UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESIS PRESENTADA PARA OBTENER EL GRADO DE MAGISTER EN INGENIERÍA DE SOFTWARE

Soporte de trazabilidad en el proceso de transformación de modelos

Lic. Omar Armando Martínez Grassi

Dra. Claudia Pons

LA PLATA ARGENTINA

Septiembre 2014

Soporte de trazabilidad en el proceso de transformación de modelos

Lic. Omar Armando Martínez Grassi

SEPTIEMBRE 2014

Dedicatorias

Después de todo tú eres la única muralla. Si no te saltas, nunca darás un solo paso..

La búsqueda de la estrella - Luis Alberto Spinetta

A mis amores, Andrea y Julieta.

Agradecimientos

A la Dra Pons, por guiarme en este largo y sinuoso camino.

A mi amigo Guillermo Fischer, y toda su banda, por el aguante de siempre. Toda esta aventura hubiera sido imposible sin su ayuda.

A mi esposa Andrea, por apoyarme siempre.

A mi familia toda, que de una u otra manera han colaborado para permitirme alcanzar este gran anhelo personal.

Resumen

El desarrollo conducido por modelos o Model-Driven Development (MDD) es una aproximación a la Ingeniería de Software cuyo concepto central son los modelos y sus respectivas transformaciones. MDD brinda los principios básicos que permiten visualizar un sistema de software como un conjunto de modelos que son sucesivamente refinados hasta alcanzar uno con el suficiente nivel de detalle como para ser implementado. La Arquitectura Conducida por Modelos o *Model-Driven Architecture* (MDA) es la propuesta del Object Management Group (OMG) para MDD. Su objetivo fundamental es separar la especificación de la funcionalidad del sistema de la especificación de la implementación de dicha funcionalidad sobre una plataforma específica. La trazabilidad, como propiedad deseable de toda transformación de modelos, tiene un rol fundamental dentro del paradigma dado que la misma permite, entre otras cosas, la posibilidad de evaluar el impacto en fases avanzadas del ciclo de vida ante cambios en los requerimientos elicitados en etapas tempranas, y el mantenimiento de la consistencia entre los distintos modelos que guían el desarrollo. Este trabajo propone un esquema que permite la obtención de información de trazabilidad a partir de la definición de una transformación de modelos escrita en lenguaje QVT Relations mediante la utilización de una estrategia de inferencia de trazas definida ad hoc. Dicho proceso es totalmente automático y no depende de la ejecución de la transformación. Las principales contribuciones del estudio incluyen la minimización de los esfuerzos manuales en la gestión de trazabilidad, actividad tediosa y muy propensa a errores, y la independencia respecto de la implementación del motor QVT.

Abstract

Model-driven development (MDD) is a Software Engineering approach consisting of models and their transformations. MDD gives the basic principles to visualize a software system as a set of models that are repeatedly refined until a model with enough details to implement. Model-driven architecture (MDA) is the MDD view of Object Management Group. MDA main goal is to separate the system functional specification from the implementation specification on an given platform. Traceability, as a desired feature of transformations, has a major role within the paradigm since it allows the possibility to evaluate the impact at advanced stages of changes in requirement specification elicited early, and keeping consistency between models that guide the development, among other benefits. This work proposes a framework to get traceability information from a transformation definition written in QVT Relations language using a trace inference strategy defined ad hoc. This process is fully automated and does not depend on the execution of transformation. The contributions of the work include the minimization of manual efforts to achieve traceability, as error-prone and time-consuming activity, and the attainment of independence regarding the QVT engine implementation.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El desarrollo conducido por modelos o Model-Driven Development (MDD) es una aproximación a la Ingeniería de Software que consiste en la aplicación de modelos y tecnologías de modelos para elevar el nivel de abstracción en el cual los desarrolladores crean y evolucionan software, con el objetivo de simplificar y formalizar las múltiples actividades y tareas que comprenden el ciclo de vida del software [47]. En esencia, como resume Atkinson *et al.* en [22], la principal motivación de MDD es el incremento de la productividad a partir de la automatización de una serie de tareas complejas, aunque rutinarias.

El desarrollo conducido por modelos enfatiza los siguientes puntos clave:

- Un mayor nivel de abstracción en la especificación tanto del problema a resolver, como de la solución correspondiente.
- El aumento de confianza en la automatización asistida por computadora para el soporte del análisis, diseño y ejecución.
- La utilización de estándares como medio para facilitar las comunicaciones, la interacción entre diferentes aplicaciones y productos, y la especialización tecnológica.

Desde el año 2000, el Object Management Group (OMG) promueve una visión particular de MDD conocida como Arquitectura Conducida por Modelos o *Model-Driven Architecture* (MDA). Esta propuesta, es el eje principal del presente trabajo de tesis.

La Arquitectura Conducida por Modelos o *Model Driven Architecture* (MDA) es un *framework* para desarrollo de software definido por el Object Management Group (OMG). El concepto central de MDA son los modelos y sus respectivas transformaciones [53]. El objetivo principal de esta propuesta es separar la especificación de la funcionalidad del sistema de la especificación de la implementación de dicha funcionalidad sobre una plataforma específica. Para esto, MDA define una arquitectura que provee guías para estructurar especificaciones expresadas como modelos [59].

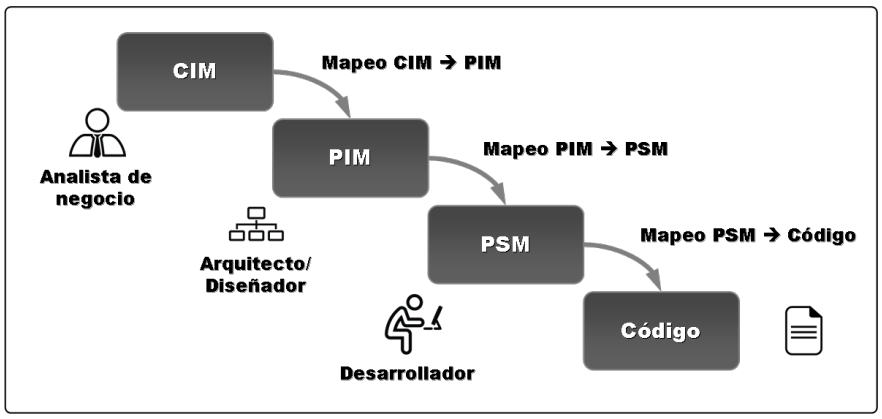


Figura 1.1: El proceso MDA

MDA propone un ciclo de desarrollo basado en transformaciones sucesivas, o refinamientos, de un modelo de alto nivel en otros, con menor nivel de abstracción, cuyo modelo final será el código fuente. Los modelos centrales caracterizados por MDA son los siguientes: un modelo independiente de la computación o CIM (Computation-Independent Model), un modelo independiente de la plataforma o PIM (Platform Independent Model), un modelo específico de la plataforma o PSM (Platform Specific Model) y el código fuente.

El CIM es el primer modelo del esquema propuesto por el framework. El mismo constituye una vista del sistema desde un punto independiente de la computación. Usualmente es conocido como modelo de dominio, y esta construido en el lenguaje específico de los expertos del negocio. El CIM juega un rol fundamental en la reducción de la brecha o *gap* entre los expertos del dominio y los expertos en el diseño y construcción de sistema.

En un primer paso, el CIM es transformado en el PIM (Figura 1.1). Este, es un modelo de alto nivel de abstracción, independiente de cualquier tecnología de implementación. El PIM se encuentra altamente ligado al negocio, y está concebido para darle a éste el mejor soporte posible. En el siguiente paso, el PIM es transformado en el PSM, el cual está pensado para especificar el sistema en términos de una tecnología de implementación particular. Un PIM puede ser transformado en múltiples PSM. Sin embargo, un PSM solo tendrá sentido para el grupo de desarrolladores que maneje la tecnología asociada. Finalmente, el último paso del proceso es la transformación del PSM en código fuente. Para ilustrar esta clasificación de modelos, podemos considerar los siguientes ejemplos de modelos MDA:

- CIM: Un modelo de casos de uso que captura los requerimientos de un sistema.
- PIM: La arquitectura del sistema de software, que describe como la funcionalidad del mismo descompuesta en componentes (arquitecturales) y conectores.
- PSM: Un modelo de implementación J2EE del sistema, expresado utilizando un perfil EJB que describe cómo los componentes (arquitecturales)

deben ser implementados por EJBs.

- Código: Los EJBs en sí mismos, sus archivos de configuración, etc, listos para el despliegue (*deploy*).

Los procesos de transformación de CIM a PIM, de PIM a PSM, y de PSM a código, no difieren en esencia de los procesos de desarrollo tradicional. El rasgo característico propuesto por MDA es que las transformaciones entre modelos, realizadas usualmente por una persona, sean enteramente automáticas y llevadas a cabo por alguna herramienta de transformación.

MDA hace sustanciales aportes a la mejora del proceso de desarrollo de software. En primer lugar, permite incrementar la productividad. En efecto, el foco del desarrollador es la especificación del PIM, dado que los PSM necesarios son generados por una herramienta de transformación, al igual que el código fuente correspondiente. Esto mejora la productividad de dos maneras: primero, porque evita la preocupación por los detalles específicos de una plataforma en particular, y segundo, porque permite a desarrolladores hacer foco en el problema del negocio, antes que en factores técnicos asociados al proceso de desarrollo del sistema de software. Además de productividad, el uso del framework favorece la portabilidad: dentro de MDA, ésta es lograda enfatizando el desarrollo de PIMs, los cuales son por definición independientes de la plataforma. El mismo PIM puede ser transformado en múltiples PSMs, para distintas tecnologías. Todo lo que sea especificado a nivel de PIM es, por lo tanto, completamente portable. A su vez, la capacidad de generar múltiples PSMs a partir de un PIM brinda otra cualidad al desarrollo con MDA: interoperabilidad. En efecto, teniendo la información disponible para transformar un PIM en más de un PSM, es posible establecer los puentes necesarios para lograr la interoperabilidad. A su vez, como eventualmente la introducción de modificaciones se realiza primero en el PIM, para luego generarse el PSM y posteriormente el código, la arquitectura conducida por modelos simplifica el mantenimiento de los sistemas. En este esquema, el PIM actúa como documentación de alto nivel, que se mantiene siempre actualizada.

MDA está basado en transformaciones. Una transformación es un proceso; cada proceso es descrito por una definición, la cual a su vez se compone de reglas, las cuales son ejecutadas por herramientas de transformación. Cada proceso de transformación tiene varias propiedades deseables, entre ellas, por orden de importancia podemos mencionar: configurabilidad (*tunability*), o la capacidad de configurar o sintonizar fácilmente la aplicación de reglas, trazabilidad (*traceability*), o la posibilidad de mapear cualquier elemento del modelo destino en su correspondiente par del modelo fuente, consistencia incremental (la información específica del modelo destino es mantenida tras las sucesivas regeneraciones del modelo) y bidireccionalidad, es decir, la capacidad de aplicar las transformaciones desde un modelo fuente a uno destino y viceversa.

En particular, a los efectos del trabajo propuesto, nos interesa el estudio de la propiedad de trazabilidad en transformación de modelos. Usualmente los PIM definidos no contienen toda la información necesaria para la implementación de un sistema completo, por lo que el desarrollador debe subsanar los *gaps* manualmente sobre el PSM. Cuando el usuario tiene la posibilidad de modificar

el PSM, puede cambiar partes del mismo que son generadas automáticamente desde el PIM.

1.1. Motivación

Reconociendo la importancia de las transformaciones en MDA, la organización OMG adoptó como estándar de transformación de modelos a QVT (Query/View/Transformation), una especificación de naturaleza híbrida declarativa/imperativa [60]. En efecto, QVT integra el estándar OCL 2.0 y lo extiende a su versión imperativa, definiendo tres lenguajes de dominio específico (DSL) llamados *Relations*, *Core* (ambos declarativos) y *Operational Mappings* (imperativo).

Desde hace unos años, el proyecto Eclipse promueve las tecnologías de desarrollo conducido por modelos (Model-Driven Development). En particular uno de los subproyectos, llamado Eclipse Modeling Framework (EMF), provee un entorno de modelado y generación de código para el desarrollo de aplicaciones basado en modelos que puede ser especificado mediante un subconjunto del lenguaje Java (conocido como Java Anotado), documentos XML o herramientas de modelado como Rational Rose™ [27]. El proyecto, entre sus principales aportes, incluye la definición de Ecore, una implementación de MOF [60], herramienta fundamental para la representación de modelos.

Pocas herramientas implementan los lenguajes definidos por el estándar QVT. Entre ellas podemos mencionar a mediniQVT [9], diseñada por la firma IKV++ Technologies AG, la cual implementa la especificación de OMG para QVT Relations; a SmartQVT [72], de la firma Orange Labs, la cual brinda una implementación *open source* del lenguaje QVT Operational Mappings, y a OptimalJ como implementación temprana de QVT Core. En general, la mayoría de estas herramientas han sido desarrollada para trabajar en forma integral con Eclipse, en forma de plugin, en el marco del proyecto EMF.

El estándar QVT define un esquema de trazabilidad interna donde las trazas son capturadas a lo largo del proceso de transformación, y las mismas pueden ser serializadas al final de la transformación para su análisis posterior. Sin embargo, el principal objetivo de este esquema es brindar soporte al mecanismo de resolución de objetos, indispensable para realizar las transformaciones. Las trazas provistas son utilizadas principalmente por el motor de transformación, y no se encuentran bien adaptadas para los usos típicos de la información de trazabilidad. En [21] se describen varios escenarios donde la que muestran cómo la información básica necesaria requerida para algunas de las principales actividades basadas en la explotación de trazas (por ejemplo la sincronización de modelos) no es provista por el esquema de trazabilidad interno de QVT. Por otro lado, la información contenida en las trazas QVT es relativa sólo a determinados tipos de elementos, como por ejemplo a entidades tipo *Class*, que modelan clases de objetos, y no *Property*, que modelan los atributos de dichas clases. En consecuencia, estos enlaces de traza o *trace links* modelan relaciones uno-a-uno entre determinados elementos de los modelos, ignorando otros tipos de relaciones existentes.

Para superar estas limitaciones, la presente tesis propone un esquema diferente: obtener la información de trazabilidad directamente desde la definición de la transformación, es decir, desde el código QVT. Este enfoque, permite además obtener no solo los enlaces de traza (o trazas, simplemente) sino también las relaciones de trazabilidad entre los elementos de modelo. Las relaciones de trazabilidad son aquellas que se encuentran definidas sobre tipos (por ejemplo, los elementos de un metamodelo), mientras que las trazas son instancias de éstas relaciones (por ejemplo, los elementos de un modelo). Como veremos más adelante, la mayoría de los enfoques de trazabilidad propuestos hasta el momento sólo permiten la obtención de enlaces de traza, y no las relaciones de trazabilidad.

En síntesis, este trabajo presenta una propuesta de soporte de trazabilidad en transformación de modelos mediante el análisis del código QVT, la cual permite la inferencia de trazas y relaciones de trazabilidad entre modelos origen y destino a partir de la especificación de la transformación, de forma sistemática, sin necesidad de código adicional ni intervención por parte del desarrollador. Dicha propuesta, además, es totalmente independiente de la implementación del motor QVT.

1.2. Objetivos

El presente estudio aborda la problemática de la implementación de trazabilidad en el proceso de transformación de modelos. En efecto, como veremos más adelante, numerosas publicaciones han estudiado esta propiedad en el contexto del desarrollo conducido por modelos, como [37, 41, 51, 45], pero no todas han propuesto una implementación concreta que fuera más allá de una visión teórica. La meta principal del trabajo es el diseño de un prototipo que pueda ser integrado a una herramienta de transformación de modelos, por ejemplo mediniQVT, en un entorno de desarrollo ampliamente utilizado como Eclipse, y asista al desarrollador automatizando el proceso de obtención de trazas o *links*, y relaciones de trazabilidad, entre elementos de los modelos origen y destino, involucrados en una transformación.

El desarrollo del trabajo de tesis propuesto comprende, por un lado, el análisis del problema de trazabilidad en el ámbito de desarrollo conducido por modelos (MDD, acrónimo de *Model-Driven Development*), seguido de la elaboración de un metamodelo de trazabilidad fácil de implementar, conciso y efectivo, que permita el aprovechamiento completo de sus beneficios sin entorpecer el proceso de desarrollo. Y finalmente, la implementación de la propuesta como herramienta complementaria para el proceso de definición de transformaciones entre modelos, en el ámbito de la Arquitectura Conducida por Modelos o *Model-Driven Architecture* (MDA).

1.3. La propuesta

El aporte fundamental del trabajo es el estudio de la problemática y la propuesta de solución, materializada en un producto de software concreto, integrado a uno

de los entornos de desarrollo integrado (IDE) más utilizados por la industria. Como hemos mencionado ya, y detallaremos en los capítulos siguientes, varios trabajos han transitado el mismo camino estudiando mecanismos de automatización del proceso de obtención de información de trazabilidad en MDD. La diferencia fundamental con ellos es el enfoque del mecanismo de obtención de trazas: mientras las mayoría aprovecha el proceso de transformación de modelos para la obtención de trazas, en algunos casos de manera explícita, mediante directivas específicas o código fuente, nuestro trabajo propone la generación de trazas a partir del análisis de la definición de la transformación, logrando automatizar la inferencia de trazas implícitas, definidas por las mismas reglas de transformación. Las contribuciones de este trabajo incluyen además la minimización de los esfuerzos manuales para lograr trazabilidad, actividad lenta y muy propensa a errores, y la independencia respecto de la implementación del motor QVT.

Para poder implementar la propuesta de trazabilidad se desarrolló un framework compuesto por un módulo analizador de modelos, un analizador de transformaciones y un componente analizador de trazas. Las entradas del proceso de obtención de trazas son los metamodelos que conforman los modelos origen y destino, y la definición de la transformación desde el modelo origen al modelo destino. El analizador de trazas o *TraceAnalizer* trabaja con la información provista según la especificación de una estrategia de trazabilidad encapsulada en un componente llamado *TraceStrategy* y, tras la inferencia de trazas, las mismas son creadas por un componente factoría llamado *TraceFactory*, que implementa un metamodelo de trazabilidad propio. Además de estos componentes, el trabajo incluye la implementación de una estrategia de trazabilidad desarrollada ad hoc llamada Análisis Basado en Variables o VBA (*Variable-Based Analysis*), la cual ha sido presentada a la comunidad científica en [58]. No obstante, el framework prevé la extensión y/o cambio de la estrategia de trazabilidad.

1.4. Publicaciones asociadas a esta tesis

Durante el desarrollo del presente trabajo de tesis, la temática del análisis basado en variables abordada en el Capítulo 5 fue presentada en el ASSE 2012 (13th Argentine Simposium of Software Engineering) de las 41 JAIIO (41° Jornadas Argentinas de Informática) mediante el paper titulado "Análisis basado en variables para trazabilidad en transformación de modelos". Dicho estudio, fue seleccionado posteriormente para la publicación de una versión extendida en idioma inglés en el Electronic Journal of SADIO (EJS) Volumen 12, la cual se encuentra disponible online en [14].

1.5. Organización del trabajo

El presente trabajo de tesis se encuentra dividido en dos partes: la primera parte define el marco conceptual asociado, revisando en el Capítulo 2 los conceptos fundamentales de la Arquitectura Conducida por Modelos o MDA (*Model-Driven Architecture*), la propuesta del OMG (Object Management Group) para

el desarrollo conducido por modelos, y en el Capítulo 3 distintos aspectos de trazabilidad, como definición, antecedentes y distintas propuestas implementadas en el marco del MDD.

La segunda parte del trabajo aborda los detalles de la propuesta presentada, implementada en una herramienta llamada QVTrace. El Capítulo 4 explica en detalle el desarrollo realizado, los aspectos de diseño considerados y el metamodelo de trazabilidad propuesto para la representación de las trazas obtenidas con esta aproximación. El Capítulo 5 describe la estrategia de trazabilidad diseñada para su utilización en QVTrace, llamada Análisis Basado en Variables o VBA. A continuación, en el Capítulo 6 se desarrollan dos casos de estudio para ilustrar el funcionamiento de la estrategia VBA. A modo de ejemplo, se presentan aquí dos transformaciones de modelos: la primera convierte un diagrama de clases UML en código Java simplificado, y la segunda transforma una especificación bibliográfica BibTeX en un documento Docbook. Sobre éstas transformaciones se aplica la estrategia desarrollada, mostrando luego las trazas inferidas. Tras el análisis de los ejemplos, el Capítulo 7 presenta un estudio comparativo entre esta propuesta y otros esquemas de trazabilidad en transformación de modelos. Seguidamente, el Capítulo 8 contiene una comparación de la traza generada por QVTrace y la traza implícita generada por el motor QVT tras la ejecución de una transformación de modelos.

Finalmente, el Capítulo 9 contiene las conclusiones del presente estudio y algunas líneas de trabajo futuro sobre el tema.

Capítulo 2

Model-Driven Architecture

La Arquitectura Conducida por Modelos o *Model-Driven Development* es un framework para desarrollo de software definido por el Object Management Group (OMG). Este, ha sido formado como una organización de estándares para ayudar a reducir complejidad, disminuir costos, y acelerar la introducción de nuevas aplicaciones de software. Una de las principales iniciativas mediante las cuales el OMG está llevando a cabo su meta principal ha sido la promoción de MDA como un framework arquitectural para desarrollo de software. Dicho framework está construido alrededor de un número de especificaciones de OMG, las cuales son utilizadas ampliamente por la comunidad de desarrolladores.

En el año 2001, el OMG adoptó la arquitectura MDA como una aproximación para el uso de modelos dentro del desarrollo de software. Sus tres principales objetivos son la portabilidad, la interoperabilidad y la reusabilidad mediante la separación arquitectural de aspectos relevantes.

El presente capítulo realiza una introducción al framework MDA, detallando sus principales características, y cómo éstas pueden ayudar a resolver problemas comunes del proceso de desarrollo de software tradicional.

2.1. Problemática de los esquemas de desarrollo tradicionales

En la actualidad el desarrollo de software sigue padeciendo un cierto número de problemas inherentes a su complejidad. Escribir software es una labor intensiva, y con la introducción de cada nueva tecnología mucho trabajo necesita ser realizado una y otra vez. Los sistemas nunca son construidos a partir de una única tecnología, y necesitan comunicarse con otros sistemas. Y todo ésto dentro de un marco de continuo cambio de requerimientos.

Para mostrar cómo MDA ayuda a abordar éstos inconvenientes, a continuación analizaremos los problemas más importantes que enfrenta el desarrollo de software y las causas de los mismos.

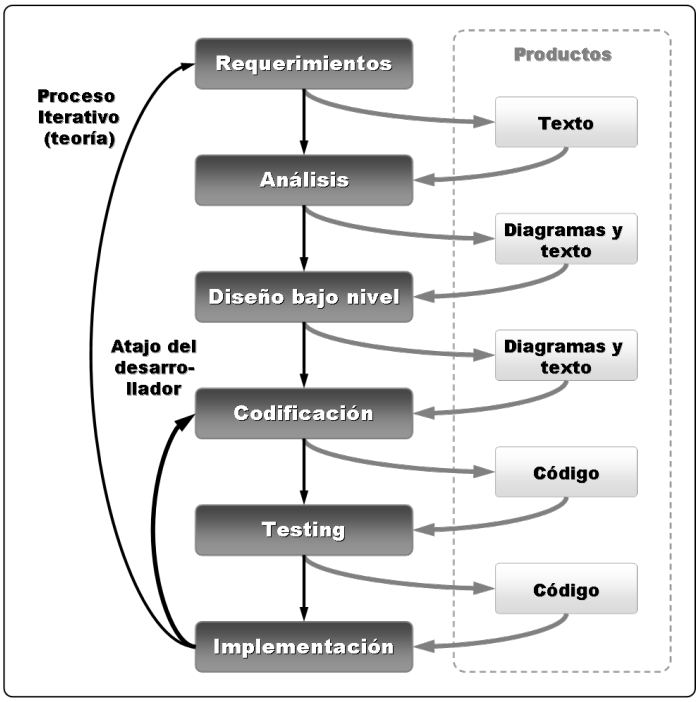


Figura 2.1: Ciclo de vida de desarrollo de software tradicional

2.1.1. El problema de la productividad

El proceso de desarrollo tradicional, como lo conocemos en la actualidad, es a menudo conducido por el diseño de bajo nivel y la codificación. Un típico proceso incluye las siguientes fases (Figura 2.1):

1. Conceptualización y obtención de requerimientos.
2. Análisis y descripción funcional.
3. Diseño.
4. Codificación.
5. Testing.
6. Despliegue/Implementación (*deployment*).

Independientemente del uso de una versión incremental o iterativa de este proceso, o el tradicional modelo cascada (*Waterfall*), los documentos y diagramas son producidos durante las fases 1 a 3. Esto incluye la descripción de los requerimientos en formato de texto o imágenes, y habitualmente muchos otros diagramas del Lenguaje de Modelado Unificado (UML) como casos de uso, diagrama de clases, de interacción, de actividades, etc.

Todo este material, diagramas y documentos, creados en las tres primeras fases rápidamente pierde su valor conforme la codificación comienza. La conexión

entre los diagramas y el código se va perdiendo a medida que las fases de codificación avanzan. Finalmente, en lugar de ser una especificación exacta del código, los diagramas usualmente se transforman en imágenes, con suerte, parecidas al resultado final.

Cuando el sistema es modificado, o se desvía de su especificación inicial, la situación se agrava aún más. Los cambios suelen ser realizados, en la mayoría de los casos, sólo a nivel del código, dado que no hay tiempo disponible para actualizar la documentación de alto nivel. Además, el valor del documentos y diagramas actualizados pasa a ser materia opinable, porque los cambios siempre comienzan, de esta manera, en el código.

Esta idea ha inspirado metodologías ágiles como Extreme Programming (XP) [23], la cual está basada en el hecho que el código es la fuerza conductora del desarrollo de software, y por tanto, las únicas fases realmente productivas del proceso son la codificación y testing. No obstante, como establece Cockburn en [32], XP resuelve sólo parte del problema. En efecto, este autor plantea que durante el desarrollo inicial el equipo tiene tanto conocimiento de alto nivel que puede fácilmente comprender el sistema. Sin embargo, los problemas comienzan cuando el equipo es desmantelado, usualmente tras la liberación de la primer versión del software. En este punto otro equipo toma la posta y necesita realizar el mantenimiento del sistema (corregir errores, mejorar funcionalidades, etc). La realización de esta tarea, sólo con código y casos de test, es compleja, y difícil de llevar adelante como corresponde.

En conclusión, las opciones son utilizar el tiempo en las primeras fases del proceso de desarrollo construyendo la documentación y los diagramas de alto nivel, o utilizar el tiempo en la fase de mantenimiento averiguando qué hace el software. Los desarrolladores a menudo sienten que sólo son productivos si generan código, mientras que escribir documentación, o modelos, no lo es. Sin embargo, en proyectos de software maduros, estas tareas necesitan ser realizadas.

2.1.2. El problema de la portabilidad

Si algo distingue a la industria de software de otras es el ritmo frenético en que las nuevas tecnologías son inventadas y se tornan populares. En muchos casos, las empresas se ven obligadas a migrar hacia éstas por determinadas razones:

- Las nuevas tecnologías son demandadas por los clientes.
- Resuelven algún problema real.
- Los *vendors* de las herramientas dejan de dar soporte a la antigua tecnología, y se enfocan en los nuevos productos.

Las nuevas tecnologías ofrecen en muchos casos beneficios tangibles, y muchas compañías no pueden permitirse quedarse atrás, por cuestiones en general de competitividad. Como consecuencia, los saltos a las nuevas tecnologías a menudo son demasiado rápidos, haciendo que la inversión de tiempo y dinero en tecnologías previas pierda valor, y se torne inútil.

Finalmente, el software existente debe ser portado a la nueva tecnología, o a una nueva versión de una tecnología existente. Eventualmente el software puede permanecer sin cambios utilizando la tecnología anterior, en cuyo caso el existente, ahora *legacy software*, necesitará interoperar con nuevos sistemas construidos bajo las nuevas tecnologías.

2.1.3. El problema de la interoperabilidad

Los sistemas de software raramente actúan aisladamente. Por el contrario, en general deben interactuar con otros sistemas. Aún cuando los sistemas son construidos desde cero, a menudo utilizan la mejor tecnología disponible para el trabajo para el cual han sido creados, lo cual conduce a una determinada multiplicidad de sistemas distintos, que necesitan comunicarse entre sí, a diferencia de las antiguas arquitecturas monolíticas de sistema único. Esto hace más fácil, o posible en algunos casos, la realización de cambios, aunque genera el requerimiento de interoperabilidad.

2.1.4. El problema del mantenimiento y documentación

Como hemos planteado anteriormente, la documentación siempre ha sido el punto débil del proceso de desarrollo de software. Se ha instalado en muchos equipos la idea que la documentación durante el desarrollo cuesta tiempo y retrasa el proceso, porque no soporta la principal tarea del desarrollador.

Esta es la principal razón por la cual la documentación, además de escasa, es a menudo de muy baja calidad. Es también la razón por la cual la documentación no suele estar actualizada, dado que con cada cambio en el código ésta debe ser actualizada de forma manual. Los desarrolladores están equivocados, obviamente. Su principal tarea es desarrollar sistemas de software que puedan ser fácilmente modificados y mantenidos posteriormente. Al contrario de lo que muchos piensan, la documentación es una de las tareas esenciales que deben llevar a cabo.

Algunos lenguajes de programación, como Eiffel o Java, proveen una facilidad para generar la documentación directamente a partir del código fuente, permitiendo asegurar así que la misma esté permanentemente actualizada. Sin embargo, esto soluciona solamente el problema de la documentación de bajo nivel. La documentación de alto nivel, como diagramas y texto, aún debe ser mantenida manualmente. Y cuanto mayor es la complejidad de los sistemas a construir, mayor debe ser la obligación de mantenerla.

2.2. Model-Driven Architecture

Como hemos mencionado, la Arquitectura Conducida por Modelos o *Model-Driven Architecture* (MDA) es un framework para desarrollo de software definida por el Object Management Group. Como su nombre indica, la arquitectura está centrada en la utilización de modelos dentro del proceso de desarrollo de

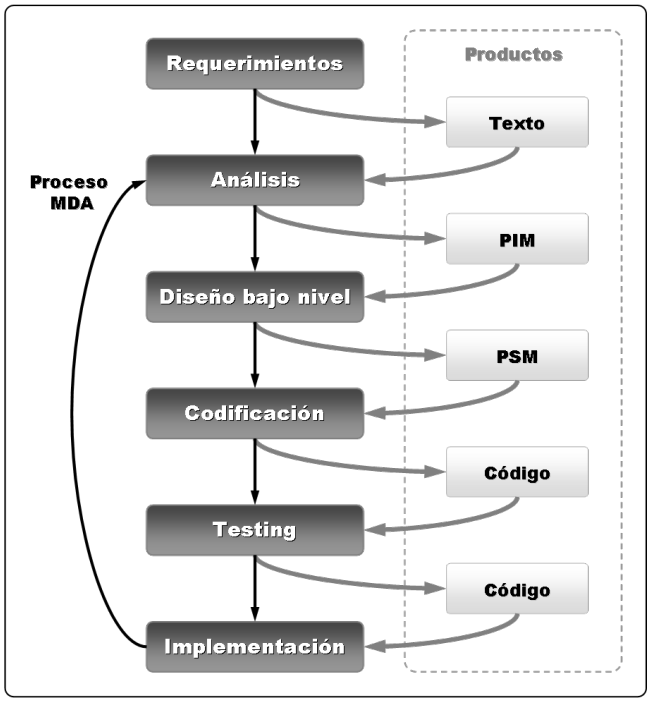


Figura 2.2: Ciclo de vida de desarrollo MDA

software, haciendo que dicho proceso sea conducido por las actividades de modelado de los sistemas de software.

A continuación analizaremos el ciclo de vida de desarrollo propuesto por MDA en contraste con los esquemas tradicionales, y se detallará cómo MDA puede ayudar a resolver gran parte de los problemas identificados en la sección anterior.

2.2.1. El ciclo de vida de desarrollo MDA

En contraste con los esquemas convencionales, el ciclo de vida de desarrollo MDA (Figura 2.2) no presenta sustanciales diferencias. Al menos en lo estructural, las fases identificadas son las mismas a la de los ciclos de vida tradicionales. La diferencia fundamental radica en la naturaleza de los productos generados durante el proceso de desarrollo. Estos productos o "artefactos" son modelos formales, y por lo tanto pueden ser comprendidos por computadoras. A continuación presentaremos los tres modelos *core* de MDA:

2.2.1.1. El modelo independiente de la plataforma o *Platform Independent Model* (PIM)

El primer modelo que define MDA es un modelo de un alto nivel de abstracción, el cual es independiente de cualquier tecnología de implementación. Un

PIM describe el sistema de software que soporta un determinado negocio. Dentro del PIM, el sistema es modelado pensando sólo en dar el mejor soporte posible al negocio. Si dicho sistema posteriormente es implementado mediante un *mainframe* con una base de datos relacional, o como aplicación web con una base de datos orientada a objetos, no tiene ninguna importancia para el PIM.

2.2.1.2. El modelo específico de la plataforma o *Platform Specific Model* (PSM)

El siguiente paso consiste en la transformación del PIM en uno o más modelos específicos de plataforma (PSMs). Un PSM está adaptado para especificar el sistema en términos de elementos de implementación disponibles para una plataforma específica. Por ejemplo, un PSM EJB (Enterprise JavaBean[1]) es un modelo del sistema en términos de estructuras EJB como "home interface", "session bean" o "entity bean". Una base de datos PSM incluye términos como "table", "column" o "foreign key". Claramente, el PSM sólo tiene sentido para un desarrollador con conocimientos sobre una determinada plataforma. Un PIM puede ser transformado en múltiples PSMs. Para cada tecnología de implementación debe ser generado un PSM diferente.

2.2.1.3. Código fuente

El paso final en el desarrollo consiste en la trasformación de cada PSM en código fuente. Dada la cercanía entre el PSM y el código, esta transformación es relativamente sencilla.

MDA define el PIM, el PSM y el código, y además define cómo están relacionados unos con otros. Un PIM debe ser creado, y posteriormente transformado en uno o más PSMs, los cuales finalmente son transformados en código fuente.

Tanto el PIM, como el PSM y el código fuente son productos generados en diferentes pasos del ciclo de vida de desarrollo, y por lo tanto, representan diferentes niveles de abstracción dentro de la especificación del sistema. La habilidad de transformar un PIM de alto nivel en un PSM eleva el nivel de abstracción al cual un desarrollador puede trabajar, permitiendo hacer frente a un mayor número de sistemas complejos con menor esfuerzo.

Hasta aquí el proceso MDA no parece demasiado distinto al desarrollo convencional. La diferencia principal radica en la transformaciones. En efecto, tradicionalmente las transformaciones de modelo a modelo, o de modelo a código, eran realizadas a mano. Muchas herramientas hoy en día pueden generar código fuente a partir de un modelo, aunque en muchos casos dicha generación no es más que un *template* parcialmente completo, que debe ser rectificado por una persona.

En MDA, las transformaciones son siempre ejecutadas por herramientas (Figura 2.3). Como hemos dicho, existen varias herramientas que convierten PSM en código. No hay ninguna novedad en eso. El punto clave de MDA es la transformación de un PIM en PSM, o en múltiples PSMs, según el caso.

[1] Los Enterprise JavaBeans (también conocidos como EJB) son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE (ahora JEE 6.0) de Oracle Corporation (inicialmente desarrollado por Sun Microsystems).

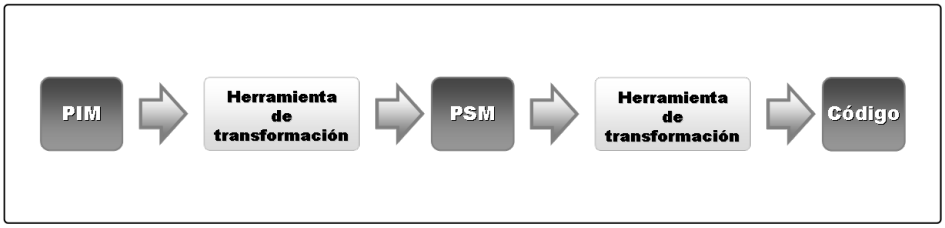


Figura 2.3: Los tres pasos fundamentales del proceso de desarrollo MDA

2.2.2. Beneficios de MDA

A continuación analizaremos los beneficios que nos ofrece MDA en términos de la mejora del proceso de desarrollo de software que supone el uso del framework, en particular en las áreas identificadas como problemáticas, mencionadas en la Sección 2.1.

2.2.2.1. Productividad

En MDA el foco de un desarrollador está basado en el desarrollo del PIM. Los PSMs necesarios son generados por la transformación de PIM a PSM. Obviamente siempre se requiere de una persona para definir ésta transformación, lo cual es una tarea compleja y especializada, pero dicha definición sólo debe ser generada una vez, y puede ser utilizada muchas veces, aún en el desarrollo de otros sistemas.

La mayoría de los desarrolladores se enfocará naturalmente en el desarrollo de PIMs. Dado que pueden trabajar de manera independiente de los detalles específicos de las plataformas, son muchos los detalles técnicos con los cuales no deberán lidiar. Estos, son luego agregados automáticamente mediante la transformación de PIM a PSM, mejorando la productividad de dos maneras:

- En primer lugar, los desarrolladores de PIMs tendrán menos trabajo dado que los detalles específicos de cada plataforma no deben ser diseñados ni escritos, sino que son direccionados en la definición de la transformación. Obviamente este tiempo puede ser aprovechado para otras tareas.
- En segundo lugar, dado que los desarrolladores cambian el foco de principal atención hacia el PIM, en lugar del código, estarán más pendientes en la resolución de los problemas del negocio. Esto redundará a la larga en un sistema que se adaptará mejor a las necesidades de los usuarios finales, los cuales podrán obtener mejor funcionalidad en menos tiempo.

Todos estos beneficios sólo pueden ser obtenidos por el uso de herramientas que automatizan completamente la generación de un PSM a partir de un PIM. Esta situación implica que la mayoría de la información sobre la aplicación debe ser incorporada en el PIM y/o a nivel de la herramienta de transformación. Dado que el modelo de alto nivel ya no es simplemente papel, sino que se encuentra directamente relacionado con el código generado, las demandas sobre la completitud y consistencia del modelo de alto nivel, es decir el PIM, son mucho mayores que en los procesos de desarrollo tradicionales.

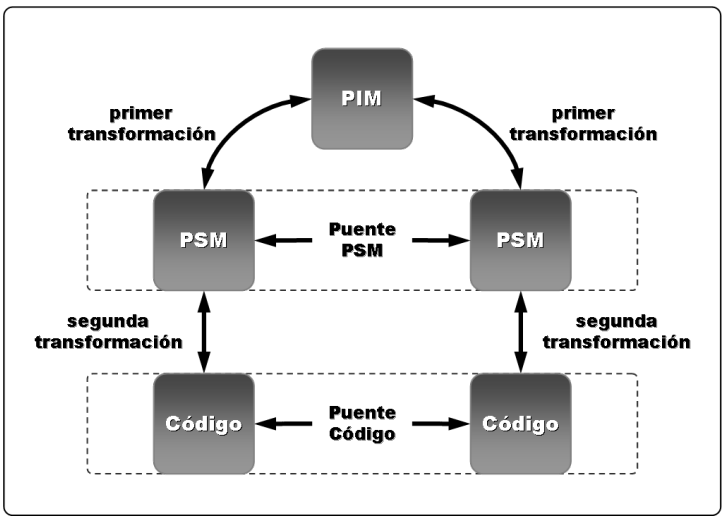


Figura 2.4: Interoperatividad en MDA utilizando puentes

2.2.2.2. Portabilidad

Dentro de MDA, la portabilidad es lograda poniendo el foco en el desarrollo de PIMs, los cuales son por definición independientes de la plataforma. El mismo PIM puede ser transformado en múltiples PSMs para distintas plataformas, por lo que todo lo que se especifique a nivel de PIM es absolutamente portable.

El alcance que la portabilidad puede tener dependerá de las herramientas de transformación automatizadas que se encuentren disponibles. Para plataformas populares, un mayor número de herramientas estarán disponibles, mientras que para plataformas menos populares, es posible que se requiera escribir las propias reglas de transformación.

2.2.2.3. Interoperatividad

Como hemos indicado anteriormente, un PIM puede transformarse en diversos PSMs. Todos estos PSMs, que tienen un origen común, pueden relacionarse entre sí mediante el concepto de puentes (*bridges*). Dos o más PSMs generados para diferentes plataformas no pueden interactuar de manera directa, de alguna forma es necesario transformar los conceptos de una plataforma en los conceptos de otra. Esto es, en esencia, interoperabilidad. MDA aborda este problema no sólo generando los PSMs , sino también los puentes correspondientes entre éstos (Figura 2.4).

El concepto de puente se basa en el hecho que si una herramienta MDA es capaz de transformar un PIM en múltiples PSMs de diversas plataformas, entonces se encuentra disponible toda la información necesaria para sortear el *gap* que separa los PSMs entre sí. De cada elemento de un PSM podemos conocer el elemento del PIM del cual se originó. De ese elemento del PIM conocemos qué elemento de otro PSM distinto generó. De esta manera podemos determinar cómo un elemento de un PSM se relaciona con un elemento de otro PSM. Dado que

además se dispone de todos los detalles específicos de cada plataforma, entonces tenemos toda la información necesaria para generar un puente entre dos PSMs distintos.

A modo de ejemplo, asumamos que tenemos un PSM que es el modelo de objetos (código) Java y otro PSM que es un modelo de base de datos relacional. Para un elemento *Cliente* del PIM, sabemos que éste es transformado en ciertas clases de nuestro modelo Java, y en ciertas tablas del modelo relacional. Podemos entonces fácilmente trazar un puente entre ambos elementos. Para recuperar un objeto desde la base de datos, sabemos que es necesario realizar una consulta sobre las tablas correspondientes e instanciar las clases correspondientes del modelo Java con estos datos. Para almacenar un objeto, es preciso tomar los datos del objeto Java y almacenarlos en la tabla *Cliente* (y demás tablas) del modelo relacional.

La interoperabilidad entre distintas plataformas puede ser soportada por herramientas que no solo generen PSMs, sino que también generen los puentes necesarios entre éstos. Esto permite afrontar los cambios tecnológicos ayudando además a preservar la inversión en el PIM.

2.2.2.4. Mantenimiento y documentación

Trabajando con el ciclo de vida MDA, los desarrolladores pueden enfocarse en el PIM, el cual es un modelo con un nivel de abstracción mucho mayor que el del código fuente. El PIM es utilizado para generar un PSM, el cual a su vez se utilizará para la creación del código. El modelo es, entonces, una representación exacta del código. Por ende, el PIM satisface la función de documentación de alto nivel necesaria para cualquier sistema de software.

La gran ventaja que ofrece este ciclo es que el PIM no es abandonado luego de la primer versión del código. Los cambios realizado al sistema, sean mejoras o correcciones, eventualmente serán realizados sobre el PIM, para luego regenerar el PSM y finalmente el código.

Al margen de que la documentación de alto nivel de abstracción naturalmente se encontrará disponible, la necesidad de escribir información adicional, que no es capturada en el PIM, aún continúa en el enfoque MDA. Esto incluye, por ejemplo, las decisiones tomadas durante el desarrollo del PIM.

2.3. Modelos

Como hemos visto anteriormente, los modelos son elementos centrales dentro del framework que estamos analizando. ¿Pero a qué llamamos exactamente modelo? Encontrar una definición que sea lo suficientemente general para agrupar todos los distintos tipos de modelo es difícil. Esta definición además, debe ser lo suficientemente específica para ayudarnos a especificar transformaciones automáticas de un modelo en otro.

A continuación presentamos el significado de modelo en el contexto del proceso MDA.

2.3.1. Concepto de modelo

La Real Academia Española da a la palabra "modelo" varias acepciones, a saber:

- Arquetipo o punto de referencia para imitarlo o reproducirlo.
- En las obras de ingenio y en las acciones morales, ejemplar que por su perfección se debe seguir e imitar.
- Representación en pequeño de alguna cosa.
- Esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento.
- Objeto, aparato, construcción, etc., o conjunto de ellos realizados con arreglo a un mismo diseño.
- Vestido con características únicas, creado por determinado modista, y, en general, cualquier prenda de vestir que esté de moda.
- En empresas, usado en aposición para indicar que lo designado por el nombre anterior ha sido creado como ejemplar o se considera que puede serlo. *Empresa modelo. Granjas modelo.*
- Figura de barro, yeso o cera, que se ha de reproducir en madera, mármol o metal.
- Persona de buena figura que en las tiendas de modas se pone los vestidos, trajes y otras prendas para que las vean los clientes.
- Persona u objeto que copia el artista.

Como podemos observar, todas las definiciones enumeradas previamente tienen varios puntos en común:

- Un modelo es siempre un abstracción de algo que existe en la realidad.
- Un modelo es diferente de aquello que está modelando. Por ejemplo: algunos los detalles son dejados de lado, o bien los tamaños son diferentes.
- Un modelo puede ser utilizado como un ejemplo para producir algo que existe en la realidad.

De todas estas observaciones, es evidente que necesitamos una palabra para indicar "algo que existe en la realidad". Dado que los modelos que nos interesan son relevantes en el contexto del desarrollo de software, utilizamos la palabra *sistema*. La mayoría de las veces, la palabra sistema se refiere a un sistema de software, aunque en el caso de un modelo de negocios, el negocio es en sí mismo el sistema.

Por otro lado, vemos que un modelo describe un sistema de una manera tal que permite producir otro sistema similar. El nuevo sistema no es igual a su

predecesor, dado que el modelo abstrae detalles del sistema original. Sin embargo, dado que las características principales se mantienen, podemos decir que el nuevo sistema es similar al anterior.

Un modelo siempre está escrito en un lenguaje. Puede ser en idioma castellano, en un lenguaje de programación, en algún lenguaje de modelado (UML [63], por ejemplo). Para posibilitar la transformación automática de modelos es necesario restringir los modelos adecuados para MDA a sólo aquellos escritos en lenguajes bien definidos. Un lenguaje bien definido es uno que tiene una forma bien definida y un significado que puede ser comprendido por una computadora. Los lenguajes naturales no son considerados bien definidos, ya que no pueden ser entendidos por computadora.

Dentro del contexto de MDA, podemos considerar las siguientes definiciones [53]:

> Un *modelo* es una descripción de (o parte de) un sistema escrito en un lenguaje bien definido.
>
> Un *lenguaje bien definido* es un lenguaje que presenta una forma (sintaxis) y un significado (semántica) bien definidos, el cual es adecuado para la interpretación automática mediante computadoras.

Esta definición es muy general. No impone restricciones sobre cómo debe ser un modelo, siempre y cuando esté escrito en un lenguaje bien definido y describa un sistema determinado.

2.3.2. Relaciones entre modelos

Para un sistema determinado es posible que haya múltiples modelos, que difieran en número de detalles, aspectos, enfoques, que éstos modelen. Asimismo, existen muchos tipos de relaciones posibles entre modelos. En particular, MDA se encuentra focalizado en un tipo de relación particular: la transformación automática de un modelo en otro. Esto no significa que esta relación sea la más importante, sólo que ha podido ser automatizada, y puede ser realizada por computadoras.

2.3.3. Tipos de modelo

La definición dada en la Sección 2.3.1 es claramente muy general. Para lograr una mayor precisión, es posible distinguir entre distintos tipos de modelos. Una técnica para ello consiste en responder las siguientes preguntas:

1. ¿En qué parte del proceso de desarrollo de software es utilizado el modelo? ¿Es un modelo de análisis o de diseño?
2. ¿Qué nivel de detalle contiene el modelo? ¿Es abstracto o detallado?
3. ¿Cuál es el sistema que está modelando? ¿Es un modelo de software o de negocios?

4. ¿Qué aspectos del sistema describe? ¿Es un modelo estructural o dinámico?
5. ¿Esta basado en una tecnología en particular? ¿Es independiente o específico de una plataforma determinada?
6. ¿Para qué plataforma está diseñado?

El sistema descripto por un modelo de negocios es, valga la redundancia, un negocio en sí mismo, o una (o parte de una) empresa. Los lenguajes utilizados para definir esta clase de modelos están basados en un vocabulario que permite modelar áreas, procesos, *stakeholders*, departamentos, unidades de negocio, y demás. Dichos modelos, no necesariamente incluyen información sobre los sistemas de software utilizados. Por esto, este tipo de modelo es conocido como *Computional-Independent Model* (CIM). Los requerimientos para un sistema de software que soporte dicho negocio siempre son derivados del modelo de negocios.

Un modelo estructural es aquél que modela aspectos estáticos de un sistema, aspectos fijos, que no dependen del funcionamiento del mismo, ni del flujo de información que atraviesa el mismo. Los modelos dinámicos están focalizados en modelar comportamiento, aspectos que dependen de lo circunstancial, del flujo dinámico del sistema. Existen lenguajes, como UML por ejemplo, que permiten especificar modelos estáticos o estructurales y dinámicos. Estos muestran diferentes visualizaciones de un mismo sistema, en un único modelo.

El estándar MDA define los términos *Platform Independent Model* (PIM) y *Platform Specific Model* (PSM) como dos tipos de modelo mutuamente excluyentes. La realidad indica que ha menudo es difícil trazar una línea entre ambos. ¿Es un modelo escrito en UML específico para una plataforma Java sólo porque uno de los diagramas de clase define una o más interfaces? Claramente la respuesta es no. De manera que sólo podemos decir que un modelo es más, o menos, *platform-specific* que otro.

2.4. Transformaciones entre modelos

Como vimos en Sección 2.2, el proceso MDA muestra el rol de cada modelo dentro del framework: PIM, PSM y el código fuente. Una transformación toma un modelo PIM y lo transforma en un modelo PSM. Otra transformación toma el modelo PSM y lo transforma en código. Estas transformaciones son esenciales dentro del proceso de desarrollo MDA.

Cuando analizamos en detalle una transformación, encontramos que existe una definición que describe cómo se realiza dicha transformación. Esa definición se conoce como la *definición de la transformación*. En resumen tenemos, por un lado, el proceso de transformación, y por otro, la definición que explica cómo se realiza dicha transformación.

En general, podemos decir que la definición de una transformación consta de un conjunto de reglas que detallan puntualmente, sin ambigüedades, cómo una parte del modelo origen es utilizado para crear parte del modelo destino. En función de éstos conceptos, arribamos a las siguientes definiciones:

> Una *transformación* es la generación automática de un modelo destino (*target*) a partir de un modelo fuente (*source*), de acuerdo a una definición de transformación.
>
> La *definición de una transformación* es un conjunto de reglas de transformación que describen cómo un modelo escrito en un lenguaje fuente puede ser transformado en un modelo basado en un lenguaje destino.
>
> Una *regla de transformación* es una descripción de cómo una o más construcciones en un lenguaje fuente pueden ser transformadas en una o más construcciones del modelo destino.

Para ser útiles, las transformaciones deben poseer ciertas características. La más importante, sin embargo, es que preserven el significado entre los modelos fuente y destino. A continuación analizaremos con mayor detalle las cualidades de las transformaciones.

2.4.1. Características deseables de las transformaciones

En la sección anterior, se definió como transformación al proceso de generación de un modelo destino a partir de un modelo fuente. Este proceso es descripto por una definición de transformación, la cual consiste en un conjunto de reglas de transformación, las cuales son ejecutadas por una herramienta de transformación. En el enfoque MDA existen un número de cualidades deseables para las transformaciones. Dichas propiedades son:

- Configurabilidad (*Tunability*), definida como el grado o la capacidad de afinar o ajustar la aplicación de cada regla general de la definición de la transformación. Por ejemplo, al transformar el tipo *String* de UML al tipo *Varchar* en un modelo entidad-relación, el desarrollador puede querer variar la longitud de la cadena de tipo *Varchar*, según la ocurrencia del atributo de tipo *String* en UML.
- Trazabilidad, especificada como la posibilidad de seguir el rastro de un elemento del modelo destino en el elemento del modelo fuente a partir del cual fue generado.
- Consistencia incremental, es decir la persistencia de información específica extra del modelo destino adicionada, luego de sucesivas regeneraciones de dicho modelo.
- Bidireccionalidad, o sea la posibilidad de aplicar una transformación no sólo desde un modelo fuente a uno destino, sino también en sentido inverso desde el modelo destino al modelo fuente.

2.4.1.1. Configurabilidad

El control sobre el proceso de transformación es un aspecto deseado por cualquier usuario de una herramienta de transformación. Existen varias maneras de controlar dicho proceso. A continuación analizaremos las más usuales.

Control manual

El control más directo que un usuario puede tener sobre una transformación es la capacidad de definir manualmente qué elemento del modelo es transformado por qué regla de transformación. Si bien este esquema brinda flexibilidad, es claramente propenso a errores. E incluso impracticable para modelos de gran envergadura.

Condiciones sobre las transformaciones

Otra manera de otorgar control al usuario es mediante la posibilidad de incorporar una condición sobre cada regla que compone la transformación. De esta forma, dicha condición permite establecer cuándo y bajo qué circunstancias dicha regla será aplicada. Preferentemente, las condiciones definidas deben ser mutuamente excluyentes, de modo de automatizar completamente el proceso de transformación. En caso que para una transformación dada se sostengan simultáneamente dos o más condiciones, este enfoque puede ser combinado con control manual.

Parámetros de transformación

Alternativamente a los dos esquemas anteriores, el proceso de transformación puede ser controlado a través de parámetros ubicados estratégicamente en la definición de la transformación.

2.4.1.2. Trazabilidad

La trazabilidad es un activo de gran ayuda en la aplicación del framework MDA. En efecto, es frecuente que los PIM no contengan toda la información necesaria que permita implementar el sistema en su totalidad, por lo que a menudo el usuario es quién debe completar ese *gap* en el PSM de forma manual. Cuando el usuario tiene la posibilidad de cambiar el PSM, puede potencialmente modificar porciones de éste que fueron generadas como parte del proceso de transformación. Obviamente, esta situación es una fuente de problemas.

Bajo estas circunstancias, la herramienta debe como mínimo alertar al usuario que la modificación realizada ha cambiado una fracción del modelo generada automáticamente. O mejor aún, indicar además qué parte del PIM ha sido impactada tras el cambio, que puede ser por ejemplo el renombrado de un elemento u operación. Para poder brindar esta ayuda, la herramienta debe ser capaz de realizar la traza desde el PSM al PIM.

En el Capítulo 3, se profundizará el estudio sobre esta propiedad tan importante, la cual es el elemento central del presente trabajo.

2.4.1.3. Consistencia incremental

Tras la generación de un modelo destino, usualmente se realizan ciertos agregados o ajustes sobre el mismo. El concepto de consistencia incremental se refiere

a la posibilidad que dichos agregados persistan en caso de una potencial regeneración del modelo destino debido a cambios en el correspondiente modelo fuente.

Cuando se produce un cambio en el modelo origen, el proceso de transformación conoce cuáles son los elementos del modelo destino que necesitan ser cambiados, y reemplaza sólo a éstos, preservando la información extra. Como se puede observar, bajo esta propiedad, los cambios en el modelo fuente tienen un impacto mínimo sobre el modelo destino.

2.4.1.4. Bidireccionalidad

Las transformaciones bidireccionales pueden ser implementadas de dos formas distintas:

1. Realizar ambas transformaciones de acuerdo a una única definición de transformación.
2. Especifican dos definiciones de transformaciones, donde cada una de ellas es la inversa de la otra.

En cualquiera de los dos casos, es difícil definir una transformación bidireccional. Bajo la primera opción, dadas las diferencias entre los lenguajes fuente y destino, es complejo confeccionar una definición de transformación que trabaje en ambos sentidos. Por ejemplo supongamos que transformamos un diagrama de estados de un modelo de negocios en código Java. En la transformación desde el modelo fuente a Java, podemos definir el estado como un atributo de una clase de tipo *boolean*. El problema surgirá al intentar transformar el código Java en un diagrama de estados, dada la imposibilidad de establecer con claridad qué atributos deberán confirmar estados en el modelo de negocios. Por otro lado, bajo la segunda opción, la implementación también resulta difícil, dada la complejidad para asegurar que cada definición de transformación es exactamente la inversa de la otra.

Existen otras razones por las cuales la bidireccionalidad es una propiedad de baja prioridad en las implementaciones de transformación de modelos. Por un lado, en casos donde se agrega información complementaria al modelo destino, o donde existe información en el modelo fuente que no es mapeada sobre el modelo destino, la bidireccionalidad no puede ser lograda. Por otro lado, la completa bidireccionalidad entre modelos sólo puede ser posible si el poder expresivo de los lenguajes de modelado fuente y destino es idéntico, lo cual a su vez implica que los niveles de abstracción de ambos lenguajes deben ser equivalentes. Teniendo en cuenta que el hecho que el PIM posea un mayor nivel de abstracción que el PSM agrega valor al uso del framework MDA, la posibilidad de restar beneficios a esta situación en procura de la bidireccionalidad hace que esto no genere gran interés, salvo en casos muy puntuales donde la aplicación de la transformación asigne por motivos específicos mayor prioridad a esta propiedad.

