Tesis

Mariano Gabriel Gili

13 de septiembre de 2014

Índice general

Índice de figuras

Agradecimientos

A mis padres, **Ana María Bloga** y **Nestor Hugo Gili**, quienes nunca dudaron en brindarme el soporte necesario para afrontar esta carrera de grado. A mi mujer **María Jimena Reimundo**, por darme el apoyo y acompañamiento que necesité para poder terminar el trabajo de tesis al mismo tiempo que empezamos a construir nuestro hermosa familia y hogar.

A la **Universidad Nacional de La Plata** y en especial a la **Facultad de Informática**, por el excelente nivel educativo al que me dio acceso. A las cátedras de **Programación** y **Programación Funcional**, que me abrieron las puertas para poder desarrollar la gratificante tarea de ayudante y colaborador de cátedra respectívamente. Al laboratorio **LIFIA**, por haberme brindado la posibilidad de realizar trabajos de investigación y de asistir a cursos extracurriculares en los que adquirí conocimientos sobre diversos temas que me resultaron apasionantes.

A la profesora **Claudia Pons**, por haberme introducido en el grupo de investigación de **Desarrollo Dirigido por Modelos**, desde donde definimos el tema que se presentará a lo largo de este trabajo de tesis, y por haber sido mi guía en el desarollo del mismo, demostrándome siempre su buena predisposición en cada consulta realizada.

A mi familia, amigos y compañeros de estudio y del trabajo, por su acompañamiento y empujón en este último tramo que tanto tiempo me llevó.

¡¡¡Muchas gracias a todos!!!

Introducción

En la Ingeniería de Software Dirigida por Modelos (*MDE* de Model-Driven Engineering), se define al modelo como artefacto principal que toma participación a lo largo de todas las tareas/procesos que conforman dicha ingeniería, ésto es, el análisis, el diseño, el desarrollo, las pruebas, el mantenimiento, etc. Una implementación particular propuesta que acompaña esta idea es la Arquitectura Dirigida por Modelos (*MDA* de Model-Driven Architecture), definida por el Object Management Group (*OMG*), cuyo ciclo de proceso de desarrollo está basado enteramente en el uso de modelos formales y transformaciones que se realizan sobre dichos modelos. Una característica muy importante de todo proceso de Desarrollo Dirigido por Modelos (*MDD* de Model-Driven Development), es lo que se conoce como ***posibilidad de rastreo*** (o de ahora en más en inglés ***traceability***), que ayuda y toma parte en todo lo que respecta a las relaciones que existen entre cada uno de los artefactos productos del proceso de desarrollo.

Cuando nos referimos al término artefacto, hablamos por ejemplo de un requerimiento de sistema, un componente de software, un caso de prueba, entre otros. El mantenimiento y la definición de las relaciones y dependencias que existen entre los artefactos no es una tarea fácil, el mismo ha sido un desafío desde principios de 1970.

En el presente documento de tesis se abordará el tema de ***traceability***, luego se presentará un análisis de los distintos problemas que aún se encuentran abiertos como así también un conjunto de soluciones encontradas a lo largo de la investigación. Finalmente, se elaborará un ***esquema de traceability*** con el fin de contribuir en la solución de alguno de los problemas nombrados.

Objetivos

El objetivo del trabajo de tesis comprende, por un lado, una introducción al tema de *traceability* conjunto con un análisis de los problemas que aún se encuentran abiertos en el ámbito de MDE, seguido de la elaboración de un *esquema de traceability* que contribuya en la solución de dichos problemas.

También se diseñará e implementará una herramienta que pueda ser integrada a otra de desarrollo MDE, y asista al desarrollador automatizando el proceso de definición de ***trazas*** (o de ahora en más en inglés ***tracelinks*** o ***links***) entre elementos de los modelos origen y destino. Esta solución proveerá un mapa de transformaciones que permitirá determinar la procedencia de cada ítem del modelo destino, y su correspondiente origen en el modelo fuente.

Capítulo 1

Traceability

1.1. Introducción

Según el Glosario Estándar de Términos de la Ingeniería de Software del Institute of Electrical and Electronics Engineers (IEEE) [1] la idea de *traceability* se define como:

> *El grado o nivel en el cual una relación puede ser establecida entre dos o más productos del proceso de desarrollo, especialmente entre productos que tengan una relación de predecesor-sucesor o principal-secundario; por ejemplo el grado en el cual el requerimiento y el diseño de un componente de software se corresponden.*

La definición anterior fue dada por *the requirements management community*, para nosotros, que necesitamos un punto de vista más cercano al contexto de MDD, usamos el término *traceability* para describir

> ***cualquiera de las complejas relaciones lógicas que existen entre los distintos artefactos que se presentan en cualquier momento del ciclo de vida del desarrollo de software, el establecimiento de estas relaciones y/o el mantenimiento de las mismas.***

Esta definición incluye a todos los productos creados durante el proceso de desarrollo, durante el proceso de despliegue o implantación, y también los creados a lo largo del mantenimiento. Además necesita que la información de *traceability* sea accesible a lo largo de toda la vida del producto de software.

En la ingeniería de software encontramos dos usos o semánticas principales que dependen del contexto de *traceability*:

- ***traceability en la Ingeniería de Requerimientos***: donde se guarda un requerimiento desde su definición hasta su implementación. En

más detalle según [6] se refiere a la habilidad de describir y seguir la vida de los requerimientos en ambas direcciones, hacia delante y hacia atrás (*forward and backward traceability*). Desde los orígenes, pasando por el desarrollo y la especificación, hacia su posterior entrega y uso, y a través de todos los períodos de refinamiento e iteración de cualquiera de estas etapas.

- ***traceability en MDD***: donde se almacenan principalmente las relaciones existentes entre los artefactos producto de las transformaciones de modelos.

Las relaciones de *traceability* pueden ser definidas de forma automática, por ejemplo producto de una transformación de modelos, o de forma manual como el caso de una relación de implementación entre un requerimiento y un componente de software.

Entre varios beneficios de *traceability* que enumeraremos mejor más adelante, podemos encontrar que ayuda a identificar las relaciones y dependencias que existen entre los artefactos de software. También *traceability* es crucial entre los requerimientos y su representación en los modelos para asegurar que el conjunto relevante de requerimientos fueron debidamente implementados en el código. Pero no solo *traceability* asegura la identificación de objetos y elementos relacionados, también puede facilitar el análisis de impactos de cambios durante el desarrollo de software.

