyecto solamente se ha implementado un subconjunto de estereotipos de la metodología KOBRA.

6.3.2 Transformación M2M fromUML2SimpleC

En el proyecto *fromUM2SimpleC* se implementa la transformación M2M que transforma modelos UML en modelos SimpleC. *fromUM2SimpleC* es un proyecto ATL. En la carpeta *transformations* se encuentran todas las reglas de transformación M2M. En concreto son 8 ficheros ATL. En la carpeta *library* están implementadas las funciones de ayuda para las reglas de transformación. Las funciones de ayuda que trabajan con conceptos del perfil MARTE se han agrupado en el fichero *UMLMarteBaremetalPlatformHelper.atl*. Las que trabajan con los conceptos de instancias de componentes se encuentran en el fichero *UMLComponentInstancesHelper.atl*. Los conceptos referentes a la definición de la estructura interna del componente se encuentran en *UMLComponentTypeHelper.atl* y *UMLComponentTypeInternalHelper.atl*,y los relativos a sus interfaces en *UMLComponentTypeInterfaes.atl*. Finalmente han sido necesarias ciertas funciones de ayuda para navegar en modelos SimpleC, localizadas en *SimpleCHelper.atl.*

En la carpeta *metamodels* se localizan el metamodelo SimpleC, *MMSimpleC*, y el perfil UML-KOBRA (*KOBRA.profile.uml*). Estas dos metamodelos también se localizan en las carpetas de sus respectivos proyectos. En la carpeta *models* se incluyen los diferentes modelos de entrada y salida de la transformación M2M. No es obligatorio que estos modelos se sitúen en esta carpeta, en caso contrario hay que indicarlo en la configuración de la ejecución de la transformación. En esta carpeta se encuentra un diseño de la arquitectura software de un sistema básico de control automático de puertas deslizantes (ficheros *doorSPL.uml* y *doorSPL.di*). Este modelo ha sido el utilizado como ejemplo de entrada para la transformación M2M.

La transformación M2M consta de 8 ficheros ATL, 40 reglas de mapeo, 30 reglas de mapeo lazy y 44 funciones de ayuda. La transformación M2M se realiza de forma incremental mediante el mecanismo de superimposición de ATL [Wag10]. La configuración del orden para la superposición está registrada en la configuración del ejecutor de la transformación ATL (*ComponentType2SimpleC* ATL Transformation Launcher).

6.3.3 Validación de la adaptación de transformaciones M2M frente a cambios en lógica de mapeo debido a cambios en los RNF de dominio: Migración de API de concurrencia

En esta apartado los resultados obtenidos al aplicar la herramienta TRANSEVOL a la transformación *fromUML2SimpleC* ante una migración del API de concurrencia son presentados. El objetivo de la validación de este apartado es validar la herramienta TRANSEVOL frente a un cambio en la lógica de mapeo sin que los metamodelos cambien.

El escenario de adaptación al que se enfrenta la herramienta TRANSEVOL es la necesidad de cambiar el API de concurrencia. La migración de API de concurrencia normalmente es debido a cambios en la plataforma de ejecución, o debida a la necesidad de cambiar de planificador de tareas para poder responder adecuadamente a los requisitos temporales. En este caso,

la migración del API requiere pasar de un planificador y API propio a *freeRTOS* [Deh09]. El sistema DSDM seleccionado utiliza el perfil MARTE para expresar la concurrencia del sistema, concretamente el paquete SRM de MARTE. En los diseños actuales utilizados como entrada en las transformaciones *fromUML2Simple* la concurrencia ya se especifica, y por ello no es necesario utilizar nuevos elementos de metamodelado para expresar el nuevo API de concurrencia en los modelos PIM. En este contexto, la decisión de negocio puede ser ofrecer una única transformación que pueda generar el código para ambas soluciones, o migrar la transformación para que solo genera el código para la nueva plataforma. En este trabajo no se ha validado el funcionamiento de TRANSEVOL para situaciones que contemplan ofrecer variabilidad. Todo indica que TRANSEVOL se puede utilizar correctamente en estas situaciones pero hace falta un trabajo de validación para ello. En este caso de estudio se supone que la decisión de negocio es una migración del producto.

En la solución *fromUML2SimpleC* se especifica cuáles son las tareas concurrentes y sus características en el diseño de arquitectura. En el diseño, también se especifica la plataforma de ejecución y el planificador, pero los detalles de implementación relacionados con el API de concurrencia están ofuscados en la propia transformación. Por lo tanto, en este caso los nuevos requisitos de mapeo se especifican, principalmente, mediante modificaciones en el modelo de salida.

Para poder ejecutar la herramienta TRANSEVOL y obtener las adaptaciones a implementar primero se deben de generar los modelos ejemplos. Para ello, lo primero es obtener una pareja de modelos de entrada y salida que representan el funcionamiento inicial de la transformación M2M legada. En esta validación el diseño UML del controlador de una puerta automática y su correspondiente modelo SimpleC fue seleccionado. El siguiente paso es deducir las modificaciones a realizar en la pareja de modelos para demostrar el nuevo requisito de transformación. La especificación de los modelos ejemplos depende de cómo se traten los RNF de concurrencia en la arquitectura del sistema de DSDM. En este caso, a pesar de que la metodología de diseño de arquitectura requiere crear un modelo del nuevo planificador en el paquete "*Platform Provider*", realmente el único cambio contemplado en el diseño ocurre en el modelo de tareas, donde se modifica el atributo que indica el tipo de tareas periódicas. En este caso, pasan de ser tareas del tipo "*PeriodicTask*" a "*xPeriodicTask*". Este cambio de atributo es el único a realizar en el modelo de entrada. Los modelos de salida SimpleC requieren modificaciones para poder ejecutarse sobre el API freeRTOS. Los cambios a realizar en el código, y por consiguiente, en el modelo *SimpleC* de salida, los decide el experto en desarrollo software para sistemas empotrados. El ingeniero de sistemas empotrados modifica el modelo *SimpleC* del controlador de puerta para que se ejecute sobre *freeRTOS*. Los cambios a realizar en los modelos *SimpleC* que representan las diferentes aplicaciones son los siguientes:

1. Sustituir las definiciones de las funciones del API de concurrencia.
2. Sustituir en todas las instancias de componentes los includes del API previo por los de *freeRTOS*.

3. Modificar el código del bucle de ejecución principal ubicado en el módulo *AppOS_TaskModel* de la instancia *AppOS*.
    a. Adaptar las funciones de creación de tareas concurrentes al nuevo API.
    b. Adaptar el código de llamada al planificador de tareas. Este cambio implica modificar el bucle principal.
4. Adaptar el código de las tareas de cada instancia al nuevo API.

En este escenario de adaptación, para poder ejecutar la herramienta TRANSEVOL correctamente, los cambios se distribuyen en 4 modelos ejemplos de salida, de forma que la herramienta TRANSEVOL se aplica incrementalmente en cuatro pasos.

El primer paso es especificar la definición de las nuevas funciones del API de freeRTOS. En este caso existe la posibilidad de evolucionar el modelo de diseño de dos formas diferentes. La primera posibilidad es modificar los nombres de las funciones del modelo de plataforma del diseño UML. Se modifican los nombres de las operaciones de la clase *TaskScheduler* estereotipadas con la anotación <<Scheduler>>, pasando por ejemplo de *addTask* a *vCreateTask*. En este caso, además de modificar también se debe de agregar la función *vTaskDelay*. Debido a que existen dos tipos de modificaciones estos cambios se deben de especificar consecutivamente en dos iteraciones, cada una con sus modelos ejemplo. Si se quisiera ofrecer la posibilidad de generar los productos software en ambas plataformas, no se modificaría el modelo de plataforma, sino que se crearía un segundo. En este segundo caso, se tendrían que agregar funcionalidades de variabilidad en el sistema de DSDM. En este trabajo no se ha abordado esta problemática. En la figura 6.2 se indican las funciones modificadas en el diseño UML.

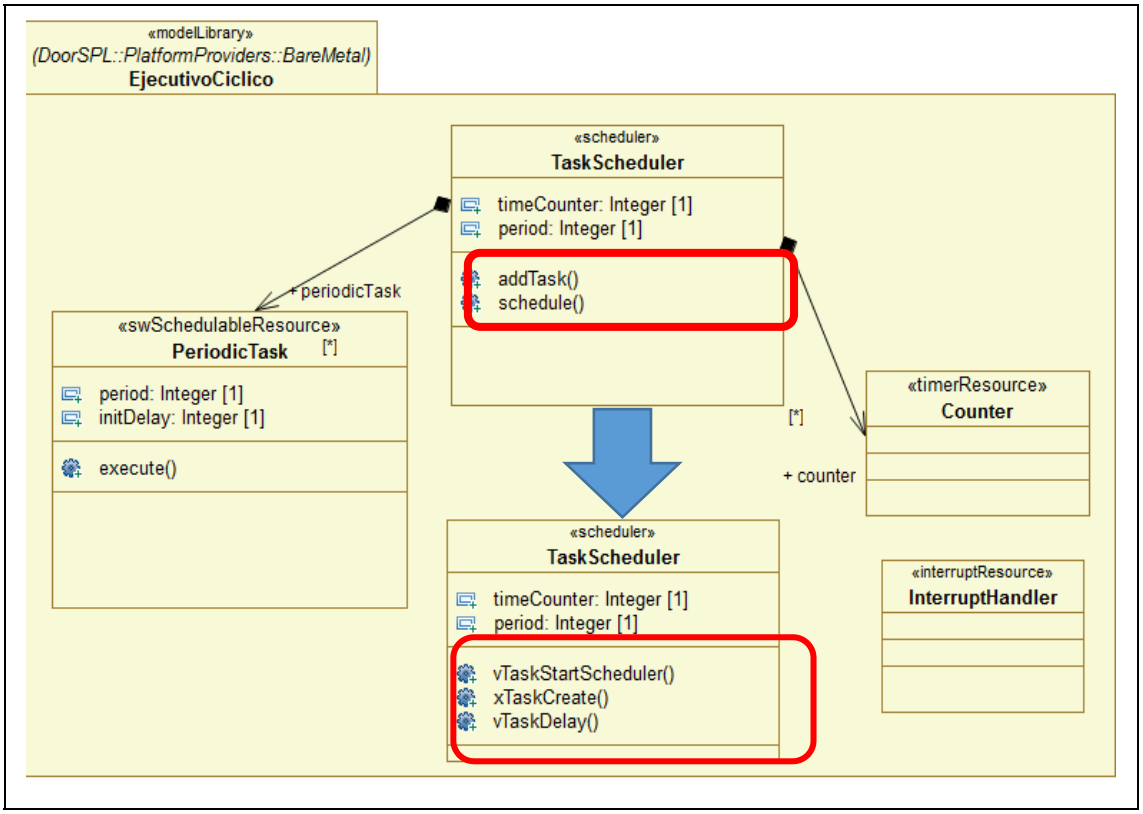


Figura 6.2 Modificación de los nombres de las funciones del API de concurrencia en el modelo de plataforma del operador de puertas (**ΔIm**).

Tras modificar los atributos del modelo de plataforma, se modifica el modelo de salida para integrar en el modelo SIMPLEC las nuevas funciones de freeRTOS. En la primera iteración el modelo de salida presenta dos modificaciones en nombres de funciones. En la segunda iteración se agrega la función *vTaskDelay* al modelo SIMPLEC. En la figura 6.3 se recogen las modificaciones realizadas en el modelo de salida en las dos iteraciones. Cuando TRANSEVOL analiza las diferencias de ambas iteraciones no detecta ninguna operación de adaptación. Esto es debido a que se ha especificado una evolución regular que no requiere ningún cambio. Para comprobar esta situación el desarrollador de transformaciones ejecuta en ambas iteraciones la transformación con su correspondiente modelo de entrada modificado, y comprueba que el modelo de salida generado es igual al modelo ejemplo de salida. Tras ejecutar la transformación M2M legada se aprecia como el modelo de salida generado ha creado en el modelo SimpleC las definiciones de las funciones del API freeRTOS. Por lo tanto, este cambio es un cambio regular, que no requiere ninguna adaptación en la transformación. Este escenario presenta en la primera iteración diferencias del tipo *MMDiff!UpdateAttribute*, concretamente Δ**Im** = 2 y Δ**Om** = 2. En la segunda iteración las diferencias son del tipo agregación *MMDiff!ModelElementChangeLeftTarget*, concretamente, Δ**Im** = 1 y Δ**Om** = 1. TRANSEVOL no presenta un escenario de adaptación para estas condiciones, y por eso deduce que no es necesaria ninguna adaptación. Este escenario es una evolución regular que no requiere ninguna operación de adaptación y que TRASNEVOL detecta correctamente.

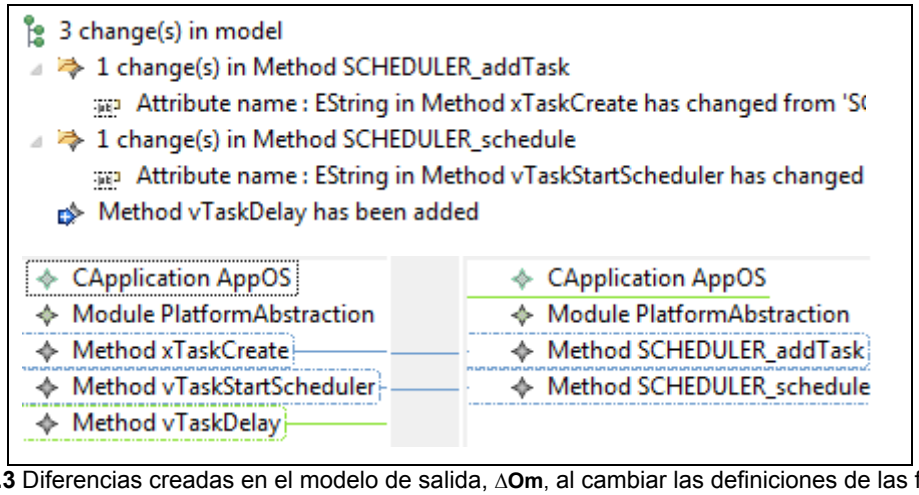


Figura 6.3 Diferencias creadas en el modelo de salida, Δ**Om**, al cambiar las definiciones de las funciones de plataforma.

En el segundo paso, el desarrollador cambia el nombre de la librería en el diseño, que pasa de llamarse “PlatformAbstraction” a “FreeRtos”. El cambio se realiza en el paquete de despliegue. Este cambio también es un cambio regular que no requiere ninguna adaptación. El efecto de este cambio en el diseño es que, los includes en las instancias pasar de ser *“#include “PlatformAbstractio.h*” a ”*#include “FreeRTOS.h*”. En este caso las diferencias son del tipo *MMDiff!UpdateAttribute*, concretamente Δ**Im** = 1 y Δ**Om** = 4. En este caso TRANSEVOL tampoco detecta ningún escenario de adaptación. De todas maneras como es un escenario de evolución regular tampoco es necesario ejecutar TRASNEVOL.