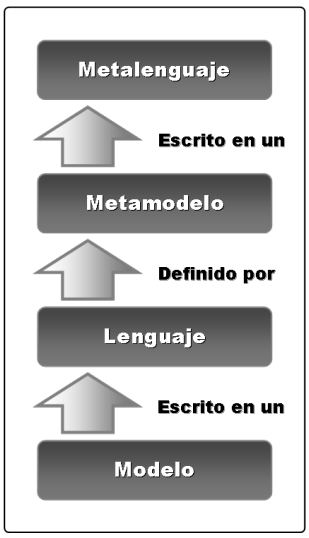


Figura 2.5: Relación entre modelos, lenguajes, metamodelos y metalenguajes

2.5. Metamodelado

El concepto de metamodelado se basa en el modelo de cuatro capas de modelado de la arquitectura del estándar MOF (Meta-Object Facility) [15]. En la presente sección analizaremos qué es el metamodelado, y por qué es relevante en el contexto de MDA.

2.5.1. Concepto de metamodelado

A menudo los lenguajes se definen a través de gramáticas en Backus-Naur Form (BNF), las cuales describen cuándo un conjunto de *tokens* conforman una expresión válida para dicho lenguaje. Al margen de su gran poder expresivo, y de la capacidad de ser interpretada por computadoras, esta herramienta posee una limitación clave para su utilización en el ámbito del framework MDA: el uso de BNF está restringido a la definición de lenguajes de texto. Dado que los lenguajes de modelado no necesariamente están basados en texto, por ejemplo UML, es necesario un mecanismo diferente para la definición de este tipo de lenguajes. Este mecanismo es conocido como *metamodelado*.

Un sistema se compone de elementos. Para ese sistema, existe un modelo que define qué elementos lo componen. A su vez, para dicho modelo existe un lenguaje que define qué elementos lo componen. Este lenguaje, por su parte, puede ser descripto por otro modelo, el cual define qué elementos pueden ser utilizados en el mismo. Pensando este último como modelo de un modelo, en el contexto del estándar del OMG, podemos decir que el mismo es un metamodelo. Finalmente, como el metamodelo es un modelo, existe un lenguaje que lo describe. Este, recibe el nombre de metalenguaje (Figura 2.5).

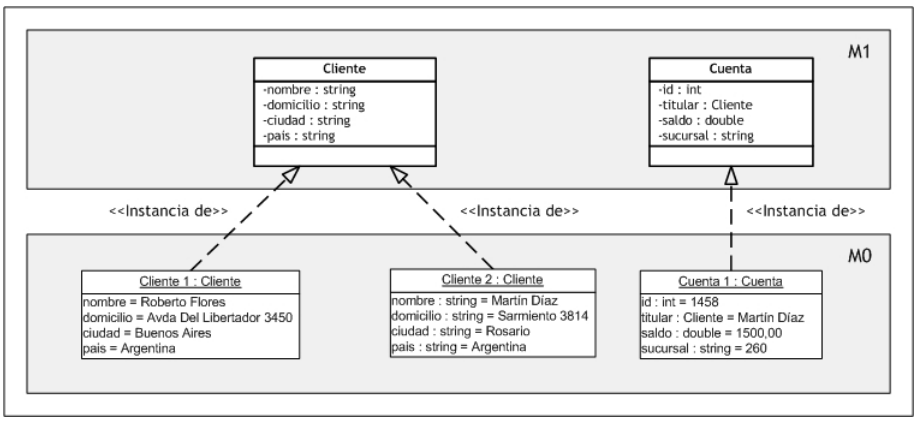


Figura 2.6: Relaciones entre capas M0 y M1

2.5.2. Las cuatro capas del OMG

El framework clásico para metamodelado está basado en una arquitectura de cuatro capas M0, M1, M2 y M3 [15]. El contenido de las capas se describe convencionalmente de la siguiente manera:

- La capa de información, M0, se compone de los datos que se desea describir.
- La capa de modelo, M1, está formada por los metadatos que describen los datos de la capa de información. Los metadatos son informalmente agregados como modelos.
- La capa de metamodelo, M2, se compone de la descripción que define la estructura y semántica de los metadatos. Estos meta-metadatos son agregados informalmente como metamodelos. Un metamodelo es un lenguaje abstracto para describir distintas clases de datos, es decir, un lenguaje sin una sintaxis o notación concreta.
- La capa de meta-metamodelo, M3, está conformada por la descripción de la estructura y semántica de los meta-metadatos. En otras palabras, se trata de un lenguaje abstracto para describir diferentes clases de metadatos.

A continuación presentaremos un ejemplo concreto para ilustrar las definiciones dadas.

2.5.2.1. Capa M0: Nivel de instancia

A nivel de capa M0 tenemos un sistema corriendo, dentro del cual existen las instancias "reales". Tenemos, por ejemplo, un cliente llamado "Roberto Flores", domiciliado en "Avda. del Libertador 3450" en la ciudad de "Buenos Aires", país "Argentina". Esta capa se conoce también como la capa de información.

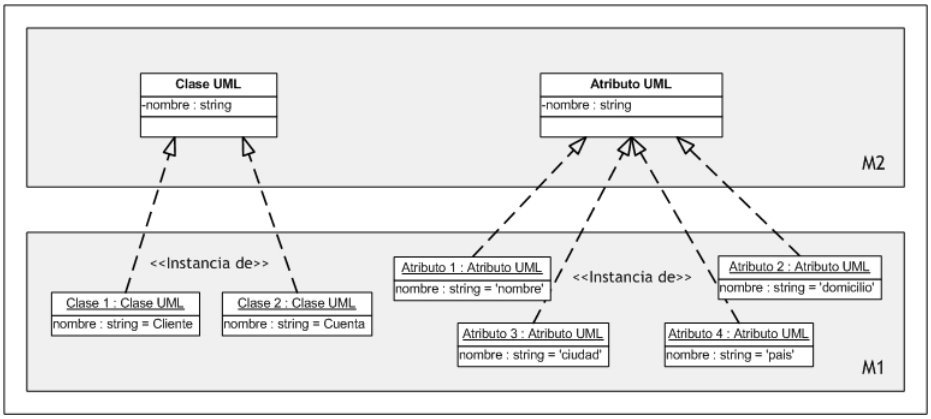


Figura 2.7: Relaciones entre capas M1 y M2

Al realizar un modelo un negocios, las instancias de M0 son los ítems del negocio en sí mismo. Al realizar modelos de software, estas instancias son las representaciones de software de los elementos del mundo real.

2.5.2.2. Capa M1: El modelo del sistema de software

La capa M1 se compone de metadatos que describen la información de la capa M0. A este nivel se definen los conceptos de negocio, por ejemplo la entidad "Cliente", con sus respectivas propiedades *nombre*, *domicilio*, *ciudad* y *país*.

Existe una marcada relación entre las capas M0 y M1. Los conceptos a nivel de M1 son categorizaciones o clasificaciones de las instancias de M0. De esta manera, los elementos de M0 son siempre una instancia de algún elemento de capa M1. Los clientes "Roberto Flores" y "Martín Díaz" son instancias del elemento de M1 "Cliente" (Figura 2.6).

2.5.2.3. Capa M2: El modelo del modelo

El modelo que reside en la capa M2 es llamado *metamodelo*. Cada modelo UML a nivel de capa M1 es una instancia de un metamodelo UML definido según una especificación de OMG. Al definir este metamodelo, de capa M2, estamos definiendo un lenguaje de modelado que nos permite escribir modelos.

Los elementos que existen a nivel de capa M1, tales como clases, atributos, y demás, son a su vez instancias de clases de M2 (Figura 2.7). La misma relación entre los elementos de M0 y M1 está presente entre los elementos de M1 y M2. Cada elemento de M1 es una instancia de un elemento de M2, y cada elemento de M2 categoriza a elementos de M1.

2.5.2.4. Capa M3: El modelo de M2

Continuando el razonamiento que aplicamos en capas previas, podemos pensar a cada elemento de capa M2 como una instancia de otro elemento de una capa

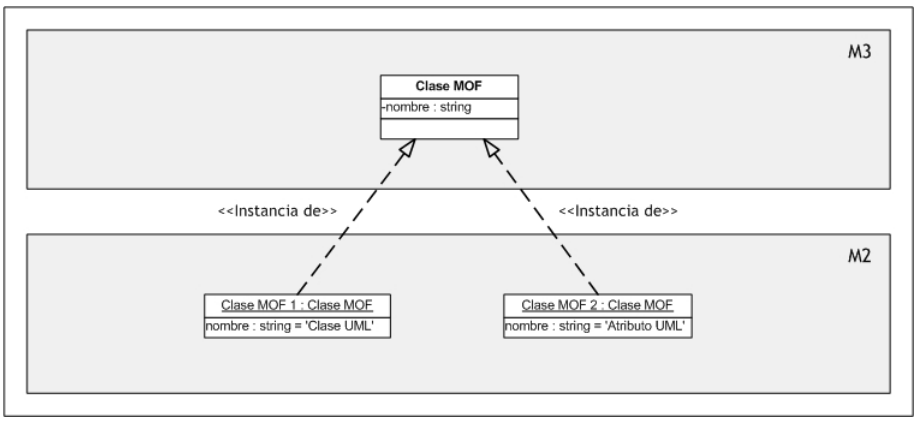


Figura 2.8: Relaciones entre capas M2 y M3

superior, la capa M3 o *meta-metacapa*. En este caso, la misma relación que encontrábamos entre los elementos de M0 y M1, y entre los elementos de M1 y M2, continúa existiendo entre capas M2 y M3. Cada elemento de M2 es una instancia de un elemento de M3, y cada elemento de M3 categoriza uno o más elementos de M2. La Figura 2.8 muestra la relación entre las capas M2 y M3. La notación utilizada para describir un meta-metamodelo es la misma que para la descripción de un metamodelo, y de un modelo. La Figura 2.9 muestra la arquitectura completa.

Dentro del esquema propuesto por el OMG, MOF [15] es el lenguaje M3 estándar. Todos los lenguajes de modelado (UML, CWM, y demás) son instancias del MOF.

En principio sería posible seguir agregando más niveles, pero la práctica ha demostrado que esto no incrementa el poder descriptivo de la arquitectura. En su lugar, el OMG propuso que todos los elementos de capa M3 deben ser definidos como instancias de conceptos de la propia capa M3. De hecho, la separación entre niveles es puramente superficial, y sólo a efectos de incrementar la compresión sobre el esquema. Lo esencial es la relación "es instancia de". En tanto cada elemento tenga su *metaelemento* clasificador a través del cual sus *metadatos* puedan ser accedidos, todo modelo podrá ser construido, y todo sistema descripto.

Por supuesto, en realidad todos los elementos de todos los niveles existen en el mundo real, y por lo tanto todos pertenecen a la capa M0. Algunos elementos en M0 son clasificaciones de otros elementos y por lo tanto también pertenecen a M1. Otros elementos de M0 son clasificaciones de un subconjunto de elementos en M0 que son a su vez clasificaciones otros elementos de M0, por lo que también pertenecen a la capa M2. La colección de elementos de M3 es un subconjunto de elementos de M2, la colección de elementos de M2 es un subconjunto de elementos de M1, y la colección de elementos de M1 un subconjunto de elementos de M0 (Figura 2.10). La arquitectura de cuatro capas es simplemente un mecanismo estructural que nos ayuda comprender los modelos y las clasificaciones.

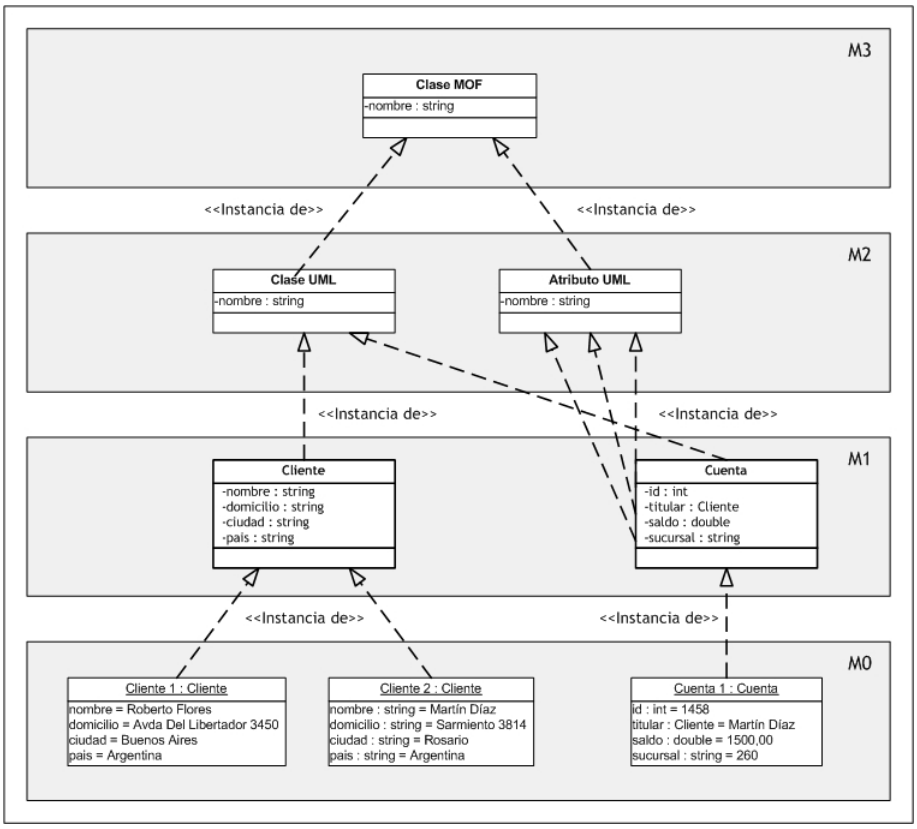


Figura 2.9: Esquema completo entre M0 y M3

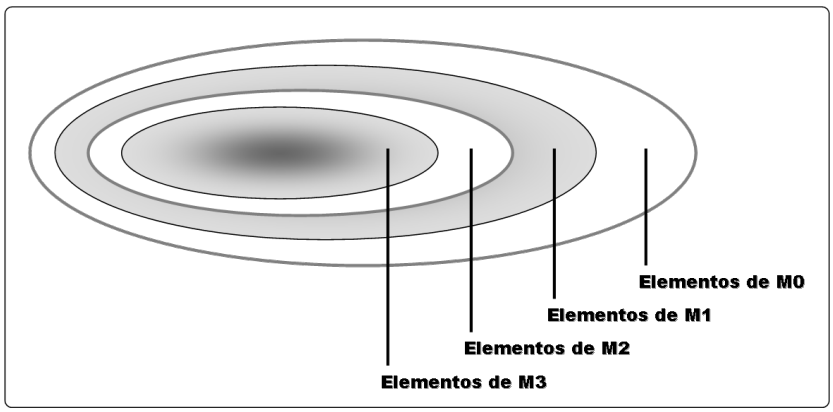


Figura 2.10: Relaciones entre subconjuntos de elementos de M0, M1, M2 y M3

2.5.3. El rol del metamodelado en MDA

Existen dos grandes razones que fundamentan la importancia del metamodelado en el framework MDA. Por un lado, es preciso un mecanismo que defina sin ambigüedad los lenguajes de modelado. Esto hace posible que una herramienta de transformación pueda leer, escribir y entender modelos. Dentro de MDA, los lenguajes se definen a través de metamodelos.

En segundo lugar, las reglas que constituyen la definición de una transformación describen cómo un modelo en un lenguaje fuente puede ser transformado en un modelo en un lenguaje destino. Estas reglas usan los metamodelos de los lenguajes fuente y destino para definir las transformaciones, al igual que la herramienta de transformación, que los requiere para la comprensión de dichos lenguajes.

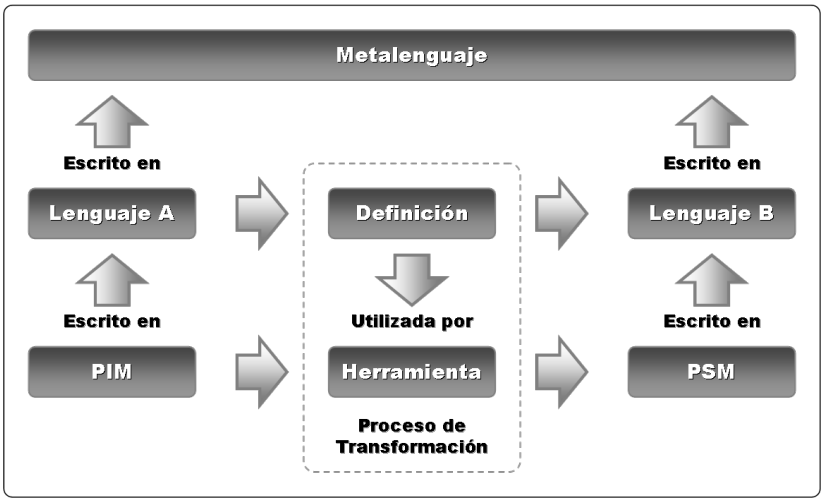


Figura 2.11: El framework MDA

La Figura 2.11 muestra cómo el framework MDA es completado con el nivel de metamodelado. En el nivel inferior del esquema tenemos la transformación de un modelo independiente de la plataforma (PIM) en un modelo específico (PSM) mediante una herramienta de transformación, la cual se basa en una definición que determina reglas de conversión. Ambos modelos están definidos por un lenguaje determinado, los cuales a su vez están escritos en un metalenguaje común.

Capítulo 3

Trazabilidad en MDD

Como observamos en el Capítulo 2, uno de los elementos centrales del desarrollo conducido por modelos son las transformaciones. Para dichas transformaciones, existen ciertas características deseables. El presente trabajo tiene como eje una de ellas: la trazabilidad (*traceability*). El concepto de trazabilidad esta ligado a la posibilidad de seguir el rastro de elementos del modelo destino (*target*) hacia atrás, a sus orígenes en el modelo fuente (*source*). De esta forma, decimos que una herramienta de transformación soporta trazabilidad si permite determinar el origen de los elementos del modelo destino.

En el presente capítulo repasaremos el concepto de trazabilidad y su importancia, identificando sus diversos aspectos. Luego, presentaremos distintos enfoques para la gestión de trazabilidad en el marco del desarrollo conducido por modelos, analizando brevemente las características más importantes de cada uno.

3.1. Concepto de trazabilidad

El Glosario Estándar de Ingeniería de Software del IEEE [68] define trazabilidad de la siguiente manera:

> (1) El grado en que una relación puede ser establecida entre dos o más productos del proceso de desarrollo, especialmente en aquellos que presentan relaciones predecesor-sucesor o maestro-subordinado entre sí; por ejemplo, el grado en que un requerimiento y el diseño de un componente de software dado coinciden.
>
> (2) El grado en que cada elemento de un producto de desarrollo de software establece su razón de existir; por ejemplo, el grado en que cada elemento de un diagrama de burbuja referencia el requerimiento que éste satisface.

Esta definición esta claramente influenciada por los orígenes de la trazabilidad, es decir, la gestión de requerimientos. En [44], Gotel y Finkelstein definen trazabilidad de requerimientos como:

> ... la habilidad de describir y seguir la vida de un requerimiento, en dirección hacia adelante o hace atrás (por ejemplo, desde sus orígenes, a través del desarrollo y especificación, hasta su subsecuente despliegue y utilización, y a través de períodos de refinamiento e iteración de alguna de estas fases).

Sin embargo, es posible encontrar definiciones más amplias, y más útiles a los efectos del desarrollo conducido por modelos. Por ejemplo, Aizenbud [16] define la trazabilidad como "cualquier relación que exista entre artefactos involucrados en el ciclo de vida de la ingeniería de software". Esta definición incluye, pero sin limitarse a:

- Relaciones explícitas o mapeos que son generados como resultado de transformaciones tanto hacia adelante (*forward*, como generación de código), como hacia atrás (*backward*, como ingeniería inversa).
- Relaciones que son computadas en base a información existente (por ejemplo, análisis de dependencia de código).
- Relaciones estadísticamente inferidas, las cuales son computadas en base a un historial provisto por algún sistema de gestión de cambios sobre ítems que fueron modificados como resultado de una solicitud de cambio.

De esta manera, la trazabilidad es lograda definiendo y manteniendo las relaciones entre los artefactos involucrados en el ciclo de vida de la Ingeniería de Software, durante el desarrollo del sistema.

3.2. La importancia de la trazabilidad

Existen varios estándares que establecen la obligatoriedad de la trazabilidad, por ejemplo el Estándar de Desarrollo de Software y Documentación del Departamento de Defensa de los EE.UU (MIL-STD-498) [69], o las prácticas recomendadas para la especificación de requerimientos de software publicadas bajo la norma IEEE/EIA 12207 [49]. Éstos, en general, derivan del modelo de ciclo de vida *Waterfall*, bajo el cual el rol de la trazabilidad se centraba en demostrar que el sistema resultante alcanzaba, o no, las definiciones de los acuerdos contractuales. Dichos estándares reflejan la visión de la práctica de la trazabilidad como una medida de la calidad del sistema y de la madurez del proceso de software.

Los distintos actores (*stakeholders*) en el proceso de desarrollo de software tienen diferentes metas de trazabilidad. Desde la perspectiva del gerente de proyecto, la rastreabilidad[1] le brinda soporte en la demostración de que cada requerimiento ha sido satisfecho, y que cada componente del sistema satisface un requerimiento. Desde la perspectiva de la gestión de requerimientos, la trazabilidad facilita el enlace entre los requerimientos y sus orígenes, y por sobre todas las cosas sus

[1] El término "rastreabilidad" proviene de la traducción al castellano de la palabra *traceability*, derivada del verbo inglés *to trace* ("seguir el rastro"), y es utilizado como sinónimo de trazabilidad.

fundamentos, capturando la información necesaria para comprender la evolución de los requerimientos, y la verificación de que los mismos han sido alcanzados. Durante la fase de diseño, la rastreabilidad permite determinar qué sucede si una solicitud de cambio es implementada previo a un rediseño del sistema. Con trazabilidad completa, la planificación y los costos de los cambios pueden ser determinados con mayor precisión, sin la necesidad de depender del conocimiento que el desarrollador tenga sobre la totalidad de las áreas afectadas por dichos cambios.

Aunque las ventajas de la rastreabilidad están bien documentadas, su práctica no se encuentra extendida. El argumento más utilizado es el alto costo de creación y mantenimiento de este tipo de información. Adicionalmente, la falta de guías acerca de qué información debe ser producida y el hecho que quiénes la utilizan no suelen ser los mismos que la generan también disminuye la motivación de aquellos que deben encargarse de generar y mantener estos datos. El desarrollo y uso de técnicas para rastrear requerimientos se originaron a principios de los 1970's con el objetivo de dar respuesta a preguntas tales como: ¿Es éste requerimiento necesario? ¿Por qué se implementó el diseño de esta manera? ¿Cuál es el impacto de cambiar un requerimiento?

El primer método utilizado para expresar y mantener información de trazabilidad fueron las referencias cruzadas, las cuales se basaban en incluir dentro de la documentación del proyecto frases del tipo "Ver Sección X". Posteriormente comenzaron a surgir nuevas herramientas para representar relaciones de rastreabilidad: matrices, bases de datos, hipervínculos, grafos, métodos formales y esquemas dinámicos, entre otros. El soporte automatizado para trazabilidad comenzó con herramientas de propósito general como procesadores de texto, planillas de cálculo o sistemas de base de datos, y se volvió más sencillo con el surgimiento del hipertexto. En la actualidad, el peor defecto de éste método continúa vigente: la información de rastreabilidad es creada y mantenida manualmente, al igual que la responsabilidad de gestionar su validez con respecto al cambio. Por lo tanto, esta información va quedando desactualizada conforme el sistema evoluciona.

El desarrollo de herramientas específicas para gestión de requerimientos tales como IBM Rational RequisitePro™y Telelogic DOORS™(luego adquirida por IBM Rational™) introdujeron soluciones de trazabilidad más avanzadas, como el soporte para la validación de la información de trazabilidad a través del control de cambios en elementos enlazados. Éstas, además, soportan la integración con otras herramientas de desarrollo para facilitar la trazabilidad a partir de requerimientos en otros productos del ciclo de vida del software. Sin embargo, a pesar de éstos avances, el trabajo de mantener la información de rastreabilidad continuó siendo una tarea manual ya que los requerimientos, comúnmente expresados como texto informal, requieren la presencia de una persona para su comprensión y determinación de validez. Por otro lado, la mayoría de las herramientas no proveen esquemas de trazabilidad sofisticados, permitiendo por ende sólo simples formas de razonamiento sobre las trazas.

3.3. Aspectos de trazabilidad en MDD

3.3.1. El rol de la trazabilidad en el desarrollo conducido por modelos

El esquema de desarrollo conducido por modelos o *Model-Driven Development* (MDD) reconoce la necesidad de tener varios tipos de modelos para representar un sistema de software desde la elicitación de requerimientos hasta la implementación final. Estos modelos representan distintos aspectos del sistema, tanto estructurales como de comportamiento, y a distintos niveles de abstracción. Debido a esto, uno de los aspectos centrales es el mantenimiento de la consistencia entre modelos y en transformaciones de modelos.

Las inconsistencias entre modelos que representan distintas vistas del sistema, o entre especificaciones a diferentes niveles de abstracción, pueden surgir durante la ejecución de una fase del proceso de desarrollo o bien entre fases, generando la necesidad de gestionar la consistencia entre diferentes modelos, y entre los modelos y el código. Algunos autores como Desfray [35] sostienen que sin una correcta gestión de trazabilidad, realizada automáticamente, los modelos y la implementación se tornan inevitablemente inconsistentes. Aunque un tanto lejano en el tiempo, Grundy *et al.* brindan en [46] un amplio espectro de herramientas y técnicas para la gestión de inconsistencias, sosteniendo que algunas inconsistencias pueden ser automáticamente corregidas, mientras que en otros casos simplemente no se puede, o bien no es recomendable, por lo que es necesario contar con mecanismos para informar a los desarrolladores de dichas inconsistencias, facilitando su detección y resolución.

MDA, la propuesta del Object Management Group (OMG) al enfoque de desarrollo conducido por modelos (MDD), se basa en el modelado y mapeo de modelos en implementaciones a través de sucesivas transformaciones de modelos. Dichas transformaciones pueden ser unidireccionales (de un modelo a otro) o bidireccionales (entre ambos modelos). La actualización de transformaciones unidireccionales y la sincronización de transformaciones bidireccionales requieren una manera de identificar los elementos existentes en el modelo destino que se encuentran relacionados con un elemento del modelo origen dado. Por esto, dichas transformaciones requieren la creación y mantenimiento de trazas entre ambos modelos origen y destino.

3.3.2. Consideraciones de diseño en un esquema de trazabilidad

En el contexto del desarrollo conducido por modelos no existe una clasificación de esquemas de transformación de modelos rígida, formalmente aceptada. El soporte de trazas es, en general, una característica inherente al lenguaje de transformación de modelos, y habitualmente en la práctica está fuertemente asociada a la implementación de dicho lenguaje.

En uno de los trabajos fundacionales en la materia, Czarnecki y Helsen [34] sugieren una posible taxonomía para la clasificación de las aproximaciones a la

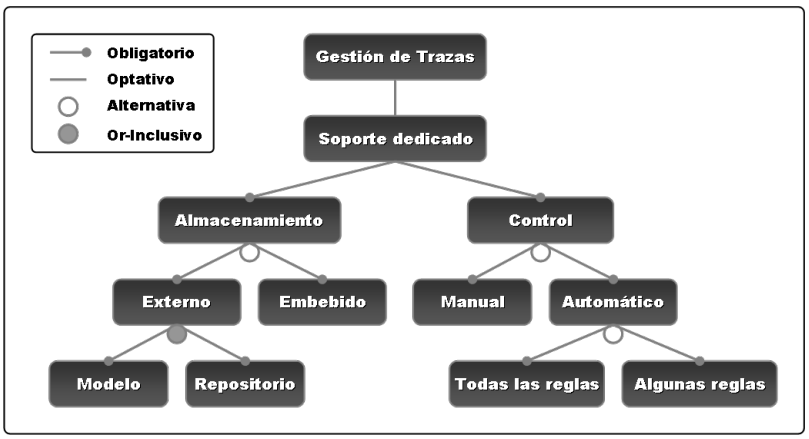


Figura 3.1: Características de diseño del soporte dedicado de trazabilidad

transformación de modelos, y la presentan como un modelo de características[2] que especifica potenciales alternativas de diseño para esquemas de transformación de modelos. La generación de trazas es una de las dimensiones de dicha clasificación. De acuerdo a los autores, las propuestas de transformación de modelos se dividen entre aquellas que brindan un soporte dedicado para trazabilidad y en las que dependen del usuario para la generación de información de trazabilidad.

Para el caso de implementaciones de soporte de trazas dedicado, el trabajo subraya dos aspectos fundamentales a tener en cuenta: por un lado el control, el cual puede ser manual (el desarrollador debe especificar las trazas) o automático, es decir, sin intervención de una persona, e inferidas de alguna desde la transformación, a partir del conjunto total de las reglas, o de un conjunto seleccionado de las mismas. Como segundo aspecto, se propone el destino del almacenamiento de las trazas generadas. En particular, éste puede estar en alguno de los modelos involucrados (origen o destino) o en almacenamiento separado, como vemos en el diagrama de características (*features*) de la Figura 3.1.

El almacenamiento de información de trazabilidad en los modelos, conocida como almacenamiento *intra-model*, se basa en mantener las trazas en forma de elementos del modelo, o como atributos de elementos de modelos, como *tags* o propiedades [36]. Al margen de su simplicidad y amigabilidad, mantener la información de esta manera puede ser problemático. Si las trazas son almacenadas en el modelo origen únicamente, la misma no estará disponible desde el modelo destino. Por otro lado, si la información es almacenada en ambos modelos, la dificultad radica en mantener la consistencia entre ambas versiones cada vez que ocurre un cambio. Por otro lado, embeber información de trazabilidad en un modelo genera una "contaminación" del mismo, es decir, la información que éste representa se ve alterada con la inclusión de elementos de una importancia secundaria respecto de lo que el modelo intenta representar.

[2] Un modelo de características o *feature model* es en esencia una representación abreviada de todos los productos de una línea de productos de software o SPL (Software Product Line), en términos de funcionalidad de un sistema.

En contraste con la estrategia de almacenamiento *intra-model*, el almacenamiento externo de información de trazabilidad consiste en la definición de un modelo separado de los modelos origen y destino para el mantenimiento de trazas. Este esquema presenta dos claras ventajas: en primer lugar, los modelos (origen y destino) involucrados en la transformación se mantienen "limpios", es decir, libres de la contaminación que implica el agregado de elementos adicionales. En segundo lugar, el almacenamiento de información de trazabilidad en un modelo que conforma un metamodelo, con una semántica claramente definida, facilita el análisis automatizado mediante herramientas.

En [51], Jouault realiza una distinción entre trazabilidad interna, la cual es mantenida por el motor de transformación durante el proceso de conversión de modelos, para asistir al algoritmo que establece la estrategia correspondiente, y entre trazabilidad externa, la cual es un mapeo persistente mantenido independientemente del proceso de ejecución de la transformación. Si bien existen propuestas de serialización[3] de información de trazabilidad interna, su principal desventaja es que el formato y granularidad de las trazas almacenadas se encuentran predeterminadas, y puede no ser compatible con la representación utilizada por otras herramientas que hacen uso de dicha información de trazabilidad, impactando negativamente sobre la interoperabilidad con otros sistema, y por lo tanto reduciendo su aplicabilidad.

Si bien se ha registrado un importante progreso en el área de la transformación de modelos, la integración con los esquemas de trazabilidad es aún débil. Las soluciones de transformación tienden a definir su propia trazabilidad interna y no se integran con soluciones de trazabilidad existentes que proveen capacidades de análisis y consulta sobre la información de trazabilidad [16].

3.3.3. Metamodelos de trazabilidad

El metamodelo de trazabilidad determina la estructura de las trazas. Las trazas derivadas como consecuencia de un proceso, sea automático o manual, deben conformar un metamodelo de trazabilidad. El modelado de trazas depende de la definición del metamodelo de trazabilidad a utilizarse. Existen dos enfoques respecto del uso de estos metamodelos: por un lado, la utilización de un metamodelo de propósito general, adaptable a múltiples situaciones, genérico, o el empleo de un metamodelo específico para cada escenario, desarrollado ad hoc, y diseñado para cumplir con determinados objetivos. La comparación entre ambos esquemas ha sido origen de varios trabajos. En particular, Drivalos *et al.* presentan en [36] un estudio donde se contrastan las dos aproximaciones.

En un metamodelo de propósito general, una traza puede conectar cualquier número de elementos, de cualquier tipo, pertenecientes a cualquier modelo. La principal ventaja de este esquema es la simplicidad y uniformidad, dado que todos los modelos de trazabilidad conforman al mismo metamodelo. Por otro lado, la aproximación facilita la interoperabilidad entre herramientas, dado que con un metamodelo de propósito general éstas pueden importar, exportar y

[3] También conocido como *marshalling*, es un proceso que consiste en la codificación de la representación de un objeto en memoria a un formato adecuado para almacenamiento o transmisión. Es un método usualmente utilizado para dar persistencia, vía almacenamiento en archivos, a los objetos de un sistema que se encuentran cargados en memoria volátil.

gestionar trazabilidad en un formato común. Sin embargo, dicho metamodelo no permite la captura de trazas específicas, fuertemente tipadas (*strongly typed*), con una semántica rigurosamente definida, posibilitando la generación de trazas potencialmente ilegítimas. Por ejemplo, asumiendo que se desea representar las trazas de una transformación de un modelo de clases en un modelo de base de datos relacional, un metamodelo genérico podría permitir el establecimiento de un enlace clase-columna, lo cual sería claramente una traza semánticamente incorrecta.

En contraste, los metamodelos específicos son definidos para cada escenario de rastreabilidad, ajustándose al enfoque del tipo de información que se desee obtener. Estos metamodelos permiten capturar trazas con una semántica bien definida que potencialmente puede incluir restricciones de correctitud que van más allá de una sencilla conformación de tipos. En esencia, capturan información específica. Como aspectos negativos, podemos mencionar que los metamodelos de rastreabilidad específicos requieren de un mayor esfuerzo para su construcción, y su falta de uniformidad, si bien puede ser positiva, atenta contra la interoperabilidad entre herramientas que soportan distintos metamodelos.

Como resultado, el estudio concluye que mientras la primera aproximación (metamodelos genéricos) facilita la interoperabilidad entre los modelos de trazas, no permite ajustarse a la realidad del escenario, dado que los modelos de traza resultan demasiado generales. En contraste, los metamodelos específicos posibilitan la generación de modelos de trazas que se adaptan mejor a escenarios concretos, aunque implican una mayor esfuerzo ya que requieren la definición de un metamodelo para cada escenario, y el consecuente problema de interoperabilidad entre modelos de distintos escenarios. Finalmente, los autores se vuelcan hacia la utilización de metamodelos de trazabilidad específicos, por tener una mayor rigurosidad a la hora de la definición de trazas.

3.3.4. Antecedentes en la creación automatizada de trazas

En [16], Aizenbud-Reshef *et al.* presentan distintos esquemas de trazabilidad automatizada y discuten el rol potencial del desarrollo conducido por modelos en este campo. En efecto, como sostienen los autores, la tarea de especificación y mantenimiento de la información de rastreabilidad genera una pesada carga que amenaza considerablemente de aceptación general de las prácticas de trazabilidad. Debido a esto, gran parte de la investigación en la materia se ha volcado a encontrar maneras de automatizar tanto la creación de trazas como el mantenimiento de las mismas.

Una de las primeras líneas de investigación se basó en el empleo de minería de texto (*text mining*) y técnicas de recuperación de información (*information retrieval*) para inferir relaciones entre artefactos de software. Alexander [18] describe un método semiautomático para la creación de trazas entre casos de uso y sus respectivas referencias, y entre términos y sus definiciones en un glosario. La técnica de marcado (*markup*) es utilizada para identificar determinados términos a enlazar, y luego una herramienta automatizada busca los elementos relacionados, generando documentos de tipo hipertexto. En [20], Antoniol *et al.* proponen un método basado en IR (*Information Retrieval*) para recuperar trazas entre código fuente y documentos de texto libre. Su propuesta consiste

en la aplicación de dos modelos de recuperación de información, uno probabilístico y otro de espacio vectorial, sobre dos casos de estudio distintos logrando generar las trazas entre código fuente C++ y páginas manuales, y entre código fuente Java y un documento de requerimientos funcionales. Otros trabajos similares pueden encontrarse en [48], donde se presenta un método para mejorar la selección de trazas candidatas mediante IR, y en [57], donde Marcus *et al.* presentan una técnica de *information retrieval* avanzada llamada análisis semántico latente o *latent semantic analysis* para extraer el significado (semántica) de la documentación y el código fuente, y así utilizar esta información para identificar trazas basándose en medidas de similaridad.

Otra de las líneas de investigación ha sido el análisis de las relaciones para la obtención de relaciones implícitas. En [67], Sherba *et al.* presentan técnicas basadas en hipermedia abierta (*open hypermedia*) e integración de información, las cuales proveen servicios que permiten el descubrimiento, creación, mantenimiento, visualización y seguimiento de relaciones. Los usuarios de este sistema pueden definir nuevas relaciones derivadas como una cadena de tipos de relaciones existentes ($rel_1 \rightarrow rel_2 \rightarrow \ldots \rightarrow rel_n$) y el sistema automáticamente descubre otras instancias de las relaciones derivadas. Otro trabajo similar puede ser encontrado en [40], donde Egyed y Grünbacher presentan un esquema que exige una mínima información de trazabilidad inicial entre distintos escenarios y los tests que los validan, y entre los tests y elementos de diseño, para luego poder establecer automáticamente las trazas entre los requerimientos y el código mediante la verificación de qué clases fueron activadas como resultado de la ejecución de un escenario de prueba para un requerimiento específico. Una vez disponible ésta información, se aplican métodos de análisis para encontrar trazas adicionales entre requerimientos, y entre requerimientos y elementos de modelo, como por ejemplo transiciones de estado.

Finalmente, el análisis de cambio (*change analysis*) a sido utilizado como fuente para la generación automática de trazas. En [74], Zimmermann *et al.* aplican diversas técnicas de *data mining* sobre el historial de cambios de un código base para determinar patrones de cambio, es decir, el conjunto de archivos que fueron modificados simultáneamente con cierta frecuencia en el pasado. Los archivos pertenecientes al mismo conjunto de cambios están relacionados y pueden ser utilizados para recomendar código potencialmente relevante a un desarrollador que está realizando tareas de modificación y/o mantenimiento. Otro trabajo similar puede encontrarse en [73].

3.4. Enfoques de trazabilidad en MDD

En la Sección 3.3.4 presentamos algunos antecedentes en la generación automática de trazas en el marco del desarrollo conducido por modelos (MDD), tratándose en todos los casos de propuestas para la gestión de trazabilidad de requerimientos. No obstante, en el contexto del desarrollo conducido por modelos es posible encontrar otros enfoques sobre trazabilidad.

Galvao *et al.* presentan en [42] un interesante estudio del estado del arte en esquemas de trazabilidad en MDD, evaluando distintas propuestas en base a los siguientes cinco criterios: representación de trazas, mapeo, escalabilidad, análisis

de impacto de cambios y herramientas de soporte provistas (*tool support*). A los efectos de la comparación, los autores proponen la clasificación de las propuestas en tres categorías, a saber:

1. Enfoques basados en requerimientos (*requirements-driven*).
2. Enfoques basados en modelos.
3. Enfoques basados en transformaciones.

Los enfoques basados en requerimientos utilizan modelos de requerimiento como abstracciones para guiar sus métodos de trazabilidad. Los esquemas basados en modelos se interesan en cómo los metamodelos, modelos y/o frameworks conceptuales interactúan en el proceso de trazabilidad. Finalmente, las propuestas basadas en transformaciones hacen uso de los mecanismos de transformación de modelos para generar información de trazabilidad.

A modo de reseña, presentaremos un breve resumen de cada una de las propuestas. En particular, en el Capítulo 7 abordaremos con mayor detalle los esquemas basados en transformaciones, efectuando una comparación con la propuesta presentada en este trabajo, la cual como se verá más adelante corresponde a esa clasificación.

3.4.1. Propuestas basadas en requerimientos

3.4.1.1. Trazabilidad de requerimientos y conformidad de trasformaciones (*Requirements Traceability and Transformation Conformance*)

En [19], Almeida *et al.* proponen un framework como base para trazabilidad de requerimientos, evaluando la calidad de las especificaciones de transformación de modelos, metamodelos y modelos. Este framework metodológico permite a los diseñadores relacionar los requerimientos elicitados en etapas tempranas del desarrollo con varios productos del proceso conducido por modelos mediante tablas cruzadas (*cross-tables*), considerando diferentes granularidades de modelo y permitiendo la definición de especificaciones de conformidad (*conformance*) de transformaciones.

3.4.1.2. Trazabilidad basada en eventos (*Event Based Traceability*)

La trazabilidad basada en eventos o EBT (*Event Based Traceability*) es un método para automatizar la generación y el mantenimiento de trazas. En [29], Cleland-Huang *et al.* presentan un estudio para trazabilidad de requerimientos que utiliza un enfoque de trazabilidad basada en eventos para la gestión de cambios evolutivos. En este método, los requerimientos y otros objetos rastreables no están directamente relacionados, sino que se vinculan mediante relaciones publicador-suscriptor provista por un mecanismo basado en el patrón de diseño *Observer* [43]. Inicialmente, todos los artefactos son registrados en un servidor de eventos por el gestor de suscripciones. Luego, el administrador de requerimientos utiliza un algoritmo de reconocimiento de eventos para manejar las

actualizaciones sobre el documento de requerimientos, y publica estos cambios en el servidor de eventos. El servidor de eventos gestiona entonces las trazas entre los requerimientos y sus artefactos dependientes, utilizando algoritmos de *information retrieval*.

Los componentes principales de este esquema son el servidor de eventos, el gestor de requerimientos y el gestor de suscripciones. El gestor de requerimientos maneja los requerimientos y es el responsable de disparar los eventos de cambio a medida que se van produciendo. El servidor de eventos es el responsable de manejar las suscripciones, recibir notificaciones de cambio, y redireccionar mensajes de eventos a los gestores de suscripción de los artefactos dependientes. El gestor de suscripciones, por su parte, es responsable de recibir notificaciones de eventos y manejarlas de manera apropiada según el tipo de mensaje recibido. Estos mensajes incluyen información tanto semántica como estructural del contexto del cambio realizado.

3.4.1.3. Trazabilidad centrada en metas (*Goal Center Traceability*)

En [31], Cleland-Huang *et al.* introducen una aproximación basada en metas para la gestión del impacto de cambios en requerimientos no funcionales de un sistema de software. El esquema de trazabilidad centrada en metas o *Goal Centric Traceability* permite modelar requerimientos no funcionales y sus respectivas dependencias utilizando un grafo conocido como *Softgoal Interdependency Graph* (SIG). La trazabilidad centrada en metas permite a los desarrolladores comprender y evaluar el impacto de cambios funcionales sobre los requerimientos no funcionales para mantener la calidad del sistema.

3.4.1.4. Trazabilidad basada en eventos con patrones de diseño (*Event Based Traceability with Design Patterns*)

Nuevamente Cleland-Hung y su equipo nos presentan en [30] una alternativa para trazabilidad en MDD. Esta propuesta, basada en EBT [29] describe un proceso diferente para la obtención dinámica de trazas de requerimientos no funcionales con patrones de diseño. Este proceso se divide en dos fases, a saber:

- Durante la fase inicial, se establecen trazas definidas por el usuario. Pero en lugar de establecer enlaces (trazas) entre cada elemento del modelo de diseño con un requerimiento no funcional en un grafo de interdependencias (SIG), los elementos son enlazados a agrupaciones o *clusters* definidos por el patrón de diseño, por lo que luego las trazas son establecidas entre los requerimientos no funcionales y el *cluster*, disminuyendo así el número de trazas entre los artefactos de diseño y requerimientos no funcionales.
- En la segunda fase, las descripciones establecidas y reglas invariantes de un patrón de diseño permiten la generación de trazas de refinamiento (*fine-grained*), de manera automática.

Los autores proponen el uso de trazas generadas de manera dinámica y estática, al igual que la utilización de patrones de diseño como modelos intermediarios para facilitar la rastreabilidad de requerimientos no funcionales a través del ciclo de vida del desarrollo de software.

3.4.1.5. Modelos de referencia para trazabilidad de requerimientos (*Reference Models for Requirements Traceability*)

Ramesh *et al.* siguen en [62] un enfoque empírico para estudiar un amplio número de prácticas de trazabilidad. Como resultado, los autores constituyen modelos de referencia que incluyen los más importantes tipos de traza para varios elementos de desarrollo de software. Una de las principales motivaciones tras este estudio es capturar los objetivos de trazabilidad de distintos *stakeholders* y presentar distintos modelos de referencia para cada una de esas necesidades. Su estudio empírico caracteriza a los participantes como usuarios de alto nivel (*high-end*) o de bajo nivel (*low-end*) de prácticas de trazabilidad, y presenta modelos de traza que reflejan las entidades capturadas por cada grupo, y luego personaliza un conjunto de cinco modelos de referencia. Los requerimientos son considerados como entidades "traceables" en todos modelos de referencia.

Los modelos de referencia para usuarios de bajo nivel se componen de cuatro elementos (requerimientos, procedimientos de verificación de conformidad, componentes de sistema y sistemas externos) los cuales están interrelacionados por enlaces que describen satisfacción, derivación, dependencias, y demás. El uso de trazabilidad de alto nivel emplea modelos más ricos: un submodelo de gestión de requerimientos, un submodelo de fundamentos, un submodelo de alocación de diseño y un submodelo de verificación de conformidad.

3.4.2. Propuestas basadas en modelos

3.4.2.1. Análisis de dependencia de trazas conducido por escenarios (*Scenario Driven Approach to Trace Dependency Analysis*)

En [39], Egyed presenta un enfoque automatizado para la generación y validación de dependencias de traza. Los elementos considerados para el análisis son: escenarios de test, elementos de modelo (flujos de dato, casos de uso y diagramas de clase) y clases de implementación. Estos elementos pueden estar relacionados por diferentes tipos de dependencias de traza. Los principales pasos del enfoque son dos: generación de trazas y validación de trazas.

El comportamiento del sistema es observado mediante el uso de escenarios de test. La ejecución de estos escenarios sobre un sistema conduce a trazas observables que relacionan estos escenarios con clases de implementación o código fuente. El esquema propuesto reduce la complejidad de la generación y validación de trazas mediante un conjunto de escenarios de test y elementos de modelo.

3.4.2.2. Semántica operacional para trazabilidad (*Operational Semantics for Traceability*)

Existen diferentes tipos y representaciones de trazabilidad, con diferentes características y propiedades. Aizenbud-Reshef *et al.* presentan en [17] una aproximación que define una semántica operacional para trazabilidad en UML que permite capturar distintos tipos de trazas, utilizando una notación común para

toda situación. Adicionalmente, el estudio propone una herramienta de soporte más completa para el monitoreo y gestión de trazabilidad.

La semántica operacional de una relación de trazabilidad describe:

- Qué información debe ser mantenida entre los elementos relacionados.
- Qué acciones deben dispararse cuando un evento afecta uno o más elementos relacionados.
- Qué acciones deben dispararse cuando un evento impacta la relación en sí misma, para asegurar que la misma siga manteniendo su validez.

El trabajo aborda tres aspectos fundamentales para trazabilidad: las consultas (por ejemplo para el análisis de impacto), el seguimiento de trazas a lo largo del ciclo de vida del proyecto, y el mantenimiento actualizado del sistema y su correspondiente documentación. Basándose en estos aspectos, los autores define dos tipos de semántica: la semántica preventiva y la semántica reactiva. Mientras la semántica preventiva indica qué no debe suceder (por ejemplo, la eliminación o modificación de un elemento), la semántica reactiva indica cómo proceder en caso que suceda algo a uno o más elementos relacionados.

La semántica operacional de una relación de trazabilidad está definida por un conjunto de una o más propiedades semánticas. La propiedad semántica (*semantic property*) de una relación es una terna {*eventos*, *condiciones*, *acciones*} donde un evento es la ocurrencia de un cambio e involucra un elemento o una relación (usualmente una operación "*create*", "*update*" o "*delete*"), una condición es una restricción lógica utilizada para definir el contexto en el cual los eventos tienen lugar, y las acciones son pasos a seguir, pudiendo ser preventivos o reactivos.

3.4.2.3. Esquema de especificación de trazabilidad unificador (*Unifying Traceability Specification Scheme*)

En [56], Limón y Garbajosa analizan varios esquemas de trazabilidad y prueban un esquema de especificación de trazabilidad basado en este análisis para facilitar la especificación de trazabilidad en un proyecto determinado, para mejorar la gestión de trazabilidad, y para ayudar a automatizar algunos procesos asociados. El punto de partida de la evaluación realizada consiste en el estudio de:

1. Trazas asociadas al proceso (*process-related*) o trazas asociadas al producto (*product-related*).
2. Categorías de relaciones de trazabilidad pre-RS (*Requirement Specification* o Especificación de Requerimientos) o post-RS.
3. El propósito de las trazas.
4. Items u objetos relacionados por la traza.

Los autores establecen la necesidad de un esquema de especificación de trazabilidad que sea unificador, y logre agrupar los aspectos comunes de la mayoría de las propuestas. En base a su definición inicial, el esquema de especificación de trazabilidad debería incluir los siguientes ítems:

- Un conjunto de datos de trazabilidad aplicable a diferentes clases de proyecto, utilizando distintos tipos de proceso.
- Un conjunto de tipos de trazas, con el objetivo de definir qué tipo de información debería contener cada traza.
- Un conjunto mínimo de trazas, buscando definir todos los tipos de traza para un proyecto especifico o línea base de trazabilidad (*traceability baseline*).
- Un conjunto de métricas para la agrupación mínima que definan qué métricas pueden o deben ser aplicadas a una estrategia de medición para verificar el correcto despliegue y gestión de trazabilidad.

3.4.2.4. Metadatos de trazabilidad de transformaciones precisos (*Precise Transformation Traceability Metadata*)

Vanhooff *et al.* definen en [71] un perfil (*profile*) UML basado en un metamodelo que brinda soporte a trazas en transformaciones de modelos. Su propuesta permite agregar trazas de transformación con mayor contenido semántico en modelos UML, manteniendo su consistencia. Estas trazas hacen posible que los motores de transformación puedan explorar mejor los modelos y realizar trasformaciones más productivas, permitiendo además una mejor comprensión de los efectos de las transformaciones de modelos.

En esencia, el trabajo propone el almacenamiento de metadatos de trazabilidad en modelos UML de manera que las unidades de transformación[4] puedan contar con dicha información y de esta forma mejorar su mapeo de elementos. Estas trazas proveen un historial de cambios de modelo completo o parcial generados por las unidades de transformación que fueron ejecutadas contra dicho modelo, brindando de esta manera la posibilidad de razonamiento sobre transformaciones pasadas.

El metamodelo sugerido se compone de unidades de transformación, elementos de entrada y salida, y mapeos de elementos. De esta forma, el perfil conceptualiza dependencias entre elementos origen (*source*) y destino (*target*), dependencias entre un mapeo y la unidad de transformación que lo generó, y la identificación de los elementos eliminados. El uso del perfil permite el mantenimiento de información de trazabilidad en los modelos UML, pudiendo éste ser extendido según las necesidades de diferentes unidades de transformación, sin depender de ningún lenguaje de transformación.

[4] Denominación genérica utilizada por los autores para describir los elementos constitutivos de una transformación, por ejemplo, las reglas.

3.4.3. Propuestas basadas en transformaciones

Dado que el desarrollo conducido por modelos soporta la automatización de la creación y descubrimiento de relaciones entre modelos, las transformaciones de modelos pueden ser consideradas un mecanismo para la generación de trazas. Por consiguiente, la mayoría de los lenguajes de transformación de modelos soportan la creación automática y la utilización de trazas entre modelos, pero esta facilidad no garantiza que las transformaciones sean bien exploradas, o estén convenientemente adaptadas, para colaborar con las prácticas y/o explotación de la información de trazabilidad.

En particular, estas propuestas centrarán todo nuestro interés dado que el esquema sugerido en el presente trabajo se enmarca en el contexto de los enfoques de trazabilidad basados en transformaciones. En nuestro caso particular, no se hace uso del proceso de transformación de modelos para la generación de información de trazabilidad, sino que la obtención de dichos datos se basa en el análisis de la definición (código fuente) de la transformación de modelos, como detallaremos más adelante.

A continuación analizaremos algunos de los trabajos relacionados, y más adelante en el Capítulo 7 realizaremos un contraste entre algunas de las propuestas de esta misma índole.

3.4.3.1. Trazabilidad escasamente acoplada (*Loosely Coupled Traceability*)

En [51], Jouault muestra cómo la información de trazabilidad puede ser agregada en programas escritos en el lenguaje de transformación de modelos ATLAS (ATL o *ATLAS Transformation Language*) [50] con el objetivo de alcanzar los límites de la trazabilidad implícita. ATL es un lenguaje de transformación de modelos que permite soporte dedicado para trazabilidad, pero su mecanismo de generación de trazas es implícito.

El autor considera a la información de trazabilidad como un modelo, más precisamente como un modelo destino o *target* de un programa de transformación, permitiendo así la generación de elementos de trazabilidad de la misma manera que son creados los elementos del modelo destino al momento de procesar la transformación. Para integrar la trazabilidad en los programas de transformación, los desarrolladores deben incorporar piezas de código ATL adicional a sus especificaciones de transformación, mediante un proceso manual.

Dado que las transformaciones son modelos, es posible que un programa ATL sea transformado automáticamente en otro programa ATL con la incorporación del código generador de información de trazabilidad, logrando así automatizar el proceso manual. El programa ATL que realiza la inserción del código de creación de trazas se llama *TracerAdder*. Dado que dicho programa puede ser utilizado antes de la compilación del código ATL, el mismo puede ser considerado un precompilador. Una de las ventajas que aporta esta solución se basa en que la generación de información de trazabilidad es explícita, pero no se encuentra fuertemente acoplada a la lógica del programa de transformación, de ahí su nombre.

3.4.3.2. Trazabilidad de fusión por demanda (*On Demand Merging of Traceability*)

Kolovos *et al.* presentan en [54] un esquema para la fusión o mezcla de modelos primarios con su correspondiente modelo de trazabilidad, generando así modelos anotados según demanda, los cuales contienen información de trazabilidad útil para propósitos de inspección. Las trazas generadas pueden ser almacenadas y gestionadas utilizando dos aproximaciones distintas, a saber:

1. En la primer alternativa, conocida como trazabilidad embebida, las trazas son almacenadas en los modelos destino (*target*) en forma de nuevos elementos de modelo. Esta opción, si bien facilita la definición y comprensión de las trazas, promueve la generación de elementos que no pertenecen al modelo.

2. La segunda alternativa, en respuesta al problema de la "contaminación" de modelos destino, propone el almacenamiento de trazas en modelos separados.

La fusión por demanda de trazas con modelos requiere que los elementos de los modelos relacionados tengan una identificación persistente, afectando la comprensibilidad de las trazas. Por esto, los autores recomiendan que la información de trazabilidad debe ser mantenida en modelos separados, los cuales pueden ser fusionados con los modelos primarios según demanda.

La fusión o *merging* de modelos es implementada a través de Epsilon Merging Language (EML), un lenguaje híbrido basado en reglas que permite realizar la fusión de modelos homogéneos o heterogéneos, y forma parte del proyecto Epsilon [12], un subproyecto de Eclipse que presenta una familia de lenguajes y herramientas para la generación de código, transformaciones entre modelos, validación, comparación y migración de modelos, entre otras propuestas. El proceso de *merging* es completado en dos fases: *matching* y *merging*. Durante la primera, se establecen las correspondencias entre elementos del modelo origen, mientras que en la segunda etapa los elementos identificados anteriormente son fusionados. El lenguaje EML está implementado como plugin del entorno de desarrollo Eclipse y soporta definiciones de modelos en varios formatos, como EMF (*Eclipse Modeling Framework*), MOF y documentos XML.

3.4.3.3. Framework de trazabilidad para transformación de modelos (*Traceability Framework for Model Transformation*)

En [41], Falleri *et al.* definen un framework de trazabilidad para facilitar la generación de trazas en transformación de modelos. Su trabajo está inspirado en la propuesta de Jouault [51] e implementado en el lenguaje orientado a modelos Kermeta [13]. El framework permite el seguimiento de cadenas de transformaciones dentro de Kermeta por medio de la especificación e implementación de un metamodelo de trazabilidad independiente del lenguaje. Dicho metamodelo define trazas de transformación de modelos como un grafo bipartito[5] en el cual los nodos se clasifican como nodos fuente y nodos destino.