Por todo lo anterior, queda demostrado que una buena solución de *traceability* que se encuentre provista de información actualizada será un servicio muy valioso tanto para jefes de proyectos, como para desarrolladores de software y/o consultores de mantenimiento.

1.1.1. Beneficios

A continuación se listan un conjunto de actividades que provienen de distintos dominios de la ingeniería, en las cuales el uso de *traceability* es muy beneficioso según [2] y [3].

- ***Análisis de Sistemas*** Ayuda a entender la complejidad de un sistema, navegando a lo largo del modelo de enlaces resultantes de la ejecución de las distintas cadenas de transformación.

- ***Análisis de Cobertura*** Por ejemplo en el momento de ejecución de los casos de prueba, el uso de *traceability* es crucial para la hora de determinar si todos los requerimientos fueron cubiertos, es decir tenidos en cuenta.

- ***Análisis de Impacto de Cambios*** *Traceability* nos ayuda a ver cómo los cambios en un modelo repercutirán en los modelos relacionados; también el uso de *traceability* nos permite saber en cualquier momento el tipo de dependencia que existe entre las entidades relacionadas, lo cual ayuda a determinar la necesidad de un cambio.

- ***Análisis de Huérfanos*** Nos permitirá encontrar fácilmente los elementos huérfanos de un modelo dado que serán los artefactos que no se encuentren enlazados a ningún *tracelink*.

- ***Comprensión del Software y la Ingeniería Inversa*** Crucial cuando sea necesario identificar todas las entidades relacionadas a una en particular, entender el tipo de relación existente, identificar las abstracciones, es decir los patrones de diseño, estilos de arquitectura, etc.

- ***Análisis de Requerimiento*** Por ejemplo para identificar el artefacto particular que demanda una propiedad específica; encontrar y resolver un conjunto de requerimientos que se contradicen, entre otros.

- ***Apoyo en la toma de decisiones*** Para justificar una decisión dado que nos facilita entender qué factores y metas influyen en la misma; también *traceability* nos será muy útil en el análisis y evaluación en los momentos que se nos encontremos con distintas propuestas de solución.

- ***Configuración del Sistema y Versionado*** En este momento el uso de *traceability* es beneficioso para identificar las restricciones entre los componentes, identificar los cambios necesarios para resolver una restricción, identificar las diferencias entre dos versiones distintas del mismo artefacto y su impacto en otros artefactos.

1.1.2. Algunos inconvenientes

Más allá que las ventajas de *traceability* hoy en día ya han sido identificadas, su puesta en práctica apenas se ha establecido. Las principales razones de lo anterior, se debería a alguno de los siguientes inconvenientes según [3]:

- El ***alto costo*** de la creación y la necesidad de ***mantenimiento manual*** de la información de *traceability*.

- La ***falta de heurísticas*** que determinen qué información de los enlaces deben ser grabados.

- Discrepancias entre los distintos ***roles*** que ejercen los usuarios de la información de *traceability*, por ejemplo entre quienes crean los enlaces y quienes los interpretan.
- Carencia de ***soporte adecuado de las herramientas***.
- Los artefactos son descriptos en ***diferentes lenguajes***, por ejemplo, los requerimientos se escriben en lenguaje natural mientras que los programas en algún lenguaje de programación.
- Los ***diferentes niveles de abstracción*** en que los artefactos describen el sistema de software, por ejemplo el nivel de abstracción de los artefactos usados durante el ciclo de diseño difiere de los niveles usados en las etapas de implementación.

Más adelante en otro capítulo se presentará con más detalle los desafíos presentes en la implementación y uso de *traceability*.

Capítulo 2

Teoría de traceability

En el presente capítulo nos introducimos en un conjunto de temas importantes de la teoría de *traceability*, las primeras dos secciones se refieren a la ***semántica de traceability***, introduciendo primero las dos distintos estrategias de ***meta-modelos de traceability*** que existen, y luego, la compleja tarea de la definición de ***tipos de tracelinks***. En la sección siguiente, se presentan dos formas distintas de ***generación de tracelinks***, y en la última, las dos ***estrategias de almacenamiento*** principales de la información de *traceability* que se pueden seguir. Estos temas han sido muy importantes y relevantes a la hora de la definición del *esquema de traceability* propuesto que se detalla en el capítulo 4.

2.1. Nivel de Granularidad

El término granularidad refiere al nivel/grado de detalle en el cual un *tracelink* se genera y/o se registra, es una característica íntimamente relacionada con el uso que se quiera realizar con *traceability*. Por ejemplo cuando se desee trabajar sobre diagramas de clases UML, se podrán generar *tracelinks* a nivel de paquetes, de clases o de métodos. La granularidad de un *tracelink* (o de ahora en más en inglés ***trace granularity***) es definido por la granularidad del artefacto origen y la granularidad del artefacto destino.

Esta propiedad debe ser definida cuidadosamente para efectivamente dar un buen soporte en las tareas de *traceability*, dado que es un factor que definirá la complejidad, y por tanto el esfuerzo, en el análsis y en la utilización del conjunto de los *tracelinks* obtenidos.

2.2. Meta-modelos de traceability

Para llegar a la definición de cualquier propuesta de *traceability* se necesita de un modelo en el cual se especifiquen los conceptos, las reglas y las relaciones que existen, por ejemplo entre los artefactos y los *tracelinks*. Éste modelo de *traceability* va a estar determinado por un meta-modelo, el cual puede llegar a ser clasificado como un ***meta-modelo de traceability de propósito general***, o como un ***meta-modelo de traceability de caso específico***, a continuación se explica cada uno.

2.2.1. Meta-modelo de propósito general

En este caso, nos encontramos con un meta-modelo genérico que permite la ***captura de tracelinks*** entre cualquier tipo de elementos de modelo. Un *tracelink* se puede conectar con cualquier número de elementos, de cualquier tipo y de cualquier modelo. Las principales ventajas de este tipo de meta-modelo son la simplicidad y la uniformidad (dado que todos los modelos conforman el mismo meta-modelo), con lo cual se mejora la interoperabilidad de las herramientas brindándole la capacidad de importar, exportar y gestionar información de *traceability* en un formato común.