El siguiente objetivo es especificar como cambia el código de creación de tareas y la llamada al planificador en el bucle principal de la aplicación. La generación del fragmento de modelo que especifica el código del bucle principal está en manos de la transformación M2M. La regla de transformación *MainLoop2Method* es la encargada de generar dicho código. En un escenario real debido a que la transformación es legada no se conocería la regla responsable. Para localizar los cambios TRANSEVOL utiliza las diferencias del modelo de salida, ya que en este paso no hay cambios en el modelo de entrada. Este requisito de mapeo se realiza ejecutando TRANSEVOL frente a dos escenarios de adaptación, dos parejas de modelos. De los 11 cambios de la figura 6.4, 9 cambios están relacionados con las llamadas a la función de creación de tareas. La otra diferencia está relacionados con la llamada al planificador. Por lo tanto, el primer escenario se define mediante 9 cambios, mientras que en el segundo escenario se define agregando el cambio restante. El controlador de puertas contiene tres tareas. Cada llamada de creación de tareas sufre tres cambios: uno en los argumentos, otro en la función llamada y otro en el identificador. Este cambio también es una evolución regular, y no requiere modificación alguna, ya que la nueva función utilizada para crear la función se ha especificado en el diseño. El siguiente cambio relacionado con la creación de tareas está relacionado con la llamada al planificador. En *freeRTOS* la llamada al planificador se realiza una única vez antes de entrar en el bucle principal, en lugar de llamarle constantemente en el bucle principal. En el modelo ejemplo de salida hay que demostrar que se quita la llamada al planificador de dentro del *while*, y se ubica justa antes. Esto se puede demostrar de dos maneras. La más sencilla, y en único paso, es mover el código de la llamada del planificador de dentro del *while* a fuera. Al realizar esto TRANSEVOL detecta un cambio de contenedor y cambia la asignación. La otra posibilidad para demostrar el cambio es quitar el código de la llamada del planificador de dentro del while. TRANSEVOL detecta la eliminación y pasa de una regla de 1 a 2, a una regla de 1 a 1. Esto le ocurre a la regla lazy rule *createPeriodicTaskCode*. A continuación el código de llamada al planificador se agrega en el modelo de salida, y TRANSEVOL detecta que la regla *lazy rule createMainLoopCode* pasa de ser una regla de 1 a 1 a una regla de 1 a 2. Ambas posibilidades fueron detectadas correctamente por TRANSEVOL. En la figura 6.5 se ven los cambios que hay que realizar en el modelo SimpleC para expresar el requisito utilizando la eliminación de elementos.

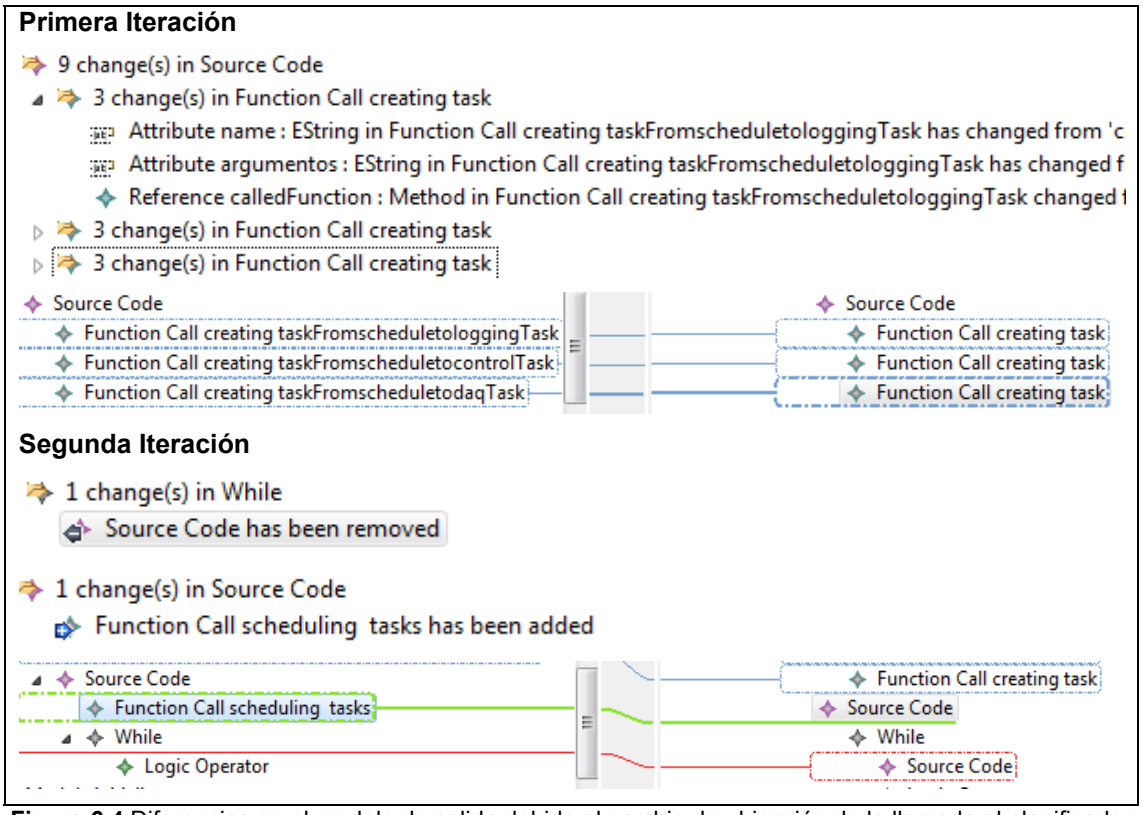


Figura 6.4 Diferencias en el modelo de salida debido al cambio de ubicación de la llamada al planificador en el bucle principal.

El último paso es muy parecido al anterior. En este caso el código de la tarea se ve modificado. En el API inicial simplemente se llama a la función de creación de tareas y el planificador se encarga de llamar a la función periódicamente. En *freeRTOS* el código es ligeramente diferente. En la función de la tarea se crea un bucle y dentro de este bucle se ejecuta el código, funcional a ejecutar periódicamente. Para poder ejecutar la funcionalidad periódicamente, se añade la llamada a una función de retardo dentro del bucle. De esta forma *freeRTOS* consigue ejecutar la funcionalidad con una periodicidad determinista. Los cambios a realizar en el modelo de salida se localizan en el código fuente de las tareas. En este caso las tres tareas se ubican en la misma instancia de componente. Los cambios realizados al modelo de salida se resumen en la figura 6.5. Los cambios a implementar requieren sustituir la llamada directa a la función de la tarea, el <<entryPoint>>, por un *while*. Después en el código del *while* se deben de ubicar la llamada a la función de la tarea y a la función de retardo. Estos requisitos se especifican mediante tres modelos ejemplos diferentes creados de forma incremental. Por lo tanto, la herramienta TRANSEVOL se ejecuta tres veces. En el primer modelo se añade el *while*. TRANSEVOL detecta que la regla *lazy rule createPeriodicTaskCodeLines* pasa de ser de 1 a 1 a 1 a 2. En esta iteración las diferencias de salida son del tipo *MMDiff!ModelElementChangeLeftTarget* y son tres Δ**Om**=3. En la siguiente iteración se mueve la llamada a la función de la tarea en el modelo ejemplo, y TRANSEVOL detecta el cambio de contenedor. En esta iteración las diferencias en Δ**Om** son tres diferencias del tipo *MMDiff!UpdateContainmentFeature*. Finalmente, se agrega una llamada

a la función de retardo de *freeRTOS* en el *while*. En este caso TRANSEVOL detecta que se pasa de una función de 1-a-2 a 1-a-3. En *freeRTOS*, la cabecera de la función tarea requiere de un argumento. Este cambio es un cambio regular, que no requiere adaptación.

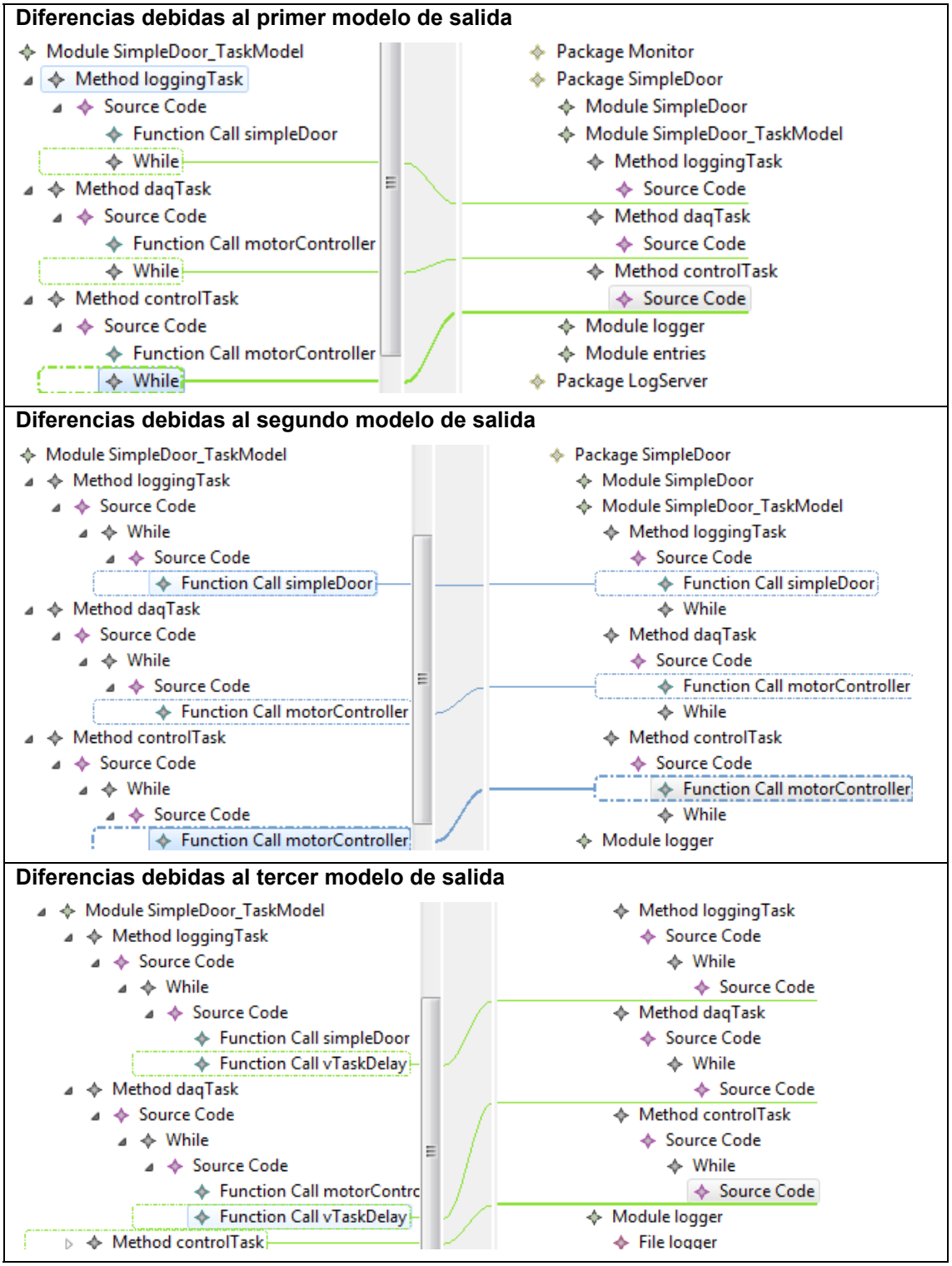


Figura 6.5 Diferencias creadas al modificar el código de la tarea *loggingTask* de la instancia *SimpleDoor*.

En cada iteración se validó la adaptación deducida por TRANSEVOL ejecutando la nueva transformación con su respectivo modelo ejemplo. El modelo de salida generado por la transformación de cada iteración fue comparado con su respectivo modelo ejemplo de salida. En todas las iteraciones el resultado fue correcto. Finalmente el modelo de salida generado se transformó

a código mediante la transformación M2T y se ejecutó en una placa de desarrollo Stellaris lm3s8962 de Texas Instruments con el sistema operativo *freeRTOS*. En la tabla 6.2 se resume el escenario de cada iteración mediante una pequeña descripción, las diferencias generadas y la operación de adaptación deducida.

Tabla 6.2. Escenarios de adaptación y operaciones de adaptación deducidas debido al cambio de API de concurrencia

Iteración	Descripción	Diferencias	Operaciones de adaptación
1	Especificación del Nuevo API de freeRTOS	*1ª Pareja de modelos ejemplo* *MMDiff!UpdateAttribute* **∆Im = 2, ∆Om = 2** *2ª Pareja de modelos ejemplo* MMDiff!ModelElementChangeRight-Target**)** **∆Im = 1, ∆Om = 1**	Ninguna (Evolución regular)
2	Cambio del nombre de la librería	*MMDiff!UpdateAttribute* **∆Im = 1,∆Om = 4**	Ninguna (Evolución regular)
3	Cambios en la creación de tareas y llamada al planificador	*1ª Pareja de modelos ejemplo:* **∆Im = 0** **∆Om = 9** *MMDiff!UpdateAttribute*	Ninguna.(Evolución regular)
		2ª Pareja de modelos ejemplo: Opción a) **∆Im = 0** **∆Om = 1 ((***MMDiff!UpdateContain-mentFeature)* Opción b) **∆Im = 0** **∆Om = 1 (**MMDiff!ModelElement-ChangeRightTarget**)** **∆Im = 0** **∆Om = 1 (**MMDiff!ModelElement-ChangeLeftTarget**)**	Cambio de contenedor Eliminación de 1 regla 1 agregación de patrón de salida
4	Modificación del código de las tareas concurrentes	1.Iteración: **∆Im = 0** **∆Om = 3 (**MMDiff!ModelElement-ChangeLeftTarget**)** 2.Iteración: **∆Im = 0** **∆Om = 3 ((***MMDiff!UpdateContain-mentFeature)* 3.Iteración: **∆Im = 0** **∆Om = 3 (**MMDiff!ModelElement-ChangeLeftTarget	1 agregación de patrón de salida 1 Cambio de contenedor 1 agregación de patrón de salida

6.3.4 Validación de la adaptación de las transformaciones frente a la aplicación de un nuevo perfil para representar RNF

En este apartado se valida la solución TRANSEVOL frente a escenarios donde se requiere expresar nuevos RNF en el sistema, y para ello se decide extender el metamodelo mediante perfiles. La herramienta TRANSEVOL se ha aplicado con éxito a dos situaciones de este tipo. En ambos casos el sistema de DSDM sobre el que se realizó el caso de estudio era el mismo: la transformación *fromUML2SimpleC*. En el primer escenario el requisito RNF consistía en evolucionar de un sistema de componentes con ejecución secuencial a una solución concurrente que se ejecutaba mediante un planificador. Para realizar el cambio se aplicó al metamodelo UML el perfil MARTE. En el segundo escenario de validación se le agregó al metamodelo de diseño UML un perfil de seguridad para especificar un control de acceso seguro en las interfaces expuestas por los diferentes componentes SW. En este apartado se va a describir el segundo escenario en detalle.

En un momento dado, los desarrolladores de la solución *fromUML2SimpleC* requerían agregar control de acceso a las interfaces de sus componentes SW. Originalmente el sistema no ofrecía dichas características, ni en el diseño ni en el código generado. Para especificar características de control de acceso se seleccionó la metodología presentada en [Bou11]. Esta metodología está basada en el uso de componentes del metamodelo UML para capturar los requisitos de seguridad e implementar patrones de seguridad. En este caso de estudio, un perfil UML asociado al patrón de seguridad RBAC es utilizado para especificar los requisitos de control de acceso. En la figura 6.7 el perfil UML utilizado para expresar el patrón RBAC es presentado. Para que el software ANSI-C generado del diseño UML ofrezca control de acceso en las interfaces de los componentes se requiere integrar en el código de las interfaces de los componentes el patrón "Proxy Protection" [Bus96].