[5]En teoría de grafos, un grafo bipartito es uno cuyos vértices pueden separarse en dos conjuntos disjuntos A y B, donde las aristas sólo pueden conectar vértices de un conjunto

En el metamodelo, cada traza es un conjunto ordenado de pasos de la cadena de transformación, que representa una transformación simple de un objeto del modelo origen sobre un objeto del modelo destino, pudiendo dicho objeto representar cualquier tipo de elemento del modelo, dependiendo del nivel de granularidad utilizado.

Los autores han implementado las siguientes funcionalidades del framework:

- Items de trazabilidad genéricos.
- Serialización de trazas.
- Visualización de trazas en transformaciones simples mediante Graphviz [7] (software *open source* para visualización de grafos).

3.4.3.4. Extracción de datos de trazabilidad basada en *Facets* (*Facet-based Traceability Data Extraction*)

Si bien no forma parte del trabajo de Galvao [42], dado que fue publicado con posterioridad, Grammel y Kastenholz presentan en [45] un framework de trazabilidad genérico para la extracción de información de trazabilidad basado en facetas (*facets*) en el marco del desarrollo conducido por modelos. Dicho framework se compone de una interfase de trazabilidad genérica (GTI, *Generic Traceability Interfase*) que provee el punto de conexión para lenguajes de transformación arbitrarios y brinda un API para conectarlos con el motor de trazabilidad. El framework, además, define un lenguaje de dominio específico (Domain-Specific Language o DSL) llamado Trace-DSL que en esencia determina qué tipo de información de trazabilidad es intercambiable entre la interfaz genérica y los conectores de los motores de rastreabilidad.

El trabajo intenta direccionar tres problemas presentes en cualquier esquema de trazabilidad en el ámbito del desarrollo conducido por modelos o MDD (Model-Driven Development): por un lado, la unificación de metamodelos de trazabilidad, dispares entre esquemas de generación de trazas implícitas (por ejemplo, en el lenguaje QVT [60]) y explícitas (como el lenguaje de transformación oAW [11]), por otro lado, la extensibilidad de dichos metamodelos de trazabilidad, que dependiendo de las metas de rastreabilidad definidas en cada caso pueden no tener la expresividad suficiente para abarcar todos los escenarios requeridos, y finalmente la minimización del esfuerzo para lograr trazabilidad, en particular en aquellos casos de generación de trazas explícitas, donde se pretenden minimizar los esfuerzos manuales. Claramente, la propuesta abarca un amplio espectro de aspectos de trazabilidad del paradigma de desarrollo conducido por modelos, y propone una solución genérica para ciertos lenguajes de transformación de modelos.

3.4.4. Cuestiones abiertas

Del análisis comparativo de las distintas propuestas para trazabilidad en el contexto del desarrollo conducido por modelos, Galvao y sus colaboradores [42]

---

con vértices del otro conjunto, es decir $\forall a_1, a_2 \in A$, $\forall b_1, b_2 \in B$ no existe arista e tal que $e = (a_1, a_2)$ o $e = (b_1, b_2)$.

han identificado las siguientes cuestiones abiertas, es decir, aspectos sin un direccionamiento concreto, o con un direccionamiento escaso, que merecen mayor atención:

- En las etapas más tempranas del ciclo de vida MDD es donde se encuentra un menor grado de automatización de la trazabilidad. La falta de modelos apropiados para requerimientos, metas, y demás características de las fases iniciales hace que, dependiendo de la forma en que el proceso conducido por modelos es aplicado y de la correctitud en la definición de los modelos, la captura de información de rastreabilidad puede ser facilitada y automatizada.
- La construcción de mejores metamodelos de trazabilidad y una mayor utilización de técnicas *model-driven* deben ser dos puntos a tener en cuenta en nuevas propuestas de trazabilidad en MDD. La semántica de los modelos de trazabilidad y su estructura es una cuestión abierta. Si bien este aspecto es independiente del proceso de desarrollo utilizado, MDD puede ayudar en la creación automática de trazas sobre la base de un metamodelo que presenta una correcta taxonomía de trazas. La necesidad de uniformidad en metamodelos de trazabilidad puede ser considerado un aspecto importante y escasamente direccionado hasta el momento.
- Otra cuestión abierta es el descubrimiento de trazas entre elementos de modelo cuando dichas trazas no se encuentran explícitamente definidas. Ninguno de los enfoques revisados explora mecanismos para la inferencia de trazas implícitas.
- Los mecanismos de evolución de trazas no están previstos en casi ninguno de los enfoques de trazabilidad analizados. Conforme los modelos y transformaciones evolucionan, es preciso que los modelos de trazabilidad se vayan actualizando, para mantener la consistencia de este tipo de información y lograr un proceso de trazabilidad de alta calidad.

El estudio muestra que el grado de implementación de las herramientas de soporte (*tool support*) de las propuestas es crucial para la automatización de trazabilidad en MDD. Por otro lado, la representación de información de trazabilidad juega un rol fundamental para la obtención de los beneficios que la aplicación de las técnicas de trazabilidad puede brindar. Si bien la taxonomía de trazas es independiente del proceso de desarrollo, las técnicas *model-driven* pueden ayudar a una buena especificación de metamodelos que permitan cubrir las necesidades específicas. Algunos de los factores que inhiben la automatización en las prácticas de trazabilidad son la ausencia, falta de precisión, o inconsistencia de la información concerniente a los elementos de modelo y las trazas.

El soporte de trazabilidad no necesariamente debe ser una propiedad del lenguaje de transformación. Puede ser provista por el motor de transformación o eventualmente por un desarrollador a cargo de la creación y administración de las trazas. No obstante, la generación dinámica es fundamental en MDD para el soporte de cadenas de transformación de modelos.

Finalmente, la representación externa de trazas, almacenadas en modelos separados que pueden ser combinados con los modelos primarios que referencia,

posibilita un bajo acoplamiento entre modelos e información de trazabilidad, generando en consecuencia mecanismos de trazabilidad más flexibles que permitan la inspección y toma de decisiones durante el proceso MDD.

3.5. Resumen

A lo largo del presente capítulo, hemos abordado distintos aspectos de trazabilidad. En primer lugar se revisó el concepto de trazabilidad, desde sus orígenes hasta su redefinición en el contexto del paradigma MDD. A continuación justificamos su importancia y rol en el marco del desarrollo conducido por modelos. Luego repasamos algunos aspectos de diseño para la implementación de un esquema de trazabilidad, y analizamos ventajas y desventajas de un metamodelo de trazabilidad de propósito general frente a uno específico.

A continuación presentamos antecedentes de propuestas de generación automatizada de trazas, las cuales en su gran mayoría se basaban en el análisis de requerimientos para la inferencia de relaciones entre diversos artefactos del proceso de desarrollo mediante el empleo de minería de texto (*text mining*) y técnicas de recuperación de información (*information retrieval*), de técnicas de análisis de relaciones, o de esquemas de análisis de cambios.

Finalmente abordamos los enfoques de trazabilidad en el contexto de MDD. Como hemos visto, existe una amplia gama de propuestas, las cuales podemos clasificar según estén basadas en requerimientos, en modelos, o en transformaciones. De cada una de ellas hemos presentado una lista de trabajos que, sin ser exhaustiva, son representativos del enfoque y nos permiten visualizar las distintas perspectivas desde las cuales se ha encarado el tema. Del estudio comparativo, han surgido algunas cuestiones abiertas o interrogantes sin direccionamiento concreto: el menor grado de automatización de trazabilidad en etapas tempranas del ciclo de vida MDD, la necesidad de unificar metamodelos de trazabilidad, la inferencia de trazas implícitas, o el escaso o nulo desarrollo de mecanismos de evolución de trazas, entre otras. Y nos ha dejado algunas conclusiones: la importancia del desarrollo de herramientas de soporte, de la representación de la información de trazabilidad y la generación dinámica de trazas, en la obtención y aprovechamiento de los beneficios de la trazabilidad.

Capítulo 4

Soporte de trazabilidad con QVTrace

El presente trabajo tiene como eje principal la obtención de información de trazabilidad en el contexto del desarrollo de software conducido por modelos propuesto por la OMG (MDA). En particular, el enfoque plantea que es posible obtener trazas a partir de la definición de una transformación QVT de un modelo origen en uno destino.

La propuesta presentada ha sido implementada en un prototipo llamado QVTrace en forma de plugin de Eclipse, entorno de desarrollo integrado que en los últimos años se ha transformado en uno de los más utilizados para el desarrollo de herramientas conducidas por modelos.

QVTrace a sido desarrollada con la visión de ser una herramienta complementaria a otras disponibles en el marco del desarrollo conducido por modelos. El hecho de estar implementado como plugin de Eclipse, le brinda versatilidad y acentúa su interoperabilidad con el resto de los programas afines. Sus entradas son la definición de una transformación (código en lenguaje QVT) y los modelos

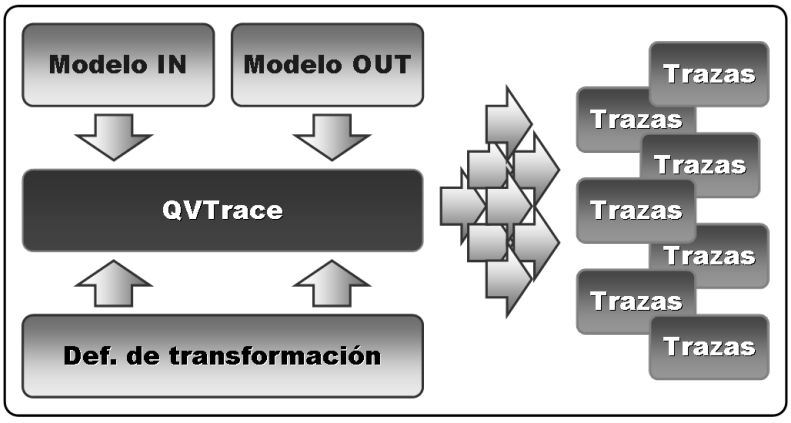


Figura 4.1: Esquema global de QVTrace

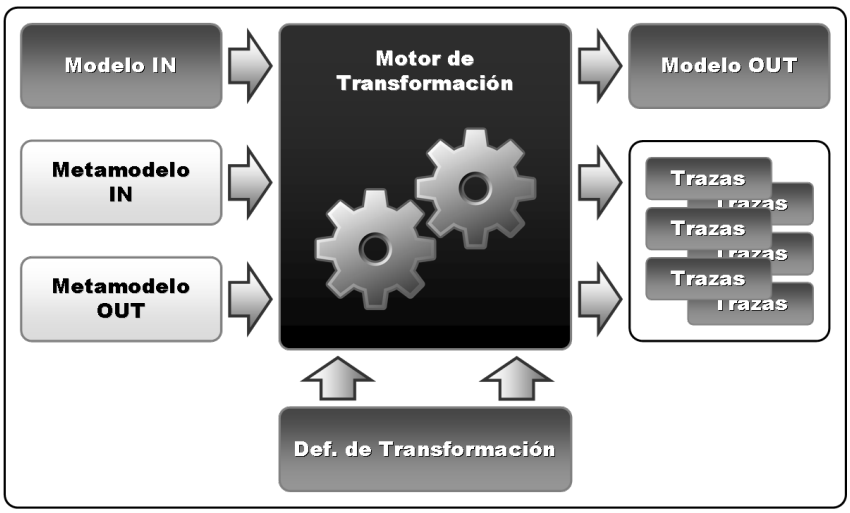


Figura 4.2: Esquemas usuales de obtención de trazas en MDD

origen y destino en formato Ecore, estándar de representación de modelos en el Eclipse Modeling Framework [27]. La salida generada es una colección de trazas (Figura 4.1) definidas en un metamodelo ad hoc.

El presente capítulo tiene como objetivo presentar los aspectos fundamentales de la propuesta. En primer lugar abordaremos el diseño general de la herramienta, sus principales componentes, y el proceso realizado para la inferencia y generación de trazas. Luego, se detallará el esquema de representación de modelos utilizado, tanto en los parámetros de entrada (modelo IN y modelo OUT), como en la representación interna de QVTrace. A continuación se presentará el esquema utilizado para la representación de las trazas, conocido como metamodelo de trazabilidad, y por último analizaremos los detalles del mecanismo de obtención de trazas desarrollado.

4.1. Obtención de información de trazabilidad

4.1.1. Esquemas tradicionales

En términos generales, la obtención de información de trazabilidad en el desarrollo conducido por modelos ha estado ligada siempre al proceso de transformación de modelos. Un enfoque totalmente lógico si consideramos a la trazabilidad como una de las características deseables del proceso de transformación de modelos.

La Figura 4.2 muestra el esquema habitual de éste tipo de implementaciones: el motor de transformación, componente encargado de llevar a cabo la transformación de un modelo en otro, es alimentado con un modelo de entrada, los respectivos metamodelos de entrada y salida, y la definición de la transformación a realizar. El resultado es el modelo de salida, resultante de la transformación del modelo de entrada, y un conjunto de trazas. En la mayoría de las implementaciones, la generación de trazas es especificada en la definición de la

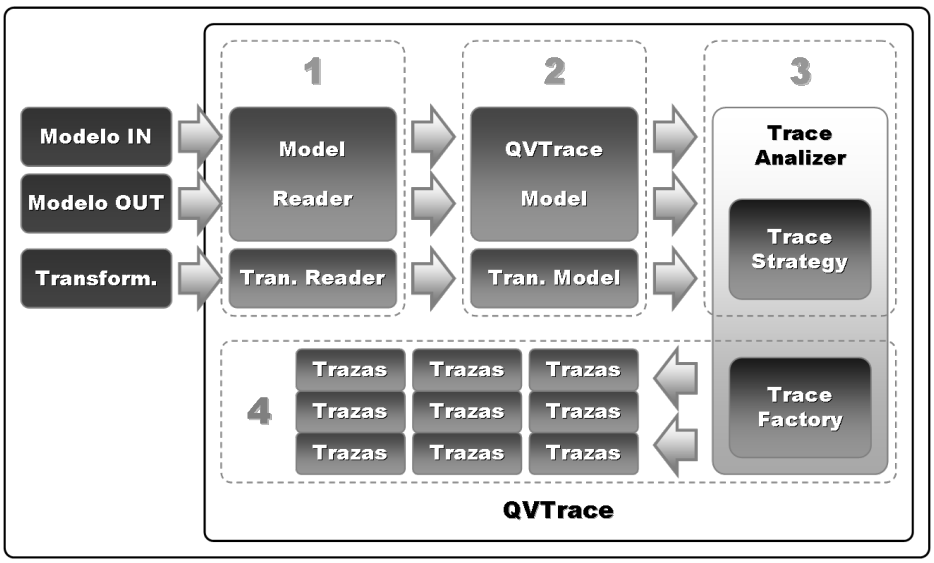


Figura 4.3: Esquema general de QVTrace

transformación, como es el caso de las propuestas de Jouault [51] y Falleri [41], que revisaremos más adelante.

En el caso de Jouault, la transformación es definida en lenguaje ATL (*Atlas Transformation Language*), e incluye un conjunto de instrucciones específicas para la generación de trazas. La propuesta de Falleri, por su parte, incluye la especificación de la transformación y generación de trazas utilizando un lenguaje de transformación de modelos llamado Kermeta.

4.1.2. La propuesta QVTrace

A diferencia de los esquemas usuales para la obtención de información de trazabilidad, este estudio sugiere que mediante el análisis de la definición de una transformación en lenguaje QVT es posible la inferencia de trazas que indiquen los mapeos de elementos del modelo origen en elementos del modelo destino, independientemente del proceso de transformación de modelos, sin necesidad de intervención alguna y de forma totalmente automática.

QVTrace requiere dos tipos de *input*: los modelos de entrada y salida, y la definición de una transformación en lenguaje QVT que detalle el mapeo de uno en el otro. El flujo de trabajo propuesto por QVTrace comienza con el procesamiento de los datos de entrada, y finaliza con la obtención de las trazas. Este proceso puede resumirse en cuatro fases:

1. Lectura y *parsing* de los modelos origen y destino, en formato Ecore, y de la especificación de la transformación QVT.

2. Creación de objetos de representación interna para los modelos de entrada y salida, y de la transformación.

3. Análisis de la definición de la transformación e inferencia de trazas según la estrategia de trazabilidad utilizada, definida en un objeto *TraceStrategy*.

4. Creación de trazas a través del objeto *TraceFactory*.

La Figura 4.3 muestra esquemáticamente los componentes de QVTrace y su interacción. La obtención de trazas es realizada por un componente llamado *TraceAnalizer*, en colaboración con dos objetos fundamentales en el proceso, y que explicaremos más adelante: uno de tipo *TraceStrategy*, el cual implementa la estrategia utilizada para la inferencia de las trazas, y otro de tipo *TraceFactory*, el cual se encarga de la creación de trazas. De esta forma, desacoplamos la creación de la traza, y por lo tanto el conocimiento del metamodelo, del proceso de inferencia de trazas.

A diferencia de otras propuestas, QVTrace no depende de la implementación del motor de transformación, ni de la ejecución de dicho proceso (transformación), del cual como vemos es totalmente independiente. Tampoco requiere directivas específicas, ni modificación alguna de la definición QVT.

A continuación analizaremos algunos aspectos relacionados con la representación de los modelos en QVTrace. En particular veremos el formato de entrada, y la representación interna utilizada, con sus respectivos fundamentos.

4.2. Representación de modelos

4.2.1. Eclipse Modeling Framework

Como hemos mencionado en el Capítulo 1, el proyecto EMF (Eclipse Modeling Framework) es un framework de modelado y generación de código fuente para construir herramientas, y otras aplicaciones, basado en modelos. Desde hace unos años, EMF se ha ganado un importante lugar en el contexto del desarrollo conducido por modelos, dado que el mismo soporta el concepto clave propuesto por el Object Management Group (OMG) a través de MDA (*Model-Driven Architecture*): el uso de modelos como datos de entrada (*input*) de herramientas de desarrollo e integración. En efecto, en el contexto del EMF un modelo es utilizado para conducir la generación de código, y su serialización para el intercambio de datos.

El framework admite la posibilidad de realizar la definición de modelos de tres maneras distintas:

1. Escribiendo la definición de los modelos en XML.

2. Utilizando diagramas UML para la especificación de los modelos.

3. Definiendo los modelos mediante interfaces de Java, utilizando un subconjunto del lenguaje conocido como Java Anotado (*Annotated Java*).

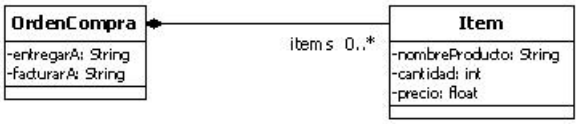
Por ejemplo, supongamos que estamos modelando una aplicación para gestionar órdenes de compra. Del análisis realizado se pudo determinar que es preciso

```
public interface OrdenCompra
{
    String getEnviarA();
    void setEnviarA(String value);
    String getFacturarA();
    void setFacturarA(String value);
    List getItems(); // List of Items

public interface Item
{
    String getNombreProducto();
    void setNombreProducto(String value);
    int getCantidad();
    void setCantidad(int valor);
    float getPrecio();
    void setPrecio(float valor);

}
```

(a) Interfaces Java



(b) Diagrama UML

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.example.com/SimplePO"
xmlns:PO="http://www.example.com/SimplePO">
    <xsd:complexType name="OrdenCompra">
        <xsd:sequence>
            <xsd:element name="enviarA" type="xsd:string"/>
            <xsd:element name="facturarA" type="xsd:string"/>
            <xsd:element name="items" type="PO:Item" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Item">
        <xsd:sequence>
            <xsd:element name="nombreProducto" type="xsd:string"/>
            <xsd:element name="cantidad" type="xsd:int"/>
            <xsd:element name="precio" type="xsd:float"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

(c) Esquema XML

Figura 4.4: Diversas representaciones de modelos en EMF

mantener de cada orden de compra una dirección de entrega, una dirección de facturación, y una colección de ítems, de los cuales se desea conocer el nombre del producto vendido, la cantidad y el precio. En el contexto del EMF, podemos modelar esta situación mediante interfaces Java (Figura 4.4a), utilizando UML (Figura 4.4b) o a través de un esquema XML (Figura 4.4c). En cualquier caso, tendremos tres representaciones de un mismo concepto: el "modelo de datos" de la aplicación de órdenes de compra. Independientemente de la manera elegida para definir los modelos, el modelo EMF es la representación común de alto nivel que combina las restantes.

4.2.2. El (meta) modelo Ecore

El modelo utilizado para representar modelos en EMF es llamado Ecore. Ecore es en sí mismo un modelo EMF, y por lo tanto, su propio metamodelo. En la Figura 4.5a se muestra un subconjunto simplificado del modelo Ecore. Si bien este modelo es más complejo, se ha evitado mostrar algunas clases base, para una mejor comprensión, sin que esto afecte su caracterización. En este caso, nos concentraremos en las cuatro clases Ecore necesarias para representar el modelo de las órdenes de compra:

1. EClass: Utilizada para representar una clase modelada. Tiene un nombre, cero o más atributos (colección *eAttributes*), y cero o más referencias (colección *eReferences*) a otras entidades EClass. Y posee una referencia a sus superclases (*eSuperTypes*), pudiendo ser éstas cero o más.

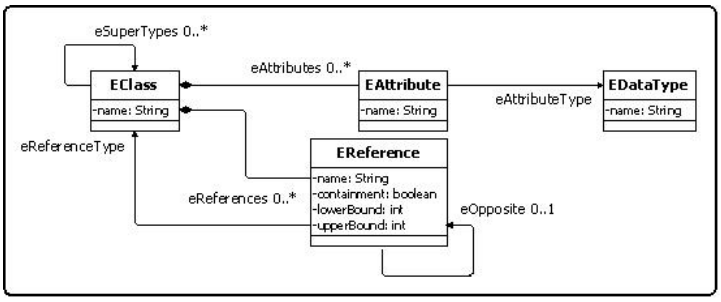
2. EAttribute: Es utilizada para representar los atributos de una clase. Los atributos tienen un nombre (*name*) y un tipo (*eAttributeType*).

3. EReference: Se utiliza para representar un extremo de una asociación entre clases. Tiene un nombre, un campo *Boolean* que identifica si la relación se trata de una composición (atributo *containment*), un tipo de referencia, que es otra clase (asociación *eReferenceType*) y una cardinalidad (atributos *lowerBound* y *upperBound*, donde el valor -1 en éste último corresponde a un valor genérico n, variable). Alternativamente, puede contener un enlace a su referencia opuesta (asociación *eOpposite*).

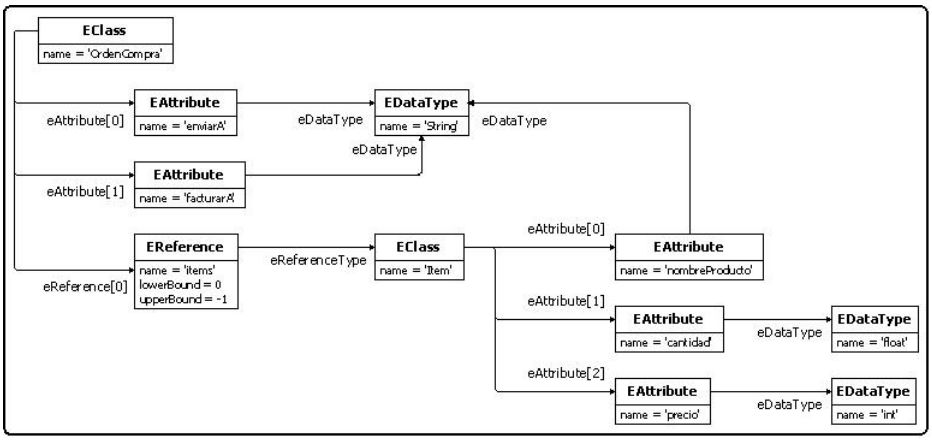
4. EDataType: Usada para representar el tipo de un atributo. Un tipo de dato puede ser primitivo como *int* o *float*, o puede ser un objeto como *java.util.Date*.

La Figura 4.5b muestra la representación de nuestro modelo de ejemplo de órdenes de compra en Ecore. El modelo Ecore permite la representación de cualquier modelo, y puede ser generado a partir de código XML, de diagramas UML o de interfaces de Java (Figura 4.6).

Como se habrá podido observar, los nombres de las clases Ecore se encuentran muy relacionadas a la terminología del UML (Unified Modeling Language). Podríamos acaso preguntarnos ¿Por qué no es UML el "modelo EMF"? ¿Por qué necesita ser EMF su propio modelo? La respuesta es sencilla: Ecore es un subconjunto pequeño del lenguaje UML completo. De hecho, UML soporta un modelado mucho más ambicioso que el soporte básico de EMF. UML, por ejemplo,



(a) El modelo Ecore simplificado



(b) Modelo de órdenes de compra representado en Ecore

Figura 4.5: El modelo Ecore

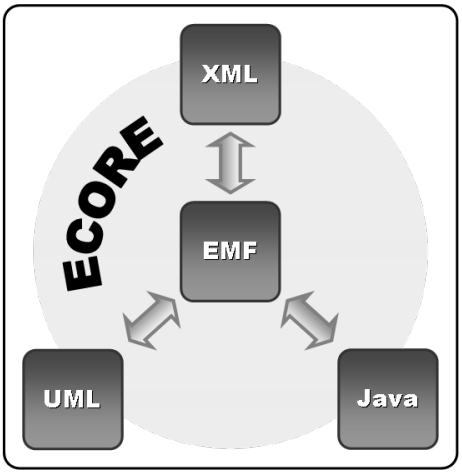


Figura 4.6: EMF unifica representaciones en Java, XML y UML mediante Ecore

permite modelar el comportamiento de una aplicación, y no sólo su estructura de clases.

4.2.3. Ecore versus MOF

En términos concretos, el estándar *Meta-Object Facility* (MOF) [15] define un subconjunto de UML que permite la descripción de conceptos de modelado de clases dentro de un repositorio de objetos. Ecore implementa una parte importante de MOF, aunque está enfocado en la integración de herramientas antes que en la gestión de un repositorio de metadatos. Debido a esto, Ecore evita alguna de las complejidades de MOF, resultando en una implementación optimizada y ampliamente aplicable. MOF e Ecore tienen muchas similaridades, en particular en su habilidad para especificar clases, y sus características estructurales y de comportamiento, herencia, paquetes y reflexión. Ambos difieren en el área de ciclo de vida, en las estructuras de los tipos de dato, en la relación entre los paquetes y en ciertos aspectos complejos de las asociaciones. MOF fue estandarizado en 1997, al igual que UML.

4.2.4. Representación de modelos en QVTrace

Una de las decisiones de diseño más importantes de QVTrace fue la representación de modelos a utilizar. En primer lugar, la representación de los modelos origen y destino de la transformación a analizar, es decir, las variables de entrada del proceso. Posteriormente, el segundo interrogante fue la representación de modelos interna de la herramienta, aspecto fundamental dado que es una de las bases del mecanismo de obtención de trazas.

En procura de incrementar la interoperabilidad de QVTrace con otras herramientas, se decidió que el formato de modelos de entrada y salida, que actúan de *input* de QVTrace, fuera Ecore. La experiencia de EMF, los aportes del proyecto en el contexto del desarrollo conducido por modelos, y el entorno de desarrollo para el cual fue pensada la herramienta QVTrace, son los motivos principales por los cuales fue elegida esta representación.

Para la representación interna de modelos, la decisión fue distinta. En este caso se confeccionó un metamodelo basado en MOF, simplificado, que pudiera ser mapeado con Ecore de manera sencilla. El eje de esta decisión fue la necesidad de brindar a la representación una mayor generalidad, permitiendo la máxima expresividad posible sin estar restringido a un formato particular. Al ser MOF un metamodelo estándar, sería posible que QVTrace en el futuro pudiera trabajar con otra representación de modelos basada en dicho esquema, con un impacto mínimo sobre el diseño de la herramienta. Por otro lado, si bien el modelo de representación interna utilizado es similar a Ecore en el aspecto estructural, el mismo fue dotado en la implementación de un comportamiento[1] específico, requerido por el *TraceAnalizer* para la inferencia de trazas, el cual no es provisto por Ecore.

La Figura 4.7 muestra el metamodelo de representación desarrollado para QVTrace. Al igual que sucede con Ecore, los nombres de las clases que componen

[1] Conjunto de métodos.

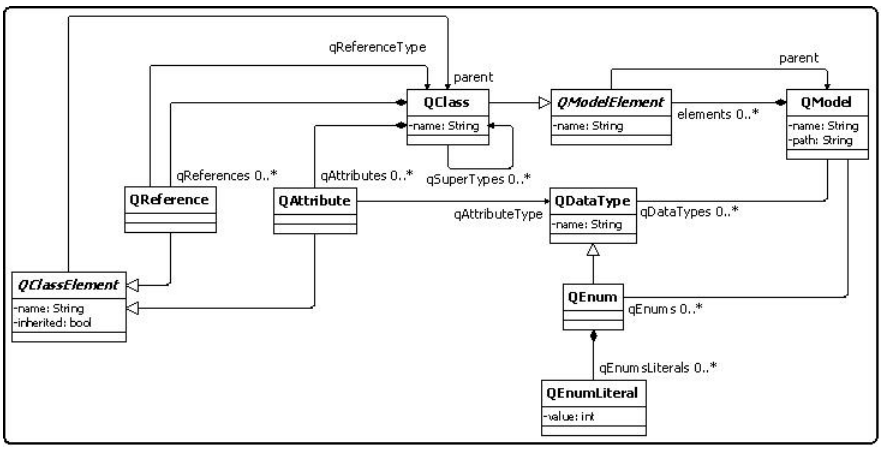


Figura 4.7: Representación de modelos en QVTrace

el metamodelo han sido denominadas con el nombre que poseen en MOF, con el agregado del prefijo "Q", letra que representa en este caso a nuestra propuesta. De esta manera, el componente *Class* (MOF) es denominado *QClass* (QVTrace *Class*). Como es posible observar, el metamodelo es similar a MOF. Las clases principales son:

- QCLASS: Es la representación de una clase dentro del modelo.
- QATTRIBUTE: Es el componente "atributo" de una *QClass*, el cual tiene un tipo asociado *QDataType*.
- QREFERENCE: Es el componente "referencia" de una *QClass*. Mantiene un enlace a la *QClass* referenciada.
- QDATATYPE: Representa el tipo de datos que un atributo *QAttribute* puede asumir.
- QENUM: Es un tipo de dato *QDataType* especial que permite el modelado de enumeraciones. Se compone de elementos simples de tipo *QEnumLiteral*.
- QENUMLITERAL: Representa los elementos literales que conforman una enumeración.
- QCLASSELEMENT: Es una clase abstracta que representa los elementos que pueden conformar una *QClass*.
- QMODEL: Representa un modelo interno QVTrace.
- QMODELELEMENT: Clase abstracta que define el tipo de elemento que un *QModel* puede tener.

Como se dijo anteriormente, el modelo de representación interna diseñado para QVTrace está basado en una versión simplificada del estándar MOF. En este caso, la propuesta está centrada en capturar las clases fundamentales para la

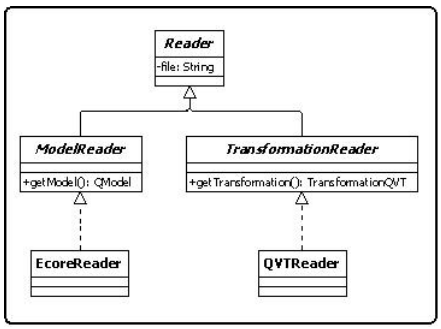


Figura 4.8: Jerarquía de *Readers* en QVTrace

definición estructural de modelos, dejando de lado el aspecto de comportamiento (*behavioral features*), el cual también es abarcado por MOF.

La transformación de los modelos de entrada y salida (de la transformación) en formato Ecore al formato interno de QVTrace basado en MOF es realizada por un componente llamado *ModelReader* (ver #1 en el esquema de la Figura 4.3). QVTrace implementa una jerarquía de lectores o *readers* basada en dos tipos: lectores de modelos (*ModelReader*) y lectores de transformaciones (*TransformationReader*). Como muestra la Figura 4.8, una clase concreta *EcoreReader* brinda una implementación para el método *getModel()* devolviendo un objeto QModel, la representación interna de un modelo en QVTrace. Para el caso de las transformaciones, una clase *QVTReader* implementa la interfase *TransformationReader* permitiendo la lectura y *parsing* de una transformación QVT.

La ventaja más importante de este diseño es la versatilidad que ofrece: a futuro sería posible modificar o ampliar los formatos de modelo aceptados por QVTrace con sólo agregar clases que implementen la interfase *ModelReader*. De la misma forma, sería sencillo modificar o extender el lector/*parser* de QVT en caso de ser necesario.

4.3. Metamodelo de trazabilidad

4.3.1. Definición del metamodelo

Para el modelado de trazas se desarrolló un metamodelo basado en la simplicidad, ajustado a las necesidades del problema abordado con la implementación de QVTrace (Figura 4.9). El mismo consta de una clase *Trace* que mantiene la información asociada a la traza, a saber:

- El nombre de la traza, dado por los nombres de los elementos que la componen, donde *nombreTraza* = *elementoOrigen2elementoDestino*.
- Los elementos origen y destino de la traza (referencias *source* y *target* en la figura).

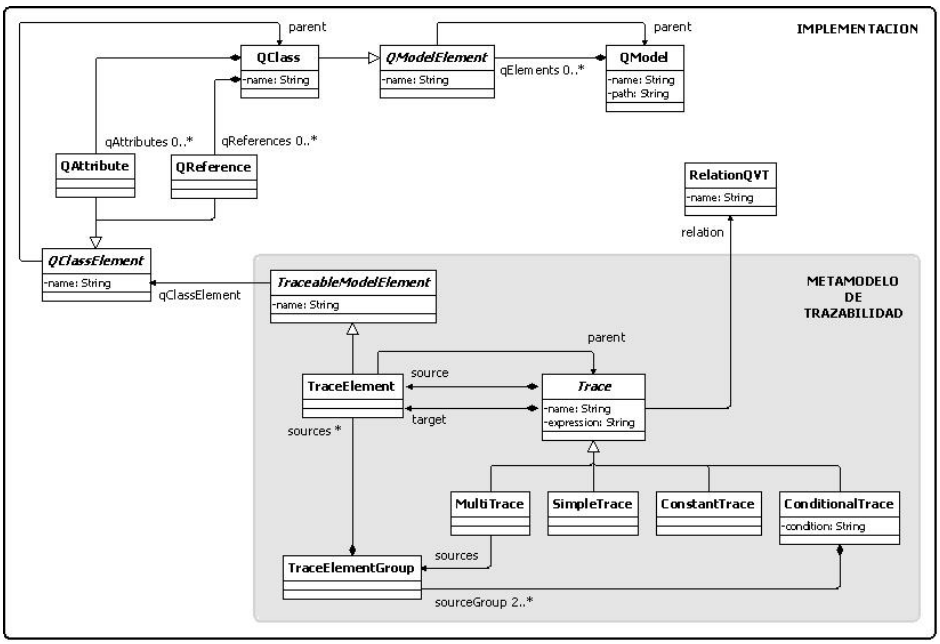


Figura 4.9: Metamodelo de trazabilidad implementado por QVTrace

- La expresión de la traza (atributo *expression*), una cadena de caracteres que representa la relación de trazabilidad entre los elementos origen y destino de la traza.
- La relación que la originó (referencia *relation*).

Los elementos relacionados por la traza son de tipo *TraceElement*, los cuales a su vez son subtipos de *TraceableModelElement*, una clase abstracta que determina qué tipo de elemento del modelo puede ser incluidos en una traza. En este caso, se decidió que los elementos "traceables" fueran de tipo *QClassElement*, es decir, aquellos elementos que componen una *QClass*. Dicha decisión tuvo como objetivo principal el dotar a las trazas con la mayor granularidad posible, es decir, permitir que se pudieran establecer las relaciones de trazabilidad entre la mínima entidad referenciable de un modelo determinado, los cuales son en este caso los elementos (atributos y referencias) que componen las clases.

El metamodelo define cuatro subtipos de traza, para contemplar diversas situaciones relacionadas con el lenguaje y con el mecanismo de inferencia de trazas. El primer tipo de traza, llamada en el modelo *MultiTrace*, representa aquellas relaciones donde múltiples elementos origen dan lugar a un elemento del modelo destino. El segundo tipo, llamadas trazas condicionales o *ConditionalTrace*, representan trazas potenciales que debido al mecanismo de inferencia de trazas utilizado no pueden ser confirmadas como tal por estar asociadas a bloques de código QVT condicional. Como explicaremos en detalle en el Capítulo 5, estas trazas tienen la forma $A \mid B \rightarrow c$, donde A y B son conjuntos disjuntos de elementos del modelo origen y c es un elemento del modelo destino. El tercer tipo de traza son las trazas constantes o *ConstantTrace*, las cuales modelan aquellos

casos donde la transformación especifica un valor constante para uno de los elementos del modelo destino, lo cual nos permite inferir con anticipación el valor de dicho elemento luego de la ejecución de la transformación. Por último, las trazas *SimpleTrace* son las convencionales, las cuales representan las relaciones 1-1 entre un elemento del modelo origen y uno del modelo destino.

4.3.2. Consecuencias

Uno de los puntos claves del diseño de un metamodelo de trazabilidad es la definición del tipo o tipos de elemento que una traza puede relacionar. En nuestro caso, buscando la mayor granularidad posible, se determinó que los elementos rastreables fueran de tipo *QClassElement*. En particular, QVTrace ha implementado sólo dos tipos de elementos de clase: *QAttribute* y *QReference*. Por tal motivo, bajo este diseño no serán posibles trazas de tipo $QClass \rightarrow QAttribute$, $QReference \rightarrow QClass$ o $QClass \rightarrow QClass$. Estas trazas, que relacionan elementos de distinto nivel de abstracción, no son a nuestro criterio lo suficientemente descriptivas como para resultar de utilidad. En particular, si una clase, como conjunto de elementos del modelo origen, determina el valor de un elemento del modelo destino, estamos en presencia de una multitraza. En lugar de modelar la traza como $A \rightarrow b$, donde A es una *QClass* y b es un elemento del modelo destino, es preferible optar por una traza $a_1, a_2, \ldots, a_n \rightarrow b$, donde $a_1, a_2, \ldots, a_n$ son elementos de la clase A.

La segunda observación está relacionada con la cardinalidad de las trazas, es decir, el número de integrantes origen y destino que éstas pueden tener. En el diseño planteado, no es posible una relación de uno o muchos elementos del modelo origen con múltiples elementos del modelo destino. Esto no responde a una restricción del lenguaje, sino a una decisión totalmente relacionada con el diseño, donde se prefirió individualizar las trazas independientemente que las mismas compartan, o no, un mismo (elemento) origen. De esta manera, cada traza de tipo $a \rightarrow b, c$ o $a, b \rightarrow c, d$ dará lugar dos trazas $a \rightarrow b$ y $a \rightarrow c$ o $a, b \rightarrow c$ y $a, b \rightarrow d$ respectivamente.

4.3.3. Diferencias con metamodelos revisados

A diferencia de los metamodelos propuestos en trabajos de similar índole, el desarrollo presentado aquí modela las trazas como una relación unívoca entre un elemento del modelo origen y uno del modelo destino, mientras que en la mayoría de los casos esta relación se generaliza como *muchos-a-muchos*. La propuesta sí contempla una posible traza entre n elementos del modelo origen y uno del modelo destino, la cual es tipificada en la clase *MultiTrace*, subtipo de clase *Trace*. Este enfoque, que podría ser considerado un limitación desde el diseño, responde en realidad a una virtud. El algoritmo de inferencia de trazas trabaja a nivel del mínimo elemento rastreable en el contexto de la representación de modelos elegida, y por las características del análisis basado en variables, utilizado en este caso para la inferencia de trazas, si uno o más elementos del modelo origen generan múltiples elementos del modelo destino, entonces se generaran múltiples trazas *SimpleTrace* o *MultiTrace*, según el caso.

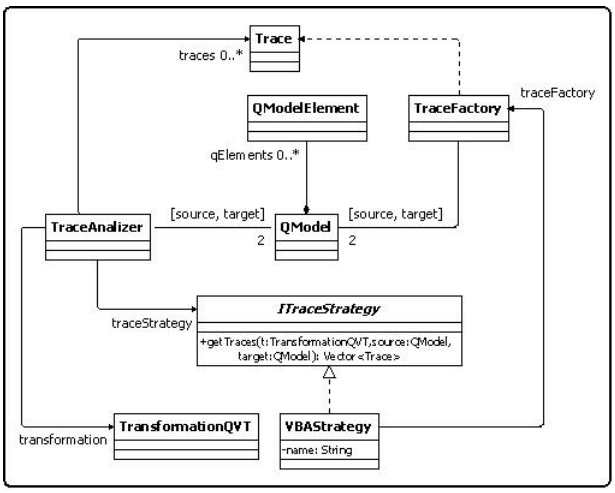


Figura 4.10: Soporte de trazas en QVTrace

La segunda diferencia con la mayoría de los metamodelos propuestos tiene que ver con la semántica de la traza. Uno de los atributos de la clase *Trace*, denominado *expression*, almacena la expresión que revela el significado de la transformación, es decir, de qué manera un elemento del modelo origen se transforma en un determinado elemento del modelo destino. Por ejemplo, asumiendo una transformación $A2B$ donde un elemento x del modelo A se convierte en un elemento y del modelo B, tendremos una traza $x \rightarrow y$ donde el atributo *expression* contendrá $y = x$, agregando significado a la relación.

El concepto de traza condicional no está presente en ninguno de los trabajos relacionados. Como hemos dicho y detallaremos a continuación, su presencia está relacionada con el mecanismo de inferencia de trazas utilizado. El Capítulo 7 presenta un estudio comparativo entre varios esquemas de gestión de trazabilidad en MDD, ahondando en las similitudes y diferencias entre cada uno.

4.4. Soporte para la inferencia de trazas

El mecanismo de inferencia de trazas propuesto es soportado a través del esquema que se observa en la Figura 4.10. El mismo consta de un componente llamado *TraceAnalizer*, el cual utilizando los modelos origen y destino (objeto *QModel*), la definición de la transformación (objeto *TransformationQVT*), y una estrategia de rastreabilidad (componentes que implementan la interfase *ITraceStrategy*) genera las trazas correspondientes.

La estrategia de trazabilidad es en esencia el mecanismo mediante el cual se obtienen las trazas. El diseño está pensado para que este componente pueda ser fácilmente extendido o reemplazado por otro que implemente el método que define la interfase mencionada, el cual como puede verse en la signatura recibe una transformación QVT y un par de modelos origen y destino, y devuelve como resultado una colección de trazas de tipo *Trace*. La responsabilidad de la creación de trazas está a cargo del objeto *TraceFactory*, el cual es el encargado de la

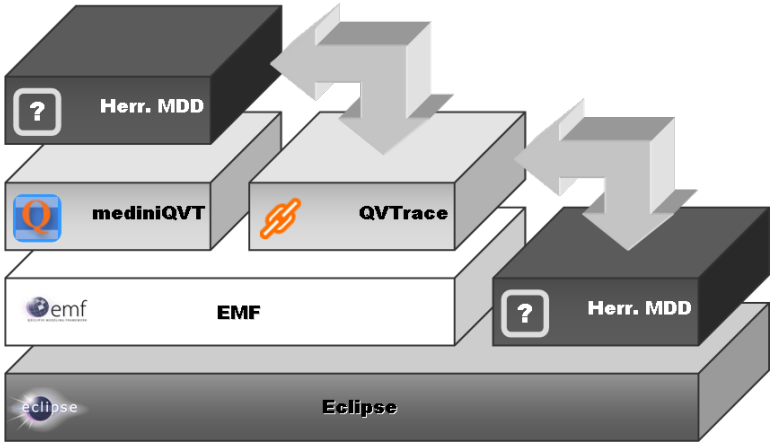


Figura 4.11: Organización general de QVTrace

generación de objetos *Trace* (ver flecha con línea punteada en el diagrama). Toda estrategia de trazabilidad implementada debe recurrir a este objeto factoría para la creación de las mismas.

El objetivo principal del diseño ha sido desacoplar el proceso de inferencia o descubrimiento de trazas, de la generación de las mismas. El primero, está relacionado con el conocimiento de los modelos y de la estructura de las transformaciones analizadas, mientras que la creación de trazas se encuentra fuertemente ligada al conocimiento del metamodelo de trazabilidad utilizado. El diseño, así planteado, permite minimizar el impacto de posibles cambios de un componente en el otro.

4.5. Implementación de QVTrace

4.5.1. Organización general de la herramienta

Como mencionamos al principio del capítulo, QVTrace ha sido implementado en forma de plugin de Eclipse. Como vemos en la Figura 4.11, utiliza a su vez los servicios de EMF (*Eclipse Modeling Framework*) como soporte para la representación de modelos en general. De la misma forma han sido desarrolladas otras herramientas MDD, como por ejemplo mediniQVT [9], la cual provee una implementación del lenguaje QVT Relations. Eventualmente es posible que QVTrace interactúe con herramientas de este tipo, u otras que no necesariamente se encuentren montadas sobre EMF. Una de las grandes ventajas del uso de Eclipse ha sido su capacidad de incorporar desarrollos de terceras partes en forma de *plugins*, unificando muchas herramientas bajo una misma plataforma, ampliando enormemente su funcionalidad.

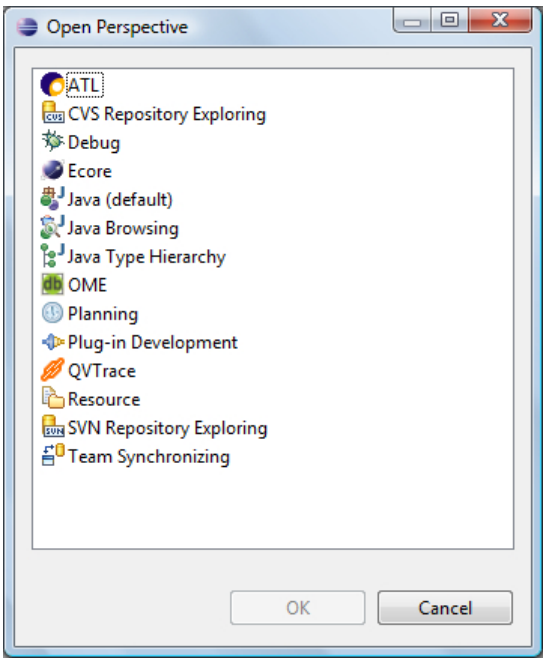


Figura 4.12: Perspectivas de Eclipse

4.5.2. Trabajando con QVTrace

QVTrace está implementado en Eclipse en forma de perspectiva (Figura 4.12). Una perspectiva es una forma de agrupar vistas y comandos Eclipse para una tarea particular [28]. El flujo de trabajo propuesto por QVTrace utiliza cuatro vistas:

- Selector de modelos o *Model Browser*: permite la selección de los modelos origen y destino, en formato Ecore, y de la definición de la transformación en lenguaje QVT.
- Vista de modelos o *Model View*: presenta gráficamente los modelos origen y destino de la transformación en forma de árbol.
- Vista de trazas o *Trace View*: muestra las trazas resultantes del proceso de análisis de código QVT.
- Vista de propiedades (*Properties*): lista las propiedades los objetos según la selección realizada. Es una vista propia de Eclipse, no de QVTrace.

La Figura 4.13 muestra las vistas de QVTrace, agrupadas por la definición de la perspectiva. Esta, actúa como una macro que abre las vistas asociadas y las ubica en una determinada posición de la pantalla.

QVTrace requiere la especificación de tres parámetros de entrada: los modelos de entrada y salida, en formato Ecore, y la definición de una transformación QVT que los relacione. Esta selección es realizada desde la vista *Model Browser*

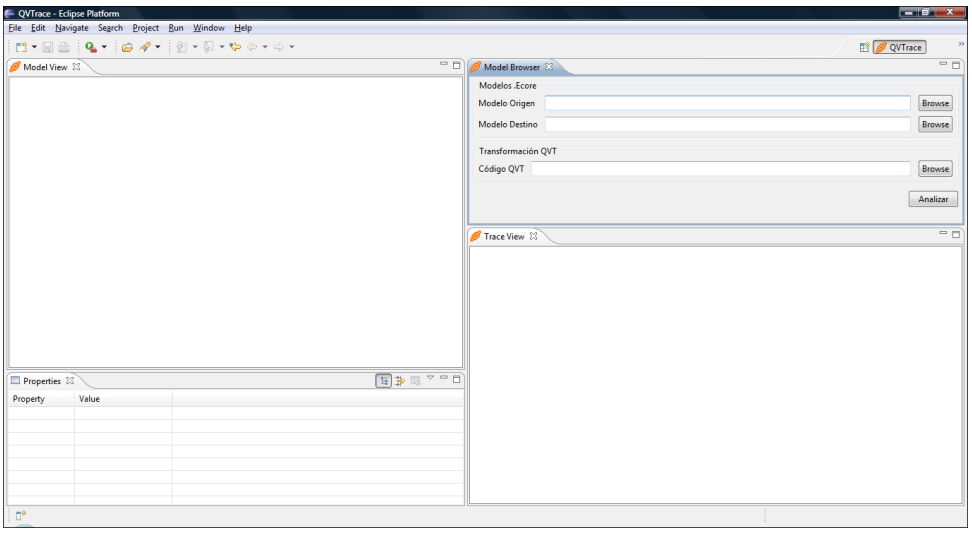


Figura 4.13: Panel principal de QVTrace

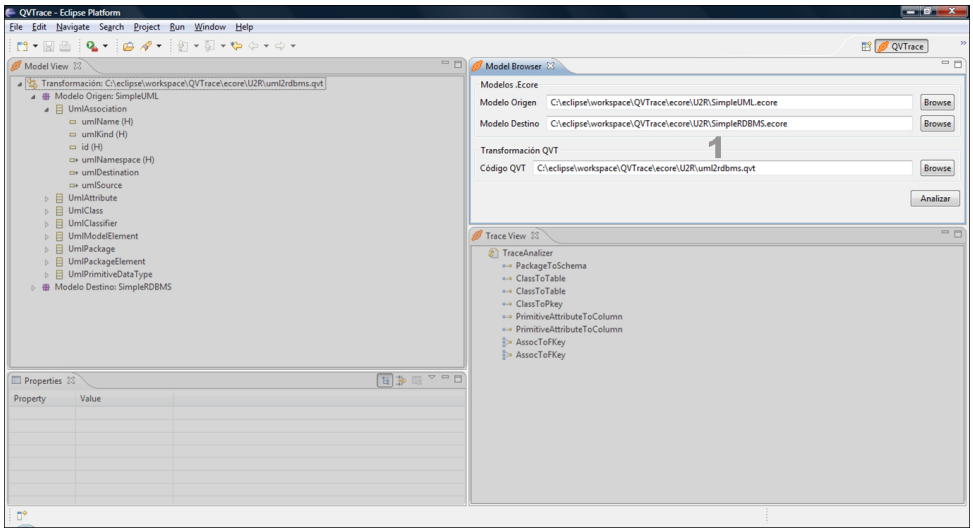
(1), la cual provee además un botón con el texto "Analizar" el cual luego de ser presionado invoca al *TraceAnalizer* para que realice la determinación de las trazas correspondientes (Figura 4.14a).

A continuación, el sistema muestra las resultados del análisis en las vistas *Model View* (2) y *Trace View* (3). La vista de modelos permite visualizar los modelos Ecore analizados durante el proceso de inferencia de trazas. Las trazas inferidas son mostradas en la vista de trazas (Figura 4.14b).

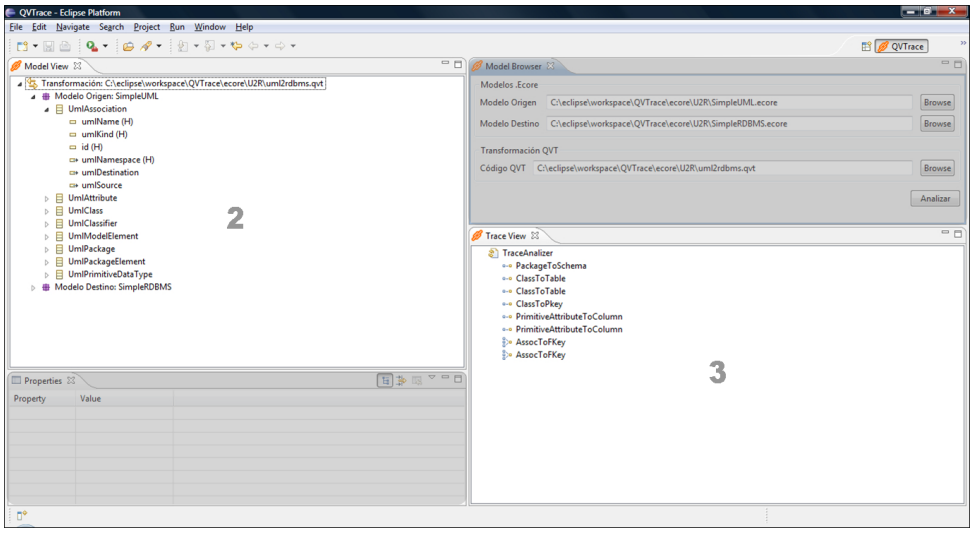
Por último, QVTrace permite la inspección de las propiedades de los distintos objetos tanto de la vista de trazas como los de la vista de modelos. La Figura 4.15a muestra la inspección de las propiedades de una de las trazas inferidas tras el análisis, llamada *PackageToSchema* (3), la cual al ser seleccionada muestra sus principales características en la vista *Properties* (4), en este caso la expresión de la traza derivada y la relación donde fue hallada. La Figura 4.15b muestra el mismo efecto pero sobre un elemento del modelo origen de la transformación, en este caso un atributo de la entidad *UmlAssociation* llamado *umlName*, del cual puede observarse en la vista *Properties* su nombre, tipo y si es heredado de una superclase o es propio de la clase.

4.6. Resumen

A lo largo del presente capítulo hemos detallado nuestra propuesta para la obtención de información de trazabilidad en el marco de la MDA, implementada en un prototipo llamado QVTrace. Respecto de dicha propuesta, hemos visto el esquema general de obtención de trazas en contraste con las soluciones tradicionales, las distintas representaciones de modelos utilizadas con sus fundamentos, el metamodelo de trazabilidad diseñado y los detalles del soporte para trazabilidad, utilizado por QVTrace.

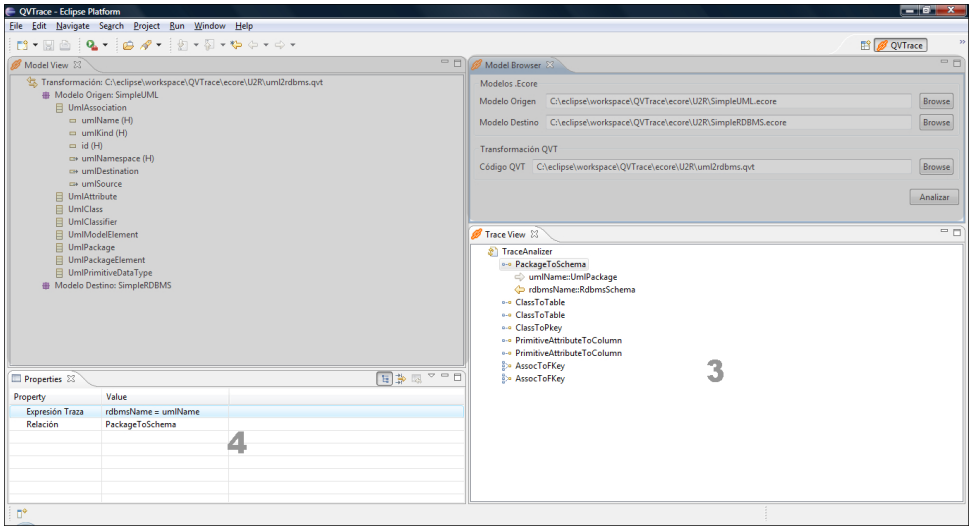


(a) Especificación de parámetros de entrada

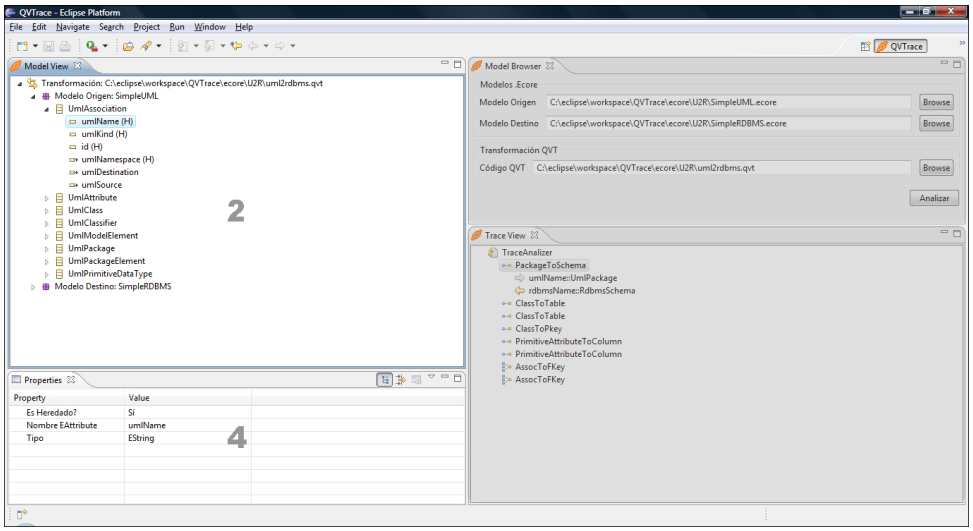


(b) Visualización de resultados

Figura 4.14: Trabajando con QVTrace



(a) Visualización de propiedades de las trazas



(b) Visualización de propiedades de los elementos de modelo

Figura 4.15: Utilización de la vista *Properties* en QVTrace

Respecto de QVTrace hemos detallado su arquitectura de alto nivel, caracterizando sus principales componentes y explicando la interacción entre éstos. En cuanto a la representación de modelos, hemos explicado los conceptos básicos detrás de Ecore, su importancia y algunas de sus semejanzas y diferencias con MOF. Además, mostramos el esquema de representación de modelos interno de QVTrace y sus principales características. Posteriormente, revisamos el metamodelo de trazabilidad propuesto para esta ocasión, y subrayamos algunas de la diferencias con metamodelos propuestos por otros autores. Estas diferencias serán abordadas con mayor profundidad en el Capítulo 7.