Por otro lado, como el *meta-modelo de propósito general* no capta casos específicos de *tracelinks* fuertemente tipados, es decir sin semántica y restricciones definidas rigurosamente, se abre la puerta a establecimientos de *links* ilegítimos. Como por ejemplo, en el caso de *tracelinks* entre un diagrama de clases y un modelo de base de datos relacional en donde existirán vínculos entre las clases del primer modelo y las tablas del segundo, un meta-modelo de *traceability* genérico permitirá el establecimiento de *links* ilegítimos entre una clase y una columna.

Para permitir un mejor soporte para el caso de los requisitos específicos, es un método de uso frecuente la provisión de mecanismos de extensión que acompañan el meta-modelo de propósito general. Sin embargo, todavía carecen de la eficacia que ofrecen los *meta-modelos de casos específicos* para capturar estos tipos de situaciones que requieren tal nivel de legitimidad entre la información y su semántica.

2.2.2. Meta-modelo de caso específico

En este caso, para cada escenario de *traceability* se define un meta-modelo específico. Este meta-modelo de *traceability* captura *links* fuertemente tipados para casos específicos con una semántica bien definida, que pueden o no

incluir restricciones de corrección. Debido a su naturaleza de tipado fuerte y las restricciones asociadas, restringe a los usuarios y las herramientas para que sólo puedan establecer *links* legítimos. Por otro lado, la definición de un meta-modelo para cada caso específico requiere mucho esfuerzo en su construcción, así como herramientas que soporten, o mejor dicho, ofrezcan la posibilidad de aceptar diferentes meta-modelos de *traceability*.

Para ser fuertemente tipado el meta-modelo de *traceability* necesita referir explícitamente a los tipos de elementos que se encuentran definidos en otros meta-modelos. Por ejemplo, consideremos que es necesario definir un meta-modelo de *traceability* que permita el establecimiento de enlaces entre instancias de A (del meta-modelo MMa) e instancias de B (a partir de MMb), pero no entre dos instancias de A o dos de B. Para lograr tal meta-modelo, la tecnología de modelado que se use no debe tomar cada meta-modelo como un espacio cerrado, sino que por el contrario debe permitir referencias *inter-meta-modelo*. Un ejemplo de tecnología que soporta referencias *inter-meta-modelo* es el framework Eclipse Modeling Framework (EMF).

Mas allá de que un meta-modelo de *traceability* definido utilizando una tecnología que permita referencias *inter-meta-modelo* puede brindar tipos seguros, encontramos frecuentemente otras restricciones que necesitan especificarse y que dicho meta-modelo no puede capturarlas. Por ejemplo, tomando como referencia el ejemplo anterior, podríamos precisar que cada instancia A de MMa sólo se puede vincular a no más de una instancia B de MMb. Para especificar tales restricciones, se requiere un lenguaje de especificación que pueda expresar restricciones que abarquen elementos que pertenezcan a modelos definidos por diferentes meta-modelos. En la actualidad el Lenguaje de Restricciones para Objetos (*OCL* de Object Constraint Language) carece de esta capacidad, ya que no proporciona las construcciones para expresiones que atraviesen modelos (*cross-model*). Ejemplos de lenguajes que soportan el establecimiento con tales restricciones incluyen el Epsilon Validation Language (EVL) y el XLinkit toolkit.

La combinación de un meta-modelo de *traceability* fuertemente tipado conjunto con la verificación de restricciones *inter-modelo* restringe a los usuarios y a las herramientas a establecer y mantener sólo *tracelinks* con sentido, los cuales pueden ser automáticamente validados para descubrir posibles omisiones e inconsistencias. Estas cuestiones pueden realizarse ya sea durante el establecimiento de los *tracelinks*, o más tarde, durante el ciclo de vida de los modelos donde ya los *tracelinks* han sido establecidos.

2.3. Tipos de tracelinks

Uno de los desafíos presentados en el capítulo 3 (sección 3.4) se refiere a la definición *semántica de los tracelinks*, tarea que nos permite descubrir los distintos *tipos de links* con lo que nos podemos llegar a encontrar, entender su significado y uso. Es importante remarcar que la determinación de la semántica de un *link* se encuentra totalmente guiada por la razón del usuario respecto de qué quiere realizar o representar con dicho *link*. No predefinir la *semántica de un tracelink* correctamente, puede resultar en fallas de razonamientos y/o entendimientos. Por ejemplo, un *link* que muestra la relación entre un requerimiento textual y un elemento de un diagrama de caso de uso UML tiene una semántica muy distinta a la de un *tracelink* que determina una relación de refinamiento dentro de un modelo de clases también UML.

La definición del conjunto de *tipos de tracelinks* por lo general es fuertemente dependiente al contexto de un proyecto, por lo cual definirlo de forma fija tendrá como consecuencia una pérdida de flexibilidad para los usuarios que deseen tipar los *tracelinks* solamente de acuerdo a las necesidades del proyecto o la compañía. Aún así, conforme un proyecto de desarrollo crece, la administración de la información de *traceability* que en consecuencia se va generando se vuelve extremadamente compleja, una jerarquía de clasificación de dicha información resultará esencial para poder entenderla y administrarla mejor. En [10] se encuentra una propuesta de *clasificación genérica de traceability* conjunto con la descripción del proceso usado para su obtención al que llamaron Traceability Elicitation and Analysis Process (TEAP) y que a continuación se explica en detalle.

2.3.1. Una clasificación genérica de traceability

Esta propuesta comienza con una clasificación o meta-modelo de *traceability* inicial, luego de forma iterativa e incremental esta clasificación se va refinando de acuerdo a las siguientes tareas: obtención, análisis y clasificación. En la obtención se identifica un *tracelink* y sus relaciones. En el análisis, se abstraen las principales características del *link* obtenido identificando las restricciones, sus relaciones y generalizaciones. Y por último, se define la clasificación a la que pertenece.