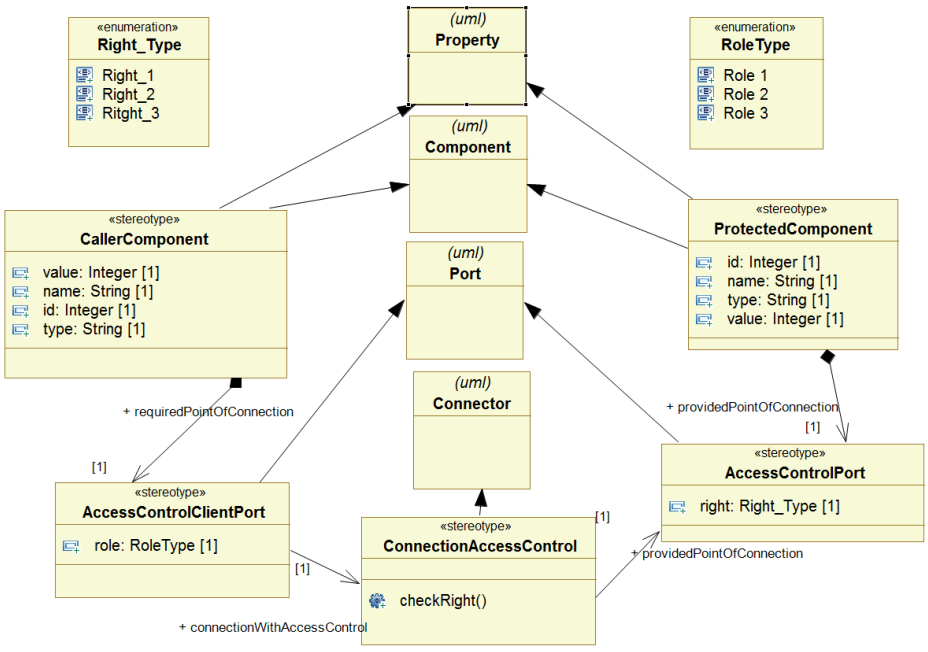


Figura 6.6 Perfil UML para el control de acceso RBAC [Bou11].

Para aplicar el método TRANSEVOL la aplicación ejemplo seleccionada es una aplicación gestora de directorios, ver figura 6.8. Esta aplicación está compuesta por un gestor de carpetas y el cliente. El gestor de carpetas ofrece servicios de listado del contenido de los directorios, navegación de directorios, creación de directorios y borrado de directorios. El modelo *SimpleC* que representa la aplicación es generado automáticamente aplicando la transformación M2M al modelo de diseño. El código en C es obtenido mediante la transformación M2T.

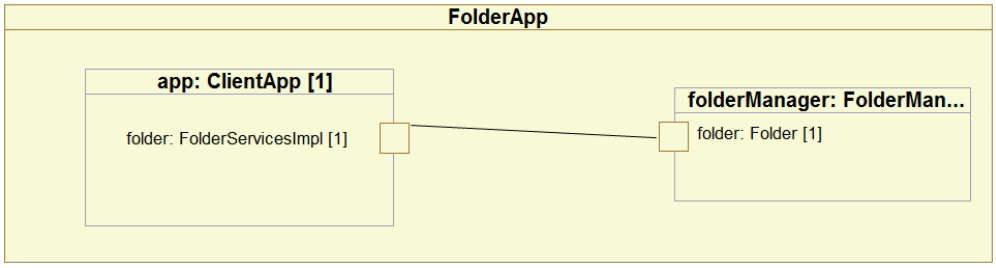


Figura 6.7 Diagrama de componentes del gestor de directorios.

Una vez implementado el perfil de seguridad y seleccionado el diseño ejemplo del gestor de carpetas y el modelo SimpleC correspondiente, se deben de incrementar manualmente los modelo ejemplo para demostrar el nuevo requisito de mapeo. La figura 6.9 representa el nuevo modelo de diseño para el gestor de directorios con características de RBAC, y el modelo SimpleC deseado con el patrón “Protection Proxy” implementado para ofrecer el control de acceso en las interfaces de sus componentes. La especificación de la transformación se puede resumir mediante los siguientes requisitos:

- Para cada <<accessControlPort>> en una instancia de un componente protegido, se debe de crear un módulo donde el patrón “Protection Proxy” debe de implementarse.
    1. Crear el modulo donde el patrón proxy debe de implementarse.
    2. Crear el atributo *actualClientRole*.
    3. Crear el método *setRole*.
    4. Crear la función envoltorio (wrapper) que llama a la función original de la interface.
    5. Crear una referencia en el fichero cabecera que define la estructura *Role*.
    6. Crear la referencia a la cabecera que define la función original de la interface.
- Para cada <<accessControlConnection>> crear:
    1. El método *checkRights* en un módulo dentro del paquete donde se ubica la instancia del componente protegido.
    2. Crear la estructura Role en un módulo dentro del paquete donde se ubica la instancia del componente protegido.
    3. Crear las enumeraciones *Roles* y *Right*.

A continuación se describe como se aplicó la herramienta TRANSEVOL para adaptar la transformación *fromUML2SimpleC*. El objetivo de este caso de estudio es agregar propiedades de control de acceso a los componentes SW que requieren protección. Los nuevos requisitos de transformación son demostrados en varias etapas. En cada fase una pareja de modelos ejemplos de entrada y salida se define. La pareja de modelos de cada etapa es un incremento del anterior.

En cada etapa, después de aplicar la herramienta, las operaciones de adaptación automáticamente derivadas son implementadas y la nueva traza de ejecución de la transformación debe de ser obtenida para ser utilizada en la siguiente iteración.

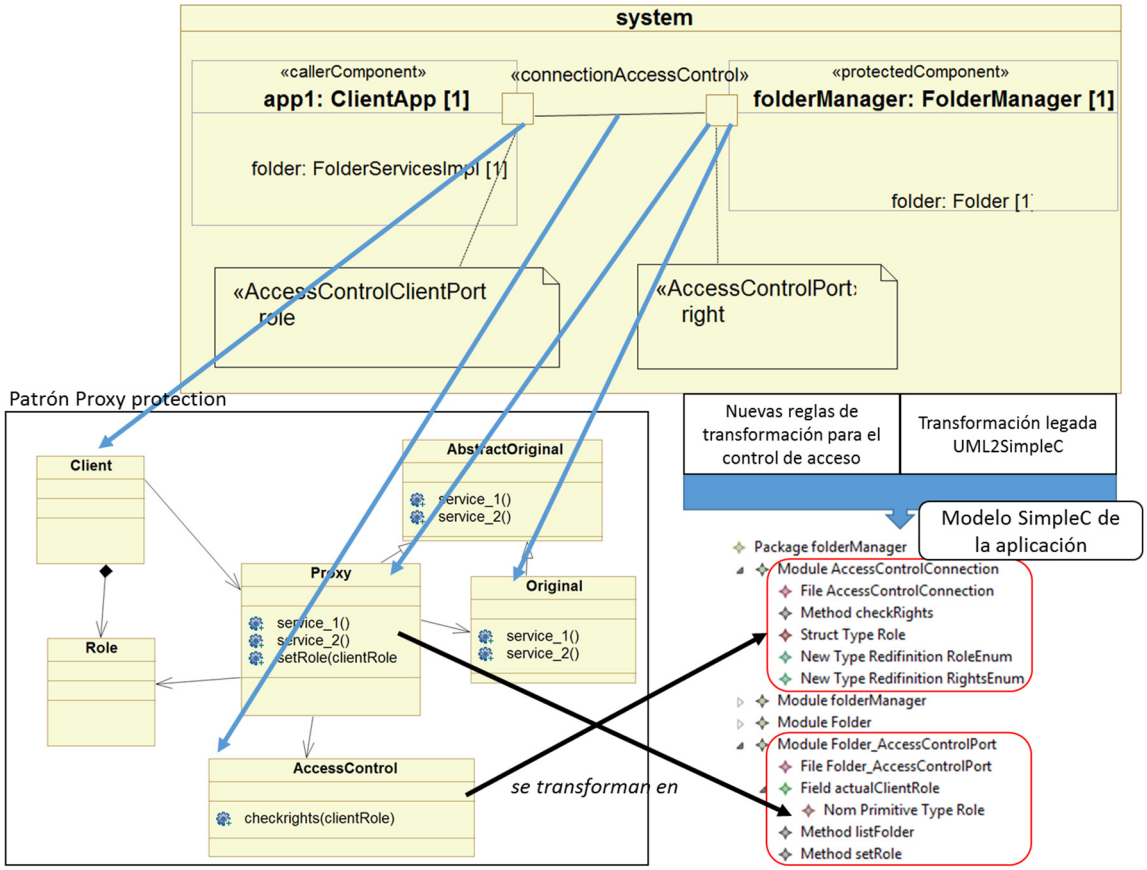


Figura 6.8 El modelo de diseño con control de acceso, el patrón "Proxy Protection" y el modelo SimpleC deseado [Agi15].

En la primera iteración el perfil de seguridad es agregado al modelo ejemplo de entrada aplicando a los componentes los estereotipados <<*callerComponent*>> y <<*protectedComponent*>>.En este caso el modelo ejemplo de salida no requiere ninguna modificación, y la transformación no se modifica. Para facilitar la comprensión, a partir de ahora solo se detallaran las acciones relacionadas con los componentes estereotipados como <<*protectedComponent*>>. En la segunda iteración el conector que requiere control de acceso es estereotipado como <<*accessControlConnection*>>. En esta iteración el modelo de salida deseado se debe de especificar, modificando manualmente el previo. El modelo de salida es modificado para ofrecer la función que chequea los permisos de un cliente para acceder a un puerto de un componente protegido. Para ello, una cabecera, representado como modulo en SimpleC, y un fichero son añadidos al directorio del componente *folderManager*. El método *checkRights*, la estructura *Role* y las enumeraciones *roles/rights* son definidos en este módulo. Los cambios presentes en el modelo de salida son obtenidos comparando el nuevo modelo con el anterior mediante EMFCompare. Seis diferencias del tipo *MMDiff!ModelElementChangeLeftTarget* existen en esta iteración en Δ**Om,** siendo **CC(ΔOmc)=3**. En el modelo de entrada una agregación de un estereotipo es detectado. Utilizando estas diferencias la herramienta detecta un mapeo de 1 a varios relacionado con la aplicación de un estereotipo a un elemento del tipo *UML!Connector* previamente transformado.

La herramienta deduce una operación de división de regla de transformación y que la nueva regla generada de la división requiere 3 nuevos patrones de salida, uno por cada tipo de elemento añadido. Para seleccionar la regla afectada el algoritmo utiliza la traza de ejecución para localizar la regla que generó la instancia del tipo *UML!Connector* estereotipada en el modelo de entrada. La regla afectada es *createConnector*.

La siguiente iteración requiere agregar el control de acceso a los puertos de los componentes protegidos. En este caso, tanto el modelo de entrada como el de salida deben de ser modificados. Esta iteración se divide en dos fases, por lo que dos parejas de modelos ejemplos son creados. Primero, en el modelo de entrada se estereotipan con <<*accessControlPort*>> aquellos puertos a securizar. En el modelo de salida ejemplo se implementa el patrón proxy pattern para cada puerto. En esta iteración inicial solo el fichero cabecera, las referencias a la interface a envolver y las referencias al módulo que implementa el método *checkRights* son añadidas al modelo de salida que representa el código. Con esta información la herramienta detecta otra vez, una operación de división de regla con filtrado. Con esta operación de adaptación la nueva regla no está completamente implementada, el método que envuelve la interfaz expuesta y la variable que representa el rol del puerto faltan. La última pareja de modelos ejemplo se genera modificando únicamente el modelo de salida. En este caso la herramienta detecta que varios elementos se han añadido (el envoltorio de la interfaz, el método *setRole*, el atributo *actualClientRole* y el fichero que implementa las funciones). En este paso los patrones de salida correspondientes son agregados a la regla creada en el paso anterior. En la figura 6.9 se puede ver una de las divisiones de regla obtenidas. En la tabla 6.3 las operaciones de adaptación derivadas en cada fase están resumidas. La nueva transformación M2M fue validada aplicando la transformación al nuevo diseño y comparando el nuevo modelo generado con el modelo de salida esperado. Para aplicar la herramienta es suficiente conocer los cambios a realizar en los modelos de entrada y salida. Previos conocimientos de la transformación M2M no son necesarios. De todas maneras la creación de modelos ejemplos queda en manos del conocimiento del desarrollador, y esto es un punto a mejorar y formalizar.

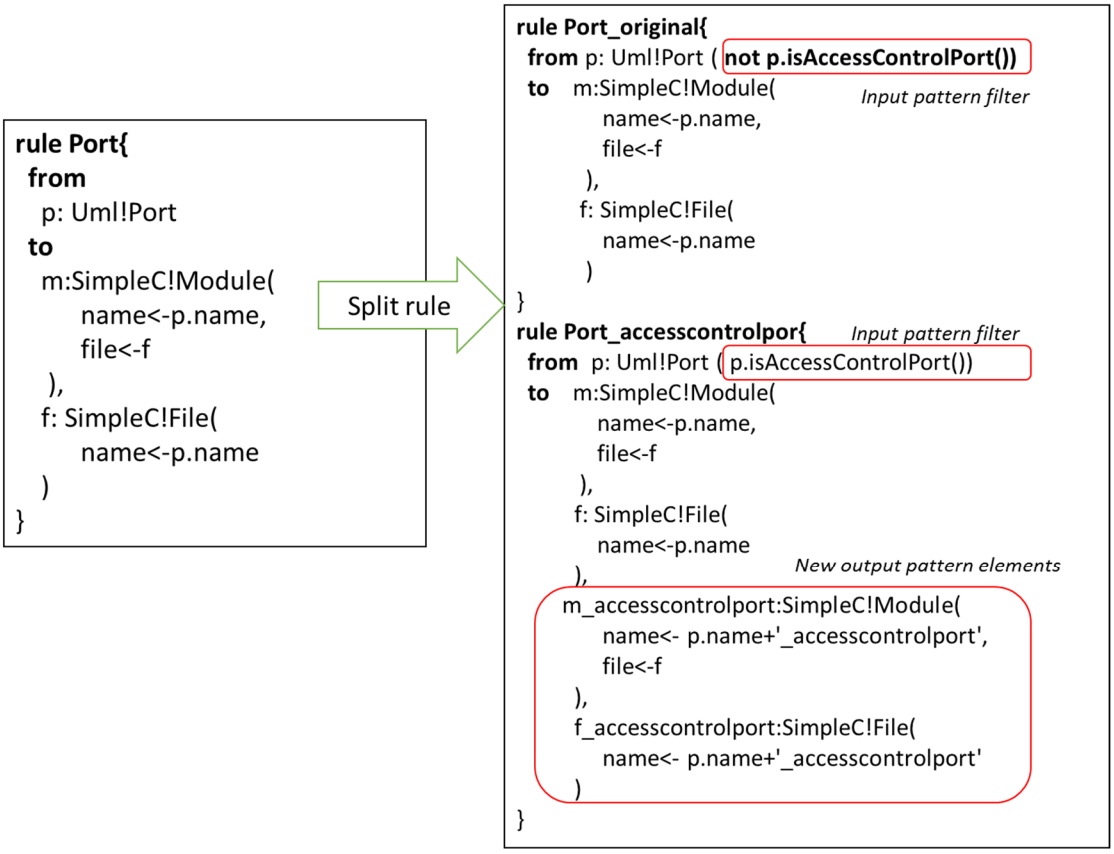


Figura 6.9 División de la regla Port en dos.

Tabla 6.3. Operaciones de adaptación deducidas al aplicar el perfil RBAC.

Iteración	Descripción	Diferencias	Operaciones de adaptación
1	El perfil de acceso de control es añadido y los estereotipos <<callerComponent>> y <<protectedComponent>> son aplicados.	*MMDiff!ModelElementChangeLeftTarget* **ΔIm = 2** **ΔOm = 0**	Ninguna (. Evolución regular)
2	El estereotipo <<AccessControlConnection>> es aplicado al conector del modelo de diseño que requiere seguridad y el módulo de acceso de control es creado en el modelo de salida.	*MMDiff!ModelElementChangeLeftTarget* **ΔIm = 1** (aplicación de estereotipo) **ΔOm = 6** **CC(ΔOm) = 3**	1 división de regla donde la nueva regla requiere 3 nuevos patrones de salida
3	El estereotipo <<AccessControlPort>> es aplicado al puerto correspondiente y el modulo proxy es creado en el modelo de salida. Los includes requeridos por el modulo proxy también son agregados en el modelo de salida.	**ΔIm = 1** (aplicación de estereotipo, *MMDiff!ModelElementChangeLeftTarget*) **ΔOm = 2** (*MMDiff!UpdateReferences*)	1 division de regla 2 cambios de asignación
4	Los métodos y atributos del módulo proxy son agregados.	**ΔIm =** 0 **ΔOm** =4 (*MMDiff!ModelElementChangeLeftTarget)*)	4 nuevos patrones de salida.