A continuación, se analizó el soporte para la inferencia de trazas. Se mostró su estructura general y se detalló la interacción entre los componentes involucrados. Por último, se presentó la estructura general de QVTrace en relación a Eclipse, plataforma sobre la cual se encuentra montado el sistema y se explicó su funcionamiento, ilustrado mediante las capturas de pantalla de la aplicación en ejecución.

En el Capítulo 5 presentaremos una estrategia de trazabilidad (*Trace Strategy*) propia llamada Análisis Basado en Variables, la cual está implementada en QVTrace y permite la inferencia de trazas a partir de la definición de una transformación de modelos escrita en lenguaje QVT.

Capítulo 5

Análisis basado en variables

El presente trabajo aborda la problemática de la obtención de información de trazabilidad de manera automática, es decir, sin tener que depender de la participación de una persona que especifique de qué manera son generados los elementos de un modelo destino a partir de un modelo origen, ni de la ejecución de una transformación.

A diferencia de otras propuestas de similar índole, como las presentadas en la Sección 3.4.3, el presente estudio plantea que dadas las características sintáctico-gramaticales del lenguaje de especificación de transformaciones estándar del OMG, Query/View/Transformation (QVT), es posible inferir cierto tipo de trazas mediante el análisis del código fuente QVT o definición de una transformación. Este análisis consiste en el reconocimiento de ciertas construcciones o estructuras dentro de la especificación de dicha transformación en el lenguaje QVT Relations que pueden ser traducidas como trazas, y permiten explicar luego el origen de algunos elementos del modelo destino. Se han determinado seis estructuras o patrones dentro del código fuente QVT que permiten inferir o reconocer trazas y relaciones de trazabilidad.

A lo largo de este capítulo se presentarán dichas construcciones del lenguaje, que son las que posibilitan la inferencia de trazas, y el mecanismo de determinación de las mismas, el cual hemos denominado "Análisis Basado en Variables". Esta técnica ha sido implementada dentro de la arquitectura de QVTrace como una estrategia de trazabilidad o *TraceStrategy*, la cual es utilizada por el componente *TraceAnalizer* para determinar las trazas asociadas a una transformación, como hemos explicado en el Capítulo 4.

Previamente al desarrollo del análisis basado en variables o VBA (*Variable-Based Analysis*, como lo hemos denominado en idioma inglés), revisaremos algunos conceptos básicos del lenguaje QVT, que permitirán una mejor comprensión del esquema sugerido en este trabajo.

5.1. El lenguaje QVT

Para entender la propuesta es preciso repasar algunos conceptos centrales del lenguaje en el cual se encuentran escritas las especificaciones de las transforma-

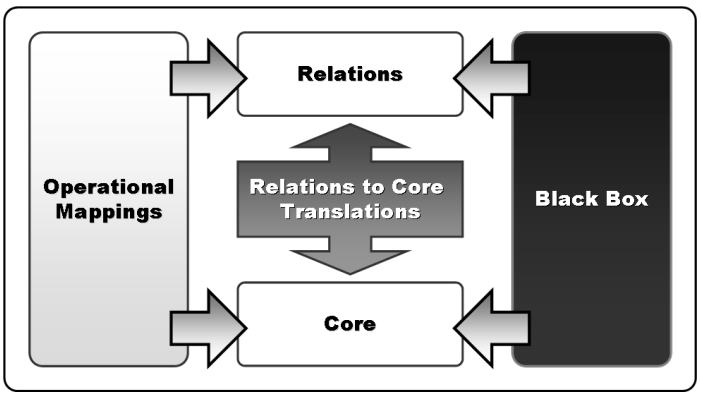


Figura 5.1: Arquitectura del lenguaje QVT

ciones de modelos analizadas. A continuación realizaremos un breve repaso de la historia del lenguaje y sus generalidades.

Como hemos mencionado, en un esfuerzo por estandarizar gran parte de los componentes del framework Model-Driven Architecture (MDA), el Object Management Group (OMG) ha liberado el estándar Meta Object Facility (MOF) para la gestión y representación de modelos y posteriormente liberó el estándar Query/View/Transformation (QVT) como lenguaje de transformación de modelos, haciendo teóricamente posible cualquier intercambio de especificaciones de transformaciones de modelos entre diferentes herramientas MDA.

Como su nombre indica, QVT permite definir consultas, vistas y transformaciones sobre modelos. Las consultas son utilizadas para buscar elementos dentro de un modelo, mientras que las transformaciones son mapeos de elementos de un modelo origen en elementos de un modelo destino, los cuales aseguran el cumplimiento de una determinada relación. La Figura 5.1 describe la arquitectura del lenguaje.

Declarativamente, una transformación puede ser expresada en el lenguaje Relations o Core, los cuales engloban la misma semántica a dos niveles de abstracción diferentes. Imperativamente, el lenguaje Operational Mappings puede ser utilizado para especificar completamente una transformación o complementar la especificación Relations o Core. Una implementación Black Box de una operación MOF con la misma signatura de una relación puede ser utilizado para complementar estos últimos.

Esta arquitectura es análoga al lenguaje Java, donde el lenguaje Core representa Bytecode, y la semántica es la especificación del comportamiento de la Máquina Virtual Java (JVM). El lenguaje Relations juega el rol del lenguaje Java, y la traducción Relations-to-Core es como especificar un compilador Java que produce Bytecode. La habilidad de invocar mapeos operacionales (*operational mappings*) y especificaciones de caja negra (*black-box*) es equivalente al llamado de la Interface Nativa Java (JNI).

5.1.1. Escenarios de ejecución

La semántica del lenguaje Relations, y por lo tanto la del lenguaje Core, permite los siguientes escenarios de ejecución:

- Transformaciones de verificación o *check-only*, para validar si los modelos están relacionados de la manera especificada.
- Transformaciones de dirección única (*single direction*).
- Transformaciones bidireccionales.
- Actualizaciones incrementales (en algún sentido) cuando uno de los modelos relacionados es modificado luego de la ejecución inicial.
- La habilidad de crear y eliminar objetos y valores, tanto como la de especificar qué objetos y valores no deben ser modificados.

Tanto los mapeos operacionales como las aproximaciones de caja negra, cuando son ejecutadas en tándem con relaciones, restringen estos escenarios haciendo sólo factibles las transformaciones de dirección única. Las transformaciones bidireccionales sólo son posibles si una implementación operacional inversa es provista de manera separada. Sin embargo, todas las otras capacidades definidas anteriormente están disponibles con ejecuciones imperativas e híbridas.

5.1.2. QVT Relations

5.1.2.1. Transformaciones y tipos de modelo

En el lenguaje Relations, una transformación entre modelos candidatos se especifica como un conjunto de relaciones que deben sostenerse para que dicha transformación sea exitosa. Un modelo candidato es un modelo que conforma un tipo de modelo, el cual a su vez es una especificación que indica qué clase de elementos de modelo puede tener un modelo que lo conforme, de forma similar como un tipo de variable especifica en un programa el conjunto de valores que dicha variable puede adoptar. Los modelos candidatos poseen un nombre, y los tipos de elemento que éstos pueden contener están restringidos a aquellos dentro del conjunto de paquetes referenciados. Veamos por ejemplo la declaración de la transformación presentada a continuación.

```
transformation umlRdbms (uml:SimpleUML, rdbms:SimpleRDBMS)
```

En esta declaración se define una transformación llamada *umlRdbms*, entre dos modelos candidatos *uml* y *rdbms*. El modelo *uml* declara al paquete SimpleUML como su metamodelo, mientras que el modelo *rdbms* declara al paquete SimpleRDBMS como su metamodelo. Una transformación puede ser invocada para verificar la consistencia entre dos modelos, o para modificar uno de los modelos y forzar la consistencia entre ambos.

Una transformación invocada para forzar consistencia (*enforcement*) se ejecuta siempre en un sentido particular, seleccionando uno de los modelos como destino

o *target*. Este modelo destino puede estar vacío, o contener elementos a ser relacionados por la transformación. La ejecución de la transformación procede primero verificando si la relación se sostiene, y para aquellas en las cuales el chequeo falla intenta hacer que la relación se sostenga creando, eliminando o modificando el modelo destino, forzando así la relación.

5.1.2.2. Relaciones y dominios

Las relaciones en una transformación declaran restricciones que deben ser satisfechas por los elementos de los modelos candidatos. Cada relación, definida por dos o más dominios y un par de predicados *when* y *where*, especifica qué vínculos deben ser mantenidos entre los elementos de los modelos candidatos, es decir, entre los elementos del modelo origen (*source*) y el modelo destino (*target*).

Un dominio es un tipo particular de variable tipeada (*typed variable*) que puede ser evaluada en un modelo de un determinado tipo. A su vez, todo dominio posee un patrón, que puede ser visto como un grafo de nodos "objeto", con sus propiedades y sus asociaciones, originados a partir de una instancia del tipo de dominio.

5.1.2.3. Cláusulas *when* y *where*

Las relaciones pueden ser limitadas además por dos conjuntos de predicados, la cláusula *when* y la cláusula *where*. La cláusula *when* especifica las condiciones bajo las cuales la relación necesita ser sostenida, es decir, las precondiciones que requiere la relación. La cláusula *where*, a su vez, indica qué condición debe ser satisfecha por todos los elementos de los modelos participantes en la relación, y puede restringir cualquiera de las variables en la relación y sus dominios.

Las cláusulas *when* y *where* pueden contener cualquier expresión OCL arbitraria, además de expresiones de invocación a otras relaciones. Estas invocaciones permiten la composición de complejas relaciones a partir de relaciones más simples.

5.1.2.4. Relaciones *Top-Level*

Una transformación contiene dos clases de relaciones: de nivel superior o *top-level*, y de nivel inferior o *non-top-level*. La ejecución de la transformación implica que todas sus relaciones *top-level* se sostienen, mientras que las relaciones *non-top-level* sólo deben ser sostenidas cuando son invocadas en forma directa o transitiva desde la cláusula *where* de otra relación.

Una relación *top-level* es distinguida sintácticamente mediante la palabra reservada *top*. En el código de la Figura 5.2a, las relaciones *PackageToSchema()* y *ClassToTable()* son definidas *top-level*, mientras que la relación *AttributeToColumn()* no.

```
transformation umlRdbms (uml:SimpleUML, rdbms:SimpleRDBMS)
{
    top relation PackageToSchema { ... }
    top relation ClassToTable { ... }
    relation AttributeToColumn { ... }

}
```

(a) Relaciones top-level en código QVT

```
relation PackageToSchema
{
    checkonly domain uml p:Package
    {
        name = pn
    }
    enforce domain rdbms s:Schema
    {
        name = pn
    }
}
```

(b) Dominios *checkonly* y *enforce*

Figura 5.2: Diferentes aspectos del lenguaje QVT

5.1.2.5. Dominios *checkonly* y *enforce*

Toda relación puede ser forzada o no dependiendo del modelo destino, el cual puede ser marcado como *checkonly* o *enforce*. Cuando una transformación es direccionada hacia un dominio *checkonly*, este es simplemente verificado para determinar si existe una coincidencia válida en el modelo, que satisface la relación. Cuando una transformación ejecuta en la dirección del modelo de un dominio de tipo *enforce*, si el chequeo falla, el modelo destino es modificado para asegurar que la relación entre ambos dominios sea satisfecha. Este último caso es utilizado para forzar una transformación, ya que por definición el modelo destino siempre contendrá aquellos elementos que satisfacen las relaciones especificadas, mientras que el primer caso se utiliza para evaluar si el modelo destino es el resultado de la aplicación de la transformación sobre el modelo origen.

En el ejemplo de la Figura 5.2b el dominio correspondiente al modelo *uml* es marcado *checkonly*, y dominio para el modelo *rdbms* es marcado como *enforce*. Esto implica que si se ejecuta la transformación en la dirección del modelo *uml*, y existe un objeto s de tipo Schema en *rdbms* para el cual no existe un correspondiente objeto p de tipo Package en *uml* con el mismo nombre, entonces esto simplemente es reportado como una inconsistencia. Es decir, no será creado el objeto Package que permita satisfacer la relación *PackageToSchema()* dado que *uml* no es un dominio de tipo *enforce*.

Por el contrario, si estamos ejecutando la transformación *umlRdbms* en la dirección del modelo *rdbms* entonces por cada objeto de tipo Package en el modelo *uml* la relación primero verificará si existe un objeto de tipo Schema en *rdbms*, y en caso que esto no suceda un nuevo objeto Schema será creado en *rdbms*, con el mismo nombre del objeto Package asociado. En caso que exista un objeto Schema, para el cual no exista el correspondiente objeto Package en *uml*,

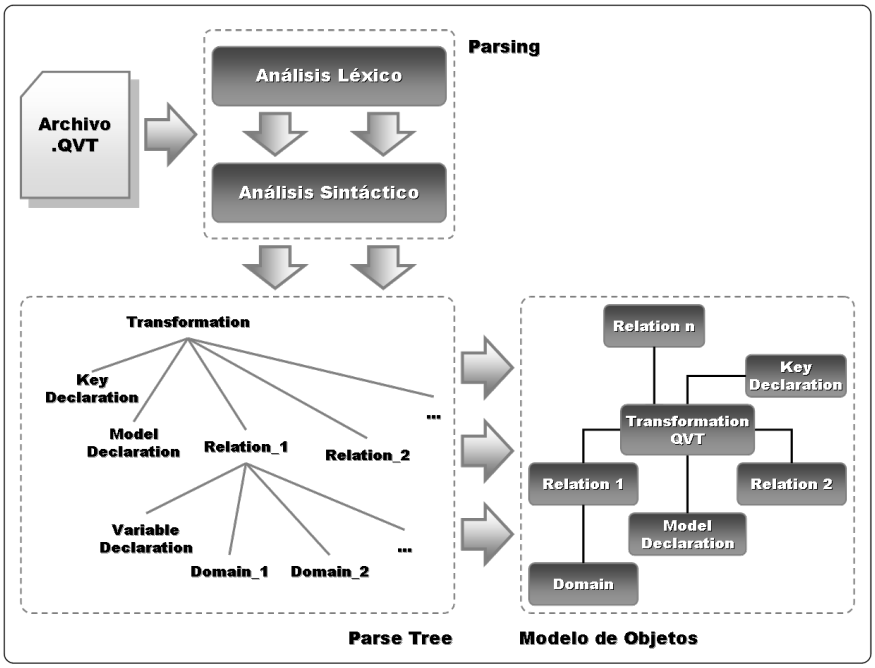


Figura 5.3: Interpretación de una transformación QVT

entonces dicho objeto (Schema) será eliminado de *rdbms*, manteniendo así la consistencia del dominio *enforce*.

Estas reglas son aplicadas según el tipo de dominio correspondiente al modelo destino. En este último escenario de ejecución, el resultado será la eliminación del objeto Schema aún si el dominio *uml* fuera marcado como *enforce*, dado que la transformación está ejecutando en la dirección de *rdbms*, por lo que la creación, modificación o eliminación de objetos sólo puede tener lugar en el modelo destino de la ejecución actual.

5.2. Modelado de transformaciones en QVTrace

Como parte del proceso de inferencia de trazas que veremos a continuación, QVTrace necesita interpretar y modelar la transformación a analizar, en un paso previo a la determinación de trazas.

Dicho análisis del código QVT se realiza en dos etapas (Figura 5.3):

1. *Parsing* del código de la transformación QVT (análisis léxico y sintáctico), cuyo resultado es un *parse tree* con la estructura de la transformación.

2. Generación del modelo de objetos en base al *parse tree*.

Para el *parsing* de la definición de la transformación en lenguaje QVT se utilizaron un conjunto de librerías desarrolladas por un grupo de investigación de la

Universidad de Kent, que trabajan sobre un framework para desarrollo conducido por modelos llamado KMF (Kent Modeling Framework) [8]. Las mismas son *open source*, y están basadas en CUP (Generador de parsers LALR para Java) [5] y Xerces (Parsers XML y componentes relacionados) [1]. Estas librerías se utilizan en varias herramientas de éste tipo, por ejemplo en mediniQVT [9], una implementación de QVT Relations para Eclipse.

La segunda fase parte del *parse tree*, y el resultado es un modelo de objetos que contiene la representación de la transformación. Los diagramas en la Figura 5.4 muestran las clases que modelan una transformación en QVTrace. A efectos de simplificar el trabajo, se restringieron algunas de las posibles expresiones OCL que el lenguaje admite. En particular, QVTrace reconoce las siguientes expresiones OCL:

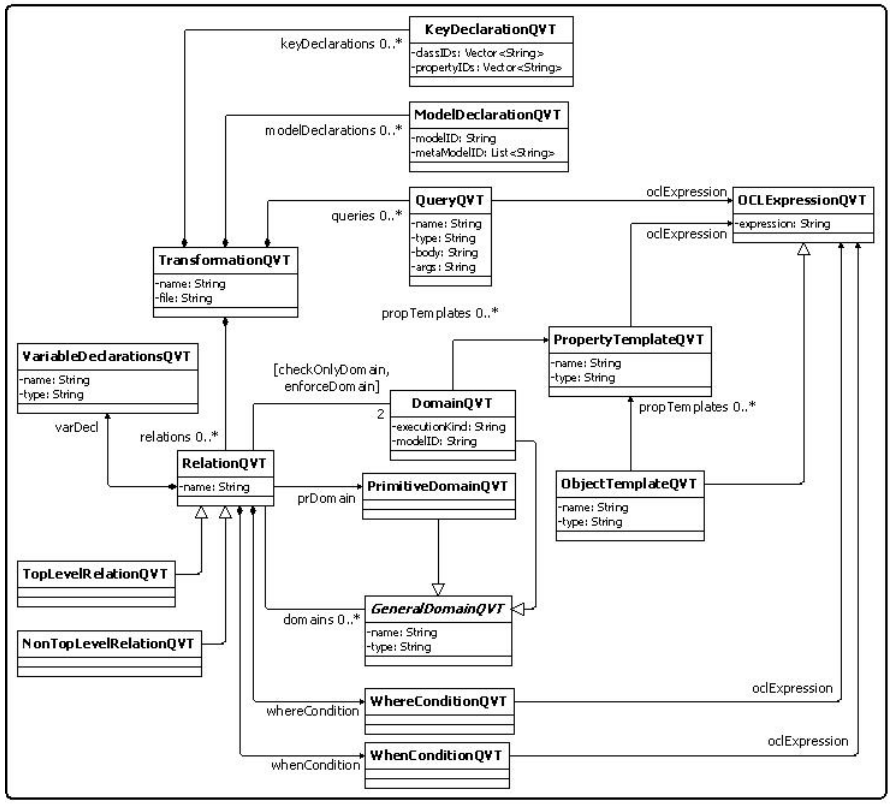
- Tipos primitivos: *boolean*, *string* (cadenas de caracteres), número enteros y reales.
- Llamadas a operaciones: expresiones n-arias, predicados, asignaciones.
- Conectores lógicos: *And*, *Or*, *Xor* y *Not*.
- Estructuras condicionales: *If-Then-Else*.
- Variables simples.

En el Apéndice A puede verse la gramática completa del lenguaje aceptado por QVTrace. Algunas de las expresiones que han sido excluidas, con sus respectivas operaciones, son:

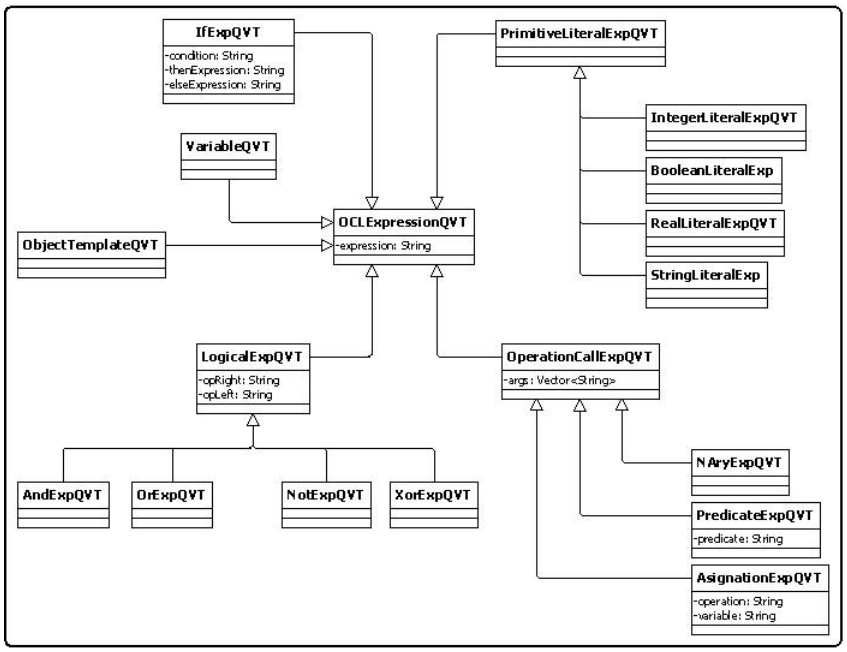
- Colecciones de objetos.
- Tuplas.

Respecto del lenguaje en sí, también por cuestiones de simplicidad se han excluido, entre otras, las siguientes características y/o capacidades:

- Importación (*imports*) de relaciones/operaciones (posibilidad de importar en un archivo definiciones QVT de otro archivo).
- Extensión (*extends*) de transformaciones. QVT permite que una transformación pueda ser una extensión de otra.
- Sobreescritura (*overriding*) de relaciones. QVT permite que una relación sobrescriba o modifique parcialmente la expresividad de otra relación.
- Soporte de paquetes.



(a) Modelo de objetos de una transformación QVT



(b) Modelo de objetos de una expresión OCL

Figura 5.4: Modelado de una transformación QVT

```
[top] relation Ri {
  v1, v2, .., vn : Type;
  checkonly domain c : Mo {
    elementMo = expression1                    (1)
  };
  enforce domain e : Md {
    elementMd = expression2                    (2)
  };
}
```

Figura 5.5: Patrón de generación de trazas en transformaciones QVT

5.3. Inferencia de trazas

A lo largo de las primeras dos secciones del capítulo hemos realizado un repaso de las principales características del lenguaje QVT, y a continuación se presentó el esquema de representación de dicha transformación como un modelo de objetos en QVTrace.

Como hemos visto, nuestro proceso incluye la lectura e interpretación de los modelos origen y destino, analizado oportunamente en la Sección 4.2, y en el *parsing* del código de la transformación QVT. El resultado de éstas dos operaciones actúa de *input* para el componente que realiza la inferencia de trazas, llamado *TraceAnalizer*. Este, aplica una estrategia de trazabilidad que permite el reconocimiento de dichas trazas. A continuación estudiaremos ciertos patrones o construcciones del código (de la transformación) QVT que permiten la derivación de trazas, y son la base de la estrategia de trazabilidad presentada en éste trabajo: el análisis basado en variables.

5.3.1. Análisis basado en variables

Como su nombre indica, esta estrategia de trazabilidad se basa en el reconocimiento de trazas, y relaciones de trazabilidad, mediante el estudio del uso de variables en el código fuente QVT. En efecto, hemos observado que determinadas variables en QVT sirven de nexo entre los modelos de entrada y salida, determinando una relación entre sus elementos, y definiendo así un traza o enlace entre éstos.

El analisis basado en variables

5.3.1.1. Generalidades

Como hemos visto, en QVT una transformación T es especificada a través de un conjunto de reglas o relaciones $R = \{r_1, \ldots, r_n\}$. La transformación de un elemento o de un modelo origen M_o en un elemento d de un modelo destino M_d tiene lugar sí y solo sí la tupla (o, d) satisface un subconjunto $R_T = \{r_1, \ldots, r_j\} \subseteq R$ de reglas obligatorias (*top-level relations*) donde $j \leq n$. Las restantes relaciones no obligatorias $R_N = \{r_1, \ldots, r_{n-j}\}$ son relaciones auxiliares, invocadas de manera directa o transitiva desde alguna de las relaciones *top-level*, que no

necesariamente se cumplirán en su totalidad, para todas las tuplas resultantes de una transformación.

El análisis basado en variables opera revisando todas las relaciones de una transformación, sean obligatorias o no, buscando ocurrencias del patrón esquematizado en la Figura 5.5. Dicho patrón o construcción del lenguaje está basado en un dominio *checkonly* c que especifica un *Template*[1] $\{elementM_o = expression_1\}$ (1), donde $elementM_o$ es un elemento del modelo origen M_o, y un dominio *enforce* e que define un *template* $\{elementM_d = expression_2\}$ (2), donde $elementM_d$ es un elemento del modelo destino M_d, y un conjunto de una o más variables $V = \{v_1, v_2, \ldots, v_n\}$, donde las expresiones $expression_1$ y $expression_2$ tienen en común alguna de las variables de V. Como veremos en breve, ante la ocurrencia de este esquema en alguna de las relaciones y, dependiendo del cumplimiento de otras condiciones que detallaremos más adelante, podremos inferir la traza $elementM_o \rightarrow elementM_d$.

5.3.1.2. El dilema de las relaciones *top-level* vs *non-top-level*

Uno de los puntos claves del análisis es el tipo de relación de la cuál es inferida una traza. En efecto, toda traza $elementM_o \rightarrow elementM_d$ derivada a partir de una relación obligatoria (*top-level*) cuenta con el respaldo que su validez estará garantizada para toda tupla ($elementM_o$, $elementM_d$) resultante de la transformación, dado que por definición dicha tupla satisface el conjunto de relaciones *top-level*. La situación para las relaciones no obligatorias difiere, dado que la validez de una traza derivada a partir de una regla *non-top-level* no puede garantizarse por definición, ya que por sus características, este tipo de relaciones no son de cumplimiento obligatorio para toda tupla de elementos origen y destino que resulte de la transformación.

No obstante, existe una condición bajo la cual una traza inferida a partir de una relación no obligatoria puede ser considerada válida para toda tupla resultante de la transformación. En efecto, si la relación desde la cual se deriva la traza es una pre-condición o una post-condición de una regla top-level, podremos asegurar que toda traza $elementM_o \rightarrow elementM_d$ derivada de ésta será valida para cualquier par ($elementM_o$, $elementM_d$) resultante de la transformación. Sean r_t y r_n dos relaciones definidas en una transformación T, donde r_t es una relación *top-level* y r_n es una relación *non-top-level*. Si r_n es invocada desde alguna sentencia de la cláusula *when* de r_t entonces todo conjunto de elementos que satisfaga r_t también verificará r_n dado que ésta actúa como pre-condición de la relación obligatoria, es decir, r_t será satisfecha solo si r_n lo es. Entonces, todo conjunto de elementos que resulte de la transformación habrá verificado r_t y por ende también r_n. Por esto, toda traza inferida desde la relación r_n será valida para cualquier grupo de elementos que resulte de la transformación. De igual manera, si r_n es invocada desde alguna sentencia de la cláusula *where* de r_t, diremos que r_n es una post-condición de r_t y por lo tanto todo conjunto que satisfaga r_t bajo esta condición también satisfará r_n.

[1] *Un Template es una construcción del lenguaje QVT que especifica uno o más igualdades <Identifier> '=' <oclExpression> (Ver Gramática en Anexo A)*

```
top relation PackageToSchema {
  pn : String;
  checkonly domain uml p : SimpleUML::UmlPackage {
    umlName = pn                                          (1)
  };
  enforce domain rdbms s : SimpleRDBMS::RdbmsSchema {
    rdbmsName = pn                                        (2)
  };
}
```

(a) Ejemplo código fuente QVT

```
top relation <nombre_relacion> '{'
  <nombre_variable> ':' <Type> ';'
  'checkonly' 'domain' <identifierDI> : <TypeI> '{'
    <IdentifierI> = <variable>                            (1)
  '};'
  'enforce' 'domain' <identifierDO> : <TypeO> '{'
    <IdentifierO> = <variable>                            (2)
  '};'
  [<when>]
  [<where>]
'}'
```

(b) Esquema general

Figura 5.6: Caso #1. Inferencia de trazas mediante una variable auxiliar

5.3.2. Casos de traza

A continuación se presentarán los casos o patrones encontrados bajo los cuales es posible derivar una traza a partir del código de una transformación QVT, y se ejemplificará su aplicación mediante el estudio de una transformación llamada *UML2RDBMS*, que establece el mapeo sencillo de un modelo de objetos llamado SimpleUML en un modelo relacional llamado SimpleRDBMS.

La palabra "patrón" posee una determinada variedad de significados para las Ciencias de la Computación: en el contexto del presente trabajo llamamos "patrón" a una forma consistente y característica de utilizar las variables dentro de un programa QVT. De esta forma, un *caso de traza* o *patrón de traza* esquematiza un determinado uso de variables dentro de una transformación QVT, el cual posibilita la obtención de información de trazabilidad.

5.3.2.1. Caso 1: Inferencia de trazas mediante una variable auxiliar

Descripción Cuando una regla obligatoria (*top-level*), o una regla no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una regla *top-level*, asigne un valor a algún elemento del modelo destino definido en el ámbito de un dominio de tipo *enforce*, mediante el uso de una variable previamente utilizada sobre un elemento del modelo origen de igual manera definido en el

ámbito de un dominio *checkonly*, podremos decir que el elemento del modelo origen mapeará directamente en el elemento del modelo destino.

Ejemplo Para clarificar este caso, consideraremos el ejemplo de la relación *PackageToSchema* definida en nuestra transformación *UML2RDBMS*. Esta regla especifica las condiciones bajo las cuales un elemento *Package*, del modelo origen, mapeará en un elemento *Schema*, del modelo destino.

Como podemos observar en el código QVT de la Figura 5.6a, vemos que la variable *pn* (1) y (2) nos permite inferir una traza entre los atributos *umlName* y *rdbmsName* de las entidades *UmlPackage* y *RdbmsSchema*. De ésta manera, con el análisis propuesto, la traza determinada será $umlName :: UmlPackage \rightarrow rdbmsName :: RdbmsSchema$, es decir, el atributo *rdbmsName* de toda entidad de tipo *RdbmsSchema* será igual al atributo *umlName* de la entidad *UmlPackage* del modelo SimpleUML (origen) a partir del cual fue derivado el modelo SimpleRDBMS (destino). Esta traza será válida para toda tupla (p, s) resultante de la transformación donde p es de tipo *UmlPackage* y s es de tipo *RdbmsSchema*.

En resumen, la información que nos provee la traza es que por cada paquete de nuestro modelo objeto tendremos, luego de la transformación, un (y solo un) esquema del modelo relacional con el mismo nombre.

Caso General En términos generales (Figura 5.6b), podemos decir que dada una transformación T, que define una relación R de tipo *top-level* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza $IdentifierI \rightarrow IdentifierO$ si existe en R un dominio *checkonly* que contiene una construcción de tipo *Property Template*[2] (1) donde $IdentifierI = variable$ y existe en R un dominio *enforce* que define un *Property Template* (2) $IdentifierO = variable$ donde $IdentifierI$ e $IdentifierO$ corresponden a elementos de los modelos origen y destino respectivamente, y *variable* es una variable definida en la relación R.

5.3.2.2. Caso 2: Inferencia de trazas mediante una expresión en función de una variable auxiliar

Descripción Este caso es una generalización del descripto anteriormente en el Caso #1. La diferencia radica en que el resultado final del elemento del modelo destino está dado, no por una sola variable, sino por una expresión (función) que afecta una variable definida en el ámbito de un dominio de tipo *enforce*.

Ejemplo Consideremos la relación *ClassToTable*, la cual determina el mapeo de una clase del modelo objeto UML en una tabla del modelo relacional RDBMS (Figura 5.7a). Como podemos observar, la expresión que describe el valor que el atributo *rdbmsName* (4) adoptará luego de la transformación estará dado por la función F de la variable *cn*, definida como $F(cn) = cn + \text{'\_id'}$,

[2] *Un Property Template es una construcción del lenguaje QVT definida como <Identifier> '=' <oclExpression> (Ver gramática en Anexo A)*

```
top relation ClassToTable {
  cn : String; prefix : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlNamespace = p : SimpleUML::UmlPackage { },
    umlKind = 'Persistent',
    umlName = cn                                              (1)
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
    rdbmsName = cn,                                           (2)
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {             (3)
      rdbmsName = cn + '_tid',                                (4)
      rdbmsType = 'NUMBER'                                    (5)
    },
    rdbmsKey = k : SimpleRDBMS::RdbmsKey {
      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}
    }
  };
}
```

(a) Código fuente QVT

```
...
'checkonly' 'domain' <identifierDI> : <TypeI> '{'
  <IdentifierI> = <variable>                                  (1)
'};'
'enforce' 'domain' <identifierDO> : <TypeO> '{'
  <IdentifierO> = F(<variable>)                               (2)
'};'
...
```

(b) Esquema General

```
...
'checkonly' 'domain' <identifierDI> : <TypeI> '{'
  <IdentifierI> = F(variable>)                                (1)
'};'
'enforce' 'domain' <identifierDO> : <TypeO> '{'
  <IdentifierO> = <Identifier> ':' <TypeO'> '{'               (2)
    <IdentifierO'> = F(<variable>)                            (3)
  '}'
'};'
...
```

(c) Uso de *Object Templates*

Figura 5.7: Caso #2. Ejemplo de utilización de una función de una variable auxiliar

donde el operador '+' representa la concatenación de cadenas de caracteres. En dicho caso, podremos inferir que todo atributo *rdbmsName* de la entidad *RdbmsColumn* (3), definida como elemento de una clase *RdbmsTable* en el modelo SimpleRDBMS, será igual al atributo *umlName* (1) de la entidad *UmlClass*, del modelo SimpleUML, concatenado al sufijo '*_tid*' o, en forma equivalente, que $rdbmsName :: RdbmsColumn = umlName :: UmlClass + \text{'\_id'}$.

Esta traza, en síntesis, nos está diciendo que toda tabla del modelo relacional tendrá una columna cuyo nombre será el mismo nombre de la clase que la originó, más el sufijo '*_tid*'. Por ejemplo, si partimos de un modelo origen que tiene una clase Persona, luego de la transformación tendremos una tabla llamada Persona (2), que contendrá una columna (3) llamada '*Persona_tid*'.

Caso General En términos generales (Figura 5.7b), podemos decir que dada una transformación T, que define una relación R de tipo *top-level* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza $IdentifierI \rightarrow IdentifierO$ si existe en R un dominio *checkonly* que contiene una construcción de tipo *Property Template* (1) donde $IdentifierI = variable$ y existe en R un dominio *enforce* que define un *Property Template* (2) $IdentifierO = F\,(variable)$ donde $IdentifierI$ e $IdentifierO$ corresponden a elementos de los modelos origen y destino respectivamente, y *variable* es una variable definida en la relación R, sobre la cual aplica la función F.

De la misma forma, como ilustramos en la Figura 5.7c este caso también aplica cuando existe en R un dominio *enforce* que define una construcción de tipo *Object Template*[3] (2), la cual especifica un *Property Template* (3) $IdentifierO' = F\,(variable)$, donde $IdentifierI$ e $IdentifierO'$ corresponden a elementos del modelo origen y destino respectivamente, y *variable* es una variable en la relación R, sobre la cual aplica la función F. En este caso, la traza generada será $IdentifierI \rightarrow IdentifierO'$.

Como puede verse en el Apéndice A, una construcción *Object Template* permite especificar elementos que forman parte de otros elementos, y no son de tipo primitivo, sino que corresponden a otras clases de elementos dentro de un modelo. En términos generales, un *Property Template* es un identificador seguido de un símbolo de igualdad '=', seguido de una expresión OCL. Una expresión OCL puede ser entre otras cosas una variable, una función de una variable, o en particular un *Object Template*, entre otros.

Esta última consideración respecto de las construcciones *Object Template*, aplica también para el Caso #1.

5.3.2.3. Caso 3: Inferencia de trazas mediante el uso de una constante

Descripción Se define para aquellos casos en los cuales un elemento del modelo destino definido en el ámbito de un dominio *enforce* es inicializado con un

[3] *Un Object Template es una construcción del lenguaje QVT que define una lista de estructuras Property Template. Es utilizado para especificar jerarquías de objetos donde un objeto actúa como atributo de otro objeto (Ver Gramática en Anexo A).*

```
...
'enforce' 'domain' <identifierDO> : <TypeO> '{'
  <IdentifierO> = <LiteralExp>                          (1)
'};'

...
```

Figura 5.8: Caso #3. Esquema general de inferencia de trazas mediante una constante

valor constante.

Ejemplo Tomando como ejemplo el código presentado en la Figura 5.7a, donde presentamos la relación que mapea una clase del modelo objeto en una tabla del modelo relacional, vemos que el atributo *rdbmsType* (5) de la entidad *rdbmsColumn* será igual a la constante 'NUMBER'. Este caso permite incrementar el nivel de información que nos provee una traza: en el ejemplo mostrado para el Caso #2 mencionamos que la traza nos indicaba que por cada entidad clase *UmlClass*, de nuestro modelo objeto, tendríamos una entidad tabla *RdbmsTable*, del modelo relacional, y que esta entidad tabla poseía un atributo de tipo *RdbmsColumn*, cuyo nombre (atributo *rdbmsName*) era igual al nombre de la clase (atributo *umlName*) concatenado al sufijo '*_tid*'. El Caso #3, nos permite ahora complementar esa información con el conocimiento que dicha entidad columna *RdbmsColumn* será siempre de tipo (atributo *rdbmsType*) numérico o 'NUMBER', en el modelo relacional.

Caso general En términos generales (Figura 5.8), podemos decir que dada una transformación T, que define una relación R de tipo *top* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza $Constante \rightarrow IdentifierO$ si existe en R un dominio *enforce* que define un *Property Template* (2) $IdentifierO = LiteralExp$ donde $IdentifierO$ corresponde a un elemento del modelo destino y $LiteralExp$ es una expresión literal (constante) definida en R.

Si observamos la gramática del lenguaje QVT aceptado del Apéndice A veremos que, por motivos de simplicidad, las expresiones literales han sido restringidas a expresiones literales primitivas, pudiendo ser éstas de tipo *String* (cadena de caracteres), enteras, flotantes o *Boolean*. El lenguaje OCL presenta además otros tipos de expresiones literales, por ejemplo expresiones de tipo colección (*CollectionLiteralExp*) y tupla (*TupleLiteralExp*), no contempladas en nuestro caso.

5.3.2.4. Caso 4: Inferencia de trazas mediante una variable auxiliar, definida como función en la cláusula *Where*

Descripción Este caso está basado en el segundo tipo de traza, descripto en el Caso #2. La diferencia radica en que la variable auxiliar es definida como una

función de otras variables asignada en alguna sentencia de la cláusula *where* de la relación.

Ejemplo Para presentar este caso consideraremos la relación *AssocToFKey*, la cual pertenece a nuestra transformación de ejemplo y se encarga de definir el mapeo entre una asociación del modelo objeto (entidad *UmlAssociation*) y una clave foránea (entidad *RdbmsForeignKey*) del modelo relacional.

Como vemos en el código de la Figura 5.9a, la relación establece que el nombre (atributo *rdbmsName*) de la clave foránea (entidad *RdbmsForeignKey*) está dado por la variable *fkn* (4), la cual es una función definida en la cláusula *where* (6) de la relación que concatena el nombre (atributo *umlName*) de la clase (entidad *UmlClass*) origen de la asociación (2), con el nombre (atributo *umlName*) de la asociación (entidad *UmlAssociation*) (1), con el nombre (atributo *umlName*) de la clase destino (entidad *UmlClass*) de la asociación (3), utilizando como separador de valores el carácter '_'.

Por otro lado, la relación indica que la clave foránea (entidad RdbmsForeignKey) tendrá una columna (entidad *RdbmsColumn*) cuyo nombre (atributo *rdbmsName*) está dado por la variable *fcn* (5), la cual es una función definida en la cláusula *where* (7) de la relación que concatena el nombre de la clave foránea (calculado en la variable *fkn*) con el sufijo '*_tid*'.

En resumen, supongamos que estamos transformando una asociación llamada *TrabajaEn*, que establece que toda persona de nuestro modelo objeto trabaja en una empresa. Esta asociación tiene una clase origen *Persona*, y una clase destino *Empresa*. La traza inferida nos dirá que el resultado de esta transformación será una entidad *RdbmsForeignKey*, que formará parte de la tabla *Persona* de nuestro modelo relacional, cuyo nombre será '*Persona_TrabajaEn_Empresa*' y que estará referenciada mediante una columna de tipo 'NUMBER' llamada '*Persona_TrabajaEn_Empresa_tid*', la cual enlaza una clave (entidad *RdbmsKey*) de la tabla *Empresa* llamada '*Empresa_pk*'.

Caso general En términos generales (Figura 5.9b), podemos decir que dada una transformación T, que define una relación R de tipo *top* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza de la forma $IdentifierI_1, .., IdentifierI_n \rightarrow IdentifierO$ si existe en R un dominio *checkonly* que contiene una construcción de tipo *Property Template* donde $IdentifierI_1 = v_1, .., IdentifierI_n = v_n$, y además existe en R un dominio *enforce* que contiene un *Property Template* donde $IdentifierO = variable$, y por último existe también en R una cláusula *where* que define un *Property Template* $variable = F(v_1, .., v_n)$, donde $IdentifierI_1, .., IdentifierI_n$, e $IdentifierO$ corresponden a elementos de los modelos origen y destino respectivamente, y $v_1, .., v_n$, *variable* son variables definidas en la relación R. Obviamente la cantidad de variables involucradas en la función $F(x)$ está relacionada con la cantidad de elementos del dominio origen involucrados en la generación del elemento del modelo destino, pudiendo ser uno o más.

```
top relation AssocToFKey {
  an : String;
  scn : String;
  dcn : String;
  fkn : String;
  fcn : String;
  checkonly domain uml a : SimpleUML::UmlAssociation {
    umlNamespace = p : SimpleUML::UmlPackage {},
    umlName = an,                                              (1)
    umlSource = sc : SimpleUML::UmlClass {
      umlKind = 'Persistent',
      umlName = scn                                            (2)
    },
    umlDestination = dc : SimpleUML::UmlClass {
      umlKind = 'Persistent',
      umlName = dcn                                            (3)
    }
  };
  enforce domain rdbms fk : SimpleRDBMS::RdbmsForeignKey {
    rdbmsName = fkn,                                           (4)
    rdbmsOwner = srcTbl : SimpleRDBMS::RdbmsTable {
      rdbmsSchema = s : SimpleRDBMS::RdbmsSchema {}
    },
    rdbmsColumn = fc : SimpleRDBMS::RdbmsColumn {
      rdbmsName = fcn,                                         (5)
      rdbmsType = 'NUMBER',
      rdbmsOwner = srcTbl
    },
    rdbmsRefersTo = pKey : SimpleRDBMS::RdbmsKey {
      rdbmsOwner = destTbl : SimpleRDBMS::RdbmsTable {}
    }
  Pl
  when {
    ClassToPkey(dc, pKey);
    PackageToSchema(p, s);
    ClassToTable(sc, srcTbl);
    ClassToTable(dc, destTbl);
  }
  where {
    fkn = scn + '_' + an + '_' + dcn;                          (6)
    fcn = fkn + '_tid';                                        (7)
  }
}
```

(a) Código fuente QVT

```
...
'checkonly' 'domain' <identifierDO> : <TypeI> '{'
  <IdentifierI1> = <v1>
  ...
  <IdentifierIn> = <vn>
'};'
'enforce' 'domain' <identifierD1> : <TypeO> '{'
  <IdentifierO> = <variable>                                   (1)
'};'
where '{'
  <variable> = F(<v1>,..,<vn>)                                 (2)
'}'

...
```

(b) Esquema general

Figura 5.9: Caso #4. Variable auxiliar y expresión en cláusula *Where*

5.3.2.5. Caso 5: Inferencia de trazas mediante una sentencia condicional *If-Then-Else*

Descripción El siguiente caso se define para aquellas situaciones donde una regla obligatoria (*top-level*), o una regla no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una regla *top-level*, asigne un valor a algún elemento del modelo destino definido en el ámbito de un dominio de tipo *enforce*, mediante el uso de sentencia condicional *If-Then-Else*, cuya expresión en caso de verificarse la condición analizada (o la expresión en caso de no verificarse, si la sentencia especifica la opción "*Else*") incluya una variable previamente utilizada sobre un elemento del modelo origen de igual manera definido en el ámbito de un dominio *checkonly*. En dicho caso, podremos decir que el elemento del modelo origen mapeará condicionalmente, o de manera parcial, en el elemento del modelo destino.

Ejemplo Para ilustrar este caso, analizaremos el código presentado en la Figura 5.10. El mismo define la regla de transformación de una clase del modelo objeto (entidad *UmlClass*) en una tabla del modelo relacional (entidad *RdbmsTable*) para lo cual debe especificar primero los mapeos de los atributos de la clase (nombre, tipo y paquete al cual pertenece) y posteriormente el mapeo de los componentes (atributos) de dicha clase en columnas de una tabla (2). Estos pueden ser primitivos (de tipos base *Integer*, *String* o *Boolean*), complejos (de tipo *UmlClass*), o bien referencias a objetos de tipo *parent* o *super* (ancestros). En particular nos abocaremos en la transformación de atributos primitivos en columnas (3).

Como podemos apreciar en la mencionada figura, la relación *PrimitiveAttributeToColumn* establece que el nombre (atributo *rdbmsName*) de una columna (5) de tipo *RdbmsColumn*, que proviene del mapeo del nombre (*umlName*) de un atributo (4) de tipo *UmlAttribute*, de una clase del modelo objeto, primitivo en este caso, está dado por la variable *cn*, la cual está afectada como post-condición de la relación en la cláusula *where*. Si analizamos la sentencia en dicha cláusula, veremos que el valor final de *cn* está condicionado por el valor asumido del dominio primitivo[4] *prefix*, el cual viene pasado en forma de parámetro desde la relación *top-level* que inició la transformación, en este caso *ClassToTable*, y se corresponde con el nombre de la clase, y de la tabla, según lo definido por la regla (1). En este caso, la relación establece que si el valor del dominio *prefix* es nulo entonces el valor del nombre de la columna (*rdbmsName*) será igual (8) al nombre (4) del atributo (*umlName*). Sino, en caso que el dominio *prefix* no sea nulo (9), entonces el valor del nombre de la columna será la concatenación $prefix + \text{`\_'} + an$, es decir el nombre de la tabla (y clase original) con el nombre del atributo, separados por un carácter '_'.

Asumamos que tenemos la clase *Persona*, cuyos atributos primitivos son *Edad* (de tipo numérico), *Nombre* (de tipo cadena de caracteres) y *DNI* (de tipo numérico). La traza inferida $UmlAttribute \rightarrow RdbmsColumn$ nos dirá que como

[4] Un dominio primitivo en QVT es un conjunto de elementos de la misma clase, de tipos básico *Integer*, *String*, *Boolean* o *Float*, utilizado comúnmente para el pasaje de parámetros entre relaciones

```
top relation ClassToTable {
  cn : String;
  prefix : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlNamespace = p : SimpleUML::UmlPackage { },
    umlKind = 'Persistent',
    umlName = cn
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsSchema = s : SimpleRDBMS::RdbmsSchema { },
    rdbmsName = cn,                                                    (1)
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName =cn + '_tid',
      rdbmsType = 'NUMBER'
    },
    rdbmsKey = k : SimpleRDBMS::RdbmsKey {
      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}
    P
  };
  when {
    PackageToSchema(p, s);
  }
  where {
    ClassToPkey(c, k);
    prefix = cn;
    AttributeToColumn(c, t, prefix);                                   (2)
  }
}
...
relation AttributeToColumn {
  checkonly domain uml c : SimpleUML::UmlClass { };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable { };
  primitive domain prefix : String;
  where {
    ComplexAttributeToColumn(c, t, prefix);
    PrimitiveAttributeToColumn(c, t, prefix);                          (3)
    SuperAttributeToColumn(c, t, prefix);
  }
}
...
relation PrimitiveAttributeToColumn {
  an : String;
  pn : String;
  cn : String;
  sqltype : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,                                                    (4)
      umlType = p :
      SimpleUML::UmlPrimitiveDataType {
        umlName = pn
      }
    }
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn,                                                  (5)
      rdbmsType = sqltype
    }
  };
  primitive domain prefix : String;
  where {
    cn = if ( prefix = '' ) then                                       (6)
      ( an )                                                           (7)
    else
      ( prefix + '_' + an )                                            (8)
    endif;
    sqltype = PrimitiveTypeToSqlType(pn);
  }
}
```

Figura 5.10: Caso #5. Inferencia de trazas en sentencias condicionales

resultado de la transformación dichos atributos mapearán en columnas (entidades *RdbmsColumn*) cuyos nombres serán respectivamente *Persona_Edad* (de tipo numérico), *Persona_Nombre* (de tipo cadena de caracteres) y *Persona_DNI* (de tipo numérico).

Caso general El presente caso puede tener dos variantes, dependiendo de la forma en que sea invocada la sentencia condicional *If-Then-Else*. El uso de esta sentencia en una cláusula *Where*, como vimos en el ejemplo, suele ser utilizada cuando se define un dominio primitivo para la relación que sirve para el pasaje de parámetros, donde el valor de dicho dominio es evaluado en la condición de la sentencia *If-Then-Else*.

En dicho caso (Figura 5.11a), podemos decir que dada una transformación T, que define una relación R de tipo *top* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza condicional $IdentifierI_1 \mid IdentifierI_2 \rightarrow IdentifierO$ si existe en R un dominio *checkonly* que contiene una construcción de tipo *Property Template* (1)(2) donde $IdentifierI_1 = v_1$, $IdentifierI_2 = v_2$, y además existe en R un dominio *enforce* que contiene un *Property Template* (2) con la forma $IdentifierO = v_O$, donde v_O es una variable definida en la cláusula *where* de R, asociada a un *Property Template* (4) $v_O = if\,(F\,(x))\,then\,(exprI_1)\,else\,(exprI_2)$, donde $F\,(x)$ es una función booleana que evalúa una determinada condición, $exprI_1$ es una expresión que involucra la variable vI_1 y $exprI_2$ es una expresión que incluye la variable vI_2, asociadas a elementos del modelo origen.

Eventualmente, es posible que tanto $exprI_1$ como $exprI_2$ incluyan más variables asociadas a elementos del modelo origen. Asumiendo que $exprI_1$ incluye el conjunto de variables $V_1 = \{v_1, .., v_n\}$ y $exprI_2$ involucra el conjunto de variables $V_2 = \{v_{n+1}, .., v_m\}$, en este caso la traza obtenida tendría la siguiente forma: $IdentifierI_1, .., IdentifierI_n \mid IdentifierI_{n+1}, .., IdentifierI_m \rightarrow IdentifierO$, donde $IdentifierI_i$ y v_i se relacionan mediante un *Property Template* cuya expresión es $IdentifierI_i = F\,(v_i)$, en el dominio *checkonly* de R.

Alternativamente, como segunda posibilidad, la sentencia *If-Then-Else* puede ser utilizada directamente en alguna construcción *Property Template* de un dominio *enforce* de R. Esta situación no altera la definición de la traza, la cual seguirá siendo la misma (Figura 5.11b) que en el caso anterior, donde la sentencia *If-Then-Else* era invocada desde la cláusula *Where* de la relación.

5.3.2.6. Caso 6: Inferencia de trazas mediante una consulta

Descripción Nuestro sexto caso contempla aquellas situaciones donde una regla obligatoria (*top-level*), o una regla no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una regla *top-level*, asigna un valor a algún elemento del modelo destino definido en el ámbito de un dominio de tipo *enforce* mediante el uso de consulta, ya sea directamente desde dicho dominio o bien en forma indirecta mediante el uso de una variable asignada en la cláusula *where* de la relación.

```
...
'checkonly' 'domain' <identifierD0> : <TypeI> '{'
  <IdentifierI1> = <vI1>                                              (1)
  <IdentifierI2> = <vI2>                                              (2)
'};'
'enforce' 'domain' <identifierD1> : <Type0> '{'
  <Identifier0> = <v0>                                                (3)
'};'
where '{'
  <v0> = if ( F(x) ) then ( <exprI1> ) else ( <exprI2> )              (4)
'}'

...
```

(a) Ejemplo 1. Traza condicional con expresión en cláusula *Where*

```
...
'checkonly' 'domain' <identifierD0> : <TypeI> '{'
  <IdentifierI1> = <vI1>                                              (1)
  <IdentifierI2> = <vI2>                                              (2)
'};'
'enforce' 'domain' <identifierD1> : <Type0> '{'
  <Identifier0> = if ( F(x) ) then ( <exprI1> ) else ( <exprI2> )     (3)
'};'

...
```

(b) Ejemplo 2. Traza condicional con expresión en dominio *enforce*

Figura 5.11: Caso #5. Esquema general

```
...
relation PrimitiveAttributeToColumn {
  an : String;
  pn : String;
  cn : String;
  sqltype : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,
      umlType = p :
      SimpleUML::UmlPrimitiveDataType {
        umlName = pn                                          (1)
      }
    P
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn,
      rdbmsType = sqltype                                     (2)
    }
  };
  primitive domain prefix : String;
  where {
    ...
    sqltype = PrimitiveTypeToSqlType(pn);                     (3)
  }
}
...
query PrimitiveTypeToSqlType (typename : String) : String
{
  if typename = 'INTEGER' then                                (4)
    'NUMBER'
  else
    if typename = 'BOOLEAN' then                              (5)
      'BOOLEAN'
    else
      'VARCHAR'                                               (6)
    endif
  endif
}
```

(a) Ejemplo código fuente QVT

```
...
'checkonly' 'domain' <identifierDO> : <TypeI> '{'
  <IdentifierI> = <vI>                                        (1)
'};'
'enforce' 'domain' <identifierD1> : <TypeO> '{'
  <IdentifierO> = <vO>                                        (2)
'};'
where '{'
  <vO> = Q(vI)                                                (3)
'}'

...
```

(b) Esquema general

Figura 5.12: Caso #6. Inferencia de trazas mediante una consulta

Ejemplo Como ejemplo tomaremos la relación *PrimitiveAttributeToColumn* (Figura 5.12a), la cual es utilizada para transformar los atributos primitivos de las clases del modelo objeto en columnas del modelo relacional. Dicha regla establece el mapeo de dos componentes del atributo: su nombre, el cual no sufre cambio alguno, y el tipo de datos asociado al atributo, el cual es transformado al tipo de dato equivalente del modelo relacional. De esta manera, un atributo tipo 'INTEGER' (4) de una clase del modelo objeto mapeará en una columna de tipo 'NUMBER' de una tabla del modelo relacional, un atributo de tipo 'BOOLEAN' (5) mapeará en una columna de igual tipo, y los restantes tipos de atributos mapearán en columnas de tipo 'VARCHAR' (6).

Esta conversión se realiza mediante una consulta QVT llamada *PrimitiveTypeToSqlType*, la cual recibe como parámetro el tipo primitivo del atributo y devuelve la equivalencia de dicho tipo en el modelo relacional (3). La relación establecida a través de esta consulta nos permitirá inferir que el tipo de dato (atributo *rdbmsType*) de toda columna (entidad *RdbmsColumn*) de una tabla (entidad *RdbmsTable*) del modelo relacional (2), será derivado a partir del nombre (atributo *umlName*) del tipo de dato primitivo (entidad *UmlPrimitiveDataType*) del atributo (entidad *UmlAttribute*) que le dio origen (1), es decir, la traza $umlName :: UmlPrimitiveDataType \rightarrow rdbmsType :: RdbmsColumn$, cuya expresión en este caso estará dada por $rdbmsType = PrimitiveTypeToSqlType\,(umlName :: UmlPrimitiveDataType)$.

Caso general En términos generales (Figura 5.12b), podemos decir que dada una transformación T, que define una relación R de tipo *top* (ejecución obligatoria), o una no obligatoria invocada desde una sentencia de la cláusula *when* o *where* de una relación *top-level*, siempre existirá una traza $IdentifierI \rightarrow IdentifierO$ si existe en R un dominio *checkonly* que contiene una construcción de tipo *Property Template* donde $IdentifierI = v_I$ (1), y existe en R un dominio *enforce* que contiene un *Property Template* de la forma $IdentifierO = v_O$, donde $v_O = Q\,(v_I)$ o eventualmente $v_O = x$ y $x = Q\,(v_I)$ (2)(3), y $Q\,(x)$ es una consulta definida en T. En dicho caso, la expresión resultante de la traza será $IdentifierO = Q\,(IdentifierI)$.