El meta-modelo inicial se puede ver en el dibujo 2.1, en el que se encuentran los siguientes conceptos fundamentales: ***Artefact***, ***Tracelink*** y ***Operation***. *Artefact* refiere tanto a artefactos MDE, por ejemplo modelos específicos de dominio, como a no MDE, por ejemplo hojas de cálculo. En *Operation* se encierran las operaciones bien manuales como automáticas que determinan qué información de *traceability* debe almacenarse. Por último

Tracelink da inicio con la clasificación de ***tracelink implícito*** (***Implicit Link***) por un lado, y ***tracelink explícito*** (***Explicit Link***) por el otro, en donde *tracelink implícito* refiere a los *links* que son creados y manipulados por la aplicación de las operaciones MDE, y por su parte *tracelink explícito* refiere a los *links* que se encuentran concretamente ya representados en los modelos.

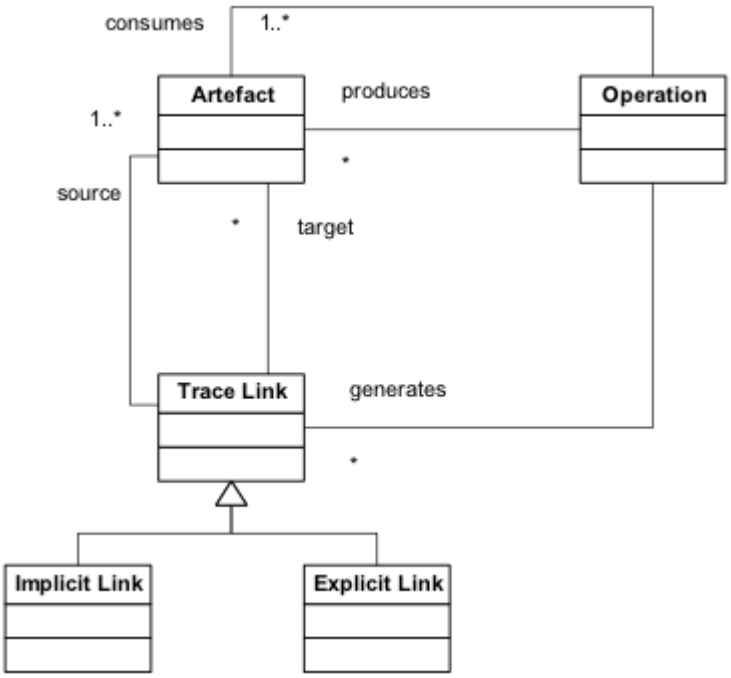

Figura 2.1: Clasificación de traceability inicial

Clasificación de tracelinks implícitos

En esta jerarquía que se muestra en el dibujo 2.2 se encuentran representados el conjunto de operaciones MDE principales: *Consulta* (***Query Link***), *Transformación modelo a modelo* (***M2M Link***), *Transformación modelo a texto* (***M2T Link***), *Composición* (***Composition Link***), *Actualización* (***Update Link***), *Creación* (***Creation Link***), *Eliminación* (***Delete Link***), entre otras.

Clasificación de tracelinks explícitos

En un principio esta clasificación se encuentra dividida en dos grandes grupos representados por las siguientes clases básicas: *Link modelo-modelo*,

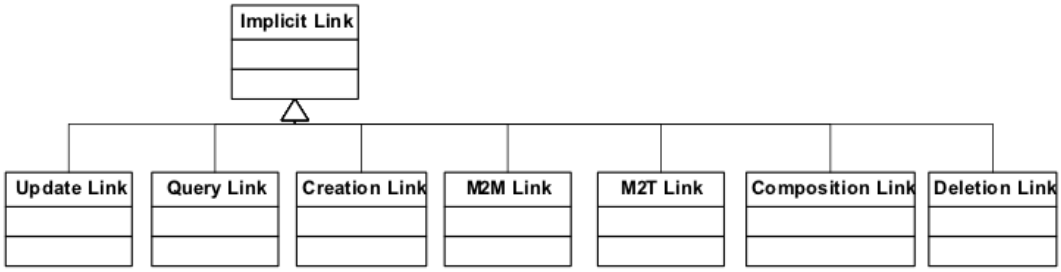

Figura 2.2: Jerarquía de links implícitos

que representa por ejemplo las relaciones de dependencias UML, y *Link modelo-artefacto*, conjunto al que pertenece por ejemplo el *link* entre un modelo y un documento de texto que dicta un requerimiento. En el dibujo 2.3 aparecen como ***Model-Model Link*** y ***Model-Artefact Link*** respectivamente.

Model-Model Link se encuentra a su vez subdividido en *Link estático* (***Static Link***) y *Link dinámico* (***Dynamic Link***). En el primero se representan relaciones estructurales que no cambian con el tiempo, en cambio en el segundo se representa información de los modelos que si puede llegar a variar.

Los *links estáticos* pueden ser, o *Links consistentes* (***Consistent-With***) donde dos modelos deben mantenerse de acuerdo o consistentes entre sí, o *Links dependientes* (***Dependency***) donde la estructura y/o la semántica de un modelo depende de otro. Por su parte, los *Links dependientes* pueden ser: *Link de subtipo* (***Is-A***), *Link de referencia*, *link de subconjunto*, de *importación* y *exportación*, de *uso*, de *refinamiento* (***Refines***), etcétera.

Entre los *links dinámicos* se incluyen los *Links de llamadas* (***Calls***) donde un modelo hace uso de métodos provistos por otro, los *Links de notificación* (***Notifies***) donde es necesario almacenar información que puede ser manejada automáticamente. También encontramos aquí las relaciones en tiempo de diseño como los *Links de generación o construcción* (***Generates***), que indican qué información de un modelo es usada para producir o deducir otro, y los *Links de sincronización* (***Synchronized With***), donde el comportamiento de un conjunto de modelos se realiza de forma sincronizada.

El alcance de *Model-Artefact Link* es muy amplio, por tal motivo es esta clasificación sólo se resumen un subconjunto de ejemplos, entre ellos: la relación *satisface* (***Satisfies***), que indica qué propiedad o requerimiento de un artefacto es satisfecha por un modelo; los *Links de asignación* (***Allocated-To***), usados cuando la información de un artefacto no modelo es asignada a un modelo específico que la representa; la relación de *realización* (***Per-***

forms), que indica qué tarea descripta por un artefacto es llevada a cabo por el modelo; las relaciones *explica* y *respalda* (***Explains*** y ***Supports*** respectivamente) que dictan qué modelo se encuentra explicado/respaldado por un artefacto no modelo.