6.4 Conclusiones

En el transcurso del desarrollo de TRANSEVOL se crearon una serie de sencillas transformaciones M2M que permitieron validar el método. Una vez implementado la herramienta prototipo se validó la herramienta utilizando una serie de transformaciones ATL M2M existentes y desarrollados por terceros. En esta validación se validaron la gran mayoría de escenarios de adaptación excepto la creación de reglas N a1 y la agregación de patrones de entrada. Este tipo de escenarios no se han validado con transformaciones de terceros. Una vez validada la herramienta TRANSEVOL con ejemplos sencillos que contemplaban todos los escenarios de adaptación y frente a 7 transformaciones M2M de terceros de mediana complejidad, se validó el método TRANSEVOL frente un sistema real de generación de código basado en DSDM para sistemas empotrados. En esta última validación el objetivo era comprobar si TRANSEVOL era capaz de deducir las operaciones de adaptación en un sistema de DSDM complejo frente a cambios en los RNF que requieren cambios en la lógica de mapeo y aplicación de perfiles. Se utilizó TRANSEVOL para adaptar la transformación M2M *fromUML2C* frente a dos evoluciones de RNF y los resultados fueron óptimos. A pesar de que se ha validado la solución TRANSEVOL ciertos factores pueden afectar el método de validación utilizado:

- *Corrección*: A pesar de que las validaciones presentan resultados prometedores, debido a que todas las operaciones de adaptación han sido correctamente identificadas, el algoritmo debe de probarse en situaciones más complejas y con escenarios diferentes para cubrir todas las posibilidades. Para ello es fundamental definir una clasificación de intenciones de mapeo, definiendo para cada tipo de intenciones su conjunto de escenarios de evolución mediante una especificación formal.
- *Escalabilidad:* El caso de estudio de mayor dimensión seleccionado en este trabajo (8 ficheros, 40 reglas de mapeo, 30 reglas lazy y 44 funciones de ayuda) presentaba menos de 30 diferencias en los modelos. A pesar de que el caso de estudio es real, una validación con casos de estudio de mayor dimensión es requerido.
- *Mapeos varios a varios*: A pesar de que la herramienta detecta escenarios de mapeo varios a varios, ocurren ambigüedades en la deducción de las operaciones de transformación. Debido a esto, este tipo de escenarios se deben de especificar mediante la combinación de varios escenarios más sencillos (uno a uno, varios a uno, etc.). Actualmente esta división del problema queda en manos del desarrollador, y esto requiere definir un método para asistir y guiar en este proceso al desarrollador.
- *Funcionalidades a mejorar*: Cuando la herramienta deduce operaciones de adaptación de adición de reglas de transformación también se generan las correspondientes asignaciones que relacionan los nuevos elementos con sus contenedores. En esos casos la expresión que determina el binding se implementa

mediante una función “helper”. La herramienta actualmente no genera el código de la función “helper”, únicamente define la cabecera y realiza la llamada a la función en la regla de transformación correspondiente. Otro aspecto a mejorar son las detecciones de las evoluciones regulares. Actualmente para detectar las evoluciones regulares el desarrollador debe de ejecutar antes de adaptar la transformación M2M la transformación legada con el modelo ejemplo de entrada y analizar si el modelo de salida generado coincide con el modelo ejemplo de salida creado. Para evitar esta verificación habría que integrar un módulo de detección de evoluciones regulares.

CUARTA PARTE
Conclusión

7. Conclusiones y líneas futuras

En este trabajo se ha presentado el método TRANSEVOL que permite deducir y localizar las operaciones de adaptación a implementar en una transformación M2M legada basada en reglas de mapeo frente a requisitos de evolución de mapeo expresados mediante modelos ejemplo. En este capítulo se resumen los objetivos alcanzados y se presentan las conclusiones obtenidas. Una vez analizados los resultados se definen las líneas futuras y los nuevos retos de investigación abiertos tras este trabajo de investigación.

TRANSEVOL se basa en las diferencias creadas por los modelos ejemplos para deducir los cambios a implementar en las transformaciones M2M. Actualmente existe una primera versión de una herramienta prototipo desarrollada en JAVA utilizando el framework EMF que implementa el método TRANSEVOL. La herramienta utiliza la versión de EMFDiff 2.1 para el cálculo de diferencias. Mediante el metamodelo *MMAdaptationGoal* basado en el metamodelo ATL 3, se expresan las operaciones de adaptación a realizar. Hasta el momento TRANSEVOL sólo se ha aplicado a transformaciones ATL.

El utilizar modelos ejemplo en la deducción de las operaciones de adaptación, permite analizar el impacto de los cambios en transformaciones M2M sin utilizar directamente la información de los metamodelos. TRANSEVOL permite adaptarse a cambios en la lógica de mapeo y deducir las nuevas reglas de transformación aunque no existan diferencias en los metamodelos. Esta característica de TRANSEVOL permite adaptar las transformaciones M2M cuando el escenario de evolución requiere cambiar únicamente la lógica de mapeo. TRANSEVOL también se puede utilizar para adaptar las transformaciones M2M cuando se debe de modificar la lógica de mapeo debido a la aplicación de un perfil en los metamodelos. Por lo tanto, TRANSEVOL se puede utilizar en aquellos escenarios de adaptación de transformaciones M2M debidos a evoluciones de RNF que afectan a la lógica de mapeo. Además, el combinar las diferencias entre modelo con las trazas de ejecución permite adaptar transformaciones M2M legadas.

El utilizar modelos ejemplos en la definición de los nuevos requisitos de mapeo permite que el artefacto de entrada utilizado para el análisis del impacto en las transformaciones M2M también se utilice en la validación de la implementación de la nueva transformación M2M. De esta forma el método TRANSEVOL se puede utilizar como base para un método de desarrollo de transformaciones M2M dirigido por validaciones.

El método TRANSEVOL se ha validado frente a pequeños ejemplos creados para cada uno de los escenarios de evolución que detecta TRANSEVOL. Con el objetivo de validar su uso en un entorno industrial del dominio de los sistemas empotrados, el método TRANSEVOL se ha validado frente a dos evoluciones de RNF en un sistema real de generación de código dirigido por modelos. La primera validación realizada sobre un sistema de DSDM del dominio de los sistemas empotrados fue una migración de API de concurrencia donde los metamodelos no se

modificaban pero una nueva lógica de mapeo era necesaria. La segunda validación fue un escenario donde se requería integrar aspectos de seguridad en las aplicaciones a generar. Para ello, se integró en el sistema de DSDM un nuevo perfil UML de seguridad y se requería modificar la lógica de mapeo para poder generar aplicaciones que integraban los nuevos conceptos de seguridad, en este caso un control de acceso RBAC. En ambas situaciones se aplicó satisfactoriamente el método TRANSEVOL. Mediante estas validaciones se han demostrado las hipótesis planteadas en este trabajo:

- **Hipótesis A***: Utilizando las diferencias entre los modelos de entrada y las diferencias entre los modelos de salida es posible detectar los cambios a implementar en la lógica de mapeo de una transformación M2M debido a cambios en los RNF de dominio cuando los metamodelos no requieren cambios.*
- **Hipótesis B:** *Utilizando las diferencias entre los modelos los modelos de entrada y las diferencias entre los modelos de salida es posible detectar los cambios a implementar en una transformación M2M al extender los metamodelos mediante perfiles para integrar nuevos RNF de dominio.*
- **Hipótesis C:** *Es posible localizar las ubicaciones de las modificaciones a realizar en las transformaciones M2M legadas combinando datos de trazabilidad de ejecución de las reglas de transformación con las diferencias de los modelos de entrada y con las diferencias de los modelos de salida.*

Es importante puntualizar estas hipótesis, ya que para una correcta deducción de las operaciones de adaptación las diferencias de los modelos de entrada, la traza de ejecución y el propio modelo de transformación también son necesarias. Las características del método TRANSEVOL son las siguientes:

1. Ofrece un **método** basado en **modelos ejemplo** para especificar los nuevos requisitos de trasformación. Al mismo tiempo la especificación se utiliza como entrada para la deducción de las operaciones de adaptación. Los modelos ejemplo de cada iteración también se utilizan en el proceso de validación de la nueva implementación de la transformación M2M.
2. El método permite analizar el impacto de los nuevos requisitos de transformación en las transformaciones **M2M legadas** mediante el uso de **trazas de ejecución**.
3. Obtiene automáticamente los **cambios a realizar** en la transformación M2M legada frente a evoluciones de RNF de dominio analizando las **diferencias entre los modelos** ejemplo y los modelos previos.
    a. Evoluciones de RNF que requieren cambios en la lógica de mapeo
    b. Evoluciones de RNF que requieren cambios en la lógica de mapeo debido a la aplicación de perfiles UML
4. Se puede automatizar la implementación de las operaciones de adaptación usando modelos que corresponden al metamodelo **MMAdaptationGoal**.

5. La herramienta TRANSEVOL permite trabajar tanto con transformaciones exógenas y endógenas.
6. El objetivo inicial de este trabajo era poder evolucionar transformaciones M2M legadas, pero el método TRANSEVOL también se puede utilizar para el desarrollo de transformaciones M2M desde cero.

El método TRANSEVOL funciona correctamente en los escenarios de evolución de transformaciones recogidos en la tabla 7.1.

Tabla 7.1 Escenarios de adaptación contemplados por TRANSEVOL.

Tipo de escenario de adaptación	Nombre del escenario
Escenarios relacionados con agregaciones en los modelos de salida	Nueva regla 1 a N
	Agregación de patrones de salida
	Nueva regla con filtrado
	División de regla
Escenarios relacionados con modificaciones de atributos en los modelos de salida	Modificación de asignaciones
	División de regla con modificación de asignaciones
Escenarios relacionados con modificación de contenedores en los modelos de salida	Cambio de contenedor
	División de regla con modificación de contenedor
Escenarios relacionados con eliminaciones en los modelos de salida	Eliminación de un patrón de salida
	Filtrado de un patrón de entrada
Escenarios donde $\Delta Om = 0$	Filtrado de patrón de entrada
Escenarios de aplicación de estereotipos	Modificación de asignaciones debido al uso de estereotipos y agregación de elementos
	División de una regla debido a la aplicación de estereotipos y la modificación de atributos de salida
	Filtrado debido a la aplicación de estereotipos

7.1 Contribución al estado del arte

Este trabajo se ubica en el ámbito de la adaptación y el desarrollo de transformaciones M2M. Existen trabajos actuales que permiten la co-evolución entre metamodelos y transformaciones M2M [Gar12] [Dir13] [Kus15]. Estos trabajos no permiten adaptar las transformaciones M2M cuando los metamodelos no varían pero la lógica de mapeo requiere ser modificada. TRANSEVOL permite adaptar las transformaciones M2M cuando únicamente se requieren cambios en la lógica de mapeo.

TRANSEVOL además de poder ser utilizado en la adaptación de transformaciones M2M, también se puede utilizar en el desarrollo de transformaciones M2M. Existen trabajos de desarrollo de transformaciones, que como TRANSEVOL, se basan en el uso de modelos ejemplo [Kes12] [Fau13]. Los trabajos del ámbito del MTBE permiten desarrollar desde cero transformaciones M2M, pero estos no se pueden aplicar a transformaciones M2M legadas.

El método TRANSEVOL se puede utilizar frente a cambios en la lógica de mapeo debido a cambios en los RNF de dominio. En [Sun13] se utilizan modelos ejemplo para demostrar la nueva lógica de mapeo de transformaciones M2M debido a RNF. [Sun13] solo se puede aplicar

a transformaciones M2M endógenas, mientras que TRANSEVOL se puede aplicar tanto a transformaciones M2M endógenas como exógenas. Otro de los objetivos de este trabajo es adaptar las transformaciones M2M al aplicar nuevos conceptos en el PIM relacionados con RNF mediante perfiles UML. En la literatura actual no existe ningún trabajo de adaptación de transformaciones M2M frente a la aplicación de perfiles UML.

En la tabla 7.2 se resume la contribución al estado del arte de TRANSEVOL a los métodos para la adaptación de transformaciones M2M.

Tabla 7.2 Análisis de soluciones para la adaptación de transformaciones M2M.

Métodos para la adaptación de transformaciones M2M		**Tipo de transformaciones M2M**			**Escenario de adaptación de transformaciones M2M debidos a la evolución de RNF**		
Tipo de solución	**Propuesta**	**Legadas**	**Exógenas**	**Endógenas**	**Sin cambios en los metamodelos**	**Extensión de metamodelos mediante perfiles**	**Cambios en los metamodelos**
Diferencias metamodelos	[Gar12]	X	X	X			X
Weaving	[Dir13]	X	X	X			X
Operaciones de diferencias	[Kus15]	X	X	X			X
MTBE	[Sun13].	X		X	X	X	X
	[Kes12]		X	X	X	X	
	[Fau13]		X	X	X	X	
	[Agi15]	**X**	**X**	**X**	**X**	**X**	

7.2 Análisis crítico de la solución

A continuación se realiza un análisis crítico de la solución con el objetivo de establecer los objetivos a abordar en los trabajos futuros.

7.2.1 Código de las asignaciones

A pesar de que se detectan los escenarios de evolución de la tabla 7.1, las operaciones de adaptación deducidas no permiten implementar automáticamente, al 100%, las nuevas reglas de transformación. Tanto la lógica de las **funciones de filtrado y asignación** se especifican mediante una llamada a una función de ayuda. Es importante remarcar que el algoritmo de estas funciones de ayuda actualmente no se implementa. La responsabilidad de implementar estas funciones queda en manos del desarrollador. Para completar la herramienta y el método se debe de analizar la integración de soluciones que permitan deducir dichos algoritmos. Esta problemática también existe en la modificación de asignaciones. En estos casos el desarrollador de transformaciones debe de analizar la situación y reimplementar la lógica de la función de asignación.

7.2.2 Numero de tipos de diferencias en los modelos

El método TRANSEVOL solo permite especificar un tipo de diferencias en los modelos ejemplos. Por lo tanto, los modelos ejemplo con más de un tipo de diferencias no son analizados por TRANSEVOL.

7.2.3 Especificación de escenarios

El método TRANSEVOL no establece como se crean los modelos ejemplo y deja esta tarea en manos del desarrollador de transformaciones M2M. Por lo tanto, puede ocurrir que el desarrollador de transformaciones M2M genere de forma incorrecta los modelos ejemplo. El desarrollador puede especificar un escenario de evolución diferente al requerido. La intención del desarrollador puede ser expresar la necesidad de un nuevo patrón de salida en una regla de transformación pero al modificar los modelos de salida no agrega en todas las instancias de salida el nuevo elemento. Ante esta situación TRANSEVOL detecta una nueva regla de transformación con filtrado en lugar de la agregación de nuevo patrón de salida. La detección de estas circunstancias no está contemplado en TRANSEVOL. Para detectar estas situaciones se puede combinar TRANSEVOL con **un lenguaje de modelado que permita expresar** los requisitos de transformación como en [Luc16] [Var14]. De esta forma una vez creados los modelos ejemplo, TRANSEVOL puede verificar si el escenario definido mediante los modelos ejemplo concuerda con el requisito de mapeo.

En caso de que el desarrollador de transformaciones especifique una situación no contemplada por la herramienta, se crea un mensaje donde se especifica que el escenario de mapeo es desconocido para la herramienta. Puede ocurrir que el desarrollador de transformaciones especifique mediante un único requisito un escenario de evolución, complejo y compuesto por un conjunto de escenarios de evolución contemplados por TRANSEVOL. Por ejemplo, los escenarios de mapeo de varios a varios no se pueden especificar en una única iteración en TRANSEVOL. Ante este tipo de situaciones el desarrollador debe de descomponer el requisito de transformación en un conjunto de requisitos cuya composición da como resultado la transformación compleja esperada tras varias iteraciones. Actualmente este trabajo de descomposición de la especificación queda en manos del desarrollador de transformaciones. Por ello, es necesario ofrecer una **metodología** que guie y asista al desarrollador de transformaciones en la definición y generación de los modelos ejemplos y permita verificar la corrección de la especificación realizada.