Al igual que en otros casos de traza, en éste también es posible que la consulta Q relacione más de elemento del modelo origen. En cuyo caso tendremos que la traza derivada será $IdentifierI_1, .., Identifier_n \rightarrow IdentifierO$, si existe un conjunto de *Property Templates* $\{IdentifierI_i = v_i\}$ donde $1 \leq i \leq n$ en el dominio *checkonly* de R y un *Property Template* $IdentifierO = v_O$ en el dominio enforce de R, donde $v_O = Q\,(v_1, .., v_n)$ o $v_O = x$ y $x = Q\,(v_1, .., v_n)$, siendo Q una consulta definida en T.

5.3.3. Esquema general de los casos de traza

Como vimos en la Sección 5.3.2, los casos de traza presentados responden a un común denominador: en todos, el análisis comienza en construcciones de tipo *Property Template* definidas sobre un dominio *enforce*, para los cuales se intenta determinar si existen en el dominio *checkonly* de la relación uno o más construcciones *Property Template* que tengan variables en común. Las variables

```
[top] 'relation' <identifierR> '{'
  <VarDeclarationList>
  'checkonly' 'domain' <identifierDO> : <TypeI> '{'
    <PropTempI1>,
    ...
    <PropTempIn>
  '};'
  'enforce' 'domain' <identifierD1> : <TypeO> '{'
    <PropTempO1>,
    ...
    <PropTempOn>
  '};'
'}'
```

Figura 5.13: Esquema general para la inferencia de trazas basada en variables

son el nexo entre ambos dominios, y permiten establecer las relaciones o el mapeo entre un dominio checkonly (origen) y un dominio enforce (destino).

Como vemos en la Figura 5.13, tenemos una relación genérica con un dominio *enforce* formado por un conjunto de elementos *Property Template* $P_O = \{PropTempO_1, .., PropTempO_n\}$, y un dominio *checkonly* que define un conjunto de *Property Template* $P_I = \{PropTempI_1, .., PropTempI_m\}$. Si existe un *Property Template* en P_O de la forma $IdentifierO_i = OCLExpression_O$ y existe un *Property Template* P_I de la forma $IdentifierI_j = OCLExpression_I$, donde $1 \leq i \leq n$ y $1 \leq j \leq m$, y $OCLExpression_I$ y $OCLExpression_O$ tienen variables en común, entonces siempre podremos inferir una traza $Identifier_I \rightarrow Identifier_O$. La expresión (forma) de la traza, dependerá de la expresión OCL definida en $OCLExpression_O$.

Aún sí $OCLExpression_I$ y $OCLExpression_O$ no compartieran variables en común es posible que las variables asociadas se encuentren relacionadas entre sí en alguna expresión de la cláusula *where*, lo cual eventualmente puede permitirnos, dependiendo del caso, como vimos anteriormente en el Caso #4 (5.3.2.4), inferir una traza.

Como puede apreciarse, los casos de traza presentados responden a las características de la expresión OCL definida en el dominio *enforce*, utilizada para especificar la trasformación de algún elemento del modelo origen en un elemento del modelo destino. En particular nuestro trabajo contempla los casos donde $OCLExpression_O$ es:

- Una variable (Caso #1 en 5.3.2.1).
- Una función (Caso #2 en 5.3.2.2).
- Una constante (Caso #3 en 5.3.2.3).
- Una expresión condicional *If-Then-Else* (Caso #5 en 5.3.2.5).
- Una consulta (Caso #6 en 5.3.2.6).

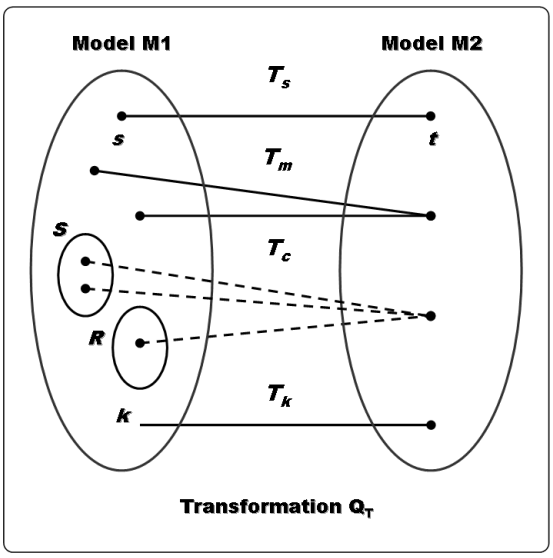


Figura 5.14: Tipos de traza detectados mediante VBA

5.3.4. Tipos de traza

Atento a los distintos casos de traza que esta técnica permite determinar, han sido tipificados cuatro tipos de traza distintos que pueden ser identificados mediante la aproximación propuesta en este trabajo. En el contexto de la presente tesis una traza es en definitiva una relación entre elementos de distintos modelos, y los distintos tipos de traza modelan las diversas relaciones que el análisis basado en variables permite reconocer.

La Figura 5.14 muestra los cuatro tipos de traza que el enfoque propuesto permite determinar: trazas simples, trazas múltiples o multitrazas, trazas condicionales y trazas constantes. A continuación detallaremos la definición de cada una de ellas.

5.3.4.1. Trazas simples

Sean $M1$ y $M2$ dos modelos que conforman los metamodelos $MM1$ y $MM2$ respectivamente, y sea Q_T una transformación QVT de $M1$ en $M2$, entonces una traza simple T_s especifica cómo un elemento s del modelo origen $M1$ es mapeado en uno t del modelo $M2$ por la transformación Q_T (relación *uno-a-uno*), donde $T_S : s \rightarrow t$. El operador $\rightarrow$ es utilizado en este caso como "determina", indicando la dependencia del valor de t respecto del valor de s, luego de la ejecución de la transformación.

5.3.4.2. Trazas múltiples o multitrazas

Una multitraza T_m especifica cómo múltiples elementos $s_1, .., s_n$ del modelo $M1$ mapean sobre un único elemento t del modelo $M2$, según la definición de la transformación T (relación *muchos-a-uno*), donde $T_m : s_1, .., s_n \rightarrow t$.

5.3.4.3. Trazas condicionales

Una traza condicional T_c representa dos trazas potenciales, determinadas dentro del código fuente QVT por sentencias condicionales *If-Then-Else*. En este caso, el valor del elemento t del modelo destino está determinado por alguno de los dos grupos $S = \{s_1, .., s_n\}$ o $R = \{r_1, .., r_m\}$ de elementos del modelo origen, donde $T_C : S \oplus R \rightarrow t$.

5.3.4.4. Trazas constantes

Por último, una traza constante T_k modela la asignación de un valor constante k sobre un elemento del modelo destino t en la definición de Q_T, donde $T_k : k \rightarrow t$.

5.3.5. La relación entre tipos y casos de traza

Naturalmente, existe una fuerte relación entre los tipos de traza que el análisis basado en variables permite identificar, y el metamodelo de trazabilidad desarrollado para QVTrace, el cual hemos detallado en la Sección 4.3. Cada tipo de traza tipificada ha sido incluida como subtipo de la clase *Trace*, permitiendo así que QVTrace pueda soportar la técnica de VBA (*Variable-based Analysis*). Por otro lado, cada uno de los casos de traza individualizados permite identificar trazas de uno o más tipos. A continuación detallaremos los tipos de traza que permiten inferir cada uno de los casos o patrones analizados:

- Caso #1: Traza simples T_s.
- Caso #2: Trazas múltiples T_m.
- Caso #3: Trazas constantes T_k.
- Caso #4: Trazas simples, múltiples, condicionales o constantes, dependiendo de la expresión definida en la cláusula *Where*.
- Caso #5: Trazas condicionales T_c.
- Caso #6: Trazas simples o múltiples, dependiendo del número elementos del modelo origen pasados como parámetros a la consulta.

5.3.6. Ventajas y desventajas del análisis basado en variables

La técnica presentada aporta valiosos beneficios al desafío de la obtención de información de trazabilidad. En primer lugar es completamente automática, es decir, en ninguna etapa del proceso de análisis se requiere la intervención de un ser humano para obtener sus resultados. Por tal, colabora con la productividad de los ingenieros al no requerir esfuerzos, y está libre de los errores que éstos pueden cometer. En segundo lugar, a diferencia de algunas implementaciones, como por ejemplo mediniQVT [9], la información de trazabilidad es generada a nivel de modelo, no de instancias, de manera que permite determinar no sólo

el mapeo de un elemento en otro, sino la expresión o forma del mismo, es decir la relación de trazabilidad entre ambos, sea cual fuere la instancia del modelo origen que se transforme o la correspondiente instancia del modelo destino. Esto, a su vez, permite la posibilidad de verificar y eventualmente forzar la consistencia e integridad de la relación entre ambos modelos, lo cual puede ser de gran ayuda en particular cuando el modelo destino es modificado unilateralmente, y no como resultado de cambios en el modelo origen luego procesados por la transformación, lo que sería el flujo natural del proceso de modificación.

Otra de las ventajas que se observan es que la obtención de trazas no depende del proceso de transformación, sino sólo de su definición. En consecuencia, esto puede ayudar al desarrollador como herramienta de *debugging* en la depuración de la especificación de la misma, brindándole la posibilidad de evaluar anticipadamente los resultados que se obtendrán tras la ejecución de la transformación. Esta independencia, a su vez, brinda flexibilidad y facilita el mantenimiento de la información de trazabilidad ya que la misma puede ser almacenada en un repositorio, o generarse ad hoc, sin contaminar los modelos ni la especificación de la transformación.

QVT, como otros lenguajes de transformación de modelos (por ejemplo, MOFScript [10]), provee la generación implícita de enlaces de traza. Por lo tanto, cualquier proceso ejecutado por fuera de una transformación implica un esfuerzo extra adicional. El beneficio en este caso es la posibilidad de contar con información de trazabilidad más amplia, y mejorada. En efecto, las trazas generadas implícitamente por QVT no se encuentran totalmente adaptadas a la mayoría de las actividades de explotación de información de trazabilidad, como se explica claramente en [21], y veremos en detalle en el Capítulo 8. Esto se debe mayormente al hecho que el propósito principal de dicho esquema de generación de trazas implícitas de QVT es el de dar soporte al mecanismo de resolución de objetos. Por otro lado, el análisis basado en variables permite obtener relaciones de trazabilidad, de varios tipos, y no sólo enlaces de traza *uno-a-uno* entre instancias de elementos de modelos. Además, dado que la aproximación VBA trabaja con código fuente QVT como *input*, no depende de ninguna implementación particular de motor de QVT.

Finalmente, si bien los casos de traza presentados abarcan un amplio espectro de las posibles trazas que pueden ser inferidas, el método no asegura la obtención de la totalidad de las trazas existentes. En este sentido, es preciso seguir trabajando en la determinación de más casos de traza, o eventualmente demostrar que la técnica permite inferir el universo completo de relaciones de trazabilidad entre elementos de los modelos origen y destino de una transformación QVT.

5.4. Implementación

A lo largo del presente capítulo hemos presentado el mecanismo de inferencia de trazas utilizado por QVTrace. En esencia este análisis se basa en el estudio del uso de las variables en un programa QVT para inferir las trazas que se encuentra implícitas en la definición de la transformación de modelos. A continuación detallaremos en términos generales como se implementó el análisis basado en

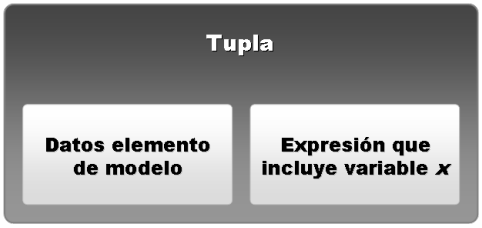


Figura 5.15: Estructura de datos *Tupla*

variables propuesto para la extracción de información de trazabilidad a partir de una transformación QVT.

5.4.1. Generalidades

El eje del análisis basado en variables consiste en determinar de qué manera las variables definidas en el contexto de una relación o regla de transformación vinculan elementos del modelo origen con elementos del modelo destino. Como explicamos anteriormente, para que dicha relación determine una traza, se requiere el cumplimiento de dos condiciones:

1. La existencia de una expresión que vincule un elemento del modelo origen con una variable de la regla de transformación, digamos x, definida en el contexto de un dominio de tipo *checkonly*.
2. La existencia de una expresión que relacione un elemento del modelo destino con dicha variable x en el contexto de un dominio tipo *enforce*, perteneciente a la misma regla de transformación del primer punto (1).

La expresión que relaciona elementos de modelo con variables puede adoptar diversas formas, las cuales hemos explicado ya en los respectivos casos de traza que han sido individualizados en la Sección 5.3.2.

5.4.2. El concepto de tupla

El esquema de inferencia de trazas desarrollado se basa en una estructura de datos que fue diseñada especialmente, la cual mantiene las relaciones entre elementos de modelo y variables. Dicha estructura fue denominada *Tupla* (Figura 5.15), y la misma contiene, en esencia, información de un elemento de modelo (nombre del atributo, clase a la cual pertenece y modelo al cual corresponde dicha clase) y de la expresión que contiene a la variable que se encuentra relacionada. Adicionalmente, es posible que la expresión asociada al elemento de modelo contenga un constante, en lugar de una variable, como veremos más adelante.

La estrategia de inferencia de trazas consiste entonces en generar las tuplas presentes en el dominio *checkonly* y en el dominio *enforce* de cada relación, y posteriormente analizar si utilizan variables en común. En caso de sostenerse

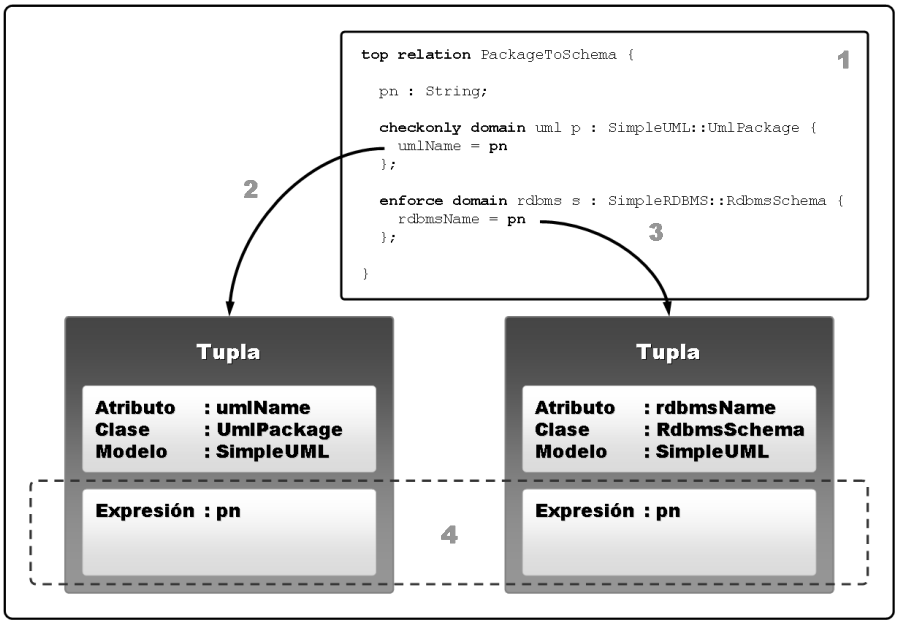


Figura 5.16: Ejemplo de utilización de tuplas

dicha condición estaremos en presencia de una traza, que implica una relación de un elemento del modelo origen con uno del modelo destino.

La Figura 5.16 ilustra la utilización de tuplas para la inferencia de trazas. En la imagen, puede verse un fragmento de la definición de una de las reglas de transformación de un modelo UML a uno Relacional (RDBMS). En este caso, la relación (1) especifica la transformación de elementos UmlPackage (del modelo UML) en elementos RdbmsSchema (del modelo RDBMS). Como resultado del análisis basado en variables, se determinan dos tuplas:

1. Una relaciona el elemento del modelo origen *umlName*, de la entidad UmlPackage, con la variable *pn* (2).
2. La segunda relaciona el elemento del modelo destino *rdbmsName*, de la entidad RdbmsSchema, con la variable *pn* (3).

Finalmente, la presencia de la variable *pn* en ambas tuplas (4) permite relacionar ambos elementos de modelo, logrando obtener una traza. En este caso, el razonamiento es muy sencillo: si $umlName = pn$ y $rdbmsName = pn$ entonces $rdbmsName = umlName$. De esta forma, podemos asegurar que para todo par ($UmlPackage$, $RdbmsSchema$) que satisface la relación $PackageToSchema$ se verifica que el nombre del esquema relacional (atributo *rdbmsName*) será igual al nombre del paquete UML origen (atributo *umlName*).

5.4.3. El algoritmo de inferencia de trazas

Como su nombre lo indica, el mecanismo de generación de trazas propuesto se basa en el análisis de las variables de las relaciones, también llamadas reglas,

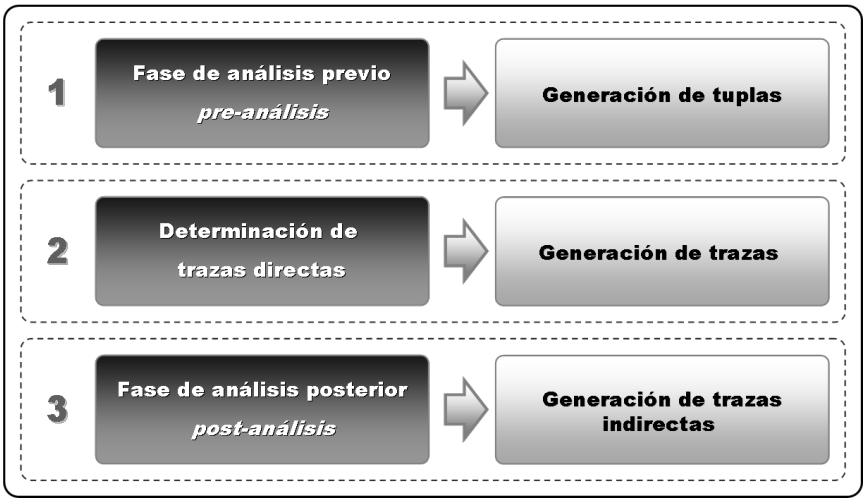


Figura 5.17: Fases del algoritmo de inferencia de trazas utilizado

que componen una transformación QVT. El algoritmo de inferencia de trazas se compone de tres fases bien definidas, ejecutadas una vez por cada variable de cada relación de la transformación QVT en estudio:

- Fase de análisis previo, donde se determinan las tuplas asociadas a la variable analizada, presentes en los dominios *checkonly* y *enforce* de la relación.
- Fase de determinación de trazas directas, donde son inferidas las trazas simples, constantes, condicionales y de consulta a partir de las tuplas generadas en el paso anterior.
- Fase de análisis posterior, o post-análisis, donde son determinadas las trazas indirectas, de tipo simple, condicional o múltiple (MultiTrace), que no pudieron ser inferidas en el paso previo.

La Figura 5.17 muestra las distintas etapas del algoritmo propuesto, y los resultados conseguidos por cada una. A continuación analizaremos las mismas con mayor nivel de detalle.

5.4.3.1. Fase de análisis previo (pre-análisis)

Durante esta etapa, son analizados cada uno de los dominios de la relación, *checkonly* y *enforce*, buscando la presencia de tuplas asociadas a una determinada variable de la relación. Como resultado de la fase, es posible encontrar tuplas en ambos dominios de la relación, o tuplas en uno u otro dominio, no en ambos, dependiendo de la utilización que se realice de cada variable.

En el primer caso, la presencia de una tupla en un dominio *checkonly*, y la presencia de otra en el dominio *enforce* de la relación, para la misma variable, dará como resultado una traza directa. En contraste, la presencia de una tupla

```
top relation PackageToSchema {
  pn : String;
  checkonly domain uml p : SimpleUML::UmlPackage {
    umlName = pn                                                    (1)
  };
  enforce domain rdbms s : SimpleRDBMS::RdbmsSchema {
    rdbmsName = pn                                                  (2)
  };
}
```

(a) Determinación de una traza directa

```
relation PrimitiveAttributeToColumn {
  an, pn, cn, sqltype : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,                                                 (1)
      umlType = p : SimpleUML::UmlPrimitiveDataType {
        umlName = pn                                                (2)
      }
    }
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn,                                               (3)
      rdbmsType = sqltype                                           (4)
    }
  };
  primitive domain prefix : String;
  where {
    cn = if prefix = '' then
      an                                                            (5)
    else
      prefix + '_' + an                                             (6)
    endif;
    sqltype = PrimitiveTypeToSqlType(pn);                           (7)
  }
}
```

(b) Determinación de una traza indirecta

Figura 5.18: Generación de tuplas en fase de pre-análisis

Relación	Variable	Tuplas (*checkonly*)	Tuplas (*enforce*)
P2S	*pn*	(umlName, pn)	(rdbmsName, pn)
P2C	*an*	(umlName, an)	-
	pn	(umlName, pn)	-
	cn	-	(rdbmsName, cn)
	sqltype	-	(rdbmsType, sqltype)

Tabla 5.1: Resultados obtenidos durante la fase de pre-análisis para cada relación

en un dominio *enforce*, sin su correspondiente tupla asociada en un dominio *checkonly*, o viceversa, indican la presencia de una traza indirecta, que no puede ser definida hasta el análisis de las post-condiciones de la regla en la cláusula *Where*, el cual es realizado durante la ejecución de la fase de post-análisis, como explicaremos más adelante.

La Figura 5.18 presenta dos relaciones de la transformación UML2RDBMS que nos permiten ilustrar los casos mencionados que se plantean durante la fase de análisis previo. La Figura 5.18a muestra el código QVT de la relación *PackageToSchema*, la cual realiza la conversión de entidades PACKAGE del modelo UML en entidades *Schema* del modelo RDBMS. Como resultado del análisis, se obtiene la tupla ($umlName :: UmlPackage$, pn) del dominio *checkonly* de la relación (1), y la tupla ($rdbmsName :: RdbmsSchema$, pn) del dominio *enforce* (2).

A continuación, la Figura 5.18b nos muestra un segundo ejemplo donde en ningún caso el análisis resulta en la obtención de dos tuplas para la misma variable en los dominios *checkonly* y *enforce*. El código fuente analizado corresponde en este caso a la relación *PrimitiveAttributeToColumn*, la cual especifica el mapeo de un elemento de tipo atributo primitivo, del modelo UML, en una columna del modelo relacional. Para este caso, el algoritmo debe realizar el análisis de los dominios de la relación para cada una de las cuatro variables definidas. La primera iteración sobre la variable *an*, nos da como resultado la tupla ($umlName :: UmlAttribute$, an) en el dominio *checkonly* (1), pero no encuentra tupla asociada a la misma variable en el dominio *enforce*. En la segunda iteración del análisis, sobre la variable *pn*, sucede una situación similar, obteniéndose la tupla ($umlName :: UmlPrimitiveDataType$, pn) en el dominio *checkonly* (2), sin una tupla correspondiente en el dominio *enforce* de la relación. Finalmente la tercer y cuarta iteración del análisis previo (sobre las variables *cn* y *sqltype* respectivamente) permiten la obtención de las tuplas ($rdbmsName :: RdbmsColumn$, cn) y ($rdbmsType :: RdbmsColumn$, $sqltype$) del dominio *enforce* (3)(4), sin hallarse tuplas asociadas a dichas variables sobre el dominio *checkonly* de la relación.

La Tabla 5.1 muestra los resultados de la ejecución de la fase de análisis previo sobre las dos relaciones analizadas. Como vemos, la primer fila nos muestra que para la regla *PackageToSchema* (indicada como P2S en la tabla) se obtienen dos tuplas relacionadas con la variable pn, una en el dominio *checkonly* y otra en el dominio *enforce* de la relación. El resto de la tabla muestra las tuplas obtenidas

en la regla *PrimitiveAttributeToColumn* (indicada en la tabla como P2C), donde se observa que para cada variable analizada sólo una tupla es hallada en cada dominio.

5.4.3.2. Fase de determinación de trazas directas

La siguiente fase del algoritmo consiste en el análisis de las tuplas generadas durante la etapa previa y en la generación de trazas directas. Denominamos traza directa a aquella que puede ser inferida a partir de las tuplas generadas durante la fase de pre-análisis sin ningún tipo de tratamiento específico o estudio adicional.

Durante esta etapa son analizadas las tuplas generadas en la fase anterior y se verifica si, para una variable determinada, existe una tupla en ambos dominios de la relación, *checkonly* y *enforce*, cuya expresión las contenga. La traza inferida, en este caso, será una traza directa. Los datos de entrada de la fase serán la lista de tuplas de dominio *checkonly* y de dominio *enforce* generadas durante el análisis previo, mientras que la salida serán las trazas directas, de tipo simple, condicional o constantes, que puedan ser inferidas a partir de las tuplas.

Continuando con nuestro ejemplo anterior, vemos que para la relación *PackageToSchema* la ejecución de esta fase logra la obtención de la traza $umlName :: UmlPackage \rightarrow rdbmsName :: RdbmsSchema$, inferida a partir de las tuplas conseguidas en la fase previa para la variable *pn*. En contraste, la operación de la fase para la relación *PrimitiveAttributeToColumn* no deja como resultado ninguna traza para ninguna de las variables. En efecto, como muestra la Tabla 5.1, si bien durante la fase preliminar se obtuvieron tuplas de ambos dominios, en ningún caso se determinó la presencia de una tupla con la misma variable para ambos dominios simultáneamente.

5.4.3.3. Fase de análisis posterior (post-análisis)

La fase final del algoritmo, llamada post-análisis, recibe como entrada aquellas tuplas generadas durante la primera etapa, o análisis previo, que no pudieron ser derivadas en trazas por falta de información que permita establecer las relaciones correspondientes entre las variables. Dicha información faltante, debe ser buscada en la sección de post-condiciones de la regla de transformación, dentro de la cláusula *Where*.

Lo que sucede durante el post-análisis es el estudio de las sentencias de la cláusula *Where*, en procura de determinar las relaciones de trazabilidad entre los elementos de modelo, especificada por las variables. Siguiendo nuestro ejemplo con la relación *PrimitiveAttributeToColumn*, vemos que las sentencias de la cláusula *Where* nos permiten terminar de asociar aquellas variables que tras la fase de determinación de trazas directas parecían aisladas. La sentencia de asignación de la variable *cn* condicionada al resultado de la cláusula *If-Then-Else* (Figura 5.18b) nos permite establecer su relación con la variable *an* (5)(6), y así poder inferir la traza condicional $umlName :: UmlAttribute \oplus prefix + \text{`\_'} + umlName :: UmlAttribute \rightarrow rdbmsName :: RdbmsColumn$. A continuación, la sentencia

de asignación de la variable *sqltype* nos permite asociarla con la variable *pn*, determinando la traza $Q\,(umlName :: UmlPrimitiveDataType) \rightarrow rdbmsType :: RdbmsColumn$, donde Q es la consulta $PrimitiveTypeToSqltype\,()$.

5.4.4. Trazas constantes

La inferencia de trazas constantes realizadas a partir del mecanismo presentado son un caso de excepción dentro del análisis basado en variables. En términos estrictos, si bien la detección de este tipo de trazas es realizada por el mismo componente que implementa la estrategia de trazabilidad, la misma no siempre corresponde a un análisis de variables. En efecto, dado que es posible inferir una traza constante en una expresión sin variables, la detección no necesariamente requiere el estudio de variables, en particular para el caso de las trazas constantes, sean directas o indirectas.

La detección de este tipo de trazas se realiza del mismo modo que para las demás. Para las trazas constantes directas, que son aquellas que pueden ser inferidas luego de la fase de pre-análisis, su detección sucede al verificar que la expresión de la tupla generada en un dominio *enforce* corresponde a una constante. En nuestro caso, hemos restringido las posibilidades a dos tipos de constantes:

- Numéricas, de valores enteros.
- De carácter.

De esta forma, la presencia de una expresión $elemento\_modelo = K$ en un dominio *enforce*, donde K es una constante numérica entera o de carácter, dará como resultado una traza $K \rightarrow elemento\_modelo :: Entidad$, siendo *Entidad* la clase a la cual pertenece el atributo *elemento_modelo*.

Para las trazas constantes indirectas, que son aquellas que son inferidas en la fase de post-análisis a partir de sentencias en la cláusula *Where* de la relación, la situación difiere un poco: en lugar de asignar a un elemento del modelo destino una constante, en el dominio *enforce*, dicho elemento tiene asignada una variable que luego es definida en la sección de post-condiciones de la regla o relación de transformación, de manera directa o a través de una sentencia condicional, según el caso.

Capítulo 6

Casos de estudio: UML2Java y Bib2Doc

Luego de haber presentado nuestra propuesta para gestión de trazabilidad en el marco del desarrollo conducido por modelos, y haber detallado el mecanismo propuesto para la inferencia automática de trazas, a continuación ilustraremos el funcionamiento del esquema mediante dos ejemplos de transformaciones de modelos que permitan visualizar las trazas obtenidas por QVTrace a partir del análisis basado en variables, aplicado sobre la definición de una transformación en lenguaje QVT.

6.1. QVTrace en acción

En el presente capítulo analizaremos dos casos de estudio de transformaciones de modelos, y veremos en detalle el resultado del proceso de inferencia de trazas aplicando el mecanismo de análisis basado en variables presentado en el Capítulo 5. Ambos casos han sido obtenidos y adaptados de la colección de ejemplos de transformaciones ATL (ATLAS Transformation Language) publicados en su portal, conocido como *ATL Transformation Zoo* [2].

En primer lugar se presentará la transformación UML2Java, la cual mapea un esquema realizado en UML sobre una modelización del código fuente Java respectivo. Nos referimos a una modelización ya que el resultado de la transformación no es código fuente Java, sino un modelo simplificado del mismo. No obstante, éste puede ser obtenido del modelo en un paso, mediante el uso de un analizador XML que lo interprete, sin mayor complejidad.

A continuación abordaremos la transformación Bib2Doc, la cual convierte un modelo de documentación bibliográfica BibTex [3] en un documento Docbook [6], el cual está basado en XML y permite la definición de texto estructurado, de forma independiente de la presentación del mismo. Para cada uno de los casos analizados se describirán los modelos de entrada y salida, la definición de la transformación, y las trazas obtenidas por QVTrace.

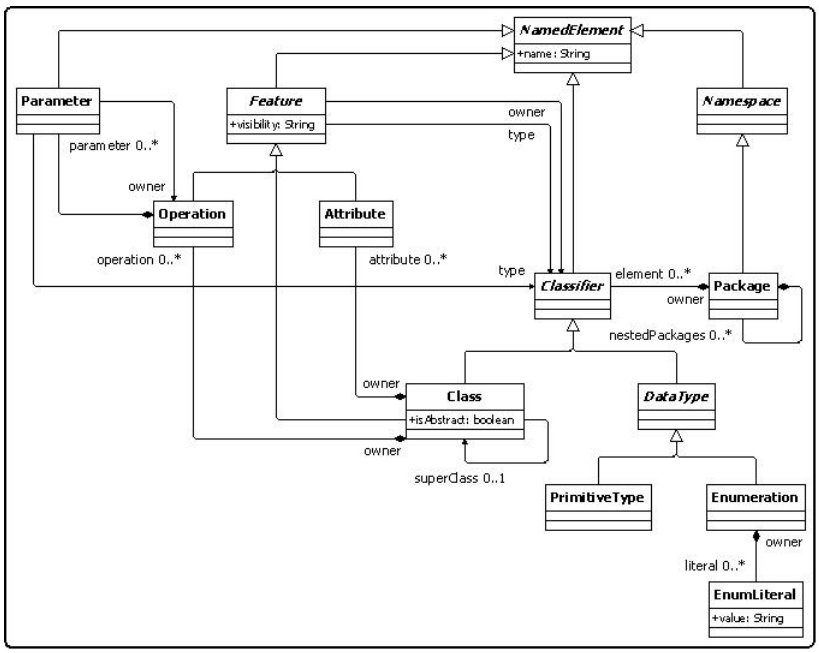


Figura 6.1: Modelo MySimpleUML

6.2. Caso #1: La transformación UML2Java

Como indicamos anteriormente, esta transformación mapea representaciones especificadas en UML sobre el correspondiente modelo de código fuente Java. En ambos casos se trabajó sobre representaciones simplificadas de cada uno de los modelos. A continuación los analizaremos en detalle.

6.2.1. Modelo de entrada: MySimpleUML

Como vemos en la Figura 6.1 el modelo MySimpleUML consta de una entidad PACKAGE, la cual se compone de elementos clasificadores (entidad CLASSIFIER) o bien de sub entidades o entidades PACKAGE anidadas. Un clasificador puede ser una clase (entidad CLASS) o un tipo de dato (entidad DATATYPE). A efectos de simplificar el modelo, hemos considerado sólo dos tipos de dato: primitivos (entidad PRIMITIVETYPE) como *Integer*, *String*, *Date*, etc, y enumeraciones (entidad ENUMERATION), conformadas por literales (entidad ENUMLITERAL).

Toda entidad CLASS se compone de atributos (entidad ATTRIBUTE) y operaciones (entidad OPERATION), y puede tener una superclase asociada, o ninguna (atributo *superClass*). A su vez, cada operación puede tener cero o más parámetros (entidad PARAMETER). Todo atributo, operación o parámetro tiene un tipo asociado, que define la clase de elemento a la cual pertenece.

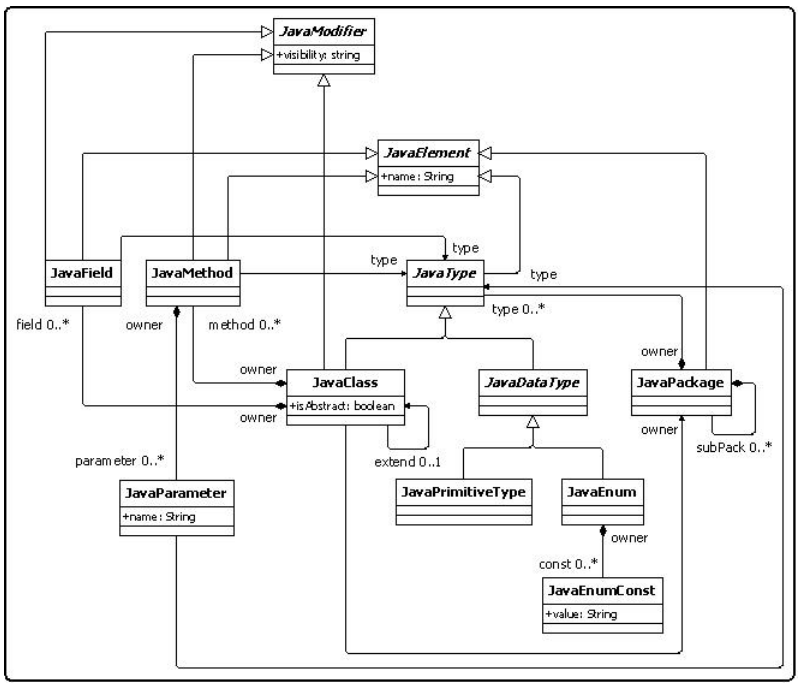


Figura 6.2: Modelo MySimpleJava

6.2.2. Modelo de salida: MySimpleJava

El modelo de salida MySimpleJava, es una versión simplificada del lenguaje de programación Java (Figura 6.2), y es en esencia estructuralmente análogo al modelo MySimpleUML. En el nivel superior tenemos un paquete Java (entidad JavaPackage) el cual se compone del elementos JavaType, o de otros JavaPackage anidados (atributo *subPack*). Un elemento JavaType puede ser una clase Java (entidad JavaClass) o un tipo de dato Java (entidad JavaDataType). Al igual que con el modelo MySimpleUML, este modelo define únicamente tipos de dato primitivos (entidad JavaPrimitiveType) o enumeraciones (entidad JavaEnum), éstas últimas compuestas por constantes (entidad JavaEnumConst).

Toda clase Java (entidad JavaClass) se compone de un conjunto de campos (entidad JavaField) y de métodos (entidad JavaMethod). Dichos métodos pueden tener cero o más parámetros (entidad JavaParameter), lo cuales pueden ser clases o tipos de datos Java.

6.2.3. Reglas de transformación

Para la definición de la transformación se consideraron las siguientes reglas:

6.2.3.1. Regla #1: Paquetes

Para cada instancia de paquete UML (Package), una instancia de paquete Java (JavaPackage) será creado bajo la siguiente condición:

- El nombre del paquete Java a crear será el mismo que el mismo que el del paquete UML de origen.

6.2.3.2. Regla #2: Clases

Para cada instancia de una clase UML (Class), se creará una instancia de una clase Java (JavaClass) que cumpla con las siguientes condiciones:

- En caso de que la clase UML no tenga superclase, el nombre de la clase Java derivada deberá ser el mismo. En caso de ser una extensión de otra clase, el nombre de la clase Java será igual al nombre de la clase de la cual deriva concatenado al de la subclase correspondiente mediante los caracteres ' :: ', quedando de la siguiente forma: $nombre(claseUML)$ :: $nombre(superClaseUML)$.
- El tipo de clase deberá corresponder (ambas clases abstractas o ambos no abstractas).
- La visibilidad de ambas clases deberá ser la misma.
- La referencia al paquete deberá mantenerse, es decir si una clase UML pertenece a un paquete UML llamado P1, entonces se deberá crear un paquete Java de nombre P1 que contenga a la clase Java generada.
- La superclase deberá corresponderse, es decir si una clase UML tiene una superclase S1, entonces deberá crearse una clase Java adicional, de nombre S1, que sea superclase de la clase generada.

6.2.3.3. Regla #3: Tipos de dato

Para cada tipo de dato UML, se creará la instancia del tipo de dato Java correspondiente, con las siguientes condiciones:

- Los nombres deberán ser los mismos.
- La referencia al paquete deberá mantenerse.
- En el caso de literales de una enumeración, deberán generarse las constantes Java correspondientes con el mismo valor.

6.2.3.4. Regla #4: Atributos

Para cada instancia de atributo UML, un campo Java será creado bajo las siguientes condiciones:

- Los nombres deberán corresponderse.
- Los tipos deberán corresponderse.
- La clase contenedora del atributo deberá corresponderse, es decir, si el atributo pertenece a una clase UML llamada C1, entonces se deberá crear una clase Java llamada C1 que contenga el campo a generarse.
- La visibilidad del campo Java deberá ser la misma que la del atributo UML.

Además del campo Java, cada atributo UML generara los métodos *setter* y *getter* correspondientes:

- El método *getter* tendrá el nombre del atributo precedido por la cadena '*get_*'. Poseerá la misma visibilidad del atributo, y su tipo se corresponderá con el tipo del atributo.
- El método *setter* tendrá el nombre del atributo precedido por la cadena '*set_*' y poseerá su misma visibilidad. Tendrá una parámetro de entrada cuyo nombre será el mismo que el del atributo precedido por la cadena '*p_*', y será del mismo tipo que el atributo. El tipo del método *setter* será nulo.

6.2.3.5. Regla #5: Operaciones y parámetros

Para cada instancia de una operación UML, se generará un método Java que satisfaga las siguientes condiciones:

- El nombre será el mismo.
- El tipo del método Java deberá corresponderse con el tipo de la operación UML.
- La visibilidad del método deberá ser la misma que la de la operación.
- La clase contenedora del método Java deberá corresponderse con la de la operación UML, es decir, si ésta pertenece a una clase UML llamada C1, entonces se deberá crear una clase Java llamada C1 que contenga el método a generarse.
- Los parámetros del método Java deberán corresponderse con los parámetros de la operación UML, si ésta los tuviera. En cuyo caso, los parámetros deberán:
  - Tener el mismo nombre.
  - Corresponderse en tipo.

6.2.4. Código fuente QVT

En base a las reglas definidas, se diseñó el código fuente QVT que presentaremos a continuación. Para facilitar la comprensión del mismo, lo explicaremos por grupo de relaciones según las reglas que implementen. La transformación definida se llama *uml2java* y recibe como entrada un modelo MySimpleUML. Como resultado de la transformación, se genera una nueva instancia del modelo MySimpleJava.

```
transformation uml2java (uml:MySimpleUML, java:MySimpleJava)
```

6.2.4.1. Relaciones asociadas con la Regla #1

Para la implementación de la regla de mapeo de paquetes se definieron tres relaciones (Figura 6.3) de tipo *top-level* (ejecución obligatoria). Dichas relaciones contemplan las tres posibles situaciones de una entidad Package del modelo MySimpleUML, a saber:

- Relación *UMLPackageToJavaPackage*: Contempla el caso de un paquete general, donde se especifica el mapeo del nombre del paquete UML en el nombre del paquete Java.
- Relación *UMLSecondLevelPackageToJavaSecondLevelPackage*: Implementa la transformación de un paquete de segundo nivel u hoja, es decir, aquellos paquetes que no tienen paquetes anidados, pero no son de nivel superior ya que pertenecen a otro paquete (atributo *owner* con valor definido).
- Relación *UMLInnerPackageToJavaInnerPackage*: Especifica el mapeo de paquetes internos, es decir, paquetes que tienen paquetes anidados y además no son de nivel superior, es decir, están incluidos dentro de otro paquete.

6.2.4.2. Relaciones asociadas con la Regla #2

A continuación analizaremos las relaciones involucradas en la transformación de una clase UML en una clase Java. Como vemos en la Figura 6.4, este mapeo implica la definición de dos relaciones:

- Relación *UMLClassToJavaClass*: Contempla el caso de aquellas clases sin superclase (atributo *superClass* indefinido). Realiza el mapeo de una clase UML en una clase Java manteniendo el nombre, la visibilidad y el tipo de clase (abstracta o no abstracta), además de asegurar la transformación de atributos en campos (1) y de operaciones en métodos (2), cuyos casos analizaremos más adelante.
- Relación *UMLSuperClassToJavaExtendClass*: Implementa el mapeo de clases para aquellas que tienen una superclase (atributo *superClass* definido), la cual a su vez puede tener o no su respectiva superclase.

```
top relation UMLPackageToJavaPackage {
  pn : String;
  umlPackage : MySimpleUML::Package;
  checkonly domain uml p : MySimpleUML::Package {
  };
  enforce domain java j : MySimpleJava::JavaPackage {
    name = pn
  };
  where {
    PmlPackage = p;
    pn = getPackageName(umlPackage);
  }
}

top relation UMLSecondLevelPackageToJavaSecondLevelPackage {
  pn : String;
  umlPackage : MySimpleUML::Package;
  checkonly domain uml p : MySimpleUML::Package {
    name = pn,
    owner = uop : MySimpleUML::Package {}
  };
  enforce domain java j : MySimpleJava::JavaPackage {
    name = pn,
    owner = jop : MySimpleJava::JavaPackage {}
  };
  when {
    UMLPackageToJavaPackage(uop,jop);
    UMLPackageToJavaPackage(p,j);
  }
  where {
    umlPackage = p;
    pn = getPackageName(umlPackage);
  }
}

top relation UMLInnerPackageToJavaInnerPackage {
  pn : String;
  umlPackage : MySimpleUML::Package;
  checkonly domain uml p : MySimpleUML::Package {
    name = pn,
    owner = uop : MySimpleUML::Package {},
    nestedPackage = unp : MySimpleUML::Package {}
  };
  enforce domain java j : MySimpleJava::JavaPackage {
    name = pn,
    owner = jop : MySimpleJava::JavaPackage {},
    subpack = jsp : MySimpleJava::JavaPackage {}
  };
  when {
    UMLPackageToJavaPackage(uop,jop);
    UMLPackageToJavaPackage(unp,jsp);
    UMLPackageToJavaPackage(p,j);
  }
  where {
    umlPackage = p;
    pn = getPackageName(umlPackage);
  }

}
```

Figura 6.3: Relaciones asociadas a la transformación de paquetes UML

```
top relation UMLClassToJavaClass {
  cn, vn : String;
  ia : Boolean;
  checkonly domain uml c : MySimpleUML::Class {
    owner = up : MySimpleUML::Package {},
    name = cn,
    isAbstract = ia,
    visibility = vn
  };
  enforce domain java j : MySimpleJava::JavaClass {
    owner = jp : MySimpleJava::JavaPackage {},
    name = cn,
    isAbstract = ia,
    visibility = vn
  };
  when {
    UMLPackageToJavaPackage(up,jp);
    c.superClass.oclIsUndefined();
  }
  where {
    AttributeToField(c,j);                                    (1)
    OperationToMethod(c,j);                                   (2)
  }
}

top relation UMLSuperClassToJavaExtendClass {
  pn, sn, cn, vn : String;
  ia : Boolean;
  checkonly domain uml c : MySimpleUML::Class {
    owner = up : MySimpleUML::Package {},
    superClass = sc : MySimpleUML::Class {
      name = sn
    },
    name = cn,
    isAbstract = ia,
    visibility = vn
  };
  enforce domain java j : MySimpleJava::JavaClass {
    owner = jp : MySimpleJava::JavaPackage {},
    extend = je : MySimpleJava::JavaClass {},
    name = jn,
    isAbstract = ia,
    visibility = vn
  };
  when {
    UMLPackageToJavaPackage(up,jp);
    UMLClassToJavaClass(sc,je) or UMLSuperClassToJavaExtendClass(sc,je);
  }
  where {
    jn = cn + '::' + sn;                                      (3)
    AttributeToField(c,j,cn);
    AttributeToField(sc,j,cn) or AttributeToFieldExtend(sc,j,cn);
    OperationToMethod(c,j,cn);
    OperationToMethod(sc,j,cn) or OperationToMethodExtend(sc,j,cn);
  }
}
```

Figura 6.4: Mapeo de clases UML en clases Java

6.2.4.3. Relaciones asociadas con la Regla #3

Para el mapeo de tipos de dato UML en tipos de dato Java se definieron las relaciones listadas en la Figura 6.5. A continuación analizaremos cada una de ellas:

- Relación *UMLPrimitiveTypeToJavaPrimitiveType*: Realiza la transformación de un tipo primitivo UML a un tipo primitivo Java.
- Relación *UMLEnumToJavaEnum*: Implementa la transformación de una enumeración UML en una enumeración Java.
- Relación *LiteralToConstant*: Especifica el mapeo de un literal UML a una constante Java.

6.2.4.4. Relaciones asociadas con la Regla #4

Para la transformación de atributos UML en campos Java se definieron cuatro relaciones, de acuerdo a los tres posibles tipos que un atributo puede adoptar dentro de este modelo simplificado, los cuales son primitivo, enumeración o clase.

La primera relación, llamada *AttributeToField*, define las distintas posibilidades de transformación de un atributo UML en un campo Java según su tipo (Figura 6.6). A continuación, las siguientes tres relaciones se encargan de los mapeos correspondientes según el tipo de atributo a generar:

- Relación *PrimitiveAttributeToField*: Realiza el mapeo de un atributo de tipo primitivo en un campo Java de tipo primitivo (Figura 6.7).
- Relación *EnumerationAttributeToField*: Implementa la transformación de un atributo de tipo enumeración en un campo Java del tipo correspondiente (Figura 6.8).
- Relación *ClassAttributeToField*: Especifica el mapeo de una atributo clase en un campo Java de tipo clase (Figura 6.9).

6.2.4.5. Relaciones asociadas con la Regla #5

El mapeo de operaciones UML en métodos Java parte de la relación *OperationToMethod*, la cual especifica dicha transformación (operación UML en método Java) de acuerdo a los posibles tipos que éstas puedan adoptar (Figura 6.10). En el caso simplificado de nuestro ejemplo, las operaciones pueden ser de tipo primitivo, enumerado o clase UML.

Posteriormente, las tres relaciones indicadas a continuación abordan cada caso puntual, realizando las conversiones correspondientes. Estas son:

- Relación *PrimitiveOperationToMethod*: Implementa la transformación de una operación de tipo primitiva en un método Java del tipo correspondiente (Figura 6.11).

```
top relation UMLPrimitiveTypeToJavaPrimitiveType {
  tn : String;
  checkonly domain uml upt : MySimpleUML::PrimitiveType {
    name = tn,
    owner = up : MySimpleUML::Package {}
  };
  enforce domain java jpt : MySimpleJava::JavaPrimitiveType {
    name = tn,
    owner = jp : MySimpleJava::JavaPackage {}
  };
  when {
    UMLPackageToJavaPackage(up,jp);
  }
}


top relation UMLEnumToJavaEnum {
  en : String;
  checkonly domain uml ue : MySimpleUML::Enumeration {
    name = en,
    owner = up : MySimpleUML::Package {}
  };
  enforce domain java je : MySimpleJava::JavaEnum {
    name = en,
    owner = jp : MySimpleJava::JavaPackage {}
  };
  when {
    UMLPackageToJavaPackage(up,jp);
  }
  where {
    LiteralToConstant(ue,je);
  L
}


relation LiteralToConstant {
  vn : String;
  checkonly domain uml e : MySimpleUML::Enumeration {
    literal = l : MySimpleUML::EnumLiteral {
      value = vn,
      owner = e
    }
  };
  enforce domain java je : MySimpleJava::JavaEnum {
    const = c : MySimpleJava::JavaEnumConst {
      value = vn,
      owner = je
    }
  };
}
```

Figura 6.5: Relaciones asociadas con la Regla #3

```
relation AttributeToField {
  checkonly domain uml c : MySimpleUML::Class {
  };
  enforce domain java j : MySimpleJava::JavaClass {
  };
  primitive domain className : String;
  where {
    PrimitiveAttributeToField(c,j,className);
    EnumerationAttributeToField(c,j,className);
    ClassAttributeToField(c,j,className);
  }
}
```

Figura 6.6: Mapeo de atributos UML en campos Java

```
relation PrimitiveAttributeToField {
  an, vn, mn, fn, cn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    attribute = a : MySimpleUML::Attribute {
      name = an,
      visibility = vn,
      type = t : MySimpleUML::PrimitiveType {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    field = f : MySimpleJava::JavaField {
      name = fn,
      visibility = vn,
      type = jpt : MySimpleJava::JavaPrimitiveType {}
    },
    method = mset : MySimpleJava::JavaMethod {
      owner = j,
      name = 'set' + mn,
      visibility = vn,
      parameter = p : MySimpleJava::JavaParameter {
        name = 'p_' + an,
        type = jpt
      }
    },
    method = mget : MySimpleJava::JavaMethod {
      owner = j,
      name = 'get' + mn,
      visibility = vn,
      type = jpt
    }
  };
  primitive domain className : String;
  when {
    UMLPrimitiveTypeToJavaPrimitiveType(t,jpt);
  }
  where {
    fn = if (className<>cn) then cn + '::' + an else an endif;
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(an) else Capitalize(an) endif;
  }
}
```

Figura 6.7: Mapeo de atributos primitivos

```
relation EnumerationAttributeToField {
  an, cn, vn, mn, fn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    attribute = a : MySimpleUML::Attribute {
      name = an,
      visibility = vn,
      type = t : MySimpleUML::Enumeration {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    field = f : MySimpleJava::JavaField {
      name = fn,
      visibility = vn,
      type = jet : MySimpleJava::JavaEnum {}
    },
    method = mset : MySimpleJava::JavaMethod {
      owner = j,
      name = 'set' + mn,
      visibility = vn,
      parameter = p : MySimpleJava::JavaParameter {
        name = 'p_' + an,
        type = jet
      }
    },
    method = mget : MySimpleJava::JavaMethod {
      owner = j,
      name = 'get' + mn,
      visibility = vn,
      type = jet
    }
  };
  primitive domain className : String;
  when {
    UMLEnumToJavaEnum(t,jet);
  }
  where {
    fn = if (className<>cn) then cn + '::' + an else an endif;
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(an) else Capitalize(an) endif;
  }
}
```

Figura 6.8: Transformación de atributos enumerados

```
relation ClassAttributeToField {
  an, vn, mn, fn, cn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    attribute a : MySimpleUML::Attribute {
      name = an,
      visibility = vn,
      type = t : MySimpleUML::Class {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    field = f : MySimpleJava::JavaField {
      name = fn,
      visibility = vn,
      type = jct : MySimpleJava::JavaClass {}
    },
    method = mset : MySimpleJava::JavaMethod {
      owner = j,
      name = 'set' + mn,
      visibility = vn,
      parameter = p : MySimpleJava::JavaParameter {
        name = 'p_' + an,
        type = jct
      }
    },
    method = mget : MySimpleJava::JavaMethod {
      owner = j,
      name = 'get' + mn,
      visibility = vn,
      type = jct
    }
  };
  primitive domain className : String;
  when {
    UMLClassToJavaClass(t,jct);
  }
  where {
    fn = if (className<>cn) then cn + '::' + an else an endif;
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(an) else Capitalize(an) endif;
  }

}
```

Figura 6.9: Conversión de atributos de tipo clase UML

```
relation OperationToMethod {
  checkonly domain uml c : MySimpleUML::Class {
  };
  enforce domain java j : MySimpleJava::JavaClass {
  };
  primitive domain className : String;
  where {
    PrimitiveOperationToMethod(c,j,className);
    EnumOperationToMethod(c,j,className);
    ClassOperationToMethod(c,j,className);
  }

}
```

Figura 6.10: Transformación de operaciones UML a métodos Java

```
relation PrimitiveOperationToMethod {
  on, vn, mn, cn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    operation = o : MySimpleUML::Operation {
      name = on,
      visibility = vn,
      type = t : MySimpleUML::PrimitiveType {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    method = m : MySimpleJava::JavaMethod {
      name = mn,
      visibility = vn,
      owner = j,
      type = jmt : MySimpleJava::JavaPrimitiveType {}
    }
  };
  primitive domain className : String;
  when {
    UMLPrimitiveTypeToJavaPrimitiveType(t,jmt);
  }
  where {
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(on) else Capitalize(on) endif;
    PrimitiveParameterToJavaParameter(o,m);
    EnumParameterToJavaParameter(o,m);
    ClassParameterToJavaParameter(o,m);
  }
}

relation EnumOperationToMethod {
  on, vn, cn, mn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    operation = o : MySimpleUML::Operation {
      name = on,
      visibility = vn,
      type = t : MySimpleUML::Enumeration {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    method = m : MySimpleJava::JavaMethod {
      name = mn,
      visibility = vn,
      owner = j,
      type = jmt : MySimpleJava::JavaEnum {}
    }
  };
  primitive domain className : String;
  when {
    UMLEnumToJavaEnum(t,jmt);
  }
  where {
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(on) else Capitalize(on) endif;
    PrimitiveParameterToJavaParameter(o,m);
    EnumParameterToJavaParameter(o,m);
    ClassParameterToJavaParameter(o,m);
  }
}
```

Figura 6.11: Mapeo de operaciones UML de tipo *Primitive* y *Enumeration*

```
relation ClassOperationToMethod {
  on, vn, cn, mn : String;
  checkonly domain uml c : MySimpleUML::Class {
    name = cn,
    operation = o : MySimpleUML::Operation {
      name = on,
      visibility = vn,
      type = t : MySimpleUML::Class {}
    }
  };
  enforce domain java j : MySimpleJava::JavaClass {
    method = m : MySimpleJava::JavaMethod {
      name = mn,
      visibility = vn,
      owner = j,
      type = jmt : MySimpleJava::JavaClass {}
    }
  };
  primitive domain className : String;
  when {
    UMLClassToJavaClass(t,jmt);
  }
  where {
    mn = if (className<>cn) then Capitalize(cn) + Capitalize(on) else Capitalize(on) endif;
    PrimitiveParameterToJavaParameter(o,m);
    EnumParameterToJavaParameter(o,m);
    ClassParameterToJavaParameter(o,m);
  }
}
```

Figura 6.12: Mapeo de operaciones UML de tipo *Class*

- Relación *EnumOperationToMethod*: Detalla la transformación de una operación de tipo enumeración en un método Java del tipo correspondiente (Figura 6.11).
- Relación *ClassOperationToMethod*: Define el mapeo de una operación de tipo clase UML en un método Java del tipo correspondiente (Figura 6.12).

Finalmente, se definieron tres relaciones adicionales para contemplar los distintos tipos que pueden adoptar los parámetros de entrada de las operaciones (Figura 6.13). A continuación detallaremos cada una:

- Relación *PrimitiveParameterToJavaParameter*: Especifica la transformación de un parámetro primitivo de una operación determinada en un parámetro del método Java correspondiente.
- Relación *EnumParameterToJavaParameter*: Implementa el mapeo de un parámetro de tipo enumeración de una determinada operación en un parámetro del método Java correspondiente.
- Relación *ClassParameterToJavaParameter*: Define la transformación de un parámetro de tipo clase UML de una determinada operación en un parámetro del método Java correspondiente.

6.2.5. Ejemplo de transformación

A efectos de ilustrar el funcionamiento de la transformación, en el presente apartado mostraremos un ejemplo de funcionamiento de la misma. La Figura 6.14 presenta una posible instancia de nuestro modelo MySimpleUML, el cual representa un paquete llamado *Personas*, el cual tiene definido dos tipos primitivos (entidades *PrimitiveType*): *Integer* y *String*, y un tipo enumerado (entidad *Enumeration*) llamada *DiasSemana*, la cual define un conjunto de cinco posibles valores, los cuales representan los días de la semana de Lunes (Lun) a Viernes (Vie).