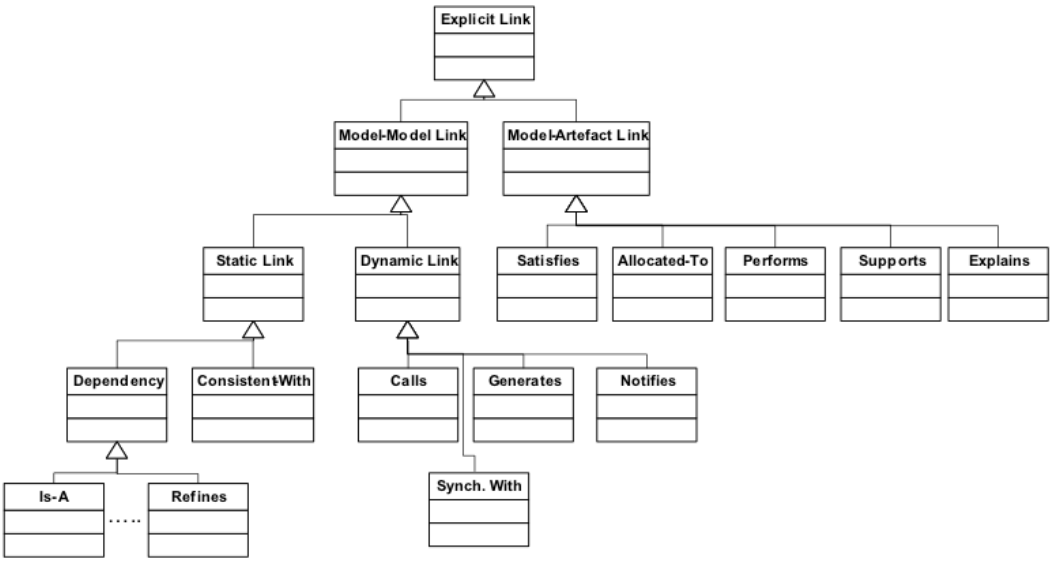

Figura 2.3: Jerarquía de links explícitos

2.4. Generación de tracelinks

Uno de los principales temas de *traceability* refiere a los dos enfoques distintos de *generación de tracelinks* que se pueden presentar o usar durante una transformación de modelos, estos son la generación ***implícita*** y la generación ***explícita***.

La diferencia entre estas dos posibilidades radica en que, por un lado en la generación *implícita* las transformaciones proveen un soporte integrado de *traceability*, mientras que por el otro, en la generación *explícita* es responsabilidad del desarrollador de la transformación de codificar explícitamente las reglas de *traceability* que generarán esta información como un modelo más de salida.

A continuación se detallan las ventajas y desventajas de cada enfoque según [3]:

2.4.1. Generación implícita

Ventajas

✓ La única y mayor ventaja de la generación *implícita* es que no es necesario ningún esfuerzo adicional para obtener los *tracelinks* que relacionan los modelos de entrada y salida, dado que son generados en paralelo y automáticamente con el modelo resultante de la ejecución de la transformación.

Desventajas

× El meta-modelo de *traceability* tiene que ser necesariamente fijo, pero dada la infinidad de enfoques de transformaciones que dan lugar a diferentes definiciones de meta-modelos, lograr un estándar entre estos diferentes enfoques es muy complejo.

× Dada la poca flexibilidad que se obtiene al hacer uso de un meta-modelo fijo, se presentan los siguientes inconvenientes a tener en cuenta:

- cuando se generan *tracelinks* para todos los elementos del modelo referenciado, obtener una cantidad considerable puede volverse incomprensible y en consecuencia inútil. También, en el caso de transformaciones de modelos grandes y complejos, podemos encontrarnos con problemas de rendimiento.
- la configuración del nivel de *trace granularity* varía de un escenario de *traceability* a otro.
- un cliente, por ejemplo aduciendo motivos de seguridad, puede requerir que para cierta información de un modelo no se le generen *tracelinks*.

2.4.2. Explícita

Ventajas

✓ Es posible el tratamiento de *traceability* como un modelo regular complementario al resultado de la transformación, que se puede obtener por la incorporación de reglas de transformación adicionales. Por lo cual, la elección del meta-modelo es responsabilidad completa del programador de la transformación y no depende del motor de transformaciones.

✓ Dada la flexibilidad del meta-modelo, el nivel de *trace granularity* de los *tracelinks* es fácil de adaptar al dominio de la transformación.

Desventajas

1. Se requiere de un esfuerzo adicional para definir las reglas de transformación específicas para *traceability*, que en consecuencia contaminan la implementación.

2. Como la definición de las reglas de *traceability* es responsabilidad del programador, es propensa a errores y puede llegar a demandar mucho tiempo. Más aun, si pensamos que esta tarea se tiene que repetir para cada transformación que se requiera.

2.5. Estrategias de almacenamiento

Según [7] hay dos tipos principales de estrategias a seguir para almacenar y administrar la información de *traceability*, ***almacenamiento intra-modelo*** y ***almacenamiento externo***. En la primera, la información de *traceability* se encuentra embebida en los modelos a los que ella refiere. En cambio en la segunda, dicha información se encuentra almacenada de forma separada de los modelos:

2.5.1. Almacenamiento intra-modelo de tracealinks

Como ya se dijo, bajo esta estrategia la información de *traceability* es almacenada dentro de los modelos a los que refiere, esto puede ser mediante elementos pertenecientes al modelo o mediante atributos de los elementos (como etiquetas o propiedades).

Es una estrategia sencilla y amigable, pero puede ser muy problemática por varias razones. Si los *tracelinks* son dirigidos y almacenados solamente en el modelo origen, éstos no son visibles en el modelo destino, a la inversa (almacenados en el destino) nos encontramos con el mismo problema pero en el origen. Por otro lado, si la información de *traceability* es almacenada en ambos modelos, entonces nos encontramos con el desafío de que dicha información se mantenga consistente por cada cambio que se suceda.

A todo lo anterior, se suma el problema de la polución que se genera en el modelo con la información de *traceability*, dado que la misma es ajena a su contexto. Dicha polución puede incluso lograr que el modelo se vuelva imposible de comprender y mantener.