También puede ocurrir que, al ser la transformación M2M legada, el desarrollador de transformaciones especifique un escenario de adaptación relacionado con una evolución regular, es decir que no requiere ninguna modificación en las reglas de transformación. Actualmente TRANSEVOL detecta este tipo de situaciones ejecutando la transformación M2M legada tras definir los modelos ejemplo y antes de adaptar las transformaciones, de forma que comprueba si el modelo de salida generado coincide con el esperado. Si coinciden significa que es una evolución regular que no requiere cambios. Este proceso de detección de evoluciones regulares

se puede agilizar si el propio método TRANSEVOL ofrece un mecanismo para la detección de evoluciones regulares.

7.2.4 Transformaciones imperativas

Existen situaciones de transformación de modelos donde el algoritmo de mapeo requiere partes implementadas de forma imperativa. Estos casos no pueden ser resueltos actualmente por TRANSEVOL. Esto es debido a que TRANSEVOL no contempla transformaciones imperativas. Este tipo de mapeos se pueden implementar de forma declarativa utilizando varias transformaciones M2M secuencialmente, definiendo para cada una de ellas los escenarios de transformación correspondientes.

7.2.5 Dependencias de lenguajes de transformación y metamodelos

Hasta el momento, TRANSEVOL solo ha sido utilizado en transformaciones ATL. En TRANSEVOL la detección de un escenario de transformación se realiza utilizando diferencias entre modelos, expresado mediante EMFDiff, y el modelo de traza de ejecución basado en el metamodelo Tracer. Por lo tanto, la detección del escenario de adaptación solo depende del formato de las diferencias y la traza de ejecución, siendo independiente del lenguaje de transformación. Por otro lado, el metamodelo creado para la especificación de operaciones de adaptación, el metamodelo *MMAdaptationGoal*, extiende el metamodelo ATL. El metamodelo *MMAdaptationGoal* solo permite expresar las operaciones de adaptación mediante conceptos del lenguaje ATL. El uso de ATL en el metamodelo *MMAdaptationGoal* limita la aplicación de TRANSEVOL a transformaciones implementadas en el lenguaje ATL. Para mejorar la reusabilidad de la herramienta frente a otros lenguajes de transformación parte de los futuros trabajos se centrarán en adaptar el sistema para ofrecer modelos de operaciones de adaptación aplicables a diferentes lenguajes de transformación.

7.2.6 Escalabilidad

Puede ocurrir que la dimensión de los modelos a modificar sea muy elevada respecto al número de instancias. En estas situaciones la modificación de modelos puede ser una tarea tediosa y propensa a errores. Por ello, al especificar escenarios de evolución se recomienda trabajar con modelos de dimensiones pequeñas siempre que sea posible. Por otro lado se deben de realizar validaciones en sistemas de DSDM con una mayor dimensión tanto de metamodelos como de reglas de transformación.

7.2.7 Perfiles EMF

Uno de los objetivos del método TRANSEVOL es localizar y deducir las adaptaciones a realizar en las transformaciones cuando se necesita cambiar la lógica de mapeo debido a la aplicación de estereotipos. Hasta este momento solo se han realizado validaciones en sistemas donde se

ha utilizado perfiles UML. Se deben de realizar validaciones en sistemas de DSDM que utilicen perfiles EMF.

7.2.8 Escenarios de evolución ΔIm = 0

Existen varios escenarios contemplados por TRANSEVOL donde **ΔIm = 0**. En estas circunstancias cuando la operación de adaptación requiere un filtrado en una regla N a N no se puede deducir cual es el patrón de entrada afectado. En estas situaciones se debe de ofrecer un mecanismo para ofrecer dicha información en los modelos ejemplo de entrada. Una solución es utilizar estereotipos para especificar el patrón de entrada en los modelos ejemplo de entrada. Otra solución puede ser utilizar un modelo de intenciones y relacionar la intención con los modelos ejemplo mediante weaving. Lo mismo ocurre cuando, siendo **ΔIm = 0**, se quiere especificar un nuevo patrón de salida en una regla de transformación que no está relacionado con el contenedor de las agregaciones de salida.

7.3 Líneas futuras

Tras analizar los objetivos alcanzados en este trabajo y las características del método TRANSEVOL nuevos horizontes se han abierto. A continuación se resumen las principales líneas futuras de trabajo que se van a tratar de seguir.

7.3.1 Implementación de una herramienta ágil y usable para la aplicación del método TRANSEVOL

El primer objetivo es convertir la herramienta TRANSEVOL en un plugin de Eclipse con una interfaz gráfica usable, ágil y amigable. El objetivo principal del plugin es asistir al desarrollador de transformaciones M2M en la evolución de las reglas de transformación. Para ello la herramienta seguirá el siguiente flujo:

1. Seleccionar la transformación M2M ATL a evolucionar.
2. Seleccionar el modelo de entrada previo.
3. Ejecutar la transformación ATL y obtener:
    a. El modelo de salida previo.
    b. La traza de ejecución.
4. Seleccionar el modelo de entrada previo para modificarlo y crear así el modelo ejemplo de entrada.
5. Seleccionar el modelo de entrada previo para modificarlo y crear así el modelo ejemplo de entrada.
6. Validar si el escenario de evolución es conocido por TARNSEVOL.
7. Ejecutar el análisis de la transformación M2M y generar el modelo de operaciones de adaptación.
8. Ejecutar un HOT para generar modificar la transformación M2M automáticamente.

9. Implementar manualmente el algoritmo de las funciones de filtrado y asignación.
10. Ejecutar la nueva transformación M2M con el modelo ejemplo de entrada y validar que su salida coincide con el modelo ejemplo de salida.
11. Guardar la nueva versión de la transformación M2M junto con sus modelos ejemplos y una descripción de lo realizado.

Actualmente la herramienta TRANSEVOL ejecuta el séptimo paso. La generación de todos los artefactos necesarios para el análisis es responsabilidad del desarrollador. En este trabajo se utiliza el HOT ATL2Tracer para obtener la traza de ejecución de transformaciones ATL en el formato TRACER. El plugin EMFCompare para comparar las diferencias entre modelos. Y el editor de modelos de EMF es el utilizado para generar los modelos ejemplos. Uno de los objetivos es integrar todos estos pasos en la herramienta. Por otro lado, para que TRANSEVOL sea usable un punto fundamental es la implementación de un HOT que presente como entrada una transformación ATL y el modelo de operaciones de adaptación y realice los cambios deducidos automáticamente. Por último, es importante remarcar que para la completa validación de la nueva implementación de la transformación M2M es necesario que tras el paso 10 se realice una validación exhaustiva de la implementación de la transformación frente a diferentes pares de modelos ejemplos. Para ello sería fundamental integrar trabajos enfocados en la validación automática de transformaciones M2M basados en la cobertura de metamodelos como en [Sen09].

7.3.2 Desarrollo de una metodología y lenguaje para la especificación de requisitos de transformación basado en TRANSEVOL

El uso de un lenguaje de alto nivel para la especificación de requisitos de transformación puede ser muy útil para la detección de escenarios de evolución erróneos. Para poder combinar la información de los modelos ejemplo con los modelos que especifican la intención de mapeo una posibilidad es utilizar el weaving entre modelos. El lenguaje de alto nivel también puede ser utilizado para dividir un escenario complejo de mapeo, por ejemplo una transformación N a N, en varios escenarios de evolución que contempla TRANSEVOL. Ante estas situaciones el sistema puede proponer una serie de escenarios que permiten construir el requisito complejo en base a varias iteraciones.

Como se ha citado previamente otro aspecto crítico es la creación manual de los modelos ejemplos. Una línea a investigar es la generación automática de los modelos ejemplo mediante los modelos de intención de mapeos. De esta forma se respondería a la problemática de modelos ejemplo erróneos y de gran dimensión.

7.3.3 Generalización del método TRANSEVOL para su aplicación a diferentes lenguajes de transformación

A pesar de que el metamodelo *MMAdaptationGoal* se puede utilizar con transformaciones basadas en lenguajes que no sean ATL, la especificación de las reglas de adaptación está limitadas a los conceptos comunes de ambos lenguajes. Por eso, uno de los trabajos fundamentales es generalizar el metamodelo *MMAdaptationGoal* de forma que permita expresar las operaciones de adaptación mediante elementos que no dependan del lenguaje de transformación. Además de generalizar el metamodelo *MMAdaptationGoal* también habría que crear diferentes HOTs, una para cada lenguaje de transformación, de forma que se pueda automatizar la implementación de las operaciones de adaptación para diferentes lenguajes.

Algo parecido ocurre con el metamodelo Tracer que utiliza la herramienta TRANSEVOL. Actualmente se utiliza un HOT implementado en ATL que permite modificar transformaciones escritas en el lenguaje ATL de forma que, éstas, al ejecutarse además de generar el modelo de salida generan un modelo con la traza de ejecución conforme al metamodelo Tracer. Una posibilidad para integrar en TRANSEVOL los metamodelos de trazas de otros lenguajes de transformación, es transformar modelos de trazas en trazas conforme al metamodelo Tracer.

8. Bibliografía

[Aba12] ABADE, Maryam Nooraei; RAJABI, Zeinab. Integrating Non-Functional Properties in Model Driven Development: A Stepwise Refinement View. En *International Journal of Information Technology and Computer Science (IJITCS)*, 2012, vol. 4, no 9, p. 1.

[Agi10] AGIRRE, Joseba Andoni; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Plataforma DSDM para la Generación de Software Basado en Componentes en Entornos Empotrados. En JI*SBD*. 2010. p. 7-15.

[Agi12] AGIRRE, J.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. A flexible model driven software development process for component based embedded control systems. *III Jornadas de Computación Empotradas JCE, SARTECO*, 2012

[Agi14a] AGIRRE, Joseba A; SAGARDUI, Goiuria; ETXEBERRIA, Leire. 2014. Generación de código ANSI-C de modelos de componentes UML para sistemas empotrados. En MOLINA, Jesús García, et al.(eds). *Desarrollo de software dirigido por modelos: conceptos, métodos y herramientas*. Ra-Ma, pp.391-423. ISBN 9788499642154.

[Agi14b] AGIRRE, Joseba A.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Transevol: A tool to evolve legacy Model Transformations By Example. En *Software Engineering and Applications (ICSOFT-EA), 2014 9th International Conference on*. IEEE, 2014. p. 234-245.

[Agi15] AGIRRE, Joseba A.; SAGARDUI, Goiuria; ETXEBERRIA, Leire. Evolving legacy model transformations to aggregate non functional requirements of the domain. En *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*. IEEE, 2015. p. 437-448.

[Amm16] AMMAR, Manel, et al. Automatic Generation of S-LAM Descriptions from UML/MARTE for the DSE of Massively Parallel Embedded Systems. En *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*. Springer International Publishing, 2016. p. 195-211.

[Ame10] AMELLER, David; FRANCH, Xavier; CABOT, Jordi. Dealing with non-functional requirements in model-driven development. En *Requirements Engineering Conference (RE), 2010 18th IEEE International*. IEEE, 2010. p. 189-198.

[Apr13] APVRILLE, Ludovic; ROUDIER, Yves. SysML-Sec: A SysML environment for the design and development of secure embedded systems. *APCOSEC, Asia-Pacific Council on Systems Engineering*, 2013, p. 8-11.

[Atk00] ATKINSON, Colin; BAYER, Joachim; MUTHIG, Dirk. Component-based product line development: the KobrA approach. En *Software Product Lines*. Springer US, 2000. p. 289-309.

[ATT10] ATTEST consortium. Report on EAST-ADL2 language definition and profile. Deliverable D4.1.1a. [online].2010. [Accedido 14 de Feberero del 2017]. Accesible desde http://www.attest.org/

[Bal09] BALOGH, Zoltán; VARRÓ, Dániel. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 2009, vol. 8, no 3, p. 347-364.

[Bol08] BOLLATI, Veronica Andrea, et al. Applying MDE to the (semi-) automatic development of model transformations. *Information and Software Technology*, 2013, vol. 55, no 4, p. 699-718.

[Bru08] BRUN, Cédric; PIERANTONIO, Alfonso. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 2008, vol. 9, no 2, p. 29-34.

[Cue10] CUENOT, Philippe, et al. 11 the east-adl architecture description language for automotive embedded software. En *Model-based engineering of embedded real-time systems*. Springer Berlin Heidelberg, 2010. p. 297-307.

[Cza03] CZARNECKI, Krzysztof; HELSEN, Simon. Classification of model transformation approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. 2003. p. 1-17.

[Del06] DEL FABRO, Marcos Didonet; JOUAULT, Frédéric. Model transformation and weaving in the AMMA platform. *Proceedings of GTTSE*, 2005, vol. 2006.

[Del09] DEL FABRO, Marcos Didonet; VALDURIEZ, Patrick. Towards the efficient development of model transformations using model weaving and matching transformations. *Software & Systems Modeling*, 2009, vol. 8, no 3, p. 305-324.

[Dem16] DEMUTH, Andreas, et al. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 2016, vol. 111, p. 281-297.

[Dir12] DI RUSCIO, Davide; IOVINO, Ludovico; PIERANTONIO, Alfonso. Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. En *International Conference on Graph Transformation*. Springer Berlin Heidelberg, 2012. p. 20-37.

[Dir13] DI RUSCIO, Davide; IOVINO, Ludovico; PIERANTONIO, Alfonso. A methodological approach for the coupled evolution of metamodels and atl transformations. En *International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2013. p. 60-75.

[Ebe15] EBEID, Emad; FUMMI, Franco; QUAGLIA, Davide. HDL code generation from UML/MARTE sequence diagrams for verification and synthesis. Design automation for embedded systems, 2015, vol. 19, no 3, p. 277-299.

[Ebe09] EBERT, Christof; JONES, Capers. Embedded software: Facts, figures, and future. *Computer*, 2009, vol. 42, no 4.

[Eck16] ECKHARDT, Jonas; VOGELSANG, Andreas; FERNÁNDEZ, Daniel Méndez. Are non-functional requirements really non-functional?: an investigation of non-functional requirements in practice. En *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016. p. 832-842.

[Eli05] ELISA, BARCHINI Graciela. Métodos “I+ D” de la Informática Revista de Informática Educativa y Medios Audiovisuales Vol. 2 (5), págs. 16-24. *Universidad Nacional de Santiago del Estero. Argentina. Consultado el*, 2005, vol. 15.

[Ell15] ELLISON, Robert, et al. Extending AADL for Security Design Assurance of Cyber-Physical Systems. 2015.

[Ern11] ERNST, Neil A.; BORGIDA, Alexander; MYLOPOULOS, John. Requirements evolution drives software evolution. En *Proceedings of the 12th International*

Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution. ACM, 2011. p. 16-20.

[Erg16] ERGIN, Huseyin; SYRIANI, Eugene; GRAY, Jeff. Design pattern oriented development of model transformations. *Computer Languages, Systems & Structures*, 2016, vol. 46, p. 106-139.

[Far06] FARAIL, Patrick, et al. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. *Embedded Real Time Software (ERTS)*, 2006, vol. 781, p. 54-59.

[Fal08] FALLERI, Jean-Rémy, et al. Metamodel matching for automatic model transformation generation. En *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2008. p. 326-340.

[Fau13] FAUNES, Martin; SAHRAOUI, Houari; BOUKADOUM, Mounir. Genetic-programming approach to learn model transformation rules from examples. En *International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2013. p. 17-32.

[Fei06] FEILER, Peter H.; GLUCH, David P.; HUDAK, John J. *The architecture analysis & design language (AADL): An introduction*. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[Fle09] FLEUREY, Franck, et al. Qualifying input test data for model transformations. *Software & Systems Modeling*, 2009, vol. 8, no 2, p. 185-203.

[Gal14] GALLINO, Juan Pedro Silva, et al. Estrategia guiada por modelos para incluir aspectos de seguridad en sistemas empotrados basados en servicios web. *Revista Iberoamericana de Automática e Informática Industrial RIAI*, 2014, vol. 11, no 1, p. 86-97.