El modelo presenta además dos clases (entidad *Class*): *Persona* y *Empresa*. La clase *Persona* presenta un atributo (entidad *Attribute*) llamado *nombre*, de tipo primitivo *String*. La clase *Empresa* posee un atributo *diasLaborables*, de tipo enumerado *DiasSemana*, y una operación (entidad *Operation*) llamada *getNroEmpleados*, de tipo primitivo *Integer*.

Una vez ejecutada la transformación, obtenemos el modelo de la Figura 6.15. El paquete Personas ahora es de tipo *JavaPackage*, y mantiene el mismo nombre. Los tipos primitivos *Integer* y *String* son ahora elementos *JavaPrimitiveType*, y la enumeración es de tipo *JavaEnum*. Los valores literales son entidades *JavaEnumConst*, pero han conservado sus mismos valores. La clase Persona es ahora una *JavaClass*, con idéntico nombre, y presenta un atributo *JavaField* llamado *nombre*. Como novedad se han agregado dos métodos *JavaMethod*: *getNombre* y *setNombre*, correspondientes a los *getters* y *setters* del atributo *nombre*. El método *setNombre* requiere un parámetro *p_nombre*, de tipo primitivo *String*, mientras que el método *getNombre* es de tipo primitivo *String*, y no posee parámetros.

```
relation PrimitiveParameterToJavaParameter {
  pn : String;
  checkonly domain uml uo : MySimpleUML::Operation {
    parameter = up : MySimpleUML::Parameter {
      name = pn,
      type = t : MySimpleUML::PrimitiveType {}
    }
  };
  enforce domain java jm : MySimpleJava::JavaMethod {
    parameter = jp : MySimpleJava::JavaParameter {
      name = pn,
      type = jpt : MySimpleJava::JavaPrimitiveType {}
    }
  };
  when {
    UMLPrimitiveTypeToJavaPrimitiveType(t,jpt);
  }
}
relation EnumParameterToJavaParameter {
  pn : String;
  checkonly domain uml uo : MySimpleUML::Operation {
    parameter = up : MySimpleUML::Parameter {
      name = pn,
      type = t : MySimpleUML::Enumeration {}
    }
  };
  enforce domain java jm : MySimpleJava::JavaMethod {
    parameter = jp : MySimpleJava::JavaParameter {
      name = pn,
      type = jpt : MySimpleJava::JavaEnum {}
    }
  };
  when {
    UMLEnumToJavaEnum(t,jpt);
  }
}
relation ClassParameterToJavaParameter {
  pn : String;
  checkonly domain uml uo : MySimpleUML::Operation {
    parameter = up : MySimpleUML::Parameter {
      name = pn,
      type = t : MySimpleUML::Class {}
    }
  };
  enforce domain java jm : MySimpleJava::JavaMethod {
    parameter = jp : MySimpleJava::JavaParameter {
      name = pn,
      type = jpt : MySimpleJava::JavaClass {}
    }
  };
  when {
    UMLClassToJavaClass(t,jpt);
  }
}
```

Figura 6.13: Transformación de parámetros de operaciones según su tipo

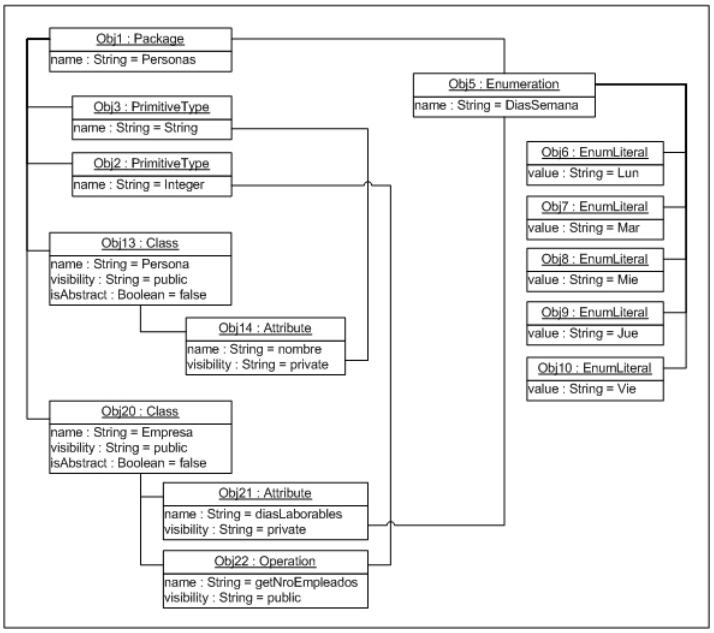


Figura 6.14: Modelo de entrada: una instancia de MySimpleUML

Por último, en el modelo de salida tenemos la clase Empresa como entidad *JavaClass*, con su atributo *diasLaborables* ahora como *JavaField*. Presenta los métodos *getDiasLaborables* y *setDiasLaborables*, este último con su correspondiente parámetro *p_ diasLaborables*, de tipo *JavaEnum*, y finalmente la operación *getNroEmpleados* la cual se ha convertido en un *JavaMethod* del mismo nombre.

6.2.6. Inferencia de trazas

Tomando como entradas los metamodelos origen (MySimpleUML) y destino (MySimpleJava), más la definición de la transformación el mecanismo de inferencia de trazas desarrollado, Análisis Basado en Variables, nos permite inferir el conjunto de trazas que detallaremos a continuación:

6.2.6.1. Relación *UMLPackageToJavaPackage*

Considerando los casos de traza descriptos en la Sección 5.3.2, vemos que la relación *UMLPackageToJavaPackage* nos permite inferir una traza encuadrada en el Caso #1 (ver Subsección 5.3.2.1), es decir, inferencia mediante una variable auxiliar. Este caso nos indica que cuando una regla obligatoria (*top-level*) asigne un valor a algún elemento del modelo destino definido en el ámbito de un dominio de tipo *enforce*, mediante el uso de una variable previamente utilizada sobre un elemento del modelo origen de igual manera, definida en el ámbito de un dominio *checkonly*, podremos decir que el elemento del modelo origen mapeada directamente en el elemento del modelo destino.

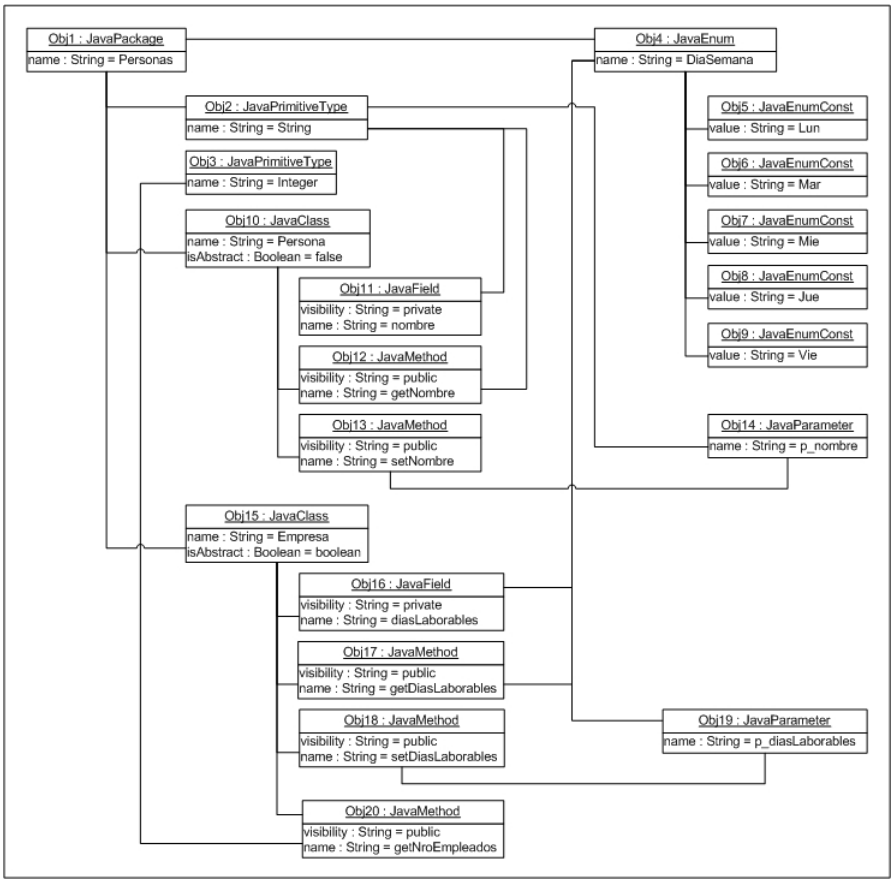


Figura 6.15: Salida de la transformación: una instancia MySimpleJava

En este caso, existe una variable *pn* de tipo *String* la cual asigna valor a un elemento del modelo destino (atributo *name* de la entidad *JavaPackage*) en el ámbito de un dominio *enforce j* de tipo *JavaPackage*, y ésta misma es también utilizada de la misma manera sobre un elemento del modelo origen (atributo *name* de la entidad *Package*) en el ámbito de un dominio *checkonly p* de tipo *Package*. Por lo tanto, podemos decir que existe una traza $name :: Package \rightarrow name :: JavaPackage$.

Observando el código de la transformación completa podemos ver que ésta misma traza puede ser inferida de las relaciones *UMLSecondLevelPackageToJavaSecondLevelPackage* o *UMLInnerPackageToJavaInnerPackage* dado que el tratamiento dado al atributo *name* es el mismo en las tres. Lo que varía en este caso es la caracterización del dominio de entrada (*checkonly*), es decir, en un caso tenemos un paquete raíz, sin sucesores ni predecesores, mientras que en el caso de la segunda relación se contempla el caso de paquetes con un predecesor, pero sin subpaquetes anidados, y que por último tenemos el caso de los paquetes con subpaquetes anidados y que poseen un predecesor.

6.2.6.2. Relación *UMLClassToJavaClass*

A partir de esta relación podemos inferir tres trazas, relacionadas con el mapeo de los elementos propios de una clase UML. Al igual que en la relación anterior, por aplicación del Caso #1 (Subsección 5.3.2.1) inferimos los siguientes enlaces:

- $name :: Class \rightarrow name :: JavaClass$, donde $name :: JavaClass = name :: Class$
- $isAbstract :: Class \rightarrow isAbstract :: JavaClass$, donde $isAbstract :: JavaClass = isAbstract :: Class$
- $visibility :: Class \rightarrow visibility :: JavaClass$, donde $visibility :: JavaClass = visibility :: Class$

En esencia, estas trazas plantean que como resultado de la transformación el nombre de una clase Java será el nombre de la clase UML de la cual deriva, al igual que su visibilidad (pública, privada, protegida, etc) y su tipo (si es abstracta o no).

6.2.6.3. Relación *UMLSuperClassToJavaExtendClass*

Al igual que la relación anterior, ésta posibilita la inferencia de trazas relacionadas con los componentes propios de una clase, como el nombre, la visibilidad y si la misma es abstracta o no.

No obstante, a diferencia de su predecesora, esta relación deriva el nombre de una clase Java de manera distinta. Tendremos aquí una multitraza, dado que el origen de la traza está dado por dos elementos del modelo fuente, que establece que el nombre de una clase Java derivada de una clase UML que extiende a otra será igual a la composición del nombre de la nombre clase UML ($name_c$) con

el de la superclase UML ($name_s$), concatenadas mediante el separador '::' (Ver sentencia 3 en Figura 6.4).

La traza resultante está encuadrada en el Caso #4, de inferencia mediante variable auxiliar definida como una función en la cláusula *Where* de la relación. En este caso, el valor del elemento destino está dado por una función de n variables asignadas en el ámbito de un dominio *checkonly*, la cual se define en la sección de post-condiciones (cláusula *where*).

En síntesis, de esta relación se derivarán las siguientes trazas:

- $name_c :: Class, name_s :: Class \rightarrow name :: JavaClass$, donde $name :: JavaClass = name_c :: Class + \text{`} :: \text{'} + name_s(Class)$
- $isAbstract :: Class \rightarrow isAbstract :: JavaClass$, donde $isAbstract :: JavaClass = isAbstract :: Class$
- $visibility :: Class \rightarrow visibility :: JavaClass$, donde $visibility :: JavaClass = visibility :: Class$

6.2.6.4. Relación *UMLPrimitiveTypeToJavaPrimitiveType*

El mapeo de tipos primitivos UML en tipos primitivos Java nos deja como resultado la traza, derivada por aplicación del Caso #1 (ver Subsección 5.3.2.1), que relaciona el atributo *name* de un elemento *JavaPrimitiveType*, del modelo MySimpleJava, con el atributo *name* de un elemento *PrimitiveType* del modelo MySimpleUML. La traza generada en este caso será:

- $name :: PrimitiveType \rightarrow name :: JavaPrimitiveType$, donde $name :: JavaPrimitiveType = name :: PrimitiveType$

6.2.6.5. Relación *UMLEnumToJavaEnum*

Al igual que la relación anterior, ésta permite inferir la traza que especifica el mapeo del nombre de una enumeración UML (atributo *name* del elemento *Enumeration*, del modelo MySimpleUML) en el nombre de una enumeración Java (atributo *name* del elemento *JavaEnum*, del modelo MySimpleJava). Por aplicación del Caso #1 de inferencia de trazas (Subsección 5.3.2.1), la traza resultante de la relación es la siguiente:

- $name :: Enumeration \rightarrow name :: JavaEnum$, donde $name :: JavaEnum = name :: Enumeration$

6.2.6.6. Relación *LiteralToConstant*

La siguiente relación especifica la transformación de un literal en una constante Java de enumeración. Nuevamente por aplicación del Caso #1 (Subsección 5.3.2.1), el algoritmo permite la inferencia de la siguiente traza:

- $value :: EnumLiteral \rightarrow value :: JavaEnumConst$, donde $value :: JavaEnumConst = value :: EnumLiteral$

6.2.6.7. Relación *PrimitiveAttributeToField*

Esta relación, la cual detalla el mapeo de un atributo primitivo de una clase UML en un campo primitivo de una clase Java, nos permite identificar varias trazas. A continuación presentamos las que son derivadas por aplicación del Caso #1 de inferencia de trazas (Subsección 5.3.2.1):

- $visibility :: Attribute \rightarrow visibility :: JavaField$, donde el valor del atributo $visibility :: JavaField = visibility :: Attribute$
- $visibility :: Attribute \rightarrow visibility :: Method$, donde el valor del atributo $visibility :: JavaMethod = visibility :: Attribute$

La primer traza nos indica que para toda tupla (*atributo*, *campo*) que satisfaga la relación, se verificará que la visibilidad del campo Java será la visibilidad del atributo UML. Es decir, por ejemplo, si el atributo UML original es un elemento "público", entonces el campo Java resultante será "público". La segunda traza nos indica que todo método generado a partir de dicho atributo, en éste caso son los métodos *getter* y *setter*, tendrán su misma visibilidad.

Esta relación nos permite también derivar una traza por aplicación del Caso #2 de inferencia de trazas (Subsección 5.3.2.2), a saber:

- $name :: Attribute \rightarrow name :: JavaParameter$, donde el valor del nombre del parámetro $name :: JavaParameter = `p\_' + name :: Attribute$

Esta traza indica que el nombre de todo parámetro generado a partir de un atributo UML, que en esta relación corresponde al parámetro generado en el método *setter* asociado a todo atributo UML, será conformado mediante la concatenación del nombre del atributo original con el prefijo 'p_'.

Por último, la relación nos permite derivar dos trazas más, de tipo condicionales, por aplicación del Caso #5 (Subsección 5.3.2.5), de inferencia de trazas en sentencias *If-Then-Else*. En este caso, se trata de la generación del nombre de un campo Java y del nombre de los métodos *getter* y *setter* correspondientes. Por razones de legibilidad, en adelante el nombre de la función $Capitalize()$ será reemplazado por $Cap()$.

Las trazas obtenidas entonces son:

- $name :: Class$, $name :: Attribute \,|\, name :: Attribute \rightarrow name :: JavaField$, donde $name :: JavaField = name :: Class + ` :: ' + name :: Attribute \oplus name :: JavaField = name :: Attribute$
- $name :: Class$, $name :: Attribute | name :: Attribute \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = `get\_' + Cap(name :: Class) + Cap(name :: Attibute) \oplus name :: JavaMethod = `get\_' + Cap(name :: Attribute)$
- $name :: Class$, $name :: Attribute | name :: Attribute \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = `set\_' + Cap(name :: Class) + Cap(name :: Attibute) \oplus name :: JavaMethod = `set\_' + Cap(name :: Attribute)$

En esencia, la primera de las trazas nos indica que todo elemento de tipo *JavaField* tendrá el nombre del elemento Attribute que le da origen y eventualmente, en caso de ser un atributo heredado, llevará además el nombre de la clase original donde pertenece.

Por otro lado, la segunda traza nos dice que todo elemento JavaMethod correspondiente al *getter* de un campo Java, originado a partir de un elemento Attribute poseerá su mismo nombre (capitalizado[1]), y eventualmente, en caso de ser heredado, llevará además el nombre de la clase original. Finalmente, la tercer traza es similar a la anterior, sólo que aplica para el método *setter* del campo Java.

6.2.6.8. Relaciones *EnumerationAttributeToField* y *ClassAttributeToField*

Dado que estas dos relaciones son análogas a la relación *PrimitiveAttributeToField* recientemente analizada, el conjunto de trazas generadas en cada caso será similar al de la relación mencionada. Naturalmente, por provenir de relaciones distintas, las trazas si bien estructuralmente son idénticas (involucran los mismos elementos del modelo origen y destino) corresponden a nuevas trazas, que tienen sentido bajo condiciones diferentes. En un caso corresponden al mapeo de un atributo de tipo primitivo sobre un campo Java de tipo primitivo, en otro corresponden al mapeo de atributos de tipo enumeración, y por último a la transformación de atributos de tipo clase.

De ésta manera, por lo expuesto, tanto las trazas generadas por la relación *EnumerationAttributeToField* como las generadas por la relación *ClassAttributeToField* tienen la siguiente estructura:

- $visibility :: Attribute \rightarrow visibility :: JavaField$, donde el valor de $visibility :: JavaField = visibility :: Attribute$
- $visibility :: Attribute \rightarrow visibility :: JavaMethod$, donde el valor de $visibility :: JavaMethod = visibility :: Attribute$
- $name :: Attribute \rightarrow name :: JavaParameter$, donde el valor de $name :: JavaParameter = `p\_' + name :: Attribute$
- $name :: Class, name :: Attribute \,|\, name :: Attribute \rightarrow name :: JavaField$, donde $name :: JavaField = name :: Class + ` :: ' + name :: Attribute \oplus name :: JavaField = name :: Attribute$
- $name :: Class, name :: Attribute \,|\, name :: Attribute \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = `get\_'+Cap(name :: Class)+Cap(name :: Attibute) \oplus name :: JavaMethod = `get\_' + Cap(name :: Attribute)$
- $name :: Class, name :: Attribute \,|\, name :: Attribute \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = `set\_'+Cap(name :: Class)+Cap(name :: Attibute) \oplus name :: JavaMethod = `set\_' + Cap(name :: Attribute)$

[1] La función de capitalizado toma como entrada una cadena de caracteres y devuelve la misma cadena, con la primer letra en mayúscula. Es decir Capitalizar("palabra") = "Palabra".

6.2.6.9. Relación *PrimitiveOperationToMethod*

La relación *PrimitiveOperationToMethod* lleva a cabo el mapeo de operaciones de tipo primitivo (aquellas que retornan un tipo primitivo) en métodos Java. Por aplicación del Caso #1 podemos inferir de la misma la siguiente traza:

- $visibility :: Operation \rightarrow visibility :: JavaMethod$, donde $visibility :: JavaMethod = visibility :: Operation$

En este caso, la traza nos indica que todo método Java originado a partir de una operación UML poseerá su misma visibilidad.

La siguiente traza inferida a partir de la relación se basa en proceso de conformación del nombre del método Java. Por aplicación del Caso #5, la traza inferida tiene la siguiente forma:

- $name :: Class, name :: Operation \,|\, name :: Operation \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = Cap(name :: Class) + Cap(name :: Operation) \oplus name :: JavaMethod = Cap(name :: Operation)$

En esencia la traza nos dice que el nombre de un método Java derivado a partir de una operación UML puede ser la composición del nombre de la clase (capitalizado) junto con el nombre de la operación (capitalizado), o bien el nombre de la operación (capitalizado). La condición que determina esta traza condicional tiene que ver con el origen de la operación a transformar, esto es, si es nativa de la clase o si es heredada.

6.2.6.10. Relaciones *EnumOperationToMethod* y *ClassOperationToMethod*

Para estas relaciones, como sucede con las reglas de transformación de atributos, las trazas inferidas tienen la misma estructura que las trazas derivadas a partir de la relación *PrimitiveOperationToMethod*. De esta manera, por aplicación de Casos #1 y #5 respectivamente, para las dos relaciones, se pueden inferir las siguientes trazas:

- $visibility :: Operation \rightarrow visibility :: JavaMethod$, donde $visibility :: JavaMethod = visibility :: Operation$
- $name :: Class, name :: Operation \,|\, name :: Operation \rightarrow name :: JavaMethod$, donde $name :: JavaMethod = Cap(name :: Class) + Cap(name :: Operation) \oplus name :: JavaMethod = Cap(name :: Operation)$

6.2.6.11. Relación *PrimitiveParameterToJavaParameter*

Esta relación se encarga de la transformación de los parámetros de una operación, cuando éstos son de tipo primitivo. Por aplicación del Caso #1 (Subsección 5.3.2.1), podemos inferir la siguiente traza:

- $name :: Parameter \rightarrow name :: JavaParameter$, donde el nombre del parámetro $name :: JavaParameter = name :: Parameter$

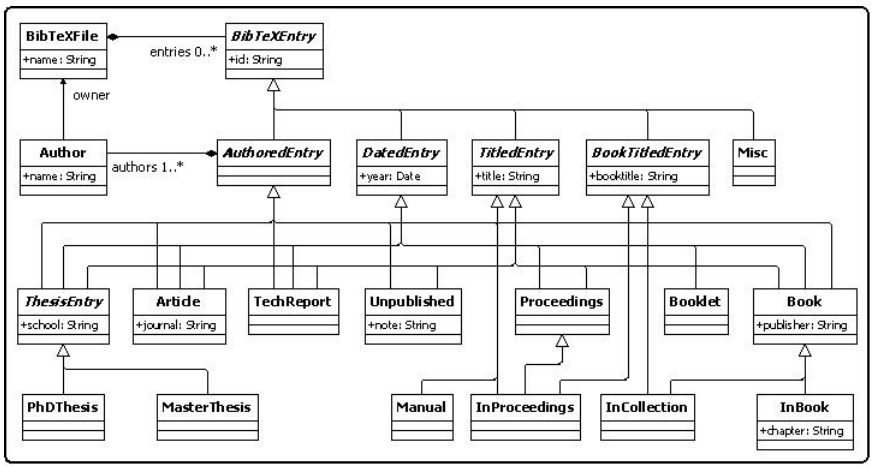


Figura 6.16: Metamodelo BibTeXML

6.2.6.12. Relaciones *EnumParameterToJavaParameter* y *ClassParameterToJavaParameter*

Al igual que sucede con las relaciones encargadas del mapeo de atributos, en las relaciones que llevan a cabo la transformación de parámetros de operaciones se vuelven a presentar trazas con la misma estructura. Tendremos entonces que las relaciones *EnumParameterToJavaParameter* y *ClassParameterToJavaParameter* permiten derivar trazas de la siguiente forma:

- $name :: Parameter \rightarrow name :: JavaParameter$, donde el nombre del parámetro $name :: JavaParameter = name :: Parameter$

6.3. Caso #2: La transformación Bib2Doc

A continuación presentamos nuestro segundo caso de estudio, la transformación de una especificación bibliográfica realizada en BibTeXML a formato Docbook. En primer lugar se describirán los metamodelos de entrada y salida, y luego se detallarán las reglas que componen la transformación. Por último analizaremos las trazas inferidas a partir del análisis basado en variables.

6.3.1. Modelo de entrada: BibTeXML

BibTeXML [4] es un esquema basado en XML que expresa el contenido del modelo BibTeX, el sistema de documentación bibliográfica utilizado en herramientas de edición de textos como LaTeX, ampliamente utilizado por la comunidad científica.

La presente transformación se basa en un metamodelo BibTeXML simplificado, el cual incluye sólo alguno de los campos obligatorios de una entrada BibTeX. Por ejemplo, una entrada de tipo *artículo* tiene como campos requeridos el

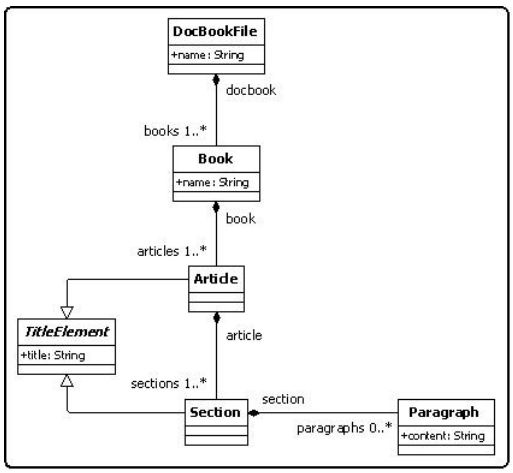


Figura 6.17: Metamodelo Docbook

autor, el título, el año de publicación y la publicación científica (*journal*) donde el mismo está incluido.

Como vemos en la Figura 6.16, una referencia bibliográfica es modelizada por un elemento BibTeXFile. Dicho elemento se compone de entradas BibTeXEntry, las cuales se encuentran identificadas mediante un ID. Todas las entradas heredan, directa o indirectamente del elemento abstracto BibTeXEntry. Las clases abstractas AuthoredEntry, DatedEntry, TitledEntry y BookTitleEntry, al igual que ThesisEntry (un nivel más abajo en la jerarquía) especifican el conjunto de campos requeridos que las entradas BibTeX concretas pueden tener.

Nuestro modelo especifica trece posibles tipos de entrada: PhdThesis, MasterThesis, Article, TechReport, Unpublished, Manual, InProceedings, Proceedings, Booklet, InCollection, Book, InBook y Misc. Por ejemplo, la entrada Proceeding posee dos campos obligatorios, los cuales son el año de publicación y el título. Por su parte, la entrada Misc no tiene campos requeridos.

6.3.2. Modelo de salida: Docbook

DocBook [6] es un esquema estándar pensado especialmente para la escritura de libros y papers. En efecto, DocBook define una serie de estructuras jerárquicas que permiten la representación de un texto escrito de manera independiente de su presentación. Para nuestro ejemplo, hemos tomado un subconjunto limitado de etiquetas o *tags* estructurales de DocBook relacionadas con la documentación de artículos.

La Figura 6.17 muestra los detalles del metamodelo de salida. Dentro del mismo, un documento DocBookFile se compone de elementos Book, los cuales a su vez se encuentran conformados por entidades Article. Estos artículos se componen de secciones Section, ordenadas, las cuales se encuentran conformadas a su vez por párrafos Paragraph, los cuales son en última instancia los

contenedores de la información (atributo *content*). Tanto los elementos DocBookFile como Book tienen asociado un nombre. Por su parte, los elementos Article y Section poseen un atributo título, heredado de la clase abstracta TitleElement.

6.3.3. Reglas de transformación

El principal objetivo de esta transformación es la conversión de un modelo BibTeXML, formado por referencias bibliográficas individuales y con distinto tipo de información asociada, en un modelo DocBook que organice los datos en cinco secciones (Sección #1 a Sección #5), agrupando referencias, autores, títulos, publicaciones y libros respectivamente.

A continuación analizaremos las reglas definidas para tal efecto.

6.3.3.1. Regla #1: Estructura del artículo

Por cada elemento *BibTeXFile*, del modelo BibTeXML, se crearán los siguientes elementos del modelo DocBook:

- Un elemento *DocBookFile*.
- Un elemento *Book*, enlazado con el elemento *DocBookFile* definido.
- Un elemento *Article*, enlazado al elemento *Book*, cuyo título sea "BibTeX to Docbook".
- Cuatro elementos *Section*, enlazados al componente *Article*, cuyos títulos sean respectivamente:
  - "References List" (lista de referencias) para la Sección #1.
  - "Authors List" (lista de autores) para la Sección #2.
  - "Titles List" (lista de títulos) para la Sección #3.
  - "Journals List" (lista de publicaciones) para la Sección #4.
  - "Books List" (lista de libros) para la Sección #5

6.3.3.2. Regla #2: Lista de autores

Por cada autor distinto (entidad *Author* del modelo BibTeXML), se generará un elemento *Paragraph* del modelo Docbook:

- Su contenido (atributo *content*) será el nombre del autor.
- Estará enlazado a la Sección #2 (lista de autores).

6.3.3.3. Regla #3: Gestión de publicaciones sin título

Por cada entrada *BibTeXEntry* sin título, es decir, entradas que no heredan de la clase *TitledEntry*, se creará un elemento *Paragraph* con las siguientes características:

- Su contenido (atributo *content*) estará formado por toda la información disponible de la entrada.
- Estará enlazado a la Sección #1 (lista de referencias).

6.3.3.4. Regla #4: Gestión de publicaciones tituladas

Por cada entrada titulada, es decir, descendiente de la clase *TitledEntry*, se generarán los siguientes componentes:

- Un elemento *Paragraph*, enlazado a la Sección #1 (lista de referencias), que contenga todo la información disponible de la entrada.
- Un elemento *Paragraph*, enlazado a la Sección #3 (lista de títulos), que contenga el título (atributo *title*) de la entrada.

6.3.3.5. Regla #5: Lista de publicaciones

Por cada artículo (entidad *Article*) del modelo BibTeXML , se creará un elemento *Paragraph* con las siguientes características:

- Su contenido (atributo *content*) estará conformado por el nombre de la publicación (*journal*) .
- Estará enlazado a la Sección #4 (lista de publicaciones).

6.3.3.6. Regla #6: Lista de libros

Por cada entrada bibliográfica que esté asociada a un libro, es decir elementos descendientes de la clase BookTitledEntry, se generará el siguiente componente:

- Un elemento *Paragraph* cuyo contenido (atributo *content*) será el nombre del libro (atributo *booktitle*).
- Dicho párrafo estará enlazado a la Sección #5 (lista de libros).

6.3.4. Código fuente QVT

A continuación presentaremos el código fuente de la transformación Bib2Doc generado según las reglas definidas.

```
top relation BibtexFileToDocbook {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc d : Docbook::DocbookFile {
  };
}

top relation BibtexFileToBook {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc bk : Docbook::Book {
    docbook = d : Docbook::DocbookFile {}
  };
  when {
    BibtexFileToDocbook(b,d);
  }
}


top relation BibtexFileToArticle {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc a : Docbook::Article {
    title = 'BibTeX to Docbook',
    book = bk : Docbook::Book {}
  };
  when {
    BibtexFileToBook(b,bk);
  }
}
```

Figura 6.18: Relaciones de generación de estructuras de artículo

6.3.4.1. Relaciones asociadas a la Regla #1 (Generación de estructura de artículo Docbook)

Esta regla incluye dos partes claramente definidas. Por un lado tenemos la formación del artículo Docbook, el cual implica la creación de una estructura *DockbookFile*, la cual contiene una entidad *Book*, y ésta a su vez contiene una clase *Article*. Esta tarea es realizada mediante las siguientes relaciones, cuyo código se encuentra detallado en la Figura 6.18:

- *BibTexFileToDocbook*.
- *BibTexFileToBook*.
- *BibTexFileToArticle*.

La segunda parte de la regla consiste en la creación de las secciones del artículo. Para esto se definieron las siguientes relaciones (Figura 6.19):

- *BibTexFileToSectionReferences*: para la creación de la Sección #1 "References List"
- *BibTexFileToSectionAuthors*: para la creación de la Sección #2 "Authors List".
- *BibTexFileToSectionTitles*: para la creación de la Sección #3 "Titles List".
- *BibTexFileToSectionJournals*: para la creación de la Sección #4 "Journals List".

```
top relation BibtexFileToSectionAuthors {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc s : Docbook::Section {
    title = 'Authors List',
    article = a : Docbook::Article {}
  };
  when {
    BibtexFileToArticle(b,a);
  }
}


top relation BibtexFileToSectionTitles {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc s : Docbook::Section {
    title = 'Titles List',
    article = a : Docbook::Article {}
  };
  when {
    BibtexFileToArticle(b,a);
  }
}


top relation BibtexFileToSectionReferences {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc s : Docbook::Section {
    title = 'References List',
    article = a : Docbook::Article {}
  };
  when {
    BibtexFileToArticle(b,a);
  }
}


top relation BibtexFileToSectionJournals {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc s : Docbook::Section {
    title = 'Journals List',
    article = a : Docbook::Article {}
  };
  when {
    BibtexFileToArticle(b,a);
  }
}


top relation BibtexFileToSectionBooks {
  checkonly domain bib b : BibTeXML::BibtexFile{
  };
  enforce domain doc s : Docbook::Section {
    title = 'Books Title List',
    article = a : Docbook::Article {}
  };
  when {
    BibtexFileToArticle(b,a);
  }
}
```

Figura 6.19: Relaciones asociadas a la generación de secciones

```
top relation AuthorToParagraph {
  an : String;
  checkonly domain bib a : BibTeXML::Author {
    name = an,
    owner = b : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = an,
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionAuthors(b,s);
  }
}
```

Figura 6.20: Generación de la lista de autores

```
top relation UntitledToParagraph {
  checkonly domain bib e : BibTeXML::BibtexEntry {
    owner = b : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = getContent(e),
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionReferences(b,s) and not e.oclIsKindOf(BibTeXML::TitledEntry);
  }
}
```

Figura 6.21: Gestión de publicaciones sin título

- *BibTexFileToSectionBooks*: para la creación de la Sección #5 "Books List".

6.3.4.2. Relaciones asociadas a la Regla #2 (Creación de la lista de autores)

Para la generación de la Sección #2, o lista de autores, se definió la relación *AuthorToParagraph*, la cual mapea las entidades *Author* del modelo BibTeXML en párrafos *Paragraph* del modelo Docbook (ver Figura 6.20), los cuales son agrupados en la sección de autores.

6.3.4.3. Relaciones asociadas a la Regla #3 (Gestión de publicaciones sin título)

La gestión de publicaciones no tituladas es realizada a través de la relación *UntitledToParagraph*, la cual realiza la transformación de toda entrada *BibtexEntry*, que no sea descendiente de la clase *TitledEntry*, en un párrafo *Paragraph* del modelo Docbook, el cual formará parte de la sección de referencias (Figura 6.21).

```
top relation TitledToSectionTitles {
  idEntry : String;
  tn : String;
  checkonly domain bib te : BibTeXML::TitledEntry {
    id = idEntry,
    title = tn,
    owner = o : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = tn,
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionTitles(o,s);
  }
}
top relation TitledToSectionReferences {
  idEntry : String;
  tn : String;
  checkonly domain bib te : BibTeXML::TitledEntry {
    id = idEntry,
    title = tn,
    owner = o : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = getContent(te),
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionReferences(o,s);
  }
}
```

Figura 6.22: Gestión de publicaciones con título

6.3.4.4. Relaciones asociadas a la Regla #4 (Gestión de publicaciones tituladas)

El mapeo de publicaciones tituladas es llevada a cabo a través de las siguientes relaciones (Figura 6.22):

- *TitledToSectionTitles*, la cual realiza la transformación de toda entidad *TitledEntry*, o sucesora, en párrafos *Paragraph* asociados a la sección de títulos, cuyo contenido consta de los nombres de los títulos de cada publicación.
- *TitledToSectionReferences*, la cual realiza la transformación de toda entidad *TitledEntry*, o sucesora, en párrafos *Paragraph* asociados a la sección de referencias, cuyo contenido agrupa toda la información disponible de la entrada.

6.3.4.5. Relaciones asociadas a la Regla #5 (Creación de la lista de publicaciones)

La generación de la lista de publicaciones, contenida en la estructura Docbook bajo la Sección #4, es realizada por la relación *ArticleToSectionJournal*, la cual mapea toda entidad *Article* en un párrafo *Paragraph*, cuyo contenido es el nombre de la publicación. La Figura 6.23 nos muestra el código fuente QVT de dicha relación.

```
top relation ArticleToSectionJournal {
  jn : String;
  checkonly domain bib te : BibTeXML::Article {
    journal = jn,
    owner = o : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = jn,
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionJournals(o,s);
  }
}
```

Figura 6.23: Creación de lista de publicaciones

```
top relation BookTitleToSectionBooks {
  bn : String;
  checkonly domain bib te : BibTeXML::BookTitledEntry {
    booktitle = bn,
    owner = o : BibTeXML::BibtexFile {}
  };
  enforce domain doc p : Docbook::Paragraph {
    content = bn,
    section = s : Docbook::Section {}
  };
  when {
    BibtexFileToSectionBooks(o,s);
  }
}
```

Figura 6.24: Generación de la lista de libros

6.3.4.6. Relaciones asociadas a la Regla #6 (Generación de la lista de libros)

Finalmente, la relación final de nuestra transformación es *BookTitleToSection-Books*, encargada de mapear entidades *BookTitleEntry* en párrafos *Paragraph*, los cuales contienen los nombre de los libros correspondientes, todos agrupados en la Sección #5, de libros (Figura 6.24).

6.3.4.7. Consultas asociadas

Como complemento de las relaciones, la transformación *Bib2Doc* incluye una consulta que recibe como parámetro una entrada *BibTex* y devuelve una cadena de caracteres formada por la concatenación de todos los atributos de la entrada, dependiendo de su tipo (Figura 6.25).

Por ejemplo, supongamos que el parámetro de la consulta es una entrada de tipo *Article*. Dado que *Article* es descendiente de *BibTexEntry*, entonces tendrá un identificador (atributo *id*) asociado. Como además todo elemento *Article* es descendiente de *TitledEntry*, el mismo tendrá un título asociado (atributo *title*). Adicionalmente, por ser un *DatedEntry*, todo artículo tendrá una fecha (atributo *year*) asociada. Por último, el atributo restante es el que indica la publicación en la cuál se encuentra el artículo (*journal*). De esta manera, la salida de la consulta *getContent()* para una entrada de tipo *Article* será la siguiente cadena de caracteres:

```
query getContent(e : BibTeXMLxmBibtexEntry) : String {
  if not e.oclIsUndefined() then
    '[' + e.id + ']'
    + if e.oclIsKindOf(BibTeXML::TitledEntry) then
      ' - ' + e.oclAsType(BibTeXML::TitledEntry).title
    else '' endif
    + if e.oclIsKindOf(BibTeXML::BookTitledEntry) then
      ' - ' + e.oclAsType(BibTeXML::BookTitledEntry).booktitle
    else '' endif
    + if e.oclIsKindOf(BibTeXML::DatedEntry) then
      ' - ' + e.oclAsType(BibTeXML::DatedEntry).year
    else '' endif
    + if e.oclIsKindOf(BibTeXML::ThesisEntry) then
      ' - ' + e.oclAsType(BibTeXML::ThesisEntry).school
    else '' endif
    + if e.oclIsKindOf(BibTeXML::Article) then
      ' - ' + e.oclAsType(BibTeXML::Article).journal
    else '' endif
    + if e.oclIsKindOf(BibTeXML::Unpublished) then
      ' - ' + e.oclAsType(BibTeXML::Unpublished).note
    else '' endif
    + if e.oclIsKindOf(BibTeXML::Book) then
      ' - ' + e.oclAsType(BibTeXML::Book).publisher
    else '' endif
    + if e.oclIsKindOf(BibTeXML::InBook) then
      ' - ' + e.oclAsType(BibTeXML::InBook).chapter
    else '' endif
  else '' endif
}
```

Figura 6.25: Consulta para obtención de contenidos

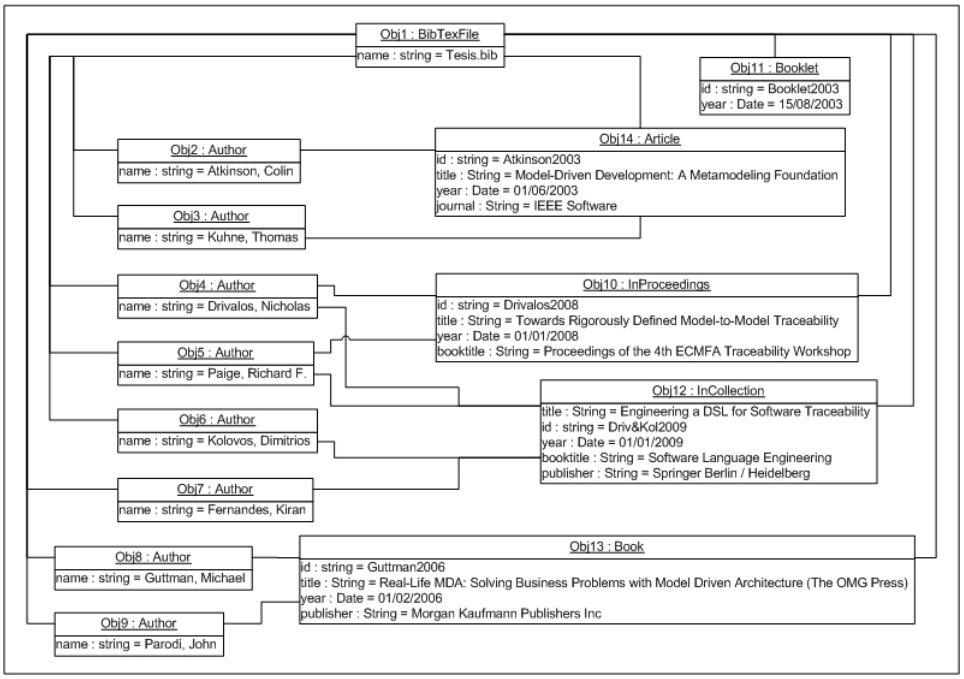


Figura 6.26: Entrada del proceso de transformación: una referencia Bibtex

```
'[' <id> ']' + '-' + <title> + '-' + <year> + '-' + <journal>
```

donde los elementos encerrados entre '<' y '>' son obviamente variables.

6.3.5. Ejemplo de trasformación

A efectos de poder dejar en claro el funcionamiento de la transformación presentaremos un ejemplo de ésta. Como vemos en la Figura 6.26, nuestra entrada a la transformación define una referencia bibliográfica BibTeX. Esta, presenta un Obj1 de tipo *BibTeXFile* que representa un archivo Tesis.bib, el cual es parte de la referencia bibliográfica de este trabajo de tesis. Este objeto tiene asociados una serie de autores (entidades *Author*) que están relacionados a las entradas de tipo *AuthoredEntry*, como veremos a continuación.

El objeto *BibTeXFile* tiene además asociadas cinco entradas (Obj10 al Obj14): un objeto *InProceeding*, uno *Booklet*, uno *InCollection*, un *Book*, y un *Article*. Todos ellos, a excepción del objeto *Booklet*, son entradas de tipo AuthoredEntry, por lo que tienen autores asociados, además de sus atributos propios.

La Figura 6.27 muestra el resultado de la transformación: un objecto *Docbook*, con un *Book* asociado el cual, a su vez, posee un elemento *Article* con cinco secciones *Section*. Las mismas están conformadas de la siguiente manera:

- *Authors List*: Contiene una serie de 8 párrafos *Paragraph* los cuales contienen en su atributo *content* los nombres de los autores del modelo de entrada.

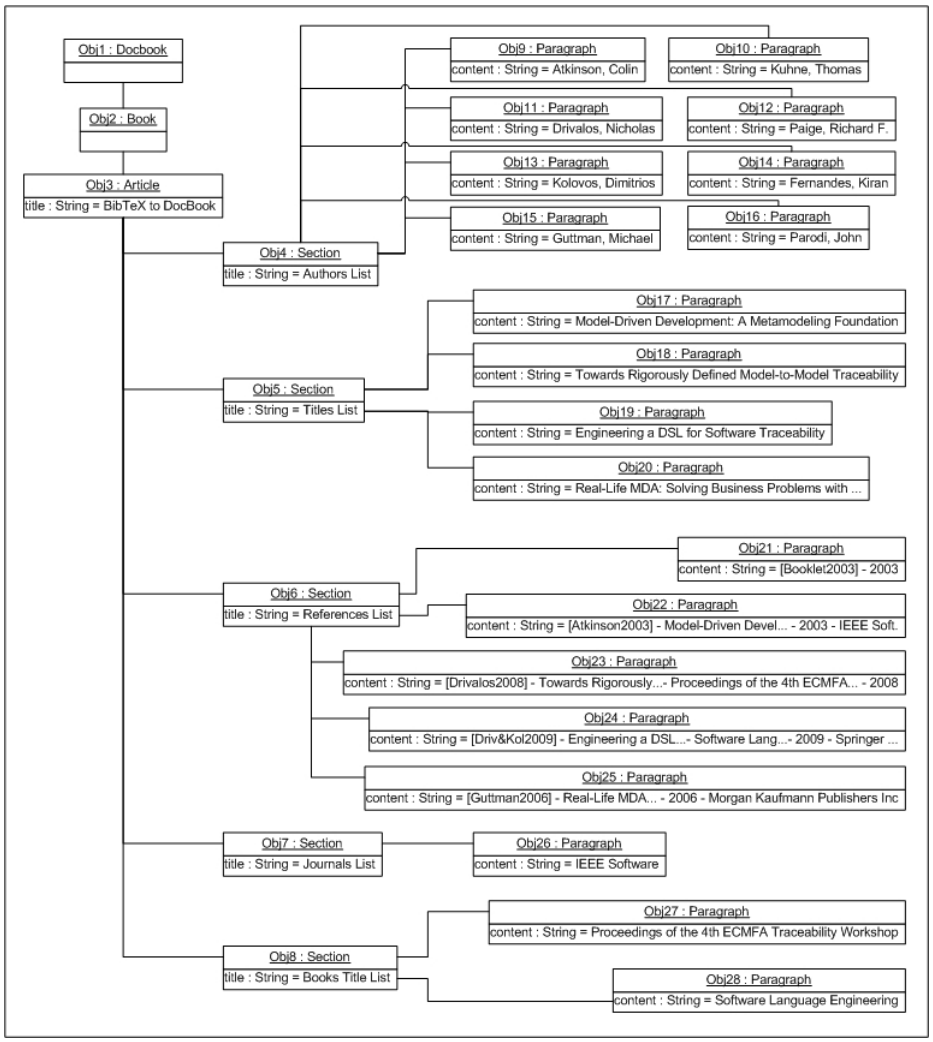


Figura 6.27: Salida del proceso de transformación: un archivo Docbook

- *Titles List*: Formada por cuatro párrafos *Paragraph* cuyo contenido son los títulos de las entradas tituladas (descendientes de la entidad *TitledEntry*).
- *References List*: Mantiene un párrafo *Paragraph* por cada entrada Bibtex del modelo origen, y su contenido (atributo *content*) mantiene una concatenación de toda la información disponible de cada una, la cual obviamente depende de su tipo (ID, título, fecha, publicación, título del libro, etc).
- *Journal List*: Posee un párrafo *Paragraph* por cada publicación.
- *Book Titles List*: Contiene un párrafo *Paragraph* por cada título de libro que esté disponible (entradas descendientes de la clase *BookTitledEntry*).

6.3.6. Inferencia de trazas

A continuación analizaremos las trazas que nos permite inferir el mecanismo de Análisis Basado en Variables. Al igual que en la Sección 6.2.6, se detallarán las trazas halladas en cada relación QVT definida dentro de la transformación.

6.3.6.1. Relación *BibtexFileToArticle*

Esta relación mapea elementos *BibTeXFile*, del modelo BibTeXML, en entidades *Article* correspondientes al modelo Docbook. La traza hallada, correspondiente al Caso #3 definido en la Subsección 5.3.2.3, de inferencia de trazas mediante una constante, nos permite inferir lo siguiente:

- '*BibTeX to DocBook*' $\rightarrow$ *title* :: *Article*, donde el título del artículo es *title* :: *Article* = '*BibTeX to DocBook*'

Esta traza nos indica que toda entidad *Article* derivada a partir de un elemento *BibTeXFile* tendrá un título constante, el cual será 'BibTeX to Docbook'.

6.3.6.2. Relaciones de generación de secciones

Al igual que en la relación anterior, todas las relaciones de creación de secciones derivan trazas de tipo "constante", correspondientes al Caso #3 de inferencia de trazas, sobre los nombres. De esta manera, tendremos las siguientes trazas:

- Para la relación *BibtexFileToSectionAuthors*, '*Authors List*' $\rightarrow$ *title* :: *Section*, donde *title* :: *Article* = '*Authors List*'
- Para la relación *BibtexFileToSectionTitles*, '*Titles List*' $\rightarrow$ *title* :: *Section*, donde *title* :: *Article* = '*Titles List*'
- Para la relación *BibtexFileToSectionReferences*, '*References List*' $\rightarrow$ *title* :: *Section*, donde *title* :: *Article* = '*References List*'
- Para la relación *BibtexFileToSectionJournals*, '*Journals List*' $\rightarrow$ *title* :: *Section*, donde *title* :: *Article* = '*Journals List*'

- Para la relación *BibtexFileToSectionBooks*, '*Books Title List*' → *title* :: *Section*, donde *title* :: *Article* = '*Books Title List*'

En esencia, todas estas trazas nos brindan información estructural del documento Docbook generado con la transformación. Cada una de estas relaciones mapean un elemento *BibTeXFile* en un elemento *Article* que posee una sección cuyo nombre es fijo y no depende del valor de ningún elemento del modelo base (origen). Dado que cada relación se verifica ("matchea") para el mismo elemento *BibTeXFile* de entrada, el resultado será la generación de un elemento *Article* con cinco secciones *Section*, cuyos nombres corresponden a los mencionados anteriormente.

6.3.6.3. Relación *AuthorToParagraph*

La relación *AuthorToParagraph* transforma cada elemento *Author* del modelo BibTeXML en un párrafo dentro de la Sección #2 (lista de autores). De ésta es posible derivar la siguiente traza:

- *name* :: *Author* → *content* :: *Paragraph*, donde *content* :: *Paragraph* = *name* :: *Author*

Dicha traza, correspondiente al Caso #1 de inferencia mediante una variable, nos indica que el nombre de todo autor presente en el archivo BibTeX (elemento *BibTeXFile*) será almacenado en un párrafo de una sección determinada. En particular dicha sección será la #2, que contiene la lista de autores, pero esta no es una información que brinda la traza, sino la relación a través de su cláusula *When*.

6.3.6.4. Relación *TitledToSectionTitles*

Esta relación transforma entradas BibTeX tituladas, es decir, de tipo *TitledEntry* o de alguna clase descendiente de ésta, en párrafos de la Sección #3 (lista de títulos). Mediante el Análisis Basado en Variables es posible obtener la siguiente traza:

- *title* :: *TitledEntry* → *content* :: *Paragraph*, donde *content* :: *Paragraph* = *title* :: *TitledEntry*

Esta traza, derivada por aplicación del Caso #1 de inferencia, nos dice que cada entrada titulada derivará en un párrafo de una sección en particular, cuyo contenido será el título de la entrada bibliográfica.

6.3.6.5. Relación *ArticleToSectionJournal*

La presente relación mapea artículos BibTeX en párrafos de la lista de publicaciones (Sección #4). La traza derivada de la relación, correspondiente al Caso #1 de inferencia, es la siguiente:

- $journal :: Article \rightarrow content :: Paragraph$, donde $content :: Paragraph = jornal :: Article$

De esta forma, la presente traza nos indica que por cada entrada *Article* del modelo BibTeXML tendremos un párrafo de una sección cuyo contenido corresponderá al nombre de la publicación del articulo (*journal*).

6.3.6.6. Relación *BookTitleToSectionBooks*

Como su nombre lo indica, esta relación transforma entradas de tipo *BookTitledEntry* en párrafos *Paragraph* de la Sección #5, la cual contiene la lista de libros. Por aplicación del Caso #1 de inferencia es posible derivar la siguiente traza:

- $booktitle :: BookTitledEntry \rightarrow content :: Paragraph$, donde $content :: Paragraph = booktitle :: BokTitledEntry$

De manera análoga a las relaciones anteriores, esta traza nos indica que la Sección #5 del documento *Docbook* resultante de la transformación contendrá los títulos de los libros de las entradas *BookTitledEntry* de modelo BibTeXML de base.

6.4. Consideraciones finales

A lo largo del presente capítulo hemos analizado dos casos de estudio de transformación de modelos. El primero, llamado UML2Java, presenta el mapeo de un modelo UML simplificado en código fuente Java (modelizado). En segundo lugar, vimos la transformación Bib2Doc, la cual presenta la conversión de un esquema de referencia bibliográfica BibTeX en un documento Docbook con determinadas características.

Para cada uno de estos casos hemos aplicado el mismo flujo de trabajo: presentación de los modelos de entrada y salida, especificación de reglas de transformación, generación del código QVT y finalmente el estudio del mismo bajo el análisis basado en variables, obteniendo como resultado un conjunto de trazas derivadas a partir del método presentado.

Como pudimos comprobar, la definición de una transformación es una tarea tediosa: el análisis del modelo de entrada, la definición de reglas para la transformación en un modelo destino, y la implementación de dichas reglas en relaciones QVT, es decir, el armado del código fuente a ejecutar. La obtención de trazas agrega al analista una tarea adicional, también tediosa, y muy propensa a errores. La posibilidad de contar con un mecanismo automatizado para la generación de información de trazabilidad agrega mucho valor al proceso, y permite a los ingenieros abocarse a otras tareas, como el aprovechamiento y explotación de dicha información.

El desarrollo de estos casos de estudio nos deja además dos cuestiones interesantes para el debate: en primer lugar la equivalencia o igualdad de trazas, es

decir las condiciones bajo las cuáles dos trazas derivadas por el algoritmo de inferencia son iguales, y por tanto una debe ser desechada. En segundo lugar, las limitaciones en la derivación de trazas a partir de la granularidad del algoritmo de inferencia.

A continuación profundizaremos cada uno de éstos aspectos planteados.

6.4.1. Igualdad de trazas

Como pudimos observar en las secciones 6.2.6.7 y 6.2.6.8, durante el desarrollo del primer caso de estudio, existen situaciones donde dos o más relaciones permiten, mediante el esquema propuesto, inferir trazas "estructuralmente" similares, es decir, trazas para las cuales los elementos origen y destino relacionados son los mismos.

En el ejemplo citado, tanto la relación *PrimitiveAttributeToField*, como las relaciones *EnumerationAttributeToField* y *ClassAttributeToField* nos permiten determinar que un elemento *Attribute* del modelo MySimpleUML mapeará en un elemento *JavaField* con el mismo nombre y visibilidad. ¿Se trata entonces de la misma traza?

La respuesta, la cual adelantamos oportunamente en la sección, es no. Al tratarse de relaciones diferentes, las trazas están asociadas a condiciones diferentes. Toda traza representa la transformación de un elemento en otro, pero además una condición bajo la cual es válida. No todo elemento *Attribute* del modelo origen necesariamente mapeará sobre un elemento *JavaField*, sino que dependerá de que la tupla de elementos evaluada satisfaga la relación que define la regla.

6.4.2. Limitaciones del análisis basado en variables

El segundo caso de estudio pone en evidencia una limitación del mecanismo de inferencia de trazas planteado. Si observamos el uso de la consulta *getContent()*, cuyo código puede verse en la Figura 6.25, veremos que la misma nos permitiría indirectamente obtener información de trazabilidad. Dicha consulta es utilizada para determinar el valor del atributo *content* de los elementos *Paragraph* en algunas relaciones, lo cual nos daría la posibilidad de encuadrarla en el marco del Caso #6 de inferencia de trazas mediante una consulta (ver Sección 5.3.2.6), sin embargo nuestro algoritmo no la detecta.

El motivo es que la consulta actúa como una función de un elemento *BibTeXEntry*, y no sobre uno de sus atributos, que es la condición que verifica el mecanismo para determinar si una consulta es candidata a generar una traza. Si bien esto restringe la potencia del mecanismo en la inferencia de trazas, el framework presentado brinda las herramientas para poder mejorar este desarrollo. La manera natural de hacerlo sería implementar una nueva *TraceStrategy* que permita el reconocimiento de estos "patrones", y aplicarla sobre la transformación previo al análisis basado en variables, o luego del mismo.

Capítulo 7

Estudio comparativo de propuestas

A efectos de poder evaluar el contenido de la propuesta presentada, a continuación se desarrollará un estudio comparativo con algunos de los trabajos relacionados que fueran presentados en la Sección 3.4.3. Dicha selección no es arbitraria, sino que responde fundamentalmente a dos cuestiones: en primer lugar, a las similitudes entre dichas propuestas y el esquema presentado en este trabajo, y en segundo lugar, a la información disponible, y el grado de detalle de cada una de ellas. Cuando hablamos de similaridad, nos referimos a los aspectos metodológicos y al enfoque dado al trabajo. Claramente cada propuesta es distinta de la otra y el objetivo del capítulo es, precisamente, contrastar esas diferencias.