Por otro lado, en un entorno MDE es común que los modelos tengan sus propias representaciones y semánticas, lo cual puede volver más complejo diferenciar la información de *traceability* de los objetos que representan el modelo del dominio.

Como resultado de los inconvenientes anteriores, el análisis automatizado de la información de *traceability* se hace muy difícil. Los principales metodologías que hacen uso de esta estrategia utilizan construcciones de lenguajes específicas, por ejemplo determinados tipos de enlaces de *traceability* están representados en los diagramas UML mediante el uso de estereotipos como «refines».

2.5.2. Almacenamiento externo de tracelinks

En esta estrategia la información de *traceability* se encuentra almacenada de forma separada a los modelos a los que refiere, en un modelo aparte. Esta propuesta tiene dos ventajas claras, la primera es que los modelos origen y destino se mantienen totalmente limpios, con lo cual la polución del *almacenamiento intra-modelo* no sucede. Y la segunda, dado que el modelo en donde se almacenan los *tracelinks* se encuentra definido por un meta-modelo con una semántica clara, logra que el proceso de análisis de la información sea mucho más fácil que en la otra estrategia.

Un requisito previo para el *almacenamiento externo de los tracelinks*, es que los diferentes elementos del modelo tengan identificadores únicos, de modo que los *links* que los relacionan se pueden resolver inequívocamente. Un ejemplo es el mecanismo propuesto por MetaObject Facility (*MOF*) y EMF en la forma de un identificador xmi.id .

Capítulo 3

Problemas y Desafíos

En este capítulo vamos a describir, por un lado, los principales problemas que se presentan en el ámbito de *traceability*, y por otro, los grandes desafíos que surgen de ellos y, que aún hoy, todavía se encuentran abiertos. Todo organizado por grupos según el conjunto de aspectos definidos en [5], donde por cada tema se listan los problemas asociados, y seguido, los desafíos en consecuencia.

3.1. Conocimiento de traceability

Problemas

× Existe poco consenso respecto a cuáles son las mejores técnicas y métodos para la aplicación de *traceability*, pocas anotaciones y documentación sobre las mejores prácticas, sumado a una falta de recursos que provean una buena base de conocimiento.

× Las definiciones semánticas no coinciden y las terminologías son dispares, todo ésto crea barreras de comunicación.

Desafíos

✓ Crear una base de conocimiento en la que se vuelquen las mejores prácticas de *traceability*, una terminología estándar y mucha información adicional, como por ejemplo casos de estudio.

3.2. Capacitación y certificación

Problemas

× Muy poca gente es competente en la tarea de definición de *tracelinks* y, por otro lado, existen disponibles pocos programas educativos que enseñen dicha tarea.

× Existen pocos programas de certificación, y de ellos, pocos incluyen componentes de *traceability*.

× No hay definido un conjunto de estrategias estándar de *traceability*.

Desafíos

✓ Identificar las áreas de conocimientos y las estrategias principales asociadas a *traceability*.

✓ Desarrollar buenos componentes educativos para la puesta en práctica de *traceability*.

✓ Desarrollar materiales pedagógicos efectivos para educar con énfasis en la importancia y administración de los costos-beneficios que conllevan la implementación de *traceability*.

3.3. Soporte y evolución

Problemas

× La información precisa, coherente, completa y actualizada sobre *traceability* es fundamental para diversos ámbitos y aplicaciones. Sin embargo, las técnicas actuales de *captura de tracelinks* aún son realizadas de forma manual y por lo tanto son propensas a errores.

× Para que los *tracelinks* sean útiles, éstos deben reflejar la dependencia actual entre los artefactos. Dado que el costo y esfuerzo para mantenerlos durante la evolución del sistema es inmenso, a menudo los *tracelinks* pasan a encontrarse en un estado erróneo o incorrecto.

× Las herramientas actuales de administración de requerimientos incluyen características como *suspect tracelinks* para ayudar a los analistas

a administrar y entender la evolución de los *tracelinks*, pero en la mayoría de los proyectos complejos, el número de enlaces marcados como *suspect tracelinks* se vuelven rápidamente excesivos, minimizándose drásticamente la utilidad de tal característica.

× Los *tracelinks* tienen que evolucionar de forma sincrónica con los artefactos que se encuentran relacionados, sin embargo los sistemas actuales de gestión de cambios y la semántica de los enlaces no son lo suficientemente sofisticados como para acompañar tal evolución.

Desafíos

✓ Desarrollar técnicas de *captura de tracelinks* automáticos para artefactos descriptos de forma textual, que sean tan precisos como el proceso manual y, a la vez, mucho más efectivos en tiempo y costo.

✓ Desarrollar la funcionalidad de *captura de tracelinks* de forma integrada a los Entornos de Desarrollo Integrados o Integrated Development Environment (IDE).

✓ Desarrollar sistemas de administración de cambios que efectivamente acompañen la evolución de los *tracelinks* sobre los múltiples tipos de artefactos que existen.

✓ Desarrollar técnicas que maximicen la reutilización de los *tracelinks* cuando un código existente se vuelva a utilizar en un nuevo producto.

3.4. Semántica de los tracelinks

Problemas

× Para efectivamente utilizar y entender las relaciones por debajo de *traceability*, es necesario definir la semántica de los *tracelinks*. Sin embargo, definir una formalidad para representar esta semántica no es una tarea fácil y puede llegar a quedar acotada a un dominio específico, cosa que no es conveniente.

× Es muy importante para mantener la consistencia en *traceability*, conocer y establecer el nivel de *trace granularity*. El problema es que no existe aún, un modelo claro de costo-beneficio que determine unívocamente cuál es el nivel correcto.

Desafíos

- ✓ Definir meta-modelos para representar la información semántica de los *tracelinks* y proveer ejemplos de instancias sobre distintos dominios específicos.
- ✓ Desarrollar técnicas y procesos para determinar el correcto nivel de *trace granularity* de un proyecto.

3.5. Escalabilidad

Problemas

- × Las técnicas corrientes de *traceability* no escalan adecuadamente en proyectos largos.
- × Las herramientas de visualización son esenciales para dar ayuda en la compresión y el uso de la gran cantidad de información de los *tracelinks*. Sin embargo, las técnicas actuales no escalan bien y no son efectivas al presentar información compleja, dado que carecen de características sofisticadas de filtrado, navegación, consultas, etc.
- × Muchos conjuntos de datos industriales son compuestos por largos e inestructurados documentos que son difíciles de enlazar mediante *tracelinks*.