[Gar12] GARCÍA, Jokin; DIAZ, Oscar; AZANZA, Maider. Model transformation co-evolution: A semi-automatic approach. En *International Conference on Software Language Engineering*. Springer Berlin Heidelberg, 2012. p. 144-163.

[Gar14] GARCÍA, Jokin; DÍAZ, Oscar; CABOT, Jordi. An adapter-based approach to co-evolve generated sql in model-to-text transformations. En *International Conference on Advanced Information Systems Engineering*. Springer International Publishing, 2014. p. 518-532.

[Gar09] GARCÍA-MAGARIÑO, Iván, et al. A tool for generating model transformations by-example in multi-agent systems. En *7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009)*. Springer Berlin Heidelberg, 2009. p. 70-79.

[Ger02] GERY, Eran; HAREL, David; PALACHI, Eldad. Rhapsody: A complete life-cycle model-based development system. En *International Conference on Integrated Formal Methods*. Springer Berlin Heidelberg, 2002. p. 1-10.

[Gue10] GUERRA, Esther, et al. transML: A family of languages to model model transformations. En *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2010. p. 106-120.

[Gue12] GUERRA, Esther, et al. Engineering model transformations with transML. *Software & Systems Modeling*, 2013, vol. 12, no 3, p. 555-577.

[Gon17] GONÇALVES, Fabíola, et al. Applying MARTE Profile for Optimal Automotive System Specifications and Design. En *Proceedings of the 50th Hawaii International Conference on System Sciences*. 2017.

[Gon14] GONZÁLEZ-HUERTA, Javier; ABRAHÃO, Silvia; INSFRAN, Emilio. Automatic derivation of AADL product architectures in software product line development. En *Proc. of the 1st Architecture Centric Virtual Integration Workshop. Valencia: CEUR*. 2014. p. 69-78.

[Gon12] GONZALEZ-HUERTA, Javier; INSFRAN, Emilio; ABRAHAO, Silvia. A multi-model for integrating quality assessment in model-driven engineering. En *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. IEEE, 2012. p. 251-254.

[Gra03] GRADECKI, Joseph D.; COLE, Jim. *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[Har93] HARKER, Susan DP; EASON, Ken D.; DOBSON, John E. The change and evolution of requirements as a challenge to the practice of software engineering. En *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. IEEE, 1993. p. 266-272.

[Has07] HAASE, Arno, et al. Introduction to openArchitectureWare 4.1. 2. En *MDD Tool Implementers Forum*. 2007.

[Hau12] HAUGEN, Øystein; WASOWSKI, Andrzej; CZARNECKI, Krzysztof. CVL: common variability language. En *SPLC (2)*. 2012. p. 266-267.

[Her09] HERRMANNSDOERFER, Markus; BENZ, Sebastian; JUERGENS, Elmar. COPE-automating coupled evolution of metamodels and models. En *European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg, 2009. p. 52-76.

[Her14] HERRERA, Fernando, et al. The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems. *Journal of Systems Architecture*, 2014, vol. 60, no 1, p. 55-78.

[ISO07] ISO/EIC 2500. *ISO/EIC 2500 System and Software quality requirements and Evaluation (SQuaRE)*. 2007. [Accedido 15 de Febrero del 2017]. Accessible desde *http://iso25000.com*.

[Ins08] INSFRAN, Emilio, et al. Towards quality-driven model transformations: A replication study. En *Proceedings of the 1st workshop on empirical studies of model-driven engineering (ESMDE'08)*. 2008. p. 51-60.

[Iov12] IOVINO, Ludovico; PIERANTONIO, Alfonso; MALAVOLTA, Ivano. On the Impact Significance of Metamodel Evolution in MDE. *Journal of Object Technology*, 2012, vol. 11, no 3, p. 3:1-33.

[Jää04] JÄÄLINOJA, Juho. Requirements implementation in embedded software development. *Espoo 2004*, 2004.

[Jou05] JOUAULT, Frédéric. Loosely coupled traceability for ATL. En *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*. 2005. p. 2.

[Jou08] JOUAULT, Frédéric, et al. ATL: A model transformation tool. *Science of computer programming*, 2008, vol. 72, no 1, p. 31-39.

[Kap12] KAPPEL, Gerti, et al. Model transformation by-example: a survey of the first wave. En *Conceptual Modelling and Its Theoretical Foundations*. Springer Berlin Heidelberg, 2012. p. 197-215.

[Kay08] KAY, Michael. XLST 2.0 and XPath 2.0, Programmer's Reference, 4th edtiion. 2008.

[Kes12] KESSENTINI, Marouane, et al. Search-based model transformation by example. *Software & Systems Modeling*, 2012, vol. 11, no 2, p. 209-226.

[Kha16] KHAN, Aamir M.; RASHID, Muhammad. Generation of SystemVerilog Observers from SysML and MARTE/CCSL. En *Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on*. IEEE, 2016. p. 61-68.

[Kle03] KLEPPE, Anneke G.; WARMER, Jos B.; BAST, Wim. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[Kru11] KRUSE, Steffen. On the use of operators for the co-evolution of metamodels and transformations. En *International Workshop on Models and Evolution*. 2011.

[Kug08] KUGELE, Stefan, et al. Optimizing automatic deployment using non-functional requirement annotations. En *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer Berlin Heidelberg, 2008. p. 400-414.

[Kus15] KUSEL, Angelika, et al. Consistent co-evolution of models and transformations. En *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*. IEEE, 2015. p. 116-125.

[Lan10] LANGER, Philip; WIMMER, Manuel; KAPPEL, Gerti. Model-to-model transformations by demonstration. En *International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2010. p. 153-167.

[Lan12] LANGER, Philip, et al. EMF Profiles: A Lightweight Extension Approach for EMF Models.*Journal of Object Technology*, 2012, vol. 11, no 1, p. 1-29.

[Lan05] LANO, Kevin; ANDROUTSOPOLOUS, Kelly; CLARK, David. Refinement Patterns for UML. *Electronic Notes in Theoretical Computer Science*, 2005, vol. 137, no 2, p. 131-149.

[Lan09] LANUSSE, Agnes, et al. Papyrus UML: an open source toolset for MDA. En *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. 2009. p. 1-4.

[Lei14] LEITE, Marcela; VASCONCELLOS, Cristiano D.; WEHRMEISTER, Marco Aurélio. Enhancing automatic generation of VHDL descriptions from UML/MARTE models. En *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*. EEE, 2014. p. 152-157

[Lev13] LEVY, F.; MUNIZ, P. Applying MTBE manually: a method and an example. *MDEBE@ MoDELS*, 2013.

[Lie14] LIEBEL, Grischa, et al. Assessing the state-of-practice of model-based engineering in the embedded systems domain. *International Conference on Model Driven Engineering Languages and Systems*. Springer International Publishing, 2014. p. 166-182.

[Luc16] LÚCIO, Levi, et al. Model transformation intents and their properties. *Software & systems modeling*, 2016, vol. 15, no 3, p. 647-684.

[Mar14] MARKO, Nadja, et al. Model-based engineering for embedded systems in practice. 2014.

[MAR11] MARTE, OMG 1.1 Specification. *UML profile for MARTE: Modeling and analysis of Real-Time Embedded Systems.* 2011. [Accedido 15 de Febrero del 2017]. Accesible desde http://www.omg.org/spec/MARTE/1.1/.

[Mil03] MILLER, Joaquin, et al. MDA Guide Version 1.0. 1. 2003.

[Mol09] MOLINA, Fernando; TOVAL, Ambrosio. Integrating usability requirements that can be evaluated in design time into Model Driven Engineering of Web Information Systems. *Advances in Engineering Software*, 2009, vol. 40, no 12, p. 1306-1317.

[Mof06] MOF, O. M. G. 2.0 core specification.*OMG Document, January*, 2006.

[Mus06] MUSSET, Jonathan, et al. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2006, vol. 2.

[Mye97] MYERS, Michael D., et al. Qualitative research in information systems. *Management Information Systems Quarterly*, 1997, vol. 21, no 2, p. 241-242.

[Old06] OLDEVIK, Jon. MOFScript eclipse plug-in: metamodel-based code generation. En *Eclipse Technology Workshop (EtX) at ECOOP*. 2006

[OMG08] OMG, MOF. Model to Text Transformation Language. *Version*, 2008, vol. 1, p. 2008-01.

[Pan08] PANACH, Jose Ignacio, et al. Dealing with usability in model transformation technologies. En *International Conference on Conceptual Modeling*. Springer Berlin Heidelberg, 2008. p. 498-511.

[Peñ10] PEÑIL, Pablo, et al. Generating heterogeneous executable specifications in SystemC from UML/MARTE models. *Innovations in Systems and Software Engineering*, 2010, vol. 6, no 1, p. 65-71.

[Ros08a] ROSE, Louis M., et al. The epsilon generation language. En *European Conference on Model Driven Architecture-Foundations and Applications*. Springer Berlin Heidelberg, 2008. p. 1-16.

[Ros08b] ROSER, Stephan; BAUER, Bernhard. Automatic generation and evolution of model transformations using ontology engineering space. En *Journal on Data Semantics XI*. Springer Berlin Heidelberg, 2008. p. 32-64.

[Ros10a] ROSE, Louis M., et al. A comparison of model migration tools. En *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2010. p. 61-75.

[Ros14] ROSE, Louis M., et al. Epsilon Flock: a model migration language. *Software & Systems Modeling*, 2014, vol. 13, no 2, p. 735-755.

[Saw01] SAWYER, Pete; KOTONYA, Gerald. Software requirements. *SWEBOK*, 2001, p. 9.

[Sen03] SENDALL, Shane; KOZACZYNSKI, Wojtek. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 2003, vol. 20, no 5, p. 42-45.

[Sen12] SEN, Sagar, et al. Reusable model transformations. *Software & Systems Modeling*, 2012, vol. 11, no 1, p. 111-125.

[She15] SHENG, Zhengguo, et al. Lightweight management of resource-constrained sensor devices in internet of things.*IEEE internet of things journal*, 2015, vol. 2, no 5, p. 402-411.

[Sch93] SCHREINER, Axel-Tobias. Object oriented programming with ANSI-C. 1993.

[Ste08] STEINBERG, Dave, et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[Ste08] STERRITT, Ashley; CAHILL, Vinny. Customisable model transformations based on non-functional requirements. En *Services-Part I, 2008. IEEE Congress on*. IEEE, 2008. p. 329-336.

[Str08] STROMMER, Michael; WIMMER, Manuel. A framework for model transformation by-example: Concepts and tool support. En *International Conference on Objects, Components, Models and Patterns*. Springer Berlin Heidelberg, 2008. p. 372-391.

[Sun09] SUN, Yu; WHITE, Jules; GRAY, Jeff. Model transformation by demonstration. En *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2009. p. 712-726.

[Sun13] SUN, Yu; GRAY, Jeff. End-User support for debugging demonstration-based model transformation execution. En *European Conference on Modelling Foundations and Applications*. Springer Berlin Heidelberg, 2013. p. 86-100.

[Sun13] SUN, Yu, et al. Automating the maintenance of nonfunctional system properties using demonstration-based model transformation. *Journal of Software: Evolution and Process*, 2013, vol. 25, no 12, p. 1335-1356.

[Sys08] SYSML, O. M. G. 1.0 Specification. *Object Management Group*, 2008.

[Tis09] TISI, Massimo, et al. On the use of higher-order model transformations. En *European Conference on Model Driven Architecture-Foundations and Applications*. Springer Berlin Heidelberg, 2009. p. 18-33.

[Tou06] TOULMÉ, Antoine; INC, I. Presentation of EMF compare utility. En *Eclipse Modeling Symposium*. 2006. p. 1-8.

[Tra05] TRATT, Laurence. Model transformations and tool integration. *Software & Systems Modeling*, 2005, vol. 4, no 2, p. 112-122.

[Thr16] THRAMBOULIDIS, Kleanthis; CHRISTOULAKIS, Foivos. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry*, 2016, vol. 82, p. 259-272.

[Uml04] UML, O. M. G. 2.0 Superstructure Specification. *OMG, Needham*, 2004.

[Yin03] YIN, K. Robert. Case study Research: Design and methods. Sage publicatons. 2003.

[Van09] VAN AMSTEL, Marcel F.; LANGE, Christian FJ; VAN DEN BRAND, Mark GJ. Using metrics for assessing the quality of ASF+ SDF model transformations. En *International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2009. p. 239-248.

[Van11] VAN AMSTEL, Marcel F.; VAN DEN BRAND, Mark. Using metrics for assessing the quality of ATL model transformations. En *Proceedings of the Third International Workshop on Model Transformation with ATL (MtATL 2011)*. 2011. p. 20-34.

[Van07] VAN DEURSEN, Arie; VISSER, Eelco; WARMER, Jos. *Model-driven software evolution: A research agenda*. Delft University of Technology, Software Engineering Research Group, 2007.

[Var06] VARRÓ, Dániel. Model transformation by example. En *International Conference on Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2006. p. 410-424.

[Var14] VARA, Juan Manuel, et al. Dealing with traceability in the MDDof model transformations. *IEEE Transactions on Software Engineering*, 2014, vol. 40, no 6, p. 555-583.

[Vol05] VOELTER, Markus; SALZMANN, Christian; KIRCHER, Michael. Model driven software development in the context of embedded component infrastructures. *Component-Based Software Development for Embedded Systems*. Springer Berlin Heidelberg, 2005. p. 143-163.

[Wad07] WADA, Hiroshi; SUZUKI, Junichi; OBA, Katsuya. A feature modeling support for non-functional constraints in service oriented architecture. En *Services Computing, 2007. SCC 2007. IEEE International Conference on*. IEEE, 2007. p. 187-195.

[Wim12] WIMMER, Manuel, et al. A Catalogue of Refactorings for Model-to-Model Transformations. *Journal of Object Technology*, 2012, vol. 11, no 2, p. 2:1-40.

[Zhu07] ZHU, Liming; GORTON, Ian. Uml profiles for design decisions and non-functional requirements. En *Proceedings of the Second Workshop on Sharing and Reusing Architectural Knowledge Architecture, Rationale, and Design intent*. IEEE Computer Society, 2007. p. 8.

Anexo: Sistema de generación de código DSDM *fromUML2SimpleC*

El sistema de generación de código *fromUML2SimpleC* ha sido utilizado para realizar la validación de TRANSEVOL en un escenario real. *fromUML2SimpleC* es un sistema de DSDM para la generación de código ANSI-C de modelos de componentes UML para sistemas empotrados. Este sistema fue presentado en [Agi12] y este anexo es una versión reducida del capítulo 19 del libro [Agi14a], donde se describe el sistema en mayor detalle junto con un ejemplo de uso completo de dicho sistema de generación de código. El sistema de generación de código combina los paradigmas de DSDM y CBD (Component Based Development) de modo que permite especificar la arquitectura software de sistemas de control concurrentes basado en componentes y generar automáticamente el código en ANSI-C desplegable en plataformas empotradas. El diseño se realiza mediante el lenguaje de modelado UML. Para poder utilizar UML con una semántica adecuada que permita especificar plataformas de ejecución se ha extendido mediante perfiles UML, concretamente se ha utilizado el perfil MARTE [Mar11]. Los diseños UML de las aplicaciones se convierten en código ANSI-C mediante una transformación M2M y otra transformación M2T. Estas transformaciones implementan las buenas prácticas y patrones de diseño necesarios para poder tratar con conceptos de componentes software en C [Sche93]. Estas transformaciones se pueden aplicar a la generación de código de múltiples productos. En este apartado se describe la arquitectura del sistema de DSDM y la generación de código con el objetivo de dimensionar la complejidad del sistema de DSDM utilizado como caso de uso en la validación final del método TRANSEVOL.