Como veremos a continuación, existen estudios de comparación muy profundos, en algunos casos exhaustivos, que han arrojado importantes resultados respecto del estado del arte en materia de implementación de esquemas de trazabilidad en MDD. La meta aquí no es identificar tendencias, o determinar la problemática de un área de estudio, ni líneas de investigación, sino poner en contraste nuestro trabajo con otros esquemas similares comparables.

7.1. Introducción

Hemos elegido tres trabajos para el presente estudio comparativo: la propuesta de trazabilidad escasamente acoplada para ATL de Jouault [51], el esquema de Grammel [45] de extracción de datos de rastreabilidad basado en *facets*, y finalmente el framework de trazabilidad para transformación de modelos propuesto por Falleri [41]. El objetivo de la comparación es mostrar de qué manera ha abordado cada enfoque el problema de obtención de información de trazabilidad, y contrastar diferencias y semejanzas de los trabajos analizados con la propuesta presentada en esta tesis.

De las propuestas indicadas en el Capítulo 3 (Sección 3.4.3), se ha excluido del análisis comparativo el trabajo de Kolovos, Paige y Polack [54]. En particular,

por sus características, la propuesta no proporciona información acerca de cómo identificar trazas. En dicho esquema, las relaciones de trazabilidad son definidas explícitamente durante el proceso de confección de modelos. A diferencia de éste, los cuatro trabajos evaluados en el presente capítulo establecen mecanismos de extracción y/o obtención de información de trazabilidad, no disponible explícitamente en los modelos intervinientes en el proceso de transformación. No obstante, son muy destacables los aportes del equipo de investigadores conformado por los autores Drivalos, Kolovos, Paige y demás en diversas áreas de la gestión de trazabilidad en transformación de modelos. Además del trabajo mencionado, el equipo ha conseguido importantes resultados en tópicos como la construcción de metamodelos de trazabilidad [37], o el mantenimiento de información de trazabilidad [38], por nombrar sólo algunos.

Existen otras propuestas de generación de trazabilidad a partir del desarrollo de transformaciones de modelos que no forman parte del presente estudio comparativo: en [25], Bondé *et al.* presentan un trabajo centrado en el ámbito del desarrollo de sistemas embebidos, donde los diseñadores deben manejar múltiples modelos a distintos niveles de abstracción; en [26], Boronat *et al.* proponen un marco de trabajo de gestión de modelos algebraicos para resolver problemas específicos; en [55], Levendovszky *et al.* presentan una aproximación para la gestión de trazabilidad en el desarrollo conducido por modelos de sistemas aéreos, donde dicha gestión se considera un aspecto obligatorio durante el desarrollo; en [61], los autores Oldevik y Neple presentan una propuesta de gestión de trazabilidad en el desarrollo de transformaciones M2T (*Model-to-Text*); en [64], Sanchez *et al.* sugieren un esquema de gestión de trazabilidad de requerimientos de seguridad en el desarrollo dirigido por modelos de aplicaciones robóticas; en [70], los autores Valderas y Pelechano presentan una propuesta enmarcada en el contexto de la gestión de trazabilidad de requerimientos en el desarrollo conducido por modelos de aplicaciones web, entre otros. Por último, cabe también mencionar el trabajo de Aranega *et. al* en [21], nombrado en varios pasajes de esta tesis, donde los autores muestran las limitaciones de las trazas implícitas de QVT y realizan una propuesta de mejora basada en un metamodelo de trazabilidad que permite el mantenimiento de información más detallada, a un mayor nivel de granularidad.

Una de las decisiones más importante de todo estudio comparativo es la elección de los criterios que se evaluarán en cada uno de los trabajos contrastados. En muchos casos, existen criterios interesantes de evaluar pero no se dispone de la información necesaria para poder determinar la valoración exacta del mismo para una propuesta dada, por lo que deben ser desestimados del alcance de la evaluación. Para poder orientar la selección de dichas pautas a analizar se revisaron una serie de trabajos basados en la comparación de propuestas de trazabilidad, a efectos de poder determinar qué aspectos son más relevantes a la hora de plantear un contraste entre esquemas de trazabilidad.

El presente capítulo se encuentra organizado de la siguiente manera: la Sección 7.2 presentada a continuación aborda los trabajos relevados con especial énfasis en los criterios de evaluación seleccionados por los autores. Luego, la Sección 7.3 detalla las pautas de comparación determinadas para la evaluación de los trabajos. A continuación, desde la Sección 7.4 a la Sección 7.6 se abordan las principales características de los trabajos de Jouault, Grammel y Falleri. Y

finalmente, la Sección 7.7 presenta el estudio comparativo entre los esquemas analizados.

7.2. Trabajos relacionados

Varios autores han realizado estudios comparativos de propuestas de trazabilidad en el contexto del paradigma de desarrollo conducido por modelos (MDD). En el Capítulo 3 presentamos el estudio de aproximaciones de trazabilidad en MDD de Galvao y Goknil [42], donde los mismos evalúan el estado del arte de una amplia variedad de esquemas de trazabilidad, y realizan una evaluación de los mismos respecto de cinco criterios: representación, mapeo, escalabilidad, análisis de impacto de cambios y soporte de herramientas. El criterio de representación compara la capacidad de representar la información de trazabilidad. El criterio de mapeo o *mapping* analiza si la propuesta es capaz de generar trazas entre los modelos a diferentes niveles de abstracción. El criterio de escalabilidad analiza si el esquema puede ser eficientemente aplicado a grandes sistemas. El criterio de análisis de impacto de cambios evalúa si la solución provee soporte para determinar el impacto de cambios sobre los artefactos a lo largo del ciclo de vida del desarrollo de software. Por último, el criterio de soporte de herramientas evalúa si la propuesta prevé algún tipo de soporte a herramientas para facilitar la trazabilidad. Las pautas de escalabilidad, análisis de impacto de cambios y soporte de herramientas son evaluadas a tres posibles valores: si, no o parcialmente. El criterio de *mapping* puede adoptar tres posibles valores: la propuesta soporta relaciones *intra-level* (trazas entre elementos del mismo nivel de abstracción), *inter-level* (trazas entre elementos de distinto nivel de abstracción), o ambas. Finalmente, el criterio de representación se basa en una descripción textual breves de la forma de representación de la información de trazabilidad que ofrece el esquema.

Otro interesante trabajo de comparación puede encontrarse en [66], donde Santiago *et al.* presentan un trabajo de evaluación de propuestas de trazabilidad que tiene como objetivo aunar criterios en torno a operaciones básicas para gestión de trazabilidad y sus aplicaciones. El trabajo analiza una amplia variedad de esquemas desde tres enfoques distintos: un grupo de propuestas son evaluadas respecto de la aplicación de la información de trazabilidad que realizan, otro grupo es analizado respecto de las operaciones de trazabilidad que implementan, y por último un tercer grupo de propuestas, todas basadas en MDD, son evaluadas con el objeto de identificar características comunes o diferenciadoras entre ellas. Sobre el primer grupo de propuestas los autores evalúan si las mismas utilizan o no la información de trazabilidad para análisis de impacto de cambios, mejora de la comprensibilidad, gestión de proyectos, gestión de la configuración o control de versiones de modelos. El segundo grupo de esquemas de trazabilidad es analizado según las operaciones básicas que implementan. De cada propuesta se analiza si permiten una serie de operaciones sobre trazas, listadas a continuación con una breve referencia:

- Definición: clasificación de los tipos de entidad que serán trazadas y el tipo de relación de trazabilidad existente entre ellos.

- Identificación: operación que permite las relaciones de trazabilidad previamente desconocidas entre entidades, ya sea manual, automática o mixta.
- Almacenamiento: operación fundamental de la gestión de trazabilidad.
- Importación/Exportación: determina las propuestas capaces de importar o exportar la información de trazabilidad en un formato adecuado para su intercambio entre distintas aplicaciones.
- Consulta: operación de recuperación de información de trazabilidad, generalmente asociada con las operaciones de identificación y almacenamiento.
- Visualización: representación de las relaciones de trazabilidad existentes entre los diferentes artefactos.
- Modificación: de las relaciones existentes de trazabilidad.

Para el último grupo de propuestas, basadas en MDD, se definieron los siguientes criterios:

- Aproximación: permite distinguir la naturaleza de los trabajos evaluados, pudiendo ser aproximaciones basadas en requerimientos o propuestas basadas en transformaciones.
- Transformación: permite identificar el tipo de transformación que es llevada a cabo por la propuesta. En particular, se consideran dos valores posibles: transformaciones entre modelos (M2M) o transformaciones de modelo a texto (M2T).
- Dimensión[1]: la cual permite determinar si la propuesta soporta trazabilidad de modelos y/o elementos de modelo a distintos niveles de abstracción. Este criterio puede asumir tres posibles valores: dimensión horizontal, para propuestas que soportan trazabilidad entre modelos y/o elementos de modelo del mismo nivel de abstracción; dimensión vertical, para esquemas que soportan trazabilidad en distintos niveles de abstracción, o eventualmente una tercer opción de ambas posibilidades simultáneamente.
- Nivel: indica si la trazabilidad se lleva a cabo entre modelos (alto nivel) o si la trazabilidad se lleva a cabo entre elementos de modelo (bajo nivel).
- Lenguajes: enumera los distintos lenguajes de transformación y/o generación de código que utilizan las propuestas.
- Representación: define las principales estructuras que son utilizadas para representar la información de trazabilidad.
- Metamodelo: indica el propósito del metamodelo presentado en cada trabajo analizado. Se establecieron dos posibles valores para el criterio de metamodelo: genérico, o específico para el dominio del problema.

[1] Este criterio es reconocido por Galvao también, bajo el nombre de *mapping*

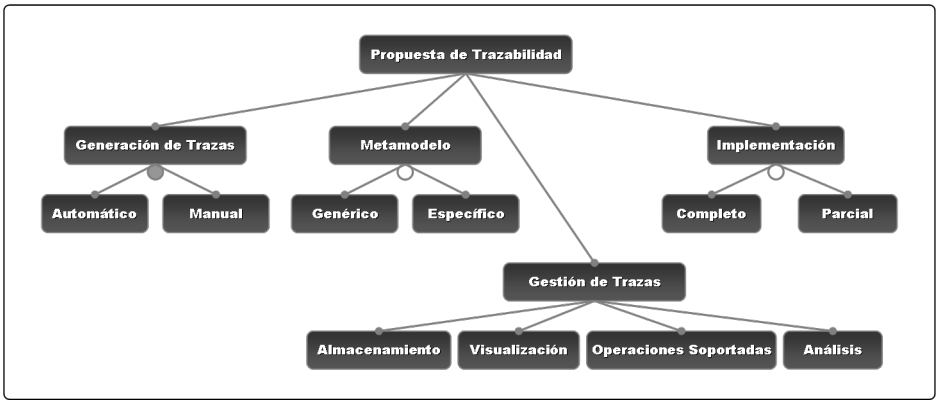
En un nuevo trabajo del su equipo de investigación, Santiago *et al.* presentan en [65] una revisión sistemática de la literatura[2] de gestión de trazabilidad en el contexto del desarrollo conducido por modelos, cuya meta principal fue descubrir cómo es gestionada y analizada la información de trazabilidad en el contexto de MDD, identificando posibles campos de mejora. A efectos de poder lograr una revisión exhaustiva de éste tópico, el trabajo presenta una revisión sistemática de la literatura de gestión de trazabilidad basada en los lineamientos propuestos por Kitchenham y Charters [52], y Biolchini *et al.* [24], el cual permite responder preguntas tales como qué nivel de automatización es recomendado para la generación de trazas o cuáles son las limitaciones actuales del estado del arte en gestión de trazabilidad en el contexto de MDD.

En su trabajo de comparación, los autores definen una serie de tópicos a evaluar en cada una de las propuestas consideradas en el estudio. Dichas pautas, algunas presentes también en los trabajos detallados anteriormente, son las siguientes:

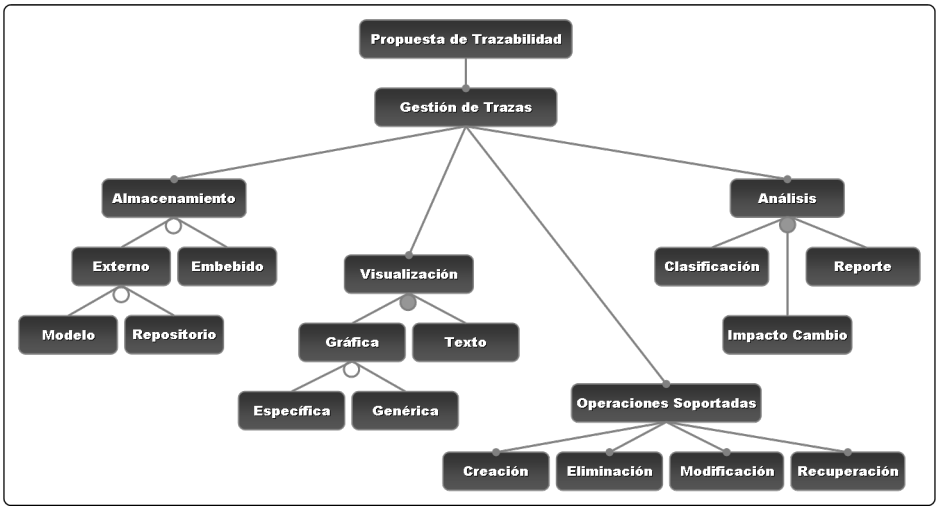
- Si las relaciones de trazabilidad son definidas automáticamente (por ejemplo a partir de transformaciones) o manualmente por usuarios.
- El tipo de transformaciones utilizadas para la gestión de trazabilidad (transformaciones modelo-a-modelo o modelo-a-texto).
- Los lenguajes de transformación utilizados para la gestión de la información de trazabilidad.
- La forma en que las trazas son almacenadas según la propuesta. Por ejemplo, en un repositorio de trazas, embebidas en los mismos modelos durante el proceso de desarrollo (es decir, sin un contenedor específico), en modelos de trazabilidad, etc.
- El tipo de metamodelo de trazabilidad propuesto (genérico o específico).
- Los lenguajes de modelado a partir de los cuales pueden ser derivadas las trazas (UML, EMF, por ejemplo).
- La manera en que la información de trazabilidad es visualizada.
- Las operaciones consideradas para la gestión de información trazabilidad (creación, modificación, eliminación, consulta, etc).
- El análisis de la información de trazabilidad realizado (clasificación, impacto de cambio, reporte).
- Si la propuesta ha sido implementada.

La Figura 7.1a ilustra mediante un diagrama de características (*feature diagram*) de Czarnecki [33] los aspectos considerados para cada propuesta de trazabilidad analizada. En particular, los aspectos relacionados con la gestión de trazas, comprenden otro conjunto de tópicos relevantes, detallados en la Figura 7.1b.

[2] Una revisión sistemática de la literatura o *systematic review* es una manera de identificar, evaluar e interpretar todo el material de las investigaciones relevantes disponibles respecto de un interrogante particular, tópico, o fenómeno de interés. Los estudios individuales que contribuyen con una revisión sistemática son llamados estudios primarios; una revisión sistemática es una forma de estudio secundario.



(a) Diagrama de *features* de los principales aspectos de una propuesta de trazabilidad



(b) *Features* de la gestión de trazabilidad

Figura 7.1: Aspectos de diseño de una propuesta de trazabilidad

7.3. Criterios de comparación elegidos

Tras la revisión de los trabajos relacionados se decidió enfocar el análisis comparativo sobre dos puntos críticos de todo enfoque de trazabilidad en el contexto del desarrollo conducido por modelos: por un lado, el metamodelo de trazabilidad planteado, es decir, la manera que cada propuesta representa las trazas, y por otro, en el mecanismo de obtención de información de trazabilidad implementado por cada uno.

Luego, de los criterios de evaluación de propuestas de trazabilidad presentes en varios de los trabajos revisados previamente, se seleccionaron aquellos inherentes o relacionados con el metamodelo de trazabilidad de una propuesta, y con el mecanismo de obtención de trazas. Así, cada uno de los ejes considerados a los efectos de la comparación se analizará de acuerdo a las siguientes pautas:

- Metamodelo de trazabilidad
  - Tipo de metamodelo.
  - Lenguajes.
  - Nivel.
  - Almacenamiento de trazas.
  - Dimensión de trazabilidad.
- Mecanismo de obtención de trazas
  - Nivel de implementación.
  - Automatización del mecanismo.
  - Tipo de obtención de trazas.
  - Visualización de la información.
  - Operaciones sobre trazas soportadas.
  - Análisis de trazas provisto.

A continuación, presentaremos un resumen de las propuestas evaluadas, con sus principales características. Finalmente, en la Sección 7.7, se detallarán cada uno de los criterios planteados, y se realizará el análisis comparativo correspondiente.

7.4. Esquema de trazabilidad escasamente acoplada

La propuesta de Jouault fue uno de los primeros trabajos de generación de información de trazabilidad automática en el contexto del desarrollo conducido por modelos. Es un trabajo de referencia, que muestra una solución absolutamente enmarcada en el paradigma MDD (*Model-Driven Development*) y por tal ha sido elegido como parámetro de comparación frente al estudio presentado en este trabajo.

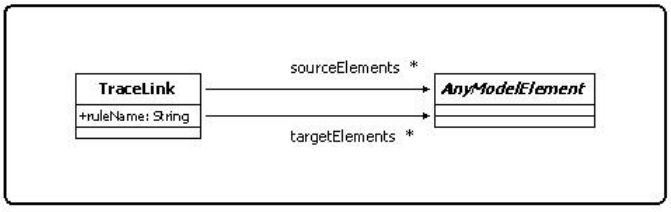


Figura 7.2: Metamodelo de trazabilidad de Jouault

7.4.1. Metamodelo de trazabilidad

El trabajo propone un metamodelo de trazabilidad simple, compuesto por una clase *TraceLink*, la cual contiene un atributo para almacenar el nombre de la regla que genera dicha traza, y mantiene dos colecciones de objetos de tipo *AnyModelElement*, denominadas *sourceElements* y *targetElements* (Figura 7.2), donde dichas colecciones almacenan los elementos origen y destino que se encuentran relacionados por la aplicación de la regla. La clase *AnyModelElement* es abstracta, y obviamente dependerá de entorno de aplicación del metamodelo. Una posible implementación concreta de la misma podría ser la clase *EObject*, en caso de trabajar en el contexto del Eclipse Modeling Framework (EMF), por ejemplo.

```
1.    module Src2Dst;
2.    create OUT : Dst from IN : Src;
3.
4.    rule A2B {
5.        from
6.            s : Src!A
7.        to
8.            t : Dst!B (
9.                name <- s.name
10.           )
11.   }
```

Figura 7.3: Definición de la transformación Src2Dst en ATL

7.4.2. Obtención de la información de trazabilidad

La propuesta sugiere la obtención de trazas mediante el agregado de, por un lado, código extra ATL a las reglas que conforman la transformación y, por otro, de un modelo de salida de trazabilidad, que permite que la información de rastreabilidad sea capturada al momento de producirse la transformación de modelos. Esta modificación no altera la lógica del programa, si bien agrega contenido extra a la definición de la transformación.

A efectos de clarificar el mecanismo, se presenta como ejemplo la definición de una transformación sencilla (*Src2Dst*) de un modelo origen A, con un único atributo llamado *name*, de tipo *String*, en un modelo destino B, con un único atributo llamado *name* también, de tipo *String*. La Figura 7.3 muestra el código

```
1.    module Src2DstPlusTrace;
2.    create OUT : Dst, trace : Trace from IN : Src;
3.
4.    rule A2BPlusTrace {
5.        from
6.            s : Src!A
7.        to
8.            t : Dst!B (
9.                name <- s.name
10.           ),
11.       traceLink : Trace!TraceLink (
12.           ruleName <- 'A2BPlusTrace',
13.           targetElements <- Sequence {t}
14.       )
15.       do {
16.            traceLink.refSetValue('sourceElements', Sequence {s});
17.       }
18.   }
```

Figura 7.4: Transformación Src2Dst con soporte de trazabilidad

fuente de la transformación ATL que convierte una instancia del modelo A en una del modelo B. En ésta podemos identificar sus dos componentes principales: por un lado tenemos la definición del modelo de entrada *Src*, a partir del cual es generado el modelo de salida *Dst*, y por otro, una única regla *A2B* la cual mapea todos los elementos del modelo de entrada *Src* en elementos del modelo destino *Dst*, para los cuales se verifica que el atributo *name* será idéntico al atributo *name* del elemento correspondiente del modelo *Src* que le dio origen.

El soporte de trazabilidad es incluido en la transformación mediante dos modificaciones sobre el código fuente (Figura 7.4): la primera consiste en el agregado de un nuevo modelo de salida *Trace*, de tipo *TraceLink*, según el metamodelo descripto en la Subsección 7.4.1 (línea 2 del código). La segunda modificación se basa en generar en paralelo los elementos del modelo destino y los elementos del metamodelo de trazabilidad (líneas 7 a 18). De esta manera, por cada elemento s del modelo origen *Src*, se generará un elemento t del modelo *Dst*, donde $t.name = s.name$, y además se creará un elemento *traceLink* de tipo *TraceLink*, donde $traceLink.ruleName = 'A2BPlusTrace'$, $sourceElement = \{s\}$ y $targetElements = \{t\}$.

De esta forma, el autor presenta una propuesta que brinda trazabilidad al programa de transformación sin necesidad de recurrir a un lenguaje específico ni motor que lo soporte. Sin embargo, el enfoque requiere el agregado manual de piezas de código ATL en la definición de las transformaciones. Para resolver este problema, el autor sugiere una solución absolutamente MDD: dado que los programas de transformación son modelos, es posible definir una transformación que permita convertir cualquier programa ATL en un programa ATL con soporte de trazabilidad, el cual incluya las sentencias adicionales. Este desarrollo ha sido implementado en un componente llamado *TracerAdder*, el cual debe ser utilizado previo a la compilación ATL, de manera que puede ser considerado un precompilador.

7.4.3. Resumen de la propuesta

Como se enfatiza en el nombre del trabajo de Jouault, una de las más importantes ventajas de la solución es que el código de generación de trazabilidad o TGC (*Traceability Generating Code*) está desacoplado de la lógica de la transformación. Es posible, incluso, incorporar distintos TGC a un mismo programa, según lo requiere una determinada aplicación. De esta forma, una transformación dada puede ser adaptada a rangos[3] o formatos específicos de trazabilidad, según sea necesario.

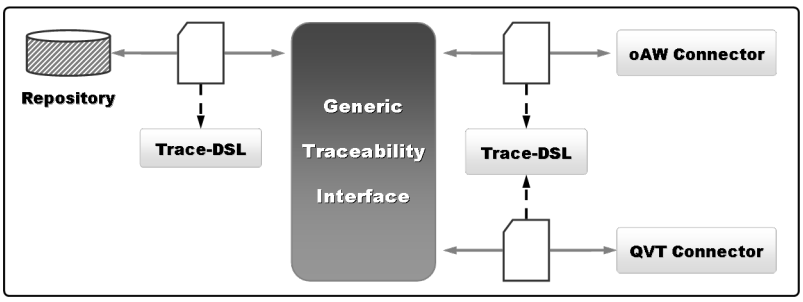
En función a todo lo dicho anteriormente, a modo de resumen, podemos realizar las siguientes observaciones acerca de la propuesta de Jouault:

- La obtención de información de trazabilidad es implementada a nivel de la transformación. Por ende, requiere de su ejecución para la generación de las trazas.
- No afecta la lógica de la definición, pero agrega información adicional que hace difusa la legibilidad de la transformación original.
- Es absolutamente independiente de los modelos involucrados en la transformación.
- Utiliza un metamodelo de trazabilidad genérico, adaptable a otros posibles esquemas.
- Si bien se requiere de la adición manual de código ATL en cada regla para la obtención de la información de trazabilidad, existe un proceso de acondicionamiento automático de reglas aplicable a cualquier transformación, que puede ser ejecutado antes de la compilación de la transformación (*TracerAdder*).
- Está basada en herramientas escritas en el mismo lenguaje de transformación de modelos ATL, sin requerir y/o depender de software de terceras partes.

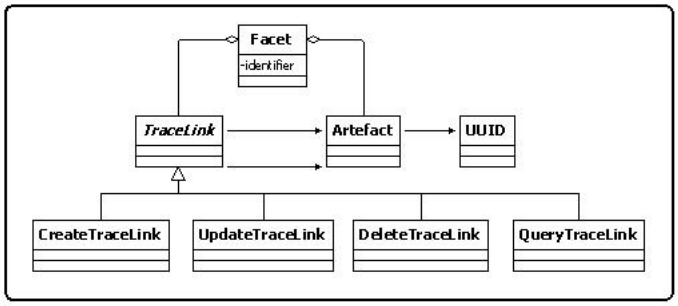
7.5. Extracción de datos de trazabilidad basado en *Facets*

El trabajo de Grammel propone un framework de trazabilidad genérico que permite aumentar esquemas de transformación de modelos arbitrarios con información de rastreabilidad. El framework está basado, por un lado, en una interfase de trazabilidad genérica (GTI, *Generic Traceability Interfase*) que provee el punto de conexión para motores de transformación arbitrarios y brinda un API (*Application Programming Interfase*) para conectarlo con el motor de trazabilidad (Figura 7.5a). Como resultado, el motor de transformación es dotado de la funcionalidad de trazabilidad. En segundo lugar, el framework se basa además

[3] El rango de uso de trazabilidad corresponde a la proporción de elementos para los cuales se mantienen trazas



(a) Arquitectura



(b) Fragmento del lenguaje Trace-DSL

Figura 7.5: Propuesta de Grammel

en un lenguaje de dominio específico (DSL, Domain-Specific Language) llamado Trace-DSL que en esencia determina qué tipo de información de trazabilidad es intercambiable entre la interfaz genérica y los conectores de los motores de rastreabilidad.

7.5.1. Metamodelo de trazabilidad

El esquema propuesto por Grammel es muy interesante dado que adopta un enfoque totalmente distinto al de la mayoría de los autores. A continuación analizaremos un fragmento del lenguaje de dominio específico Trace-DSL desarrollado para dar soporte de trazabilidad (Figura 7.5b). En este modelo, las trazas son representadas por el componente *TraceLink*, el cual es considerado una abstracción de la transición de un artefacto hacia otro. El objeto *Artefact*, individualizado con un único identificador universal (*UUID*) representa cualquier producto rastreable generado en el proceso de desarrollo, como un requerimiento o clase, o un componente de otro artefacto (por ejemplo un método de una clase). La transición representada siempre es dirigida, por lo que una traza entre artefactos origen y destino siempre da lugar a una relación *desde-hasta* entre dichos artefactos. Considerando a la trazabilidad como el seguimiento de todos los cambios posibles aplicados a elementos de modelos durante una transformación, Grammel propone dividir dicha transformación en un conjunto de operaciones elementales eo_s y definir un tipo de traza para cada operación, con

los correspondientes elementos de los modelos origen y destino involucrados. De acuerdo a estas consideraciones, el autor propone los siguientes tipos:

- *CreateTraceLink*: Para definir las operaciones que dan lugar a un nuevo modelo destino.
- *UpdateTraceLink*: Para las operaciones que realizan modificaciones sobre elementos existentes.
- *DeleteTraceLink*: Para las operaciones que eliminan elementos del modelo destino.
- *QueryTraceLink*: Para operaciones de consulta que no generan ningún tipo de cambio en los elementos del modelo destino.

Para asignar tipos a los artefactos y trazas, la propuesta utiliza el concepto de faceta (*facet*), donde el lenguaje Trace-DSL asigna un conjunto de facetas a cada artefacto y traza, simplificando la jerarquía de tipos y buscando dotar de extensibilidad al metamodelo propuesto.

7.5.2. Obtención de la información de trazabilidad

Como hemos visto, el uso de conectores permite al enfoque la interacción eventualmente con cualquier lenguaje de transformación de modelos arbitrario. En particular, y considerando que nuestra propuesta, QVTrace, ha sido definido en el contexto de QVT, nos concentraremos en determinar cómo lleva a cabo la obtención de trazas este mecanismo para transformaciones escritas en dicho lenguaje. El conector QVT desarrollado está escrito en el lenguaje de mapeos operacionales que provee QVT, Operational Mappings. Dicho lenguaje permite definir transformaciones utilizando una aproximación imperativa o bien complementar transformaciones relacionales (escritas en lenguaje QVT Relations) con operaciones imperativas (enfoque híbrido), cuando es difícil proveer una especificación completamente declarativa de una relación. Cada relación define una clase que será instanciada para la traza entre elementos de modelo que están siendo transformados, y esta tiene un mapeo unívoco con una operación que el mapeo operacional implementa.

La Figura 7.6 muestra el código correspondiente al conector QVT escrito en Operational Mappings. En primer lugar, cada registro TraceRecord (trazas internas QVT) es recogido y mapeado en un objeto TraceLink del metamodelo definido (línea 2). Dado que los archivos de trazas contienen ejecuciones exitosas del mapeo de elementos de modelo con el resultado de nuevos elementos creados, los registros TraceRecords son invariablemente convertidos en objetos CreateTraceLink (líneas 4 a 11). Una transformación de actualización requeriría la definición de una mapeo UpdateTraceLink convenientemente. Los artefactos relacionados son creados también en base a TraceRecords. Para el caso de los elementos origen, el UUID es extraído del contexto del elemento del modelo (línea 16), mientras que en el caso de los elementos destino el UUID es derivado de los elementos resultado del TraceRecord (línea 20). Por último, las líneas 7, 8, 17 y 21 generan diferentes facetas. En particular, las trazas son enriquecidas con

```
1.  main () {
2.    qvt.objectsOfType(QVTTraceDSL::Trace).traceRecords -> map TraceRecord2TraceLink();
3.  }
4.  mapping TraceRecord::TraceRecord2TraceLink() : CreateTraceLink {
5.    init {
6.      var facetsList := List {
7.        self.map TraceRecord2TraceLinkNameFacet(),
8.        self.map TraceRecord2TraceLinkMappingFacet()
9.      }
10.   }
11.   sources := self.map TraceRecord2SourceArtefact() -> asList();
12.   targets := self._result._result.value.map EValue2TargetArtefact( self ) -> asList();
13.   facets := facetsList;
14. }
15. mapping TraceRecord::TraceRecord2SourceArtefact() : Artefact {
16.   uuid := self._context._context.value.modelElement.map EObject2UUID();
17.   facets := self.map TraceRecord2SourceArtefactTypeFacet() -> asList();
18. }
19. mapping TraceRecord::TraceRecord2TargetArtefact() : Artefact {
20.   uuid := self._result._result.value.modelElement -> first().map EObject2UUID();
21.   facets := self.map TraceRecord2TargetArtefactTypeFacet() -> asList();
St. }
23. mapping TraceRecord::TraceRecord2SourceArtefactTypeFacet(): Facet {
24.   id := "com.sap.traceframe.facet.TypeFacet";
25.   value := self._context._context.type;
26. }
```

Figura 7.6: Transformación QVT a Trace-DSL

información como el nombre del mapeo que las generó (*TraceLinkNameFacet*) y con la operación correspondiente, ya sea creación, actualización, eliminación o consulta (*TraceLinkMappingFacet*).

Bajo este esquema, la extracción de información de trazabilidad ocurre en dos pasos: en primer lugar, la transformación de modelos es ejecutada permitiendo la conversión de cualquier instancia del metamodelo de trazabilidad interno de QVT en una instancia de Trace-DSL mediante la definición de un mapeo operacional entre ambos metamodelos. El segundo paso consiste en la importación de la instancia Trace-DSL generada en su correspondiente repositorio.

7.5.3. Resumen de la propuesta

Como hemos visto, Grammel y Kastenholz proponen en este trabajo un framework de trazabilidad genérico para aumentar motores de transformación arbitrarios con un mecanismo de trazabilidad basado en el lenguaje de dominio específico Trace-DSL. Dicho framework está basado en una interfase de trazabilidad genérica que provee puntos de conexión entre motores de transformación y de trazabilidad arbitrarios.

Del análisis del esquema, surgen las siguientes observaciones:

- Al igual que con la propuesta de Jouault, ésta opera a nivel de transformación. Es decir, requiere la ejecución de la transformación para la generación de las trazas.

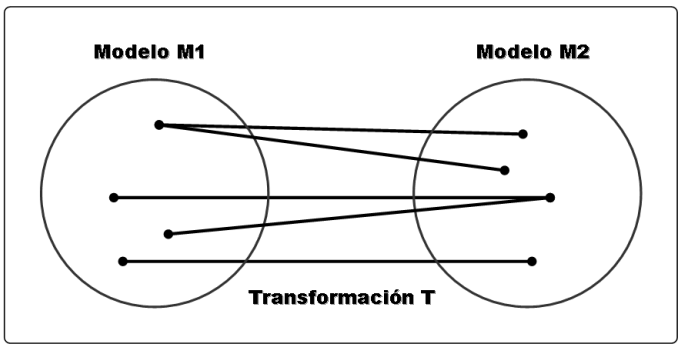


Figura 7.7: Esquema de transformación de modelos

- Permite la interacción con lenguajes de transformación de modelos arbitrarios, dependiendo fuertemente del conector que posibilita el enlace con el motor de transformación.
- En el contexto de QVT, utiliza el lenguaje Operational Mappings para la generación de trazas.
- Define un metamodelo de trazabilidad genérico, tipificando los distintos escenarios de traza (generación de nuevos elementos, actualización, eliminación y consulta).
- Es independiente de los modelos involucrados.
- No agrega información adicional que pudiera alterar los modelos o la lógica de la transformación.

7.6. Framework de trazabilidad para transformación de modelos en Kermeta

Falleri *et al.* proponen en [41] un framework de trazabilidad para transformación de modelos basado en un metamodelo de trazabilidad independiente del lenguaje utilizado. Su trabajo, si bien está inspirado en el esquema de Jouault [51] presenta una sustancial mejora, que es el soporte de trazabilidad para encadenamiento de transformaciones o *transformation chaining*, muy frecuentes en procesos MDA[4], los cuales hacen casi imposible, aún para un desarrollador, reconocer qué elementos del modelo origen del inicio de la cadena conducen a la generación de un elemento dado del último modelo de la cadena.

Una interesante aplicación de la información de trazabilidad en encadenamientos o composición de transformaciones puede encontrarse en el trabajo de Vanhooff *et al.* [72], donde podemos ver una valiosa utilización de las relaciones *inter* e *intra* modelos contenidas en modelos de trazabilidad para subsecuentes transformaciones.

[4] Es decir, procesos que tienen lugar en el contexto de la arquitectura conducida por modelos

7.6.1. Metamodelo de trazabilidad

El principal objetivo de diseño de la propuesta de Falleri y su equipo fue el desarrollo de un metamodelo de trazabilidad independiente del lenguaje. Para esto, los autores proponen un simple y eficiente método para definir un modelo, que consiste en verlo como un conjunto compuesto de elementos (Figura 7.7).

En función de esta visión, se proponen las siguientes definiciones:

Definición 1. *Un modelo M es un conjunto de elementos.*

Definición 2. *Sean M_1 y M_2 dos modelos. Una transformación de modelos es una relación T, $T \subseteq M_1 \times M_2$*

Definición 3. *Una traza de transformación es un grafo bipartito. Los nodos son particionados en dos categorías: nodos fuente y nodos destino.*

En base a estas definiciones, se propone un metamodelo de trazabilidad que represente una estructura de grafo bipartito, cuyo resultado puede observarse en la Figura 7.8a. Como vemos, un paso de traza o *Step* se compone de objetos *Link*. Cada elemento *Link* referencia dos elementos *Object*, representando el origen y el destino de la traza. La clase *Object* es el tipo de elemento más general que puede encontrarse en el lenguaje Kermeta, de manera de asegurar que todo elemento podrá ser almacenado en una traza. Por último, para soportar trazas en encadenamientos de transformaciones, se agregó al metamodelo un objeto *Trace* el cual se compone de elementos *Step* o pasos de traza.

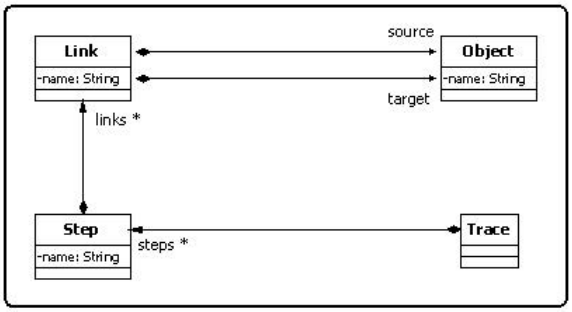
Especificado el metamodelo de trazabilidad, se definen dos operaciones sobre las trazas. Considerando el ejemplo de la Figura 7.8b tenemos que:

Definición 4. *Los predecesores directos (direct parents) de un elemento son todos aquellos elementos que se encuentran directamente enlazados a éste. Por ejemplo, parents $(C3) = \{B3, B4\}$.*

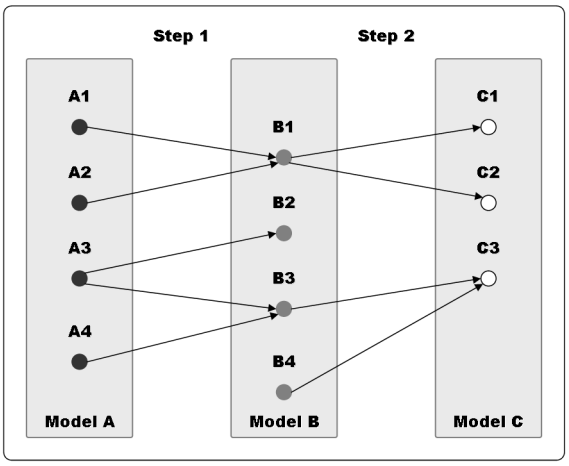
Definición 5. *Los predecesores (parents) de un elemento son los predecesores directos del elemento y los predecesores directos de los predecesores directos (recursivamente). Por ejemplo allparents $(C3) = \{B3, B4, A3, A4\}$.*

7.6.2. Obtención de la información de trazabilidad

En este esquema la obtención de trazas se realiza de manera explícita, incorporando en el código fuente de la transformación las directivas correspondientes para la generación de información de trazabilidad, al igual que en la propuesta de Jouault. A diferencia de ésta, no existe ningún mecanismo que permita automatizar esta tarea, quedando a cargo del desarrollador la responsabilidad de incluirla como parte de la definición de la transformación.

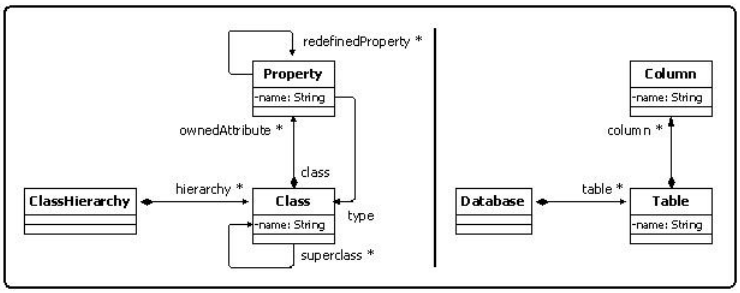


(a) Metamodelo de trazabilidad



(b) Un ejemplo de encadenamiento de transformaciones

Figura 7.8: Propuesta de Falleri



(a) Metamodelos *minuml* y *mindb*

```
1. operation transform ( source : ClassHierarchy ) : DataBase is do
2.   result := DataBase.new // Initialize the target model
3.   trace.initStep("minuml2mindb") // Trace Generating Code
4.   source.hierarchy.each { cls | // Iterate on every class of the source model
5.     var table:Table init Table.new // Create a Table
6.     table.name := String.clone(cls.name) // Copy the name of the Class to the table
7.     result.table.add (table) // Add the table in the target model
8.     trace.addlink("minuml2mindb","class2table", cls, table) // Trace Generating Code
9.     cls.ownedAttribute.each { prop | // Iterate on every Property of the Class
10.       var col:Column init Column.new // Create a new Column
11.       col.name := String.clone(prop.name)
12.       table.column.add(col) // Add the Column to the relative Table
13.       trace.addlink("minuml2mindb", "property2column", prop, col) // Trace Gen. Code
14.   } // End Iterate on every Property of the Class
15.  } // End Iterate on every class of the source model

16. end
```

(b) Transformación *minuml2mindb*

Figura 7.9: Ejemplo código fuente Kermeta

La Figura 7.9b nos muestra el código de la transformación que permite transformar un modelo *minuml* en un modelo *mindb* en Kermeta (Ver definición en Fig. 7.9a). Como vemos en las líneas 3, 8 y 13 del mismo, las directivas para la generación de trazas deben ser incluidas explícitamente por el desarrollador. Como indican los autores en las conclusiones, al ser Kermeta un lenguaje imperativo, dicho código resulta inevitable.

7.6.3. Resumen de la propuesta

Como hemos visto, el esquema de Falleri *et al.* propone un framework interesante, con un metamodelo de trazabilidad sencillo pero que a diferencia de propuestas similares soporta trazas en encadenamiento de transformaciones. La obtención de información de trazabilidad no es automática, se realiza mediante porciones de código fuente en el programa de transformación, cuyo agregado está a cargo del desarrollador.

En resumen, podemos observar como puntos más salientes de la propuesta:

- La generación de trazas opera a nivel de transformación de modelos, es decir, las trazas son generadas al momento de la ejecución de la transformación.
- Presenta un metamodelo genérico, que permite almacenar trazas de cualquier granularidad, es decir, entre elementos de cualquier tipo dentro de la jerarquía de objetos de Kermeta.
- La incorporación del código de soporte de trazabilidad no afecta la lógica de la transformación, aunque afecta su legibilidad.

Por último, podemos mencionar como ventajas adicionales ofrecidas por el esquema analizado la implementación de serialización de trazas mediante XMI 2.0 y la visualización de trazas a través de Graphviz.

7.7. Análisis comparativo de las propuestas

A lo largo del presente capítulo hemos analizado tres enfoques al problema de la gestión de trazabilidad en transformación de modelos. De la amplia variedad de trabajos revisados, hemos seleccionado las propuestas de Jouault, Grammel y Falleri por sus enfoques, afines a la propuesta propia (QVTrace) que detallamos en los Capítulos 4 y 5.

Para poner en contraste los distintos esquemas se seleccionaron dos ejes amplios, pero fundamentales en toda solución de estas características: el metamodelo de trazabilidad propuesto, y los mecanismos definidos para la obtención de trazas.

Luego de repasar los principales detalles de los tres trabajos de referencia seleccionados, a continuación contrastaremos las soluciones según las pautas definidas anteriormente.

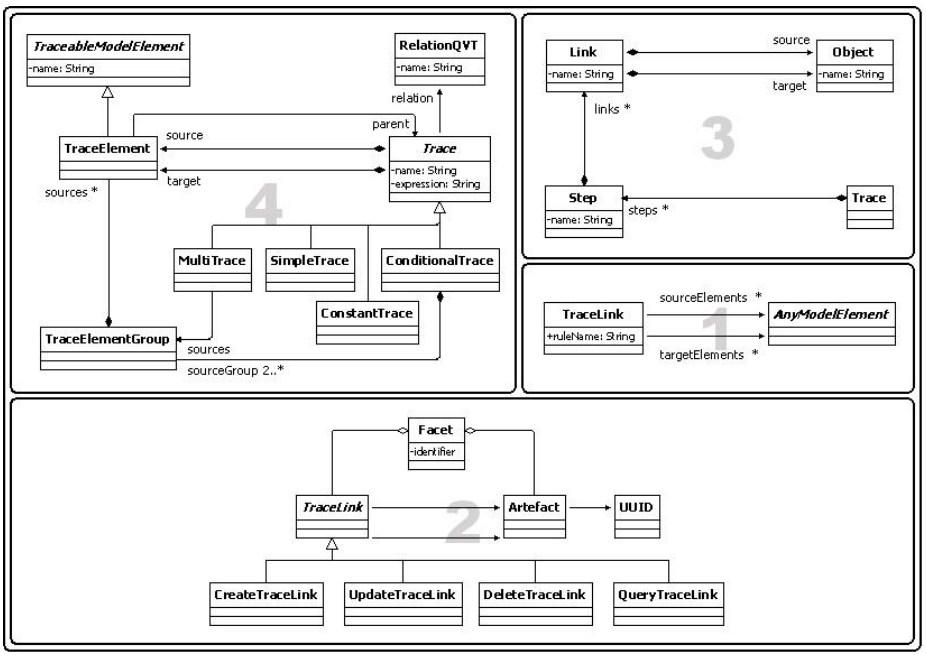


Figura 7.10: Metamodelos de trazabilidad evaluados

7.7.1. Metamodelo de trazabilidad

A lo largo de las secciones hemos descripto las principales características de los metamodelos evaluados. La Figura 7.10 presenta en un mismo cuadro los cuatro esquemas comparados. En primer lugar hemos visto un planteo de Jouault (1) muy genérico, simple, y flexible, cuyo énfasis está puesto en el mantenimiento de la información de trazabilidad desde el punto de vista de la relación entre elementos de los modelos origen y destino.

En segundo lugar, analizamos el metamodelo de Grammel (2), con un enfoque por escenarios que caracteriza distintas operaciones sobre trazas, donde la asignación de tipos a las trazas y a los elementos rastreables es realizada mediante el concepto de facetas o *facets*. Posteriormente presentamos el esquema de Falleri (3), inspirado en el metamodelo de Jouault, pero más completo, con soporte de trazabilidad en encadenamientos de transformaciones.

Finalmente, como una propuesta intermedia, un desarrollo propio implementado en la herramienta QVTrace (4), con un enfoque similar al de Jouault y Falleri, aunque más restrictivo respecto de los participantes de la relación de traza y cuyo eje es el significado de la relación, no solo los integrantes de la misma, esto es la manera en que un elemento del modelo origen se transforma en uno del modelo destino. La propuesta, a diferencia de las otras tres, presenta una tipificación de las trazas en multitrazas (relaciones *muchos-a-uno* entre modelos origen y destino), trazas simples (relaciones *uno-a-uno*), trazas constantes, y finalmente trazas condicionales, aquellas que por el método de inferencia utilizado no pueden ser confirmadas por estar asociadas a porciones de código condicionales (sentencias *If-Then-Else*).

Propuesta	**Jouault (1)**	**Grammel (2)**	**Falleri (3)**	**QVTrace (4)**
Metamodelos	Genérico	Genérico	Genérico	Especif.
Lenguajes	ATL	Trace-DSL, oAW, QVT	Kermeta, XMI	EMF, QVT, Java
Nivel	Alto	Alto	Alto	Bajo
Almacenamiento	Modelo	Repositorio	Modelo	Repositorio
Dimensión	Horiz/Vert.	Horiz/Vert.	Horiz/Vert.	Horiz.

Tabla 7.1: Características de los enfoques evaluados

De acuerdo a lo establecido, los metamodelos en estudio fueron evaluados según los siguientes criterios:

- Tipo de metamodelo: Indica el propósito del metamodelo de trazabilidad utilizado por el enfoque. Se considerarán como opciones: metamodelo genérico o metamodelo específico para el dominio del problema.
- Lenguajes : Enumera los lenguajes de representación de modelos y/o transformación utilizados en la propuesta.
- Nivel: Especifica si la trazabilidad se lleva a cabo entre modelos (alto nivel) o entre elementos de modelos (bajo nivel).
- Almacenamiento de trazas: Indica el tipo de estructura utilizado para salvar las trazas. En particular consideraremos dos opciones: repositorio de trazas específico o modelos de trazabilidad.
- Dimensión de trazabilidad: Evalúa si la propuesta soporta trazabilidad de modelos y/o elementos a diferentes niveles de abstracción. Los posibles valores a asumir por este criterio son horizontal, para el caso que la propuesta soporte trazabilidad entre entidades de igual nivel de abstracción, o vertical, para el caso en que la propuesta soporte trazabilidad entre entidades modelos y/o elementos de distinto nivel de abstracción.

La Tabla 7.1 resume las características de los esquemas comparados. Como podemos observar, los tres metamodelos analizados son menos restrictivos en cuanto a las relaciones que permiten representar dado que permiten las trazas en elementos de distintos niveles de abstracción (criterio "Dimensión"). Nuestra propuesta, en contraste, es más restrictiva y de bajo nivel, dado que permite trazas sólo entre elementos de modelo. Lejos de ser una limitación, esta decisión está relacionada con la obtención del significado o semántica de la relación, la cual sólo puede ser obtenida con una granularidad más fina de los componentes de las trazas.

En cuanto a los lenguajes utilizados para la representación de trazas y modelos, vemos que hay una amplia variedad de opciones. Nuestro criterio en este punto ha sido seguir las recomendaciones de los organismos reconocidos, en este caso el uso de QVT como lenguaje estándar de transformación de modelos propuesto

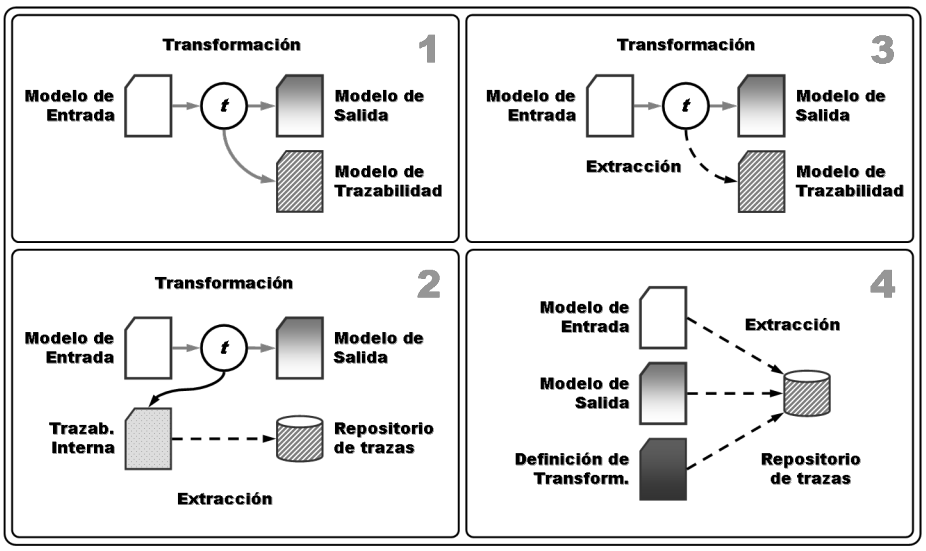


Figura 7.11: Esquemas de obtención de información de trazabilidad

por el Object Management Group (OMG) y el Eclipse Modeling Framework (EMF) para representación de modelos, la respuesta de Eclipse al paradigma de desarrollo conducido por modelos, herramienta ampliamente utilizada por la comunidad de desarrolladores MDD.

Respecto del tipo de metamodelo desarrollado, a diferencia de los demás trabajos nuestra propuesta sugiere un diseño específico que está relacionado con la forma en que son derivadas las trazas. A nuestro criterio, no existe "el" metamodelo de trazabilidad, sino distinto enfoques que enfatizan aspectos puntuales en función de la utilización y los resultados que busquen obtenerse. En particular, el esquema presentado ha sido diseñado con el objetivo de mantener no sólo las relaciones entre elementos, sino su semántica. Por otro lado, como hemos mencionado, el mecanismo de inferencia de trazas ha condicionado el metamodelo, requiriendo una tipificación de trazas no considerada en otros esquemas similares, que contempla las trazas que hemos denominado *condicionales*, y las trazas constantes.

7.7.2. Obtención de la información de trazabilidad

Si el metamodelo de trazabilidad define qué información de rastreabilidad obtener, el segundo criterio de comparación considerado ha sido el "cómo" obtenerla. Como hemos visto, en primer lugar se analizó el mecanismo de Jouault, netamente conducido por modelos, definido con las mismas herramientas del lenguaje de transformación, automatizado, implementado a nivel de transformación, que agrega información extra a la definición de dicha transformación, alterando no la lógica pero sí la legibilidad de la misma. La obtención de la información de trazabilidad es realizada durante el proceso de transformación, siendo el modelo de trazabilidad una de las salidas de dicho proceso, como podemos observar en el esquema de la Figura 7.11 (1).

El trabajo de Grammel, por su parte, está basado en un DSL desarrollado ad hoc, genérico, adaptado a un número arbitrario de lenguajes de transformación de modelos, aunque dependiente de las posibilidades y herramientas que éste pueda proveerle para la construcción de un componente de conexión fundamental[5], implementado a nivel de transformación. La Figura 7.11 (2) ilustra el esquema de obtención de trazas de Grammel para transformaciones QVT: en paralelo al proceso de transformación de modelos t, un conector escrito en Operational Mappings extrae las trazas de un modelo de trazabilidad interna de QVT, las cuales llegan mediante una interfase (omitida en el esquema) al repositorio de trazas definido para el framework.

A continuación se analizó el esquema de Falleri, que incorpora en el programa de transformación las directivas para la generación de trazas, cuyo metamodelo no participa de la transformación, como sí sucede en el trabajo de Jouault. A diferencia de éste, además, el agregado de las sentencias para la captura de trazas es manual. Como vemos en Figura 7.11 (3), el proceso la extracción de trazas es realizado durante la transformación de modelos, pero no como parte de ésta, sino mediante instrucciones adicionales en el programa que generan las trazas y las derivan a un modelo de trazabilidad especialmente definido.

Finalmente la propuesta presentada en este trabajo, QVTrace, esquematizada en la Figura 7.11 (4), que sugiere que desde el análisis de la definición de la transformación y los modelos involucrados es posible identificar ciertos patrones o construcciones del lenguaje que permiten reconocer trazas de manera independiente a la transformación, caracterizando no sólo los participantes de la misma, sino también el significado o forma, que no altera de ninguna manera los modelos ni la lógica de la transformación, automatizada, implementada por fuera de las posibilidades del lenguaje de transformación de modelos, pero integrada con uno de los principales entornos de desarrollo utilizados en el marco del paradigma conducido por modelos.

Como vimos anteriormente, hemos definido una serie de pautas a considerar en la evaluación de los mecanismos de obtención de trazas, a saber:

- El nivel de implementación, el cual especifica si la propuesta se encuentra implementada a nivel de la transformación de modelos, es decir, la obtención de la información de trazabilidad tiene lugar durante el proceso de transformación, o si es independiente del mismo.
- La automatización del mecanismo, un indicador de tipo *Boolean* que establece si la propuesta requiere de algún proceso manual, o si el proceso de obtención de trazas es automático.
- Tipo de obtención de trazas, para el cual consideraremos dos posibles valores: explícito, cuando la obtención de información de trazabilidad es producido mediante instrucciones o directivas específicas, o implícito, cuando la obtención de trazas no requiere ninguna indicación especial en el código fuente.
- Visualización de la información, es decir, la forma en que las trazas obtenidas son presentadas.

[5] Llamado conector

Propuesta	**Jouault (1)**	**Grammel (2)**	**Falleri (3)**	**QVTrace (4)**
Implementación	Transf.	Transf.	Transf.	Indep.
Automatizada?	Sí	Sí	No	Sí
Tipo	Explíc.	Explíc.	Explíc.	Implíc.
Visualización	Modelos gráficos (EMF)	No definida	Graphviz	Modelos gráf. (EMF) y texto
Operaciones	C	C, U, D, Q	C	C
Análisis	No provisto	No provisto	No provisto	Semántica, clasif. y rep.

Tabla 7.2: Características de los mecanismos de obtención de trazas

- Operaciones sobre trazas soportadas. Entre las posibles opciones tenemos creación (C), modificación (U), eliminación (D), consulta (Q), exportación (E), importación (I) y validación (V).
- Análisis de trazas. Algunos de los valores posibles pueden ser: clasificación, reporte de trazabilidad, análisis de impacto de cambios.

La Tabla 7.2 presenta los resultados de la evaluación de los criterios definidos en cada uno de los mecanismos analizados, revelando algunas diferencias importantes. Como podemos observar, la propuesta presentada en este trabajo se distingue claramente de las otras en tres aspectos:

1. La generación de trazas es independiente del proceso de transformación de modelos.
2. La obtención de información de trazabilidad es realizada de manera implícita, es decir, sin instrucciones ni directivas específicas que dirijan el proceso.
3. El esquema sugerido incluye una aproximación al análisis de trazas, en particular en cuanto a la determinación semántica de la trazas, y en menor medida en cuanto a clasificación y reporte de las mismas, en contraste con el resto de propuestas, las cuales no ofrecen este tipo de funcionalidad.

¿Qué ventajas supone la independencia de la obtención de información de trazabilidad respecto del proceso de transformación de modelos? En esencia, el principal beneficio es que la gestión de trazabilidad deja de depender de las posibilidades y limitaciones del lenguaje de transformación. Por otro lado, su naturaleza implícita hace que el proceso de obtención de trazas no afecte de ninguna manera en los modelos origen y destino, ni la lógica de la transformación, permitiendo así obtener trazabilidad en el proceso *model-driven* de manera totalmente transparente. Como contrapartida, la separación con el motor de transformación hace que la propuesta no siempre puede asegurar la totalidad de las relaciones entre los elementos de los modelos intervinientes, dando lugar a

un subtipo de trazas que denominamos *condicionales*, que pueden adoptar una u otra forma dependiendo del modelo de entrada y de la transformación.