Desafíos

- ✓ Obtener conjuntos de datos de escala industrial desde varios dominios y usarlos para investigar la escalabilidad de las técnicas disponibles actualmente y, si es necesario, crear nuevas aproximaciones que escalen más eficientemente.
- ✓ Desarrollar mecanismos visuales efectivos que permitan la navegación y consulta de un gran número de *tracelinks* y los artefactos asociados.
- ✓ Desarrollar técnicas escalables para marcar los *tracelinks* independientemente de que el conjunto de datos sea heterogéneo, de tamaño considerable y/o se encuentre débilmente estructurado.

3.6. Factores humanos

Problemas

× Los métodos automáticos de *traceability* por lo general trabajan produciendo *tracelinks* candidatos. Sin embargo, el proceso es inútil si el analista no es capaz de evaluarlos correctamente y lograr diferenciar los buenos de los malos, o si no puede confiar en la completitud y precisión de los resultados.

× Idealmente la captura de *tracelinks* debería ser invisible durante todo el proceso de desarrollo. Sin embargo, la generación y el uso de los *tracelinks* es continuamente interrumpido por interacciones humanas dado que, en los ambientes de desarrollo actuales, aún no es posible automatizar todo el proceso.

× Los *tracelinks* por lo general enlazan artefactos semánticamente diferentes, a su vez estos artefactos son creados por diferentes personas y frecuentemente escritos en diferentes documentos. Como resultado, los usuarios de un lado de los *tracelinks* no entienden bien los artefactos que pertenecen al otro lado de la relación.

Desafíos

✓ Basándose en el estudio del uso de las herramientas actuales de *traceability*, crear nuevas que reúnan las necesidades prácticas que vayan surgiendo.

✓ Entender las vulnerabilidades a fallas humanas y sus impactos del proceso de *traceability*, y desarrollar técnicas que ayuden a los analistas a prevenir tales errores y/o minimizar el impacto de los mismos cuando ocurran.

✓ Desarrollar técnicas que ayuden a las personas a superar las barreras semánticas que se pueden presentar en el proceso de desarrollo completo.

3.7. Análisis de costo-beneficio

Problemas

× En un escenario de *traceability* completo, los *tracelinks* son creados entre artefactos que se encuentran en un nivel bajo de abstracción, lo

que puede ser deseable para propósitos de comprensión, sin embargo este nivel tan bajo no es por lo general práctico y efectivo en costo.

× Se carece de un modelo de costo-beneficio que ayude en el análisis que sea necesario realizarse sobre un proyecto cualquiera que implemente y/o use *traceability*, por ejemplo, que guíe en el tratamiento y selección de un conjunto de *tracelinks* potenciales que puedan surgir de un proyecto.

Desafíos

✓ Definir y desarrollar técnicas de generación y mantenimiento de la información de *traceability* eficientes.

✓ Definir un modelo de costos que sea práctico y aplicable en la generación y el mantenimiento de los *tracelinks*, que tome en consideración factores tales como el tamaño del proyecto, el tiempo, el esfuerzo y la calidad.

✓ Definir un modelo de beneficios del uso de *tracelinks*, que tome en consideración la crítica y la volatilidad, e incorpore el valor logrado gracias al uso de *traceability*.

3.8. Métodos y herramientas

Problemas

× Los métodos de recuperación de *tracelinks* que relacionan artefactos multimedia no son lo suficientemente sofisticados y/o soportados. Más aún, se ha realizado poco por incorporar tales técnicas multimedia en las herramientas de *traceability*.

× Que *traceability* sea automático es esencial, sin embargo, su puesta en práctica se hace difícil dada la falta de consistencia entre los artefactos relacionados y la imprecisión de los modelos.

× El uso de *traceability* implica todas las siguientes actividades: construcción y/o generación, evaluación, mantenimiento y uso de los *tracelinks*. Sin embargo, no existe aún una sola herramienta que pueda cubrir todas estas tareas por completo.

Desafíos

- ✓ Desarrollar métodos efectivos que enlacen artefactos multimedia.
- ✓ Construir métodos y herramientas con altos niveles de automatización que soporten el ciclo de vida entero: la construcción, la evaluación, el mantenimiento y el uso de los *tracelinks*.
- ✓ Desarrollar métodos de *traceability* que den soporte a requerimientos no funcionales.

3.9. Procesos

Problemas

- × *Traceability* no se incluye frecuentemente como una parte integral del ciclo de vida del desarrollo.
- × *Traceability* automático provee una alternativa eficiente en comparación a la metodología manual, pero la práctica ha mostrado que algunos conjuntos de datos son difíciles en el momento de procesar los *tracelinks* usando métodos automáticos, ésto se debe a las inconsistencias en terminología y estándares, la carencia de estructuras, los formatos heterogéneos, etc.

Desafíos

- ✓ Construir modelos de proceso que definan el ciclo de vida completo de *traceability*.
- ✓ Desarrollar técnicas que permitan evaluar la posibilidad/capacidad que tiene un conjunto de datos dado de soportar los métodos automáticos de *traceability*.

3.10. Conformidad

Problemas

- × El uso de estándares aseguran procesos consistentes y completos, aunque abundan los estándares en el ámbito de *traceability*, no está claro si los investigadores y/o profesionales están completamente consientes de la existencia de los mismos.

× En la comunidad de *traceability* se encuentran eruditos sobre técnicas y procesos del tema, pero tienen poca influencia sobre los contenidos de *traceability* relacionados con los procesos estándar de ingeniería de software.

× No está claro cómo se puede demostrar el cumplimento de los estándares y regulaciones.

Desafíos

✓ Establecer un mecanismo de comunicación para hacer que la comunidad de expertos de *traceability* dictamine los estándares relacionados con la tecnología.

✓ Sumarse a la comunidad que define normativas y estándares con el fin de influir y/o desarrollar los estándares de *traceability*.

✓ Como comunidad, desarrollar y promover escenarios válidos para probar que las herramientas, las técnicas y las metodologías de *traceability* cumplen con los estándares.