A.1 Estructura global MDA utilizada para la generación de código

El proceso de generación de código se ha dividido en dos etapas, ver figura A.1. Primero se utiliza una transfortmación M2M, donde los modelos de componentes, basados en el metamodelo *UML2+MARTE*, se transforman en modelos *SimpleC*, basados en el metamodelo *SimpleC*. *SimpleC* es un metamodelo que permite expresar un subconjunto de ANSI-C. En una segunda etapa una plantilla, implementa en XPAND2, es aplicada a los modelos *SimpleC* para generar el código ANSI-C. El objetivo de esta separación en etapas es doble, por un lado, simplifica el proceso de generación de código, y por otro, las transformaciones M2T pueden reutilizarse en otros proyectos.

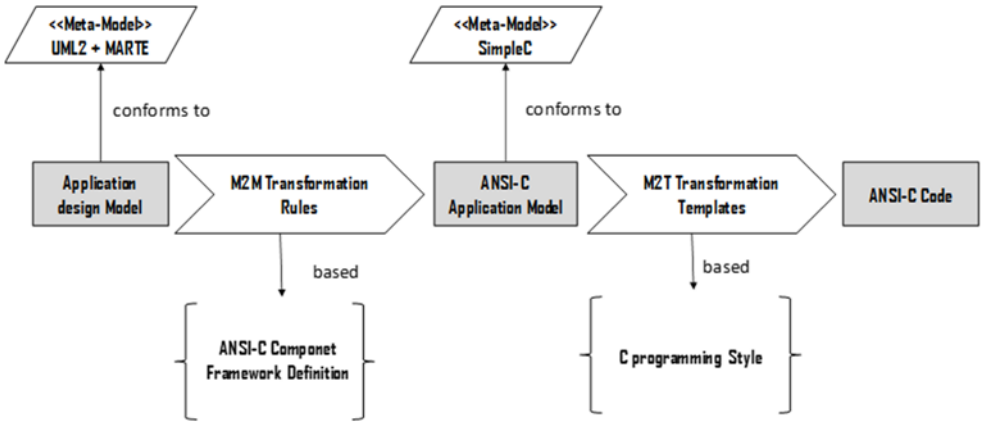


Figura A.1 Proceso de generación de código *fromUML2C*.

A continuación se enumeran las herramientas y tecnologías utilizadas para la implementación del sistema de generación de código. Para el metamodelado se ha utilizado EMF. Para el metamodelo UML se ha utilizado la versión 2.3 basada en EMF. El metamodelo *SimpleC* ha sido definido e implementado en un proyecto EMF. Para el modelado UML de la arquitectura software se ha utilizado la herramienta Papyrus 1.12. Esta herramienta se puede utilizar tanto como plugin de eclipse o como una adaptación de eclipse. Este plugin además de permitir un diseño visual de diagramas UML, presenta una implementación del perfil UML-MARTE, permitiendo estereotipar diseños UML mediante el perfil MARTE. Para las transformaciones M2M se ha utilizado ATL. Para las transformaciones M2T se ha utilizado XPAND2 de OpenArchitectureWare. El sistema de DSDM consta de cuatro proyectos:

a) *fromUML2SimpleC*: Proyecto ATL donde se implementa la transformación M2M. Esta transformación convierte modelos de componentes UML estereotipados y diseñados conjuntamente con el perfil MARTE en modelos SimpleC.
b) *MMSimpleC*: Proyecto EMF donde se define el metamodelo *SimpleC*, el cual define un subconjunto del lenguaje de programación ANSI-C.
c) *SimpleC2Code*: Proyecto XPAND2 que permite generar código ANSI-C teniendo como entrada modelos SimpleC.
d) *KOBRAProfile*: Debido a que UML no diferencia entre instancia de componente y tipo de componente ha sido necesario crear un proyecto UML-Profile de Papyrus para la definición de los estereotipos para el uso de ambos conceptos. El diseño de los componentes UML se ha realizado utilizando la metodología KOBRA [Atk00], donde es fundamental diferenciar entre instancia y tipo de componente. A pesar del nombre del proyecto solamente se ha implementado un subconjunto de estereotipos de la metodología KOBRA.

Todos los ficheros correspondientes a este proyecto se encuentran en el blog del máster de sistemas embebido de MU (http://mukom.mondragon.edu/master-sistemas-embebidos/). En la figura A.2 se puede ver la estructura de carpetas de cada uno de los proyectos

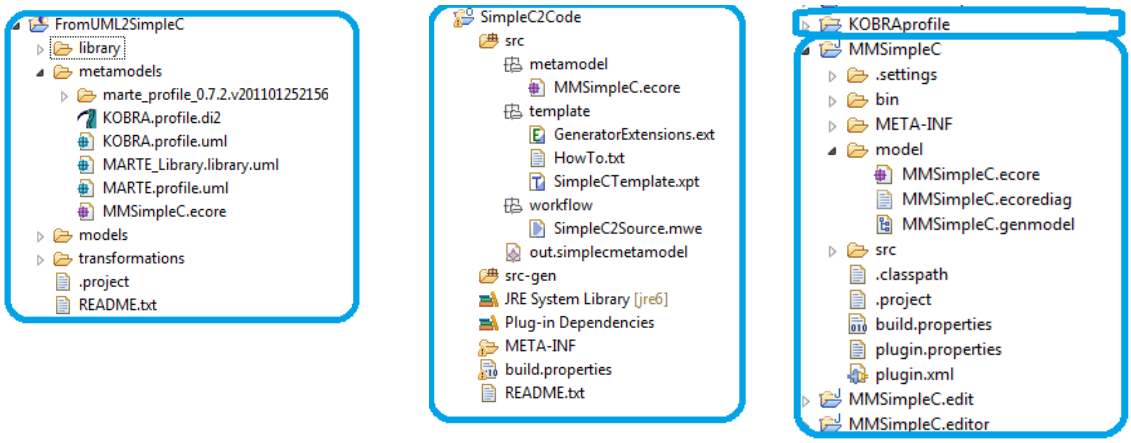


Figura A.2 Estructura del proyecto de generación de código *fromUML2SimplreC*.

A.2 Proyecto fromUML2SimpleC

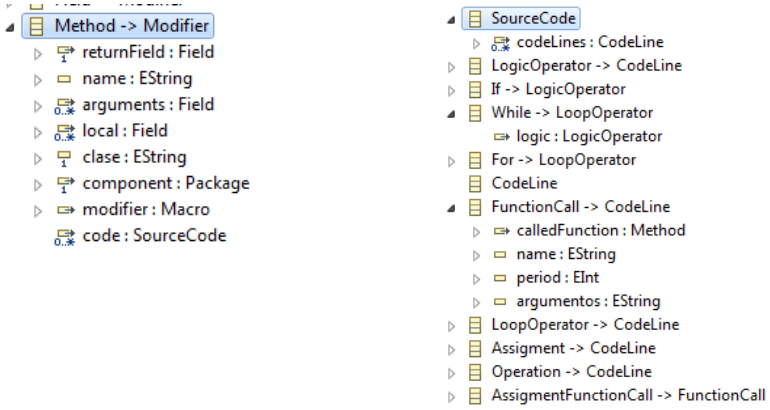
En el proyecto *fromUM2SimpleC* se implementa la transformación M2M que transforma modelos UML en modelos SimpleC. *fromUM2SimpleC* es un proyecto ATL. En la carpeta *transformations* se encuentran todas las reglas de transformación M2M. En concreto son 8 ficheros ATL. En la carpeta *library* están implementadas las funciones de ayuda para las reglas de transformación. Las funciones de ayuda que trabajan con conceptos del perfil MARTE se han agrupado en el fichero *UMLMarteBaremetalPlatformHelper.atl*. Las que trabajan con los conceptos de instancias de componentes se encuentran en el fichero *UMLComponentInstancesHelper.atl*. Los conceptos referentes a la definición de la estructura interna del componente se encuentran en *UMLComponentTypeHelper.atl* y *UMLComponentTypeInternalHelper.atl*,y los relativos a sus interfaces en *UMLComponentTypeInterfaes.atl*. Finalmente han sido necesarias ciertas funciones de ayuda para navegar en modelos SimpleC, localizadas en *SimpleCHelper.atl.*

En la carpeta *metamodels* se localizan el metamodelo SimpleC, *MMSimpleC*, y el perfil UML-KOBRA (*KOBRA.profile.uml*). Estas dos metamodelos también se localizan en las carpetas de sus respectivos proyectos. En la carpeta *models* se incluyen los diferentes modelos de entrada y salida de la transformación M2M. No es obligatorio que estos modelos se sitúen en esta carpeta, en caso contrario hay que indicarlo en la configuración de la ejecución de la transformación. En esta carpeta se encuentra un diseño de la arquitectura software de un sistema básico de control automático de puertas deslizantes (ficheros *doorSPL.uml* y *doorSPL.di*). Este modelo ha sido el utilizado como ejemplo de entrada para la transformación M2M.

El proyecto además de presentar las carpetas y ficheros anteriormente mencionados presenta una configuración de ejecución ATL (Configuración *ComponentType2SimpleC* ATL Transformation Launcher) para la transformación del *doorSPL.uml* en un modelo SimpleC (out.simplecmetamodel). También se presenta un fichero ANT que se puede duplicar y modificar para ejecutar la transformación con las entradas y salidas necesarias, sin tener que abrir el launcher de ATL.

A.3 Proyecto MMSimpleC

En este proyecto EMF se define el metamodelo *SimpleC* que se utiliza para especificar aplicaciones ANSI-C. El metamodelo está definido en el fichero *MMSimpleC.ecore* ubicado en la carpeta *models* del proyecto. El metamodelo *SimpleC* se usa para definir modelos que contienen conceptos básicos de un programa C: módulo, fichero, campo, método, macro, modificador, línea de código, etc. A modo ilustrativo, en la figura A.3 se presenta la parte del metamodelo correspondiente a las funciones y al código fuente.



A.3 Figura Parte del metamodelo SimpleC: funciones y código fuente.

A.4 Proyecto SimpleC2Code

Este proyecto XPAND2 se encarga de transformar modelos *SimpleC* en código ANSI-C. La transformación de modelos *SimpleC* a código ANSI-C se realiza en base a plantillas XPAND2. Para cada elemento del metamodelo *SimpleC* se ha creado una plantilla que define el texto relacionado. Las plantillas crean estructuras, ficheros de cabecera, campos de datos, funciones, líneas de código, etc. A continuación se presenta la estructura del proyecto. En la carpeta *src* del proyecto se encuentran los directorios *templates*, *workflow* y *metamodel*.

En la carpeta *Template* se encuentra el fichero-plantilla XPAND *SimpleCTemplate.xpt* donde se define en que texto se transforma cada elemento del metamodelo *SimpleC*.

En la carpeta *workflow* se encuentra el fichero *SimpleC2Source.mwe*, el cual es el encargado de ejecutar la transformación modelo a texto. En este fichero se indica cual es el metamodelo para los modelos de entrada, el modelo simpleC de entrada, el fichero donde se encuentran definidos las plantillas a aplicar y también se especifica el elemento principal sobre el cual comienzan a aplicarse las plantillas XPAND2.

En la carpeta *metamodel* existe una copia del metamodelo SimpleC (*MMSimple.ecore*). En el directorio *src* se encuentra el fichero *out.simplecmetamodel*. Actualmente el fichero que define el workflow de ejecución de la transformación recibe como entrada este fichero. El fichero *out.simplecmetamodel* es una copia del modelo resultado obtenido al ejecutar la transformación

M2M *fromUML2SimpleC*. Por lo tanto, este fichero es una representación bajo el metamodelo *SimpleC* del diseño de la arquitectura software del controlador de puertas deslizantes definido en el fichero *doorSPL.uml*. El código ANSI-C generado al aplicar al modelo de *entrada out.simplecmetamodel* las transformaciones M2T se sitúa en el directorio *src-gen*.

A.5 Análisis de la transformación M2M: De UML2+MARTE a SIMPLEC

El objetivo de este apartado es ofrecer una descripción del diseño de la transformación M2M implementada. La lectura de este apartado es necesario en el caso que se quiera profundizar en la implementación de las reglas de transformación M2M; en caso contrario se puede prescindir de la lectura del mismo.

La transformación M2M se realiza de forma incremental mediante el mecanismo de superimposición de ATL [Wag10]. La transformación M2M está constituida por 8 ficheros. La configuración del orden para la superposición está registrada en la configuración del ejecutor de la transformación ATL (*ComponentType2SimpleC* ATL Transformation Launcher). Antes de describir el contenido de cada fichero ATL es conveniente explicar, sin demasiado detalle, como se implementan utilizando elementos ANSI-C el concepto de tipo de componente e instancias de componente. Las reglas de transformación M2M del proyecto *fromUML2SimpleC* están implementadas para crear modelos SimpleC que representan arquitecturas software basadas en componentes implementados bajo el patrón descrito en el siguiente sub-apartado.

A.5.1 Como crear componentes Software mediante elementos ANSI-C

ANSI-C y SimpleC carecen de la noción de componente y clase. Para mantener la encapsulación se combinan diferentes elementos del metamodelo SimpleC para implementar clases. La implementación de una clase UML mediante SimpleC se expresa combinando los siguientes elementos:

- Fichero de cabecera: describe la estructura de datos que contiene las propiedades de la clase UML. Los métodos de la clase UML se convierten en métodos SimpleC. Todos los métodos reciben como primer parámetro un puntero a la propia estructura de datos de manera que se mantenga el concepto de clase.
- Fichero código fuente: contiene la implementación de las funciones (implementaciones de los métodos UML).

En las figura A.4 y A.5 se representa simplificado el mapeo entre un componente UML y el código ANSI-C. A partir de la noción de clase, los diferentes conceptos relacionados con los componentes UML se mapean de la siguiente forma en elementos ANSI-C:

1. Estructura interna de un componente: Todos los ficheros correspondientes a un componente se localizan en una carpeta con el nombre del componente. La estructura

interna de un componente se compone de una clase SimpleC por cada componente (clase componente) y una clase SimpleC por cada una de las clases internas del componente. Por cada clase interna que contiene el componente es necesario incluir el fichero de cabecera de la clase SimpleC interna.

2. Interfaces ofrecidas por un componente: se implementan de igual manera a las clases SimpleC pero el fichero de cabecera contiene un puntero a la clase componente. Cuando alguien requiere algún servicio, la interfaz llama a la función del componente que ofrece dicha funcionalidad.
3. Interfaces requeridas por un componente: es una clase SimpleC. En el fichero de cabecera se incluye el fichero de cabecera del interfaz ofrecido que se va a usar y se crea una estructura con un puntero a la interfaz que ofrece los servicios que se requieren. En la estructura de la clase de componente se incluye una referencia a las interfaces requeridas.
4. Instancias de componentes: Se van a crear componentes estáticos que se definen mediante un fichero .h y .c. En el fichero de cabecera se define una variable que representa la instancia y para ello es necesario incluir el fichero de cabecera de la clase componente correspondiente. En caso de que esta instancia ofrezca interfaces, se tiene que implementar el puerto correspondiente a esa interfaz. El uso de puertos permite referenciar a servicios concretos de una instancia de componente reduciendo el acoplamiento. Los puertos UML se traducen en SimpleC como un fichero cabecera con una variable del tipo de la interface.
5. Relación entre las interfaces de las instancias (binding): en las interfaces ofrecidas se asigna la instancia del componente al que pertenece. En el caso de las interfaces requeridas, hay que incluir el fichero cabecera del puerto donde se encuentra la interfaz ofrecida correspondiente.