Respecto de la tercer diferencia identificada, como hemos visto, a diferencia de las tres propuestas revisadas el esquema presentado incluye una aproximación al análisis de trazas. Si bien el objetivo de QVTrace no ha estado exclusivamente centrado en este tópico, desde el diseño del metamodelo de trazabilidad hasta el desarrollo del mecanismo de obtención de trazas se tuvo en cuenta con el objetivo de aumentar la información de trazabilidad para registrar no solo la relación entre los elementos, sino también su significado o semántica. Adicionalmente la propuesta incluye, desde su diseño también, una facilidad de clasificación de trazas en dos dimensiones: por un lado la distinción entre trazas condicionales y no condicionales, y por otro lado según la multiplicidad de elementos relacionados por las mismas.

7.8. Consideraciones finales

A lo largo del presente capítulo se realizó un análisis comparativo de la propuesta QVTrace con otros trabajos de similar índole, con el objetivo de validar el esquema propuesto a través de una serie de criterios relevantes utilizados en varios de los trabajos de evaluación y análisis de propuestas de trazabilidad en MDD disponibles en la literatura específica del área.

Como hemos visto, nuestra propuesta QVTrace satisface las pautas seleccionadas y en algunos casos ofrece alguna funcionalidad no contemplada en los otros trabajos. No obstante, no es conceptualmente correcto hablar de esquemas mejores ni peores, sólo enfoques diferentes que ofrecen distintas soluciones a una problemática común que es la gestión de información de trazabilidad en transformación de modelos, cada uno con sus ventajas y desventajas. Desde su concepción, tanto el análisis basado en variables como el framework de soporte de éste implementado en QVTrace, han sido diseñados para superar desventajas de otras propuestas. Sin embargo, también tienen sus debilidades, como vimos.

En esencia, la principal diferencia entre QVTrace y los demás esquemas analizados es su enfoque: desacoplar el proceso de obtención de información de trazabilidad de la ejecución de la transformación. De esta forma, la propuesta logra independizarse completamente del motor de QVT, y por ende de la herramienta que lo implemente. Ahora que hemos comparado la propuesta para trazabilidad implementada en QVTrace con otras similares, analizaremos a continuación las diferencias entre las trazas generadas por dicho esquema frente a las trazas implícitas generadas por QVT, según la definición del estándar en [60].

Capítulo 8

La traza QVT vs la traza QVTrace

En el capítulo anterior hemos presentado un análisis comparativo de QVTrace con propuestas de índole similar: esquemas focalizados en la obtención de información de trazabilidad a partir de modelos y transformaciones en el marco del desarrollo conducido por modelos o MDD. QVT, al igual que otros lenguajes de transformación de modelos como MOFScript [10], provee generación implícita de enlaces de traza. Esto significa que como parte del proceso de transformación de un modelo en otro, el estándar QVT [60] prevé la generación trazas. Estas trazas pueden ser serializadas al término de la ejecución de la transformación con el objetivo de una inspección posterior.

La pregunta entonces es qué sentido tiene el desarrollo de un mecanismo de obtención de información de trazabilidad sobre un lenguaje que la genera implícitamente como consecuencia del proceso de transformación de modelos. La respuesta a dicho interrogante fundamenta la presencia de este capítulo dedicado, como veremos a continuación.

8.1. Introducción

Como hemos mencionado anteriormente, y explican claramente Aranega *et al.* en [21], el propósito principal de la traza QVT (a partir a ahora llamaremos así las trazas generadas implícitamente por el lenguaje) es el de brindar soporte al mecanismo de resolución de objetos, indispensable para llevar a cabo la transformación. La trazas provistas por QVT son utilizada internamente por el motor de transformación, y no se encuentran bien adaptadas a propósitos relacionados con la explotación de la información de trazabilidad. Esta limitación es la razón principal que motivó el desarrollo del mecanismo para la gestión de trazabilidad que se presenta en este trabajo de tesis.

A lo largo del presente capítulo desarrollaremos un ejemplo concreto para ilustrar las limitaciones de las trazas QVT, y cómo la solución provista por QVTrace permite complementar el lenguaje con un mecanismo de soporte de trazabilidad

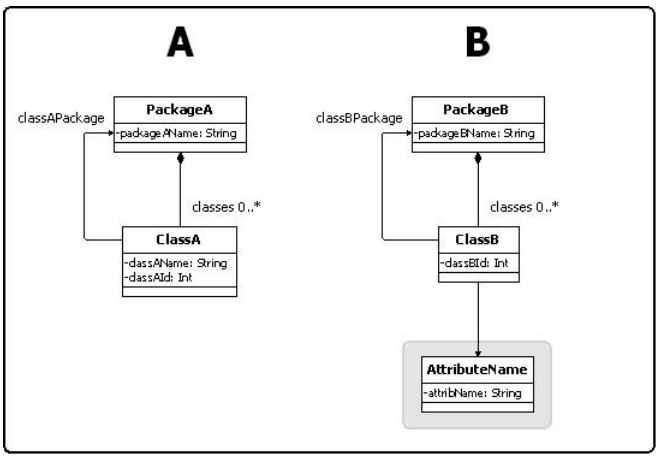


Figura 8.1: Modelos A y B utilizados en la transformación A2B

independiente de la implementación del motor QVT, completamente automático, y que permite explotar al máximo la información de trazabilidad obtenida. El escenario que utilizaremos, que es la sincronización de modelos, ha sido tomado de [21]. En nuestro caso utilizaremos una transformación distinta a la planteada en dicho trabajo, desarrollada especialmente para ilustrar la problemática de las trazas QVT.

8.2. Las limitaciones de la traza QVT

En esencia, la principal limitación de las trazas QVT está relacionada con su diseño: en efecto, éstas han sido creadas como parte del mecanismo de resolución de objetos. Son utilizadas internamente por el motor de transformación y pueden ser invocadas por el desarrollador utilizando directivas específicas. La información contenida en estas trazas es relativa sólo a algunos tipos de elementos, como veremos más adelante, como por ejemplo a clases pero no a atributos.

La información de trazabilidad no es una meta en sí misma, sino que su mayor interés está centrado en su capacidad de ser utilizada como *input* de un algoritmo [21]. Para el caso de QVT, el algoritmo es el de resolución de objetos, y dicha traza se encuentra perfectamente adaptada para eso. En el caso de otro tipo de algoritmos basados en trazas, la traza QVT puede no proveer suficiente información, como mostraremos a continuación.

8.3. La transformación A2B

8.3.1. Los modelos A y B

Durante el presente capítulo, utilizaremos una transformación especialmente desarrollada para la ocasión, que hemos denominado A2B. La transformación

```
transformation a2b(modeloA : A, modeloB : B) {
  top relation PackageToPackage {                                (1)
    pname : String;
    checkonly domain modeloA a : A::PackageA {
      packageAName = pname
    };
    enforce domain modeloB b : B::PackageB {
      packageBName = pname + ' en B'
    };
  }
  top relation PrimitiveClassToWrapperClass {                    (2)
    n : String;
    i : Integer;
    checkonly domain modeloA a : A::ClassA {
      classAName = n,
      classAId = i,
      classAPackage = pa : A::PackageA {}
    };
    enforce domain modeloB b : B::ClassB {
      attribB = attb : B::AttributeName { attribName = n },
      classBId = i,
      classBPackage = pb : B::PackageB {}
    };
    when {
      PackageToPackage(pa,pb);
    }
  }
}
```

Figura 8.2: Código QVT de la transformación A2B

A2B, como su nombre lo indica, mapea instancias de un modelo A, básico, en un modelo B ligeramente diferente. Como vemos en la Figura 8.1, el modelo A es un modelo sencillo, basado en una clase paquete PackageA compuesta por cero o más clases ClassA. Cada ClassA posee un atributo identificador *classAId* y un atributo nombre *classAName*. La transformación A2B permite la transformación de A en un modelo B compuesto por una clase paquete PackageB, conformado por cero o más clases ClassB. Dichas clase ClassB es similar a ClassA, con un atributo identificador *classBId*, pero donde el atributo nombre ha sido empaquetado en otra clase AttributeName, la cual a modo de clase *wrapper*[1] posee la información del nombre de la clase en un atributo llamado *attribName*.

8.3.2. La transformación

La transformación A2B, cuyo código fuente QVT puede verse en la Figura 8.2, consta de dos reglas:

1. *PackageToPackage*: Especifica la transformación de un paquete A en un paquete B. Como puede verse, por cada paquete A la transformación asegurará la creación de un paquete B cuyo nombre (atributo *packageBName*) sea igual al nombre del paquete A (atributo *packageAName*) concatenado con el sufijo "en B".

[1] Una clase *wrapper* o "envoltorio" es una clase utilizada para empaquetar un elemento en un objeto, usualmente utilizada para empaquetar tipos de datos primitivos.

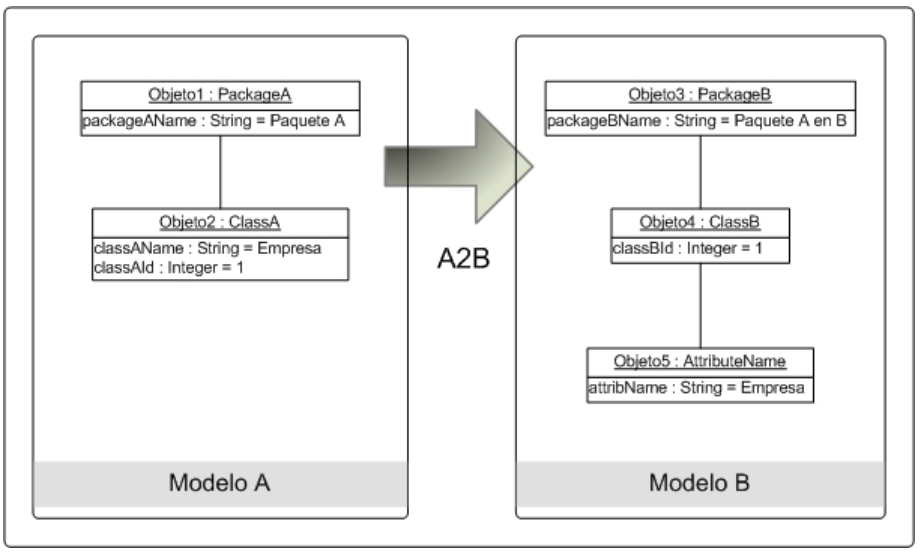


Figura 8.3: La transformación A2B en acción

2. *PrimitiveClassToWrapperClass*: Determina la transformación de una clase A, perteneciente a un paquete A, en una clase B agrupada en un paquete B. Esta última clase tendrá el mismo identificador que la clase generadora (atributo *classAId*), y poseerá el mismo nombre (atributo *classAName*). En este caso, a diferencia de la clase A, en la clase B el atributo nombre es empaquetado en una entidad AttributeName.

8.3.3. A2B en acción

A continuación veremos la transformación A2B en acción. Supongamos que partimos de una instancia del modelo A formada por un objeto PackageA (Objeto1 en Figura 8.3) cuyo nombre es "Paquete A", y por una instancia de la clase ClassA de nombre "Empresa", con ID 1 y perteneciente al paquete de nombre "Paquete A" (Objeto2). Luego de la ejecución de la transformación A2B, por aplicación de la regla *PackageToPackage* se generará un objeto (Objeto3) PackageB cuyo nombre será "Paquete A en B" (es decir, el nombre del objeto origen concatenado al sufijo "en B"). Posteriormente, por aplicación de la regla *PrimitiveClassToWrapperClass* se generaran dos nuevos objetos: uno de tipo ClassB (Objeto4) con identificador 1 (es decir, con el mismo identificador de la instancia ClassA origen), y uno de tipo AttributeName (Objeto5) cuyo *attribName* sera "Empresa" (esto es, con el mismo nombre que el objeto ClassA origen).

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:a2b="urn:a2b.ecore">
  <a2b:PackageToPackage>
    <a href="../metamodel/A.xmi#/"/>
    <b href="../metamodel/B.xmi#/"/>
  </a2b:PackageToPackage>
  <a2b:PrimitiveClassToWrapperClass>
    <a href="../metamodel/A.xmi#//@classes.0"/>
    <b href="../metamodel/B.xmi#//@classes.0"/>
    <pa href="../metamodel/A.xmi#/"/>
    <attb href="../metamodel/B.xmi#//@classes.0/@attribB"/>
    <pb href="../metamodel/B.xmi#/"/>
  </a2b:PrimitiveClassToWrapperClass>
</xmi:XMI>
```

Figura 8.4: Trazas QVT generadas (trace.A2B)

8.3.4. Trazas generadas

8.3.4.1. Trazas QVT

Luego de ejecutar la transformación podemos analizar las trazas QVT generadas. En esencia, QVT Relations genera una traza por cada aplicación de una regla de transformación que da como resultado la creación de un objeto nuevo. En nuestro caso ambas reglas generan objetos nuevos, y son instanciadas una vez cada una, generando de esta forma dos trazas:

1. $Objeto_{PackageA} \rightarrow Objeto_{PackageB}$
2. $Objeto_{ClassA} \rightarrow (Objeto_{ClassB}, Objeto_{AttributeName})$

Como vemos, la traza QVT relaciona objetos. La primer traza indica que una instancia de un elemento PackageA mapea en otra instancia de un objeto PackageB, mientras que la segunda indica que una instancia de un elemento ClassA mapea en un conjunto de objetos que instancian las clases ClassB y AttributeName. La Figura 8.4 muestra el contenido del archivo XML con las trazas QVT generado por la herramienta mediniQVT, la cual ofrece como salida de una transformación QVT las trazas resultantes, de acuerdo a la especificación del lenguaje.

8.3.4.2. Trazas QVTrace

A diferencia de las trazas QVT, las trazas generadas a partir del análisis basado en variables pueden ser obtenidas en cualquier momento ya que la ejecución de la transformación no es un pre-requisito necesario para su funcionamiento. Dado que QVTrace analiza las trazas a nivel de modelos, no de instancias, la cantidad de objetos instanciados no afecta el número de trazas generadas. En este caso, QVTrace ha detectado tres trazas simples (ver Figura 8.5):

1. $packageAName :: PackageA \rightarrow packageBName :: PackageB$, donde $packageBName = packageAName + \text{`}en\ B\text{'}$.

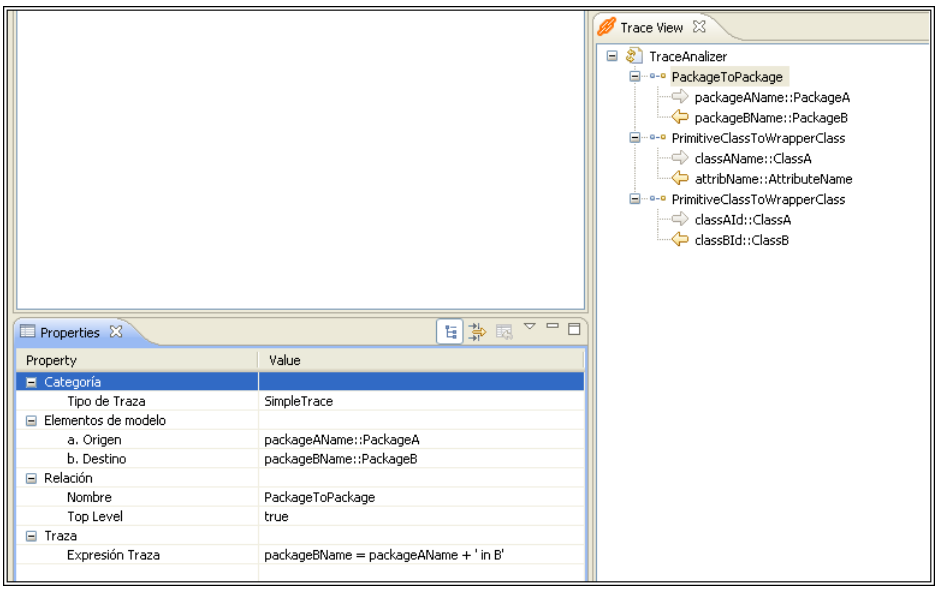


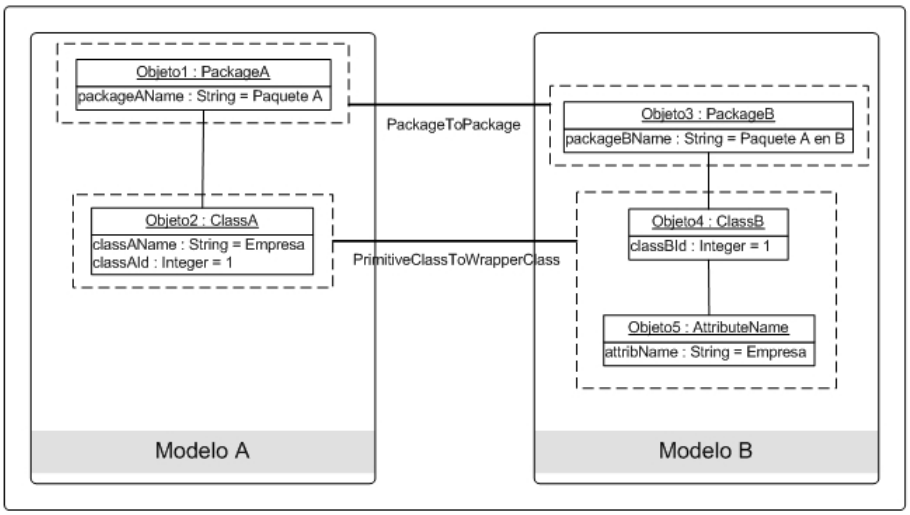
Figura 8.5: Detalle de la traza *PackageToPackage* obtenida con QVTrace

2. $classAId :: ClassA \rightarrow classBId :: ClassB$, donde $classBId = classAId$.
3. $classAName :: ClassA \rightarrow attribName :: AttributeName$, para el cual $attribName = classAName$.

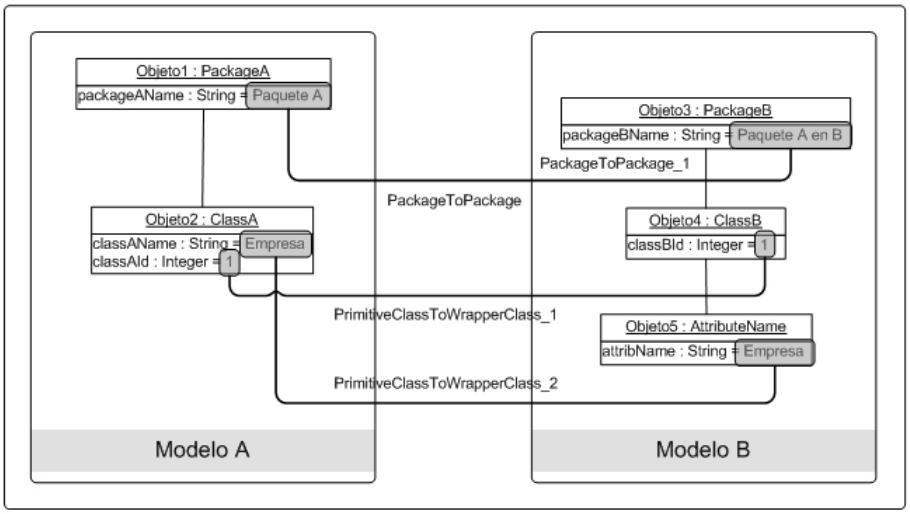
La primer traza indica que el nombre (atributo *packageBName*) de todo objeto *pb* de tipo PACKAGEB será igual al nombre (atributo *packageAName*) del objeto *pa* de tipo PACKAGEA origen, concatenado al sufijo "en B", para toda tupla (pa, pb) que satisface la relación *PackageToPackage*. La segunda traza nos indica que el nombre (*attribName*) de todo objeto ATTRIBUTENAME es igual al nombre (*classAName*) del objeto CLASSA que le da origen. Finalmente la última traza nos indica que el ID (*classBId*) de todo objeto de tipo CLASSB será igual al ID (*classAId*) del objeto CLASSA origen.

8.3.4.3. Diferencias

Luego de generar ambos tipos de traza, a continuación contrastaremos los dos enfoques. La principal diferencia, sustancial, es el nivel de granularidad de cada una de las trazas: mientras la traza QVT relaciona elementos a nivel de instancia, la traza QVTrace establece la relación de trazabilidad entre los elementos, la cual está determinada por la definición de la transformación. La Figura 8.6 (a) ilustra gráficamente este concepto: las trazas QVT relacionan instancias, por ejemplo, Objeto1 con Objeto3 (u Objeto2 con Objeto4 y Objeto5, para el caso de la regla *PrimitiveClassToWrapperClass*). Para el caso de las trazas QVTrace, las relaciones entre los elementos de los modelos se encuentran perfectamente determinadas: como vemos en la Figura 8.6 (b) el nombre de la entidad PACKAGEA (atributo *packageAName*) forma parte del nombre de la entidad PACKAGEB (atributo *packageBName*). La gráfica muestra además las



(a) Trazas QVT de la transformación A2B



(b) Trazas QVTrace de la transformación A2B

Figura 8.6: Las trazas QVT frente a la trazas QVTrace

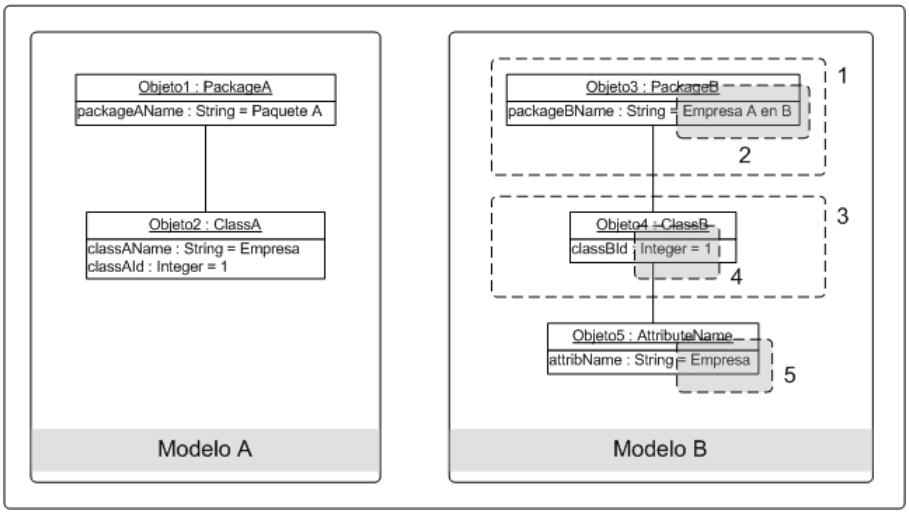


Figura 8.7: Escenarios de sincronización de modelos

otras dos relaciones determinadas con QVTrace, en ambos casos a partir de la regla *PrimitiveClassToWrapperClass*.

8.4. Caso de estudio: Sincronización de modelos

A los efectos de ilustrar las limitaciones de la traza QVT frente a la traza QVTrace analizaremos un caso de explotación de información de trazabilidad como es la sincronización de modelos, una aplicación habitual de las trazas. Para ello, utilizaremos una estrategia de sincronización de modelos muy simple:

- Si un elemento es eliminado en el modelo de salida (*output*), entonces el o los elementos del modelo de entrada (*input*) que dieron lugar a su creación deben ser eliminados. Si éstos están involucrados en el generación de otros elementos del modelo de salida, los mismos deben ser eliminados también.
- Si un elemento del modelo de salida es modificado, entonces el o los elementos que dieron lugar a su creación deben ser modificados. Si estos últimos están involucrados en la generación de otros elementos del modelo de salida, éstos deben ser modificados también.

Si bien la sincronización de modelos implica muchos otros aspectos, esta sencilla estrategia servirá para ilustrar las diferencias entre ambos tipos de trazas. Asumiendo nuestros modelos de ejemplo A y B (Figura 8.7) a continuación estudiaremos cinco posibles escenarios de eliminación/modificación sobre elementos del modelo de salida:

1. Escenario #1: Eliminación de la instancia PackageB del modelo B (1).
2. Escenario #2: Modificación del nombre (atributo *packageBName*) del paquete B instanciado en el ejemplo (2).

3. Escenario #3: Eliminación de la instancia ClassB del modelo B (3).
4. Escenario #4: Modificación del ID (atributo *classBId*) de la instancia ClassB (4).
5. Escenario #5: Modificación del nombre (*attribName*) del atributo AttributeName correspondiente a la instancia ClassB (5).

A continuación evaluaremos la factibilidad de implementación de la estrategia de sincronización propuesta para cada uno de los escenarios planteados, teniendo en cuenta la información de trazabilidad disponible en cada uno de los casos contrastados.

8.4.1. Escenario #1

El escenario plantea la eliminación de la instancia de la clase PackageB, Objeto3, en nuestro modelo B. Para sincronizar el modelo de entrada (*input*), es preciso eliminar los elementos de dicho modelo implicados en la creación de la instancia PackageB, en nuestro caso, la instancia de la clase PackageA, Objeto1, del modelo A. Asimismo, como la instancia del paquete A participa indirectamente en la creación de la instancia de la clase ClassB (todo objeto ClassB debe depender de un objeto PackageB, el cual a su vez satisfaga la relación *PackageToPackage* con un paquete del modelo A[2]), también es preciso eliminar el Objeto4 y el Objeto 5, dependiente de éste último. En todos los casos, tanto las trazas QVT como las trazas QVTrace nos permiten determinar el elemento que originó una instancia dada de nuestro modelo destino, B. De esta manera, la implementación del escenario planteado es viable bajo cualquiera de los dos esquemas de trazabilidad.

8.4.2. Escenario #2

Este escenario propone la modificación de un atributo de un objeto del Modelo B, en este caso, el nombre del paquete B, el cual tiene como valor asignado la cadena de caracteres "Paquete A en B". En el caso de las trazas QVTrace, el escenario es perfectamente implementable: la traza indica qué elemento dio origen al Objeto3 (fue el elemento Objeto1), el atributo de dicho elemento que está relacionado (*packageAName*), y la relación entre ambos, que es $packageBName = packageAName + \text{`}en\ B\text{'}$.

En contraste, la traza QVT no brinda información detallada acerca de qué atributo dio origen al atributo a modificar. Sólo indica el elemento del cual proviene, en este caso, del Objeto1. Como vemos, la información que provee la traza QVT es limitada, y no soporta un escenario sencillo de sincronización de modelos como el planteado.

[2]Definido como pre-condición de la regla *PrimitiveClassToWrapperClass*, en la cláusula *When* de la misma.

8.4.3. Escenario #3

El escenario sugiere la eliminación del Objeto4 del Modelo B. Para este caso, al igual que para el Escenario #1, ambas trazas QVT y QVTrace soportan la sincronización de modelos, dado que permiten la identificación de la instancia del Modelo A que dio origen al Objeto4, o sea el Objeto2, el cual es el objeto que se debe eliminar para lograr la sincronización de (los modelos) A y B. La ejecución del escenario dará como resultado además la eliminación de la instancia Objeto5, también del Modelo B, dado que este no tiene razón de ser por sí mismo, y su existencia depende de la existencia de una instancia de la clase ClassB.

8.4.4. Escenario #4

Este caso plantea la modificación del atributo ID (*classBId*) de la instancia de la clase ClassB en el Modelo B (Objeto4). Al igual que para el Escenario #2, sólo la traza QVTrace provee la información suficiente como para poder identificar al elemento que le dio origen a la instancia (Objeto2), el atributo asociado (*classAId*, de la instancia de clase ClassA), y la relación entre ambos atributos $classBId = classAId$ (Traza No.2 en Subsección 8.3.4.2). La traza QVT sólo permite la identificación del elemento que le dio origen al objeto en cuestión, pero de ninguna manera el atributo asociado (Traza No.2 en Subsección 8.3.4.1).

8.4.5. Escenario #5

Finalmente, el Escenario #5 plantea un caso donde el mapeo definido en la transformación no es *uno-a-uno*, es decir, el concepto relacionado no previene de una clase análoga o "espejo". Nuevamente, el escenario propuesto es soportado sólo por la traza QVTrace, la cual provee la información necesaria para poder identificar el origen del atributo *attribName* de la instancia de la clase AttributeName (Objeto5) en el Modelo A (Traza No.3 en Subsección 8.3.4.2). La traza QVT (Traza No.2 en Subsección 8.3.4.1) permite conocer sólo al elemento que dio origen a la instancia en cuestión, pero al igual que para el Escenario #4, no brinda ninguna referencia acerca del atributo del cual proviene.

8.5. Consideraciones finales

A lo largo del presente capítulo hemos planteado un ejemplo de transformación sencillo, de un modelo A en un modelo B, y mostramos la información de trazabilidad generada por QVT y obtenida por QVTrace, para dicha transformación. Posteriormente presentamos un caso de estudio de sincronización de modelos, el cual se apoya en la información de trazabilidad disponible, requisito indispensable para su implementación. Como hemos visto, para los escenarios que planteaban modificación de atributos en elementos de modelo la traza QVT resultó limitada, dado que la información que provee no es suficiente para la identificación de los conceptos a modificar. En contraste, la traza provista por QVTrace soporta todos los escenarios planteados por el caso de estudio, dado que ofrece un nivel de granularidad mayor al de la traza QVT, y permite

obtener una mayor cantidad de relaciones de trazabilidad, fundamentales para poder determinar el origen de cada elemento en el modelo destino, luego de una transformación.

La traza QVT cumple perfectamente el rol para el cual ha sido diseñada. En la búsqueda de alternativas que maximicen la explotación de información de trazabilidad hemos desarrollado un mecanismo, complementario, para la obtención de información de trazabilidad, el cual permite subsanar la limitaciones de la traza QVT para este propósito, y posibilita un mejor aprovechamiento de estos datos.

Capítulo 9

Conclusiones y trabajo futuro

Finalmente, después de un largo camino recorrido, hemos llegado al último tramo. A modo de cierre, detallaremos en el presente capítulo un resumen del trabajo realizado, las principales contribuciones del mismo, y la propuesta de trabajo futuro.

9.1. El trabajo de tesis

El presente trabajo abordó la problemática de la implementación de un sistema de soporte de trazabilidad en transformación de modelos en el contexto del desarrollo conducido por modelos.

A lo largo del mismo se ha realizado un repaso de las características más destacadas del paradigma MDD, detallando el rol central de los modelos, y de sus transformaciones. Asimismo, se presentó la propuesta del Object Management Group (OMG) para el desarrollo *model-driven*, la Arquitectura Conducida por Modelos o MDA (Model-Driven Architecture), y se subrayaron los esfuerzos del consorcio por consolidar la adopción de un lenguaje estándar de transformación de modelos, el cual finalmente fue Query-View-Transformation o QVT.

A continuación se abordó el concepto de trazabilidad en términos genéricos, y su significado en el marco del MDD. Se puntualizó la importancia de contar con esta información, y la problemática asociada a su obtención. Se abordaron diversas consideraciones de diseño de un esquema de trazabilidad y se presentó el concepto de metamodelo, con sus principales características. Luego, se mostraron diversos esquemas de trazabilidad en MDD agrupados según su enfoque: propuestas basadas en requerimientos, propuestas basadas en modelos y finalmente las basadas en transformaciones, con especial énfasis en éstas últimas, que plantean un enfoque similar a nuestra propuesta.

Presentada la información contextual y del estado del arte, se detalló nuestra propuesta para trazabilidad en transformación de modelos, implementada en el prototipo QVTrace. La misma consta, en síntesis, de un marco para la obtención de información de trazabilidad en transformación de modelos, soportado por un metamodelo específico orientado a las relaciones de trazabilidad, y de un

mecanismo de inferencia de trazas a partir del análisis de código fuente, llamado Análisis Basado en Variables.

Posteriormente, a efectos de ilustrar el funcionamiento del mecanismo de inferencia de trazas presentado, se desarrollaron dos casos de estudio: la transformación de un modelo UML en código fuente Java, y la transformación de una referencia bibliográfica BibTeX en un formato DocBook. Cada uno de los ejemplos abarcó la definición de la transformación de un modelo en otro, y el resultado obtenido tras procesar la misma con QVTrace.

A continuación, se validó la propuesta mediante un estudio comparativo frente a tres esquemas similares, de gestión de trazabilidad en transformación de modelos, utilizando pautas y criterios de comparación propuestos por otros autores en varios de los estudios de evaluación más importantes sobre trazabilidad en MDD. Como resultado, pudimos ver que no sólo nuestra propuesta se encuentra a la altura de otras de su misma índole, por los aspectos que abarca, sino que presenta características originales y funcionalidad no presente en ningún otro trabajo de la materia.

Por último, se comparó la traza generada por QVTrace frente a la traza implícita generada por QVT al momento de una transformación. A través de un caso de estudio de sincronización de modelos, con cinco escenarios distintos, se mostraron las principales limitaciones de la traza QVT para su utilización en un contexto de explotación de información de trazabilidad, en contraste con ventajas ofrecidas por las trazas QVTrace.

9.2. Principales contribuciones

Los resultados obtenidos en este proyecto de tesis son varios. No obstante, las contribuciones más importantes son a nuestro criterio, y por orden de importancia: el desarrollo de un marco de trabajo o framework de trazabilidad para transformación de modelos, la propuesta de un mecanismo de inferencia de trazas original, y la confección de un metamodelo específico orientado a las relaciones de trazabilidad.

A continuación analizaremos cada uno de ellos en detalle, indicando los logros alcanzados, sus ventajas y desventajas.

9.2.1. QVTrace como framework de trazabilidad

¿Qué hay de nuevo en QVTrace? En esencia, su enfoque: desacoplar el proceso de obtención de información de trazabilidad de la ejecución de la transformación. Como hemos visto, QVTrace es más que un mecanismo de inferencia de trazas. QVTrace propone un marco de trabajo de trazabilidad para transformación de modelos integrado con el entorno de desarrollo más utilizado por la comunidad MDA, Eclipse. Con una arquitectura compuesta por un módulo lector/analizador de modelos, un módulo lector/analizador de transformaciones, y por un componente que analiza y procesa esta información, *TraceAnalizer*, permite al desarrollador la obtención de trazas a nivel de modelos, posibilitando

su utilización como herramienta de *debbugging* en la definición de transformaciones, permitiendo la obtención de información sobre el modelo destino, previo a la realización de la transformación, y por otro lado, que la misma pueda servir para verificar y/o asegurar la consistencia entre los modelos origen y destino en caso que éste último sea modificado fuera del flujo natural que propone el paradigma.

Si bien QVTrace está implementado para trabajar con un lenguaje de transformación de modelos específico, QVT, sus características de diseño hacen que sea factible su modificación, con relativamente poco esfuerzo, para brindar soporte a otro lenguaje de transformación, y otra representación de modelos. Su diseño es flexible, permitiendo la incorporación de nuevos algoritmos de inferencia de trazas, más allá del propuesto, o facilitando la extensión del metamodelo de trazabilidad, cuya gestión se encuentra desacoplada del proceso de generación de trazas en un componente *TraceFactory*, que es el que conoce el metamodelo y se encarga de la creación de las trazas. Por otro lado, su integración con Eclipse incrementa su extensibilidad y permite su interacción con herramientas de terceros.

9.2.2. QVTrace como mecanismo de inferencia de trazas

La premisa fundamental sobre la cual fue concebido QVTrace es que la definición de una transformación de modelos contiene información de trazabilidad implícita. Bajo este concepto se trabajó en un mecanismo que permitiese explotarla, obteniéndose como resultado el denominado Análisis Basado en Variables o VBA. En efecto, se detectó que mediante el análisis de las variables declaradas en el contexto de las reglas o relaciones de una transformación QVT es posible inferir relaciones de trazabilidad entre los elementos de los modelos origen y destino involucrados.

La propuesta fue contrastada frente a esquemas similares presentados por otros autores. Tres características la diferencian del resto: es independiente de la transformación, por lo que no depende del motor y ni de una implementación del lenguaje QVT particular, la obtención de información de trazabilidad es realizada de manera implícita, es decir, sin instrucciones ni directivas específicas que dirijan el proceso, y propone una aproximación al análisis de trazas.

En contrapartida el esquema no provee una forma de priorizar las trazas, por lo que es posible que se generen múltiples trazas triviales o equivalentes. Por otro lado, el mecanismo no asegura que el conjunto de trazas inferidas sea exhaustivo, es decir, que permita la inferencia de la totalidad de las trazas posibles.

Las trazas resultantes obtenidas por QVTrace fueron comparadas con las trazas implícitas definidas por el estándar QVT. Como vimos, las trazas QVTrace presentan ventajas respecto de la información de trazabilidad que contienen respecto de las trazas QVT, logrando subsanar las limitaciones de éstas frente a un escenario de aplicación de trazabilidad, como puede ser una sincronización de modelos.

9.2.3. El metamodelo de trazabilidad QVTrace

Como todo esquema de gestión de trazabilidad QVTrace es soportado por un metamodelo de trazabilidad, el cual ha sido especialmente diseñado para alcanzar los objetivos propuestos para la herramienta. El metamodelo de trazabilidad utilizado por QVTrace es de bajo nivel, dado que sólo permite relacionar entre sí elementos de modelo, y es de diseño específico, ya que se encuentra condicionado fuertemente por el mecanismo de obtención de la información de trazabilidad. Más allá de algunos matices, el análisis comparativo con otros metamodelos mostró dos grandes diferencias con el resto: la tipificación de trazas, en concretas y condicionales, y la orientación a las relaciones de trazabilidad, es decir, la iniciativa de proveer el significado, la semántica, de la traza.

Naturalmente, la propuesta también tiene sus puntos débiles. Como todo diseño específico para una determinada aplicación, el metamodelo propuesto podría no ser lo suficientemente flexible como para adaptarse a otra aplicación. La descripción de la relación de trazabilidad es textual. Y no soporta operaciones sobre trazas más allá de la creación de las mismas. Nos preguntamos entonces si estamos frente a un buen o mal metamodelo. ¿Existen mejores y peores metamodelos de trazabilidad? Definitivamente la respuesta es no. Existen metamodelos diseñados para propósitos específicos, que en algunos casos son más genéricos y flexibles, y en otros sólo resultan de utilidad en el medio para el cual fueron diseñados. En este caso, el metamodelo de trazabilidad implementado en QVTrace satisface los objetivos para los cuales fue diseñado. Pero obviamente es susceptible de ser mejorado.

9.3. El balance resultados vs objetivos

Desde el comienzo, nos planteamos como meta principal del trabajo ".. el diseño de una herramienta que pueda ser integrada a una herramienta de transformación de modelos, por ejemplo mediniQVT, en un entorno de desarrollo ampliamente utilizado como Eclipse, y asista al desarrollador automatizando el proceso de obtención de trazas o *links* entre elementos de los modelos origen y destino ..".

Como indicamos, la obtención de este amplio y ambicioso objetivo comprendía ".. por un lado, el análisis del problema de trazabilidad en el ámbito de desarrollo conducido por modelos (MDD, acrónimo de *Model-Driven Development*), seguido de la elaboración de un metamodelo de trazabilidad fácil de implementar, conciso y efectivo, que permita el aprovechamiento completo de sus beneficios sin entorpecer el proceso de desarrollo. Y finalmente, la implementación de la propuesta como herramienta complementaria para el proceso de definición de transformaciones entre modelos ..".

A la luz de los resultados, consideramos que se ha alcanzado la meta propuesta: en efecto, QVTrace aborda y propone una solución a la problemática de la obtención de información de trazabilidad automatizada y se encuentra totalmente integrada a un entorno de desarrollo ampliamente utilizado por la comunidad *model-driven*, permitiendo que la misma pueda ser utilizada de forma complementarias a otras herramientas para desarrollo conducido por modelos.

9.4. Trabajo futuro

Existen dos grandes líneas base sobre las cuales debemos seguir trabajando. La primera de ellas consiste en el mejoramiento de la obtención y mantenimiento de la información de trazabilidad, área que todavía ofrece algunas limitaciones. Como hemos indicado, el analizador de transformaciones sólo acepta un subconjunto del lenguaje QVT, limitando así la naturaleza de las transformaciones que QVTrace puede procesar. La ampliación de la gramática soportada por el analizador, que potenciará la herramienta, está indicada como una línea de trabajo futuro.

Como segundo paso a seguir podemos indicar el mejoramiento del mecanismo de inferencia de trazas. En particular, sería de mucho interés poder determinar nuevas construcciones o casos de traza que permitan inferir nueva información de rastreabilidad no identificada hasta el momento, o en su defecto, en caso de encontrar el límite de los posibles tipos de trazas que la técnica permite generar, poder demostrarlo formalmente.

Por otro lado, como subrayamos en la Sección 6.4.2, las restricciones sobre los integrantes de las relaciones de trazabilidad limitan la cantidad de trazas que pueden ser inferidas. En particular, todas las trazas generadas por QVTrace son relaciones entre elementos de modelo. La posibilidad de incorporar relaciones *modelo-elemento* y su detección son un buen punto para trabajo futuro, sobre todo considerando que el diseño de la solución nos brinda las facilidades para hacerlo mediante la definición de una nueva estrategia de trazabilidad (*trace strategy*), la cual puede ser incorporada al analizador de trazas incrementando su capacidad de inferencia de trazas.

En la misma línea, aunque no como mejora sino como nueva funcionalidad, cabe mencionar la posibilidad de incorporar nuevas operaciones sobre trazas. En efecto, dado que QVTrace sólo implementa operaciones de creación de trazas, sería de interés la implementación de, por ejemplo, operaciones de modificación y eliminación de trazas, las cuales sirven de complemento a la generación de trazas y facilitan el mantenimiento de la consistencia de los modelos origen y destino.

La segunda línea base de trabajo futuro consiste en el aprovechamiento de la información de trazabilidad. En particular resulta de mucho interés la incorporación de una mayor funcionalidad en el análisis de trazas, déficit indicado en la mayoría de los esquemas de trazabilidad revisados, y también en QVTrace. En este sentido, un buen punto de partida sería el estudio de distintas posibilidades para la utilización de la información de trazabilidad en el contexto del desarrollo conducido por modelos, y su correspondiente implementación en QVTrace.

Apéndice A

Lenguaje QVT

A.1. Gramática del lenguaje QVT aceptado por QVTrace

```
<TopLevel>                        ::= <TransformationList>
<TransformationList>              ::= <TransformationList> <Transformation>
<Transformation>                  ::= 'transformation' <Identifier>
                                      '{' <ModelDeclarationList> '}'
                                      '{' <KeyDeclarationList> <RelationOrQueryList> '}'
<ModelDeclarationList>            ::= <ModelDeclaration> | <ModelDeclarationList> ',' <ModelDeclaration>
<IdentifierList>                  ::= <Identifier> | <IdentifierList> ',' <Identifier>
<KeyDeclarationList>              ::= | <KeyDeclarationList> <KeyDeclaration>
<RelationOrQueryList>             ::= | <RelationOrQueryList> <Relation>
                                      | <RelationOrQueryList> <Query>
<ModelDeclaration>                ::= <Identifier> ':' <MetaModelIdList>
<MetaModelIdList>                 ::= <Identifier> | <MetaModelIdList> ',' <Identifier>
<KeyDeclaration>                  ::= 'key' <PathName> '{' <PropertyIdList> '}' ';'
<PropertyIdList>                  ::= <Identifier> | <PropertyIdList> ',' <Identifier>
<Relation>                        ::= <IsTopLevel> 'relation' <Identifier>
                                      '{' <VarDeclarationList> <DomainOrPrimitiveTypeDomainList>
                                      <When> <Where> '}'
<DomainOrPrimitiveTypeDomainList> ::= <Domain>
                                      | <PrimitiveTypeDomain>
                                      | <DomainOrPrimitiveTypeDomainList> <Domain>
                                      | <DomainOrPrimitiveTypeDomainList> <PrimitiveTypeDomain>
<IsTopLevel>                      ::= | 'top'
<VarDeclaration>                  ::= <IdentifierList> ':' <Type> ';'
<VarDeclarationList>              ::= | <VarDeclarationList> <VarDeclaration>
<Domain>                          ::= <CheckEnforceQualifier> 'domain' <Identifier>
                                      <OptionalIdentifier> ':' <Type>
```

```
                                    '{' <PropertyTemplateList> '}' ';'
<OptionalIdentifier>            ::= | <Identifier>
<PropertyTemplateList>          ::= | <PropertyTemplate> | <PropertyTemplateList> ',' <PropertyTemplate>
<PrimitiveTypeDomain>           ::= 'primitive' 'domain' <Identifier> ':' <Type> ';'
<CheckEnforceQualifier>         ::= 'checkonly' | 'enforce'
<When>                          ::= | 'when' '{' <Pattern> '}'
<Where>                         ::= | 'where' '{' <Pattern> '}'
<Pattern>                       ::= | <Pattern> <Predicate>
<Predicate>                     ::= <OclExpression> ';'
<Query>                         ::= 'query' <PathName> '{' <ParamDeclarationList> '}' ':'
                                    <Type> <SemiOroclExpressionInBra>
<SemiOroclExpressionInBra>      ::= ';' | '{' <OclExpression> '}'
<ParamDeclaration>              ::= <Identifier> ':' <Type>
<ParamDeclarationList>          ::= | <ParamDeclaration> | <ParamDeclarationList> ','
<VariableDeclarationList>       ::= <VariableDeclaration> | <VariableDeclarationList> ','
                                    <VariableDeclaration>
<VariableDeclaration>           ::= <Identifier> ':' <Type> '=' <OclExpression>
                                    | <Identifier> ':' <Type>
                                    | <Identifier> '=' <OclExpression>
                                    | <Identifier>
<Type>                          ::= <PathName>
<Template>                      ::= <ObjectTemplate>
<ObjectTemplate>                ::= ':' <Type> '{' <PropertyTemplateList> '}'
                                    | <Identifier> ':' <Type> '{' <PropertyTemplateList> '}'
<PropertyTemplate>              ::= <Identifier> '=' <OclExpression>
<OclExpression>                 ::= <Template> | <LiteralExp> | '{' <OclExpression> '}'
                                    | <PathName> | <OclExpression> '(' ')'
                                    | <OclExpression> '(' <OclExpression> ')'
                                    | <OclExpression> '(' <OclExpression> ','
                                    <ArgumentList> ')'
                                    | <OclExpression> '{' <OclExpression> ','
                                    <VariableDeclaration> '|' <OclExpression> '}'
                                    | <OclExpression> '(' <OclExpression> ':'
                                    <Type> ',' <VariableDeclaration> '|' <OclExpression> ')'
                                    | <OclExpression> '(' <VariableDeclaration>
                                    '|' <OclExpression> ') | 'not' <OclExpression>
                                    | '-' <OclExpression> | <OclExpression> '*' <OclExpression>
                                    | <OclExpression> '/' <OclExpression>
                                    | <OclExpression> 'div' <OclExpression>
                                    | <OclExpression> 'mod' <OclExpression>
                                    | <OclExpression> '+' <OclExpression>
                                    | <OclExpression> '-' <OclExpression>
                                    | 'if' <OclExpression> 'then' <OclExpression>
                                    'else' <OclExpression> 'endif'
```

```
                                    | <OclExpression> '<' <OclExpression>
                                    | <OclExpression> '>' <OclExpression>
                                    | <OclExpression> '<=' <OclExpression>
                                    | <OclExpression> '>=' <OclExpression>
                                    | <OclExpression> '=' <OclExpression>
                                    | <OclExpression> '<>' <OclExpression>
                                    | <OclExpression> 'and' <OclExpression>
                                    | <OclExpression> 'or' <OclExpression>
                                    | <OclExpression> 'xor' <OclExpression>
<ArgumentList>                  ::= <OclExpression>
                                    | <ArgumentList> ',' <OclExpression>
<LiteralExp>                    ::= <PrimitiveLiteralExp>
<PrimitiveLiteralExp>           ::= INTEGER | REAL | STRING | TRUE | FALSE
<PathName>                      ::= <Identifier>
                                    | <PathName> '::' <Identifier>
<Identifier>                    ::= <Letter><Alpha>*
<Letter>                        ::= <Lowercase> | <Uppercase> | [_]
<Alpha>                         ::= <Letter> | <Digit>
<Digit>                         ::= [0-9]
<Lowercase>                     ::= [a-z]
<Uppercase>                     ::= [A-Z]
```

Bibliografía

[1] Apache Xerces. Parsers XML y componentes de software relacionados. http://xerces.apache.org/.

[2] ATL Transformations Zoo. http://www.eclipse.org/atl/atlTransformations/.

[3] BibTeX. http://www.bibtex.org/.

[4] BibTeXML. http://bibtexml.sourceforge.net/.

[5] CUP. Generador de Parsers LALR para Java. http://www2.cs.tum.edu/projects/cup/.

[6] The DocBook Project. http://docbook.sourceforge.net/.

[7] Graphviz - Graph Visualization Software. http://www.graphviz.org/.

[8] Kent Modelling Framework. Universidad de Kent. Canterbury, UK. http://www.cs.kent.ac.uk/projects/kmf/.

[9] mediniQVT. http://projects.ikv.de/qvt.

[10] MOFScript. http://www.eclipse.org/gmt/mofscript/.

[11] openArchitectureWare. http://www.openarchitectureware.org/.

[12] Proyecto Epsilon. http://www.eclipse.org/epsilon/.

[13] Triskell Metamodeling Kernel. http://www.kermeta.org/.

[14] Variable-Based Analysis for Traceability in Model Transformations. http://www.sadio.org.ar/wp-content/uploads/2013/11/6-Variable-Based_Analysis.pdf.

[15] Meta-Object Facility (MOF) 1.4 Specification. http://www.omg.org/technology/documents/formal/mof.htm, April 2002.

[16] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. *IBM System Journal*, 45(3):515–526, July 2006.

[17] N. Aizenbud-Reshef, R. F. Paige, J. Rubin, Y. Shaham-Gafni, and D. S. Kolovos. Operational Semantic for Traceability. In *ECMDA-TW: Traceability Workshop at European Conference on Model Driven Architecture*, pages 7–14, Nuremberg, Germany, November 2005.

[18] Ian Alexander. Towards Automatic Traceability in Industrial Practice. In *Proceedings of the 1st International Workshop on Traceability*, pages 26–31, Edinburgh, Scotland, September 2002.

[19] Joao Paulo Almeida, Pascal van Eck, and Maria-Eugenia Iacob. Requirements Traceability and Transformation Conformance in Model-Driven Development. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '06, pages 355–366, Washington DC, USA, 2006. IEEE Computer Society.

[20] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng*, pages 970–983, 2002.

[21] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an Alternative Trace for QVT. In *Workshop on Multi-Paradigm Modeling*, Olso, Norway, October 2010.

[22] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[23] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[24] Jorge Biolchini, Paula Gomes Mian, and Ana Candida Cruz Natali. Systematic review in software engineering. Technical Report RT-ES 679/05, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, May 2005.

[25] Lossan Bondé, Pierre Boulet, and Jean-Luc Dekeyser. Traceability and interoperability at different levels of abstraction in model-driven engineering. In A. Vachoux, editor, *Applications of Specification and Design Languages for SoCs*, pages 263–276. Springer Netherlands, 2006.

[26] Artur Boronat, José Á. Carsí, and Isidro Ramos. Automatic support for traceability in a generic model management framework. In *Proceedings of the First European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'05, pages 316–330, Berlin, Heidelberg, 2005. Springer-Verlag.

[27] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[28] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 3 edition, 2008.

[29] J. Cleland-Huang, C.K. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *Software Engineering, IEEE Transactions on*, 29(9):796 – 810, sept. 2003.

[30] J. Cleland-Huang and D. Schmelzer. Dinamically tracing non-functional requirements through design pattern invariants. In *Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering*, October 2003.

[31] Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. Goal-centric traceability for managing non-functional requirements. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 362–371, New York, NY, USA, 2005. ACM.

[32] Alistair Cockburn. *Agile software development.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[33] Krzysztof Czarnecki. *Domain Engineering.* John Wiley & Sons, Inc., 2002.

[34] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[35] Philippe Desfray. MDA - When a major software industry trend meets our toolset, implemented since 1994, 2001.

[36] Nicholas Drivalos, Richard F. Paige, Kiran J. Fern, and Dimitrios S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proceedings of the 4th ECMFA Traceability Workshop*, Berlin, Germany, June 2008. ECMDA'08.

[37] Nikolaos Drivalos, Dimitrios Kolovos, Richard Paige, and Kiran Fernandes. Engineering a DSL for software traceability. In *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 151–167. Springer Berlin / Heidelberg, 2009.

[38] Nikolaos Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. A state-based approach to traceability maintenance. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, pages 23–30, New York, NY, USA, 2010. ACM.

[39] Alexander Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng.*, 29(2):116–132, February 2003.

[40] Alexander Egyed and Paul Grünbacher. Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *Proceedings of the 17th IEEE international conference on Automated Software Engineering (ASE)*, pages 163–171, Edinburgh, Scotland, September 2002.

[41] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a traceability framework for model transformations in Kermeta. In *ECMDA-TW: Traceability Workshop at European Conference on Model Driven Architecture*, pages 30–39, Bilbao, Spain, July 2006.

[42] I. Galvao Lourenco da Silva and A. Göknil. Survey of traceability approaches in model-driven engineering. In *Proceedings of the Eleventh IEEE International EDOC Enterprise Computing Conference*, pages 313–324, Los Alamitos, October 2007. IEEE Computer Society Press.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[44] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101, 1994.

[45] Birgit Grammel and Stefan Kastenholz. A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, pages 7–14, Paris, France, 2010.

[46] John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11):960–981, November 1998.

[47] B. Hailpern and P. Tarr. Model-driven Development: the good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, 2006.

[48] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne. Improving requirements tracing via information retrieval. In *in Proceedings of the International Conference on Requirements Engineering (RE*, pages 151–161, 2003.

[49] IEEE. IEEE Recommended Practice for Software Requirements Specifications, 1998.

[50] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2005. Springer Verlag.

[51] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, Nuremberg, Germany, November 2005.

[52] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[53] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[54] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (eml). In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 215–229, Berlin, Heidelberg, 2006. Springer-Verlag.

[55] Tihamer Levendovszky, Daniel Balasubramanian, Kevin Smyth, Feng Shi, and Gabor Karsai. A Transformation Instance-Based Approach to Traceability. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, pages 55–60, New York, NY, USA, 2010. ACM.

[56] A.E. Limón and J. Garbajosa. The Need for a Unifying Traceability Scheme. In *ECMDA-TW: Traceability Workshop at European Conference on Model Driven Architecture*, pages 47–56, Nuremberg, Germany, November 2005.

[57] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[58] Omar Martínez Grassi and Claudia Pons. Análisis basado en variables para trazabilidad en transformación de modelos. In *Proceeding of the 13th Argentine Symposium on Software Engineering (ASSE 2012)*, JAIIO'41, La Plata, Buenos Aires, 27-31 Agosto 2012.

[59] Stephen J. Mellor, Scott Kendall, Axel Uhl, and Dirk Weise. *MDA Distilled.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[60] Object Management Group. Meta object facility (MOF) 2.0 Query/View/transformation specification version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/, April 2008.

[61] Jon Oldevik and Tor Neple. Traceability in Model to Text Transformations. In *ECMDA-TW: Traceability Workshop at European Conference on Model Driven Architecture*, pages 63–68, Bilbao, Spain, July 2006.

[62] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, January 2001.

[63] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley Professional, 2004.

[64] Pedro Sanchez, Diego Alonso, Francisca Rosique, Barbara Alvarez, and Juan A. Pastor. Introducing Safety Requirements Traceability Support in Model-Driven Development of Robotic Applications. *IEEE Transactions on Computers*, 60(8):1059–1071, 2011.

[65] Iván Santiago, Álvaro Jiménez, Juan Manuel Vara, Valeria De Castro, Verónica A. Bollati, and Esperanza Marcos. Model-driven engineering as a new landscape for traceability management: A systematic literature review. *Inf. Softw. Technol.*, 54(12):1340–1356, December 2012.

[66] Iván Santiago Viñambres, Valeria de Castro, Juan Manuel Vara, and Esperanza Marcos. Evaluación de propuestas para la gestión de trazabilidad en el contexto de la Ingeniería Dirigida por Modelos. In *XVI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2011 (SISTEDES)*, La Coruña, Spain, September 2011.

[67] Susanne A. Sherba, Kenneth M. Anderson, and Maha Faisal. A framework for mapping traceability relationships. In *2 nd International Workshop on Traceability in Emerging Forms of Software Engineering at 18th IEEE International Conference on Automated Software Engineering*, pages 32–39, 2003.

[68] The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology.* New York, USA, September 1990.

[69] U. S. Department of Defense. MIL-STD-498 - DI-IPSC-81433 - Software Requirements Specification, December 1994.

[70] Pedro Valderas and Vicente Pelechano. Introducing requirements traceability support in model-driven development of web applications. *Information & Software Technology*, 51(4):749–768, 2009.

[71] B. Vanhooff and Y. Berbers. Supporting Modular Transformation Units with Precise Transformation Traceability Metadata. In *ECMDA-TW: Traceability Workshop at European Conference on Model Driven Architecture*, pages 15–27, Nuremberg, Germany, November 2005.

[72] Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as input for model transformation. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, Haifa, Israel, June 2007.

[73] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, September 2004.

[74] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.