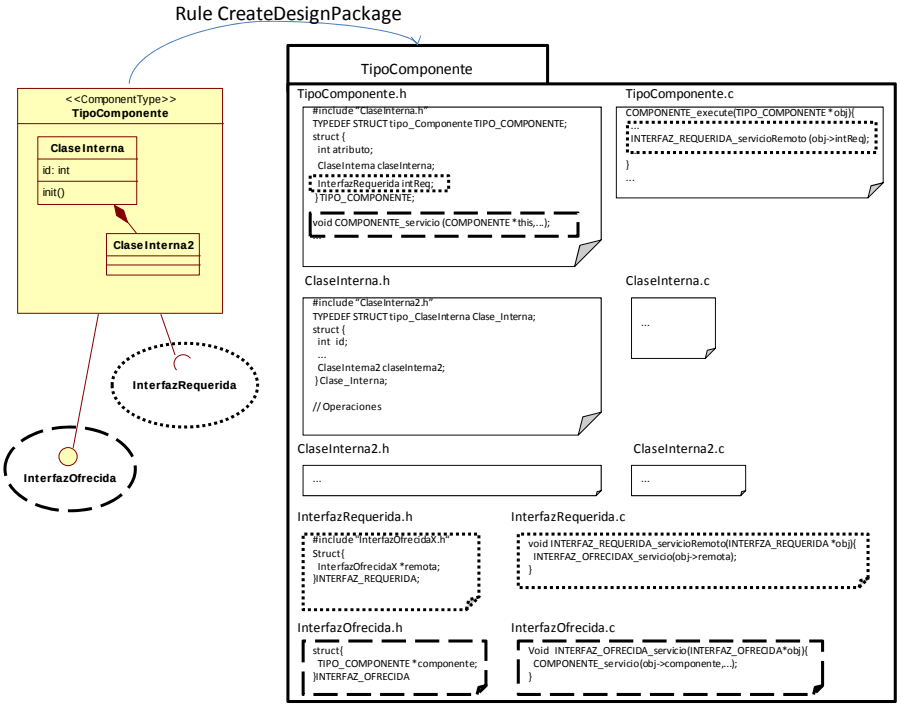


Figura A.4 Implementación en ANSI-C: tipos de componente y sus interfaces.

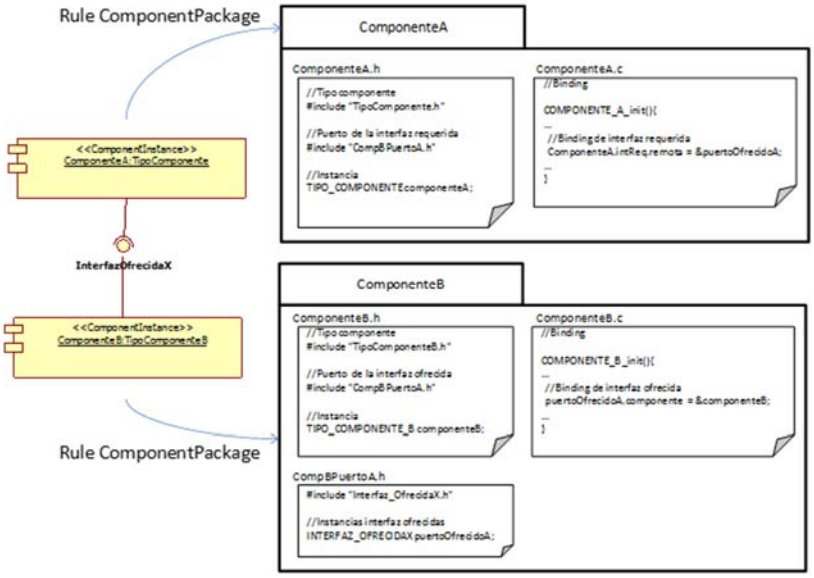


Figura A.5 Instancias de componentes y binding.

A.5.2 Implementación de las reglas de transformación M2M

A continuación se describe el objetivo de cada uno de los ficheros ATL que componen la transformación de modelos UML-MARTE a modelos SimpleC. Las reglas de transformación están basadas en el patrón de diseño para la implementación de componentes software basados en ANSI-C presentado en la sección anterior.

En el fichero *TypesCreation* se encuentran las reglas invocadas a utilizar para crear tipos de datos de *SimpleC* por cada elemento UML que lo requiera. Por ejemplo, por cada clase se debe de generar una estructura, lo mismo ocurre con las interfaces. Además existen reglas

invocadas que permiten asociar a una variable, propiedad o parámetro un tipo de dato. Estas reglas se utilizan en la creación de la estructura interna, instancias de componentes y demás.

Las regla de transformación del fichero *ComponentTypeProperties* crean las variables, elementos del tipo *SimpleC!Field*, por cada propiedad del modelo UML, exceptuando los elementos existentes en el modelo de tareas. En *ComponentTypeOperation* se crean todos los métodos *SimpleC* relacionados con el diseño de la aplicación, es decir el diseño de los tipos de componentes. También se ofrecen una serie de reglas invocadas que permiten crean el valor de retorno de las funciones y el argumento puntero que referencia a una clase en la función.

Para la creación de lo referente a las interfaces se utilizan los ficheros *ComponentTypeRequiredInterfaces* y *ComponentTypeProvidedInterfaces*. En el fichero ATL referente a las interfaces requeridas se crea un módulo, conjunto de fichero cabecera y fichero código fuente, por cada interfaz. En cada módulo se crea la estructura que representa la interfaz, se relaciona el módulo con sus funciones y también se genera el puntero a la interfaz remota que deben presentar todas las interfaces requeridas.

Las reglas de transformación relacionadas con las interfaces ofrecidas crean elementos muy parecidos al de las interfaces requeridas, con una excepción, generan un puntero al componente que implementa realmente el servicio expuesto de manera que pueda utilizar sus funciones para implementar el servicio correspondiente. A cada módulo se le agregan todos los includes necesarios (en el metamodelo SimpleC se utiliza el atributo *externals* para indicar los diferentes *includes* en un module SimpleC).

Los tres ficheros restantes se encargan de crear los elementos SimpleC necesarios para expresar tipos de componentes junto con su estructura interna (*ComponentType2SimpleC.atl*), todos los elementos SimpleC relacionados con el modelo de tareas MARTE diseñado (*ComponentInstanceTaskModel2SimpleC.atl*) y los elementos relacionados con las instancias (*ComponentInstance2SimpleC.atl*).

Las transformaciones agrupadas en el fichero *ComponentType2SimpleC.atl* crean la estructura interna de los componentes en SimpleC. Por cada clase estereotipada como <<*Komponent*>> (la función helper *isKomponent()* valida si este estereotipo está establecido o no) se crea un elemento package de SimpleC que representa un directorio. En este directorio se crean los módulos (conjunto fichero .c y .h) que representan la clase del componente y un módulo por cada clase interna del componente. En la figura A.6 se representa el código de esta transformación.

(a)

```
rule CreateDesignPackage{
    from
        c :  Uml!Class ( c.isKomponent()  )
    to
        p : SimpleC!Package(
            name<-c.name,
            path <- c.name
            ,modules <- thisModule.createKomponentModule(c)
            ,modules <- c.getInternalClasses()
        )
}

lazy rule createKomponentModule{
    from
        c :  Uml!Class ( c.isKomponent()  )
    to
        m : SimpleC!Module(
            name <- c.name
            ,path <- c.name
            ,file <- thisModule.ClassFile(c)
            ,newTypes <-thisModule.ClassStruct(c)
            ,methods <- c.getClassOperations()
            ,externals <- c.getPropertiesClasses()
        )
}
```

(b)

```
helper context Uml!Class  def : getInternalClasses(className : String) : Sequence(Uml!Class) =
    self.getAllCompositeProperties()
        ->iterate( p ; a : Sequence(Uml!Class) = Sequence{} |
            then
                a->including(p.type)->including( p.getPropertyClass().getInternalClasses() )
            else
                a->including(p.type)
            endif
        )
;
```

Figura A.6 (a) Reglas de transformación para crear los componentes (b) Función helper "getInternalClasses.

Para obtener todas las clases internas de un tipo de componente se utiliza la función helper *getInternalClasses* (librería *ComponentTypeInternalHelper.atl*) que realiza una navegación por todas las asociaciones relacionadas con el tipo de componente, localizando todas las clases internas. La regla *ComponentInstanc*e del fichero *ComponentType2SimpleC.atl* construye la instancia, creando una variable que representa la instancia, incluyendo todos los ficheros cabecera que necesita (fichero de cabecera que define el tipo de componente, los ficheros cabecera de los puertos ofrecidos y de los puertos requeridos y los ficheros cabecera que definen las tareas) como elementos los elementos *externals* que representan la inclusión de los ficheros cabecera. También crea las funciones *init-exec-destroy* de cada instancia de componente. La regla invocada *createInitMethod* crea la función *create* de cada instancia, que se encarga de realizar las inicializaciones pertinentes y de establecer las referencias necesarias, asociando punteros, para que la comunicación entre instancias sea correcta. El código de asociación de punteros establece por cada interfaz requerida de la instancia el puntero hacia la interfaz ofrecida de la instancia remota que implementa el servicio. En cada interfaz ofrecida se establece el puntero de la instancia que implementa el servicio. El código de asociación de punteros son asignaciones, denominadas en SimpleC como *SimpleC!Assigment*. En la imagen A.7 se presenta el código de la reglas de transformación *ComponentInstanc*e y *createInitMethod*.

(a)

```
rule ComponentPackage {
    from
        c : Uml!Component (not c.isApplication() )
    to
        m :SimpleC!Package(
            name <-c.name,
            path <- c.name ,
            modules <- thisModule.ComponentInstance(c) ,
            modules <- c.getComponentTaskModelInterface(),
            modules <- c.getComponentPorts()
        )
}


lazy rule ComponentInstance{
    from
        c : Uml!Component (not c.isApplication() )
    to
        m :SimpleC!Module(
            name <- c.name
            ,path <- c.name ,
            externals <-  c.getComponentTaskModel(),
            externals <-  c.getComponentPorts()
            ,externals <- c.getKomponentModule()
            ,methods <- thisModule.createInitMethod(c)
            ,methods <- thisModule.createDestroyMethod(c)
            ,methods <- thisModule.createExecMethod(c)
            ,file <-  thisModule. InstanceModuleFile(c)
            ,fields <- thisModule.createComponentField(c)
    )
}
```

(b)

```
lazy rule createInitMethod{
    from
        c : Uml!Component (not c.isApplication() )
    to
        m : SimpleC!Method(
            name <- c.name + '_init'
            ,code <- thisModule.createBindingCode( c )
        )
}
lazy rule createBindingCode{
    from
        c : Uml!Component (not c.isApplication() )
    to
        sc: SimpleC!SourceCode(
            codeLines <- c.getComponentPorts()->select(p | p.isRequiredPort() )->
                                                collect ( pa | thisModule.createRequiredPortBindingCode( pa )
            ,codeLines <- c.getComponentPorts()->select(p | p.isProvidedPort() )->
                                                collect ( pa |  thisModule.createProvidedPortBindingCode(pa) )
    )
}
```

Figura A.7 Figura (a) Reglas de transformación para crear las instancias (b) Reglas de transformación para la función *create* y para crear el código para el *binding*.

Las transformaciones agrupadas en el fichero ComponentInstanceTaskModel2SimpleC.atl se encargan de generar el código relacionado con el modelo de tareas. Cada instancia de componente tiene un fichero cabecera (nombreInstacia_TaskModel.h) donde se definen las funciones de la instancia relacionadas con el modelo de tareas. El *mainLoop* es el encargado de crear y planificar las tareas. La regla *ComponentInstanceTaskModelInterface* crea el módulo *SimpleC* que define el modelo de tareas de cada instancia. Esta regla además recoge los métodos que están relacionados con sus elementos del modelo de tareas. Para ello se utiliza el helper ATL *getComponentPlatformTaskModelElement*. Para que la aplicación generada pueda gestionar la ejecución de las tareas y realizar su planificación periódica el planificador crea la información necesaria por cada tarea. En este caso, la plataforma ofrece la función SCHEDULER_addTask. El elemento software encargado de crear las tareas y controlar la ejecución de la aplicación es el componente etiquetado con el estereotipo <<Application>>. Este componente contiene la función *appLoop*. La función *main* de la aplicación simplemente llama a esta función para que el sistema comience su funcionamiento. El modelo de tareas que contiene el elemento *appLoop* que incluye las definiciones de las tareas a planificar para poder utilizar dichas definiciones en la creación de tareas (*getScheduledTaskModules*). En la figura A.8 se puede ver parte del código de la regla que genera bucle principal de la aplicación.

```
rule ComponentTaskModelInterface{
    from
        r: Uml!Realization ( r.supplierIsTaskModel() )
    to
        m : SimpleC!Module{
            name <- r.client->first().name+'_TaskModel'
            ,path <- r.client->first().name
            ,methods <-  r.client->first().getComponentPlatformTaskModelElements()
            ,externals <- r.getScheduledTasksModules()
            ,externals <- r.client->first().getKomponentModule()
        )
}

rule MainLoop2Method {
    from
        p: Uml!InstanceSpecification ( p.refImmediateComposite().isClassifierTaskModel()
                        and p.isMainLoop()
                    )
    to
        m : SimpleC!Method(
            name <- p.name
            ,code <- thisModule.createSimpleOSTaskCreationCode(p)
            ,code<- thisModule.createMainLoopCode(p)

        )
```

Figura A.8 Reglas de transformación para definir el modelo de tareas de cada instancia.

A.5.3 Transformando el modelo de tareas UML-MARTE en una descripción SimpleC

Para cada elemento definido en el modelo de tareas se ejecuta una regla declarativa que genera un método por elemento: *PeriodicTask2Method* para las tareas periódicas, *InterruptTask2Metho*d para las interrupciones y *MainLoop2Method* para el planificador y gestor de tareas. La regla *MainLoop2Method*, ver figura A.8, crea los elementos *SimpleC* de la función gestora de tareas: *appLoop*. Esta regla genera las llamadas SCHEDULER_addTask() necesarias y el bucle principal donde se le llama a la función Schedule() mediante reglas invocadas: *createSimpleOSTaskCreationCode* crea tantas llamadas *addTaks* como relaciones <<Usage>> existan entre el gestor y las tareas periódicas en el modelo de tareas, y la regla invocada *createMainLoop* crea el bucle que llama a la función *SCHEDULER_schedule()*. En el caso de las tareas periódicas y las interrupciones el código de la función a implementar se deriva de las relaciones abstractas estereotipadas como <<EntryPoiny>>, ver figura A.9. El método de la instancia que implementa la tarea periódica o la interrupción se comporta como una función envoltorio, siendo su finalidad llamar a la función que realmente implementa el comportamiento de la tarea. Esta función está definida en el diseño del tipo de componente. Por ejemplo en la figura 6.9 se define la tarea periódica *controlTask* en el modelo de tareas. Mediante la relación <<*entryPoint*>> se le relaciona a una clase interna de una instancia del componente, *MotorController* en este caso. Finalmente, la etiqueta <<*entryPoint*>> de MARTE presenta una propiedad denominada *routine* que permite especificar la función que implementa el comportamiento real de la tarea, el método *executeState* de MotorController. Las reglas que construyen estas funciones envoltorio se recogen en las reglas invocadas *PeriodicTask2Method*, regla *entryPoint* y *createPeriodicTaskCode*.

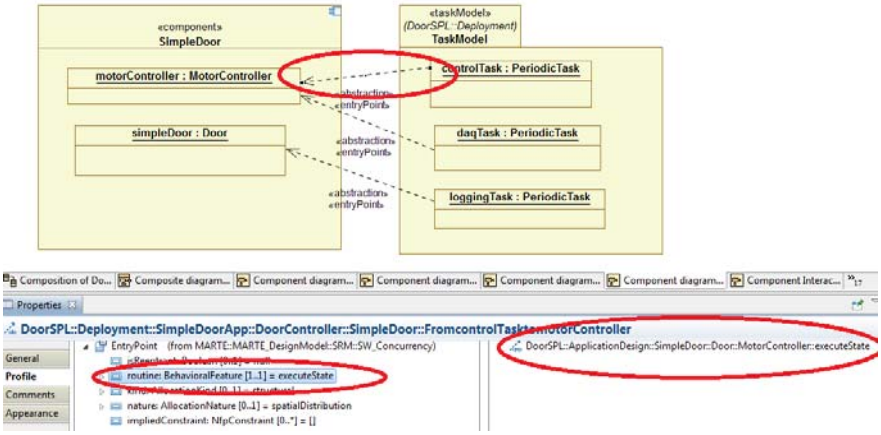


Figura A.9 Tarea relacionada con la función que implementa la tarea.