3.11. Mediciones y Benchmarks

Problemas

× Los estudios empíricos se necesitan para demostrar la eficacia de los métodos de *traceability* y así, facilitar el trabajo colaborativo y evolutivo entre los investigadores y profesionales. Sin embargo, no se dispone de diseños comunes de experimentación, metodologías, ni *cotas de referencia* (o de ahora en adelante en inglés *benchmarks*) para poder realizar dichos estudios.

× Los *benchmarks*, métodos y métricas propuestas actuales no han sido validadas a través de estudios o pruebas empíricas.

× No existen o no se han realizado buenas pruebas y *benchmarks* para *traceability* y, las que existen, no son compatibles.

× No existen *benchmarks* que ayuden en la comparación de métodos y técnicas ya definidas y desarrolladas de *traceability*.

× La detección de errores en los *tracelinks* determina la eficacia del producto y el proceso, sin embargo los modelos de detección de errores actuales son primitivos e inválidos.

Desafíos

- ✓ Definir procesos estándares para la realización de estudios empíricos en el ambiente de investigación de *traceability*.
- ✓ Crear *benchmarks* para evaluar los métodos y las técnicas de *traceability*.
- ✓ Definir medidas que ayuden en la evaluación de calidad de *tracelinks*, tanto de uno como de un conjunto de ellos.
- ✓ Desarrollar técnicas de evaluación de métodos y procesos de *traceability*.

3.12. Transferencia de tecnología

Problemas

- × Uno de los objetivo de la investigación de *traceability* es lograr transferir soluciones eficaces a la industria. Sin embargo, en la realidad en la industria son reacios a probar técnicas nuevas donde la eficacia aún no fue demostrada.
- × La carencia de diálogo entre los investigadores y los profesionales limita, por un lado a los investigadores a acceder a un conjunto de datos reales para testear nuevas técnicas, y por otro, inhibe la retroalimentación de la industria hacia los investigadores.
- × Los prototipos de *traceability* son generalmente diseñados para mostrar pruebas de conceptos en el campo de la investigación. Sin embargo, estos prototipos no son lo suficientemente rigurosos para el campo de pruebas de la industria.

Desafíos

- ✓ Crear una infraestructura y un conjunto de métodos relacionados con el fin de organizar el proceso de transferencia de tecnología.
- ✓ Identificar los casos de estudio exitosos y darlos a conocer con el fin de demostrar la eficacia de rentabilidad y técnica que ofrecen las metodologías de *traceability* en el ámbito industrial.
- ✓ Identificar los usuarios de *traceability* y definir sus necesidades en términos de calidad, ciclo de vida, comunicación, etc.

- ✓ Incorporar las herramientas de *traceability* que se encuentren a la vanguardia en los IDEs más comunes (como Eclipse) y en las herramientas de administración de requerimientos industriales.

Capítulo 4

Manos a la obra

En este capítulo introducimos, desarrollamos y fundamentamos el *esquema de traceability* propuesto en la presente tesis. Luego, haremos lo mismo con el prototipo de la herramienta desarrollada que acompaña y ejemplifica las ideas del esquema presentado. La herramienta, desarrollada como un ***plug-in*** del Eclipse, asiste en la generación de *tracelinks* sobre transformaciones de modelos existentes, y luego, presenta un mapa en el que se muestran cómo se relacionan las transformaciones, los *links* y los artefactos orígenes y resultados de la ejecución de las transformaciones.

4.1. Requerimientos

4.1.1. Introducción

La idea fundamental es lograr un meta-modelo bien simple en el que se capture solamente los conceptos más relevantes de *traceability* y se almacene la información mínima necesaria. Gracias a todo lo anterior, y solo así, logramos obtener el concepto deseable de independencia del modelo con respecto al dominio en el que se quiera utilizar la funcionalidad de *traceability*. Concepto necesario para poder llegar a definir un *esquema de propósito general* (2.2.1) que apunte principalmente a dar solución a los siguientes desafíos:

- Lograr un esquema que represente el conjunto de *tracelinks* independientemente de cómo fueron generados, *explícita* o *implícitamente* (2.4), su *trace granularity* (2.1) y/o semántica.
- Mantener la comunicación lo más simple posible, tanto sea la que se realiza entre los distintos usuarios con sus respectivos roles, como la de los distintos tipos de herramientas que necesiten aplicar *traceability*.

- Cuando se desee agregar la funcionalidad de *traceability* mediante el uso de este esquema, la adaptación de la herramienta, cualquiera sea, tiene que ser una tarea sencilla.
- Aún dada la infinidad de artefactos que se pueden presentar, el esquema siempre tiene que ofrecer información semántica de los mismos, con el fin de facilitar la validación de los *links* legítimos.
- Lograr identificar unívocamente cada artefacto sobre el que se desee realizar *traceability*.

4.2. El esquema de traceability propuesto

Con el fin de cumplir con las consignas fundamentales listadas antes en 4.1.1, se definió una propuesta de *esquema de traceability* que se encuentra representado en el diagrama de clases del dibujo 4.1.

Como se puede observar, el meta-modelo se puede dividir lógicamente en dos partes, la principal en la que se encuentran las abstracciones fundamentales de los elementos mínimos necesarios en cualquier herramienta que desee ofrecer *traceability*, y una secundaria en la que se engloba la configuración del esquema. Esta configuración permite ofrecer un meta-modelo flexible a cualquier escenario/dominio en el que se desee hacer uso de *traceability*.

Clases fundamentales

Entre los elementos fundamentales se encuentra la clase *TraceLink*, la cual representa un tracelink. Esta clase está compuesta por un nombre que sirve como identificador (*name*), un tipo con el cual se determina su semántica (*type*), una referencia de la transformación a la que pertenece la cual es optativa porque un tracelink puede o no ser producto de una transformación, y por último dos conjuntos de artefactos: el de los orígenes/fuentes (*sources*) y el de los objetivos/resultados (*targets*).

Los artefactos se encuentran representados por la clase *Artefact*, cuya identificación también pasa por su nombre (*name*), su descripción también por un diccionario de clave-valor, y por último su semántica también puede ser definida por un tipo definido en la configuración.

Por último entre los elementos fundamentales, tenemos la clase *Transformation* que abstrae de las transformaciones un nombre para su identificación, una descripción vía un diccionario y la colección de *TraceLinks* que son producto de su ejecución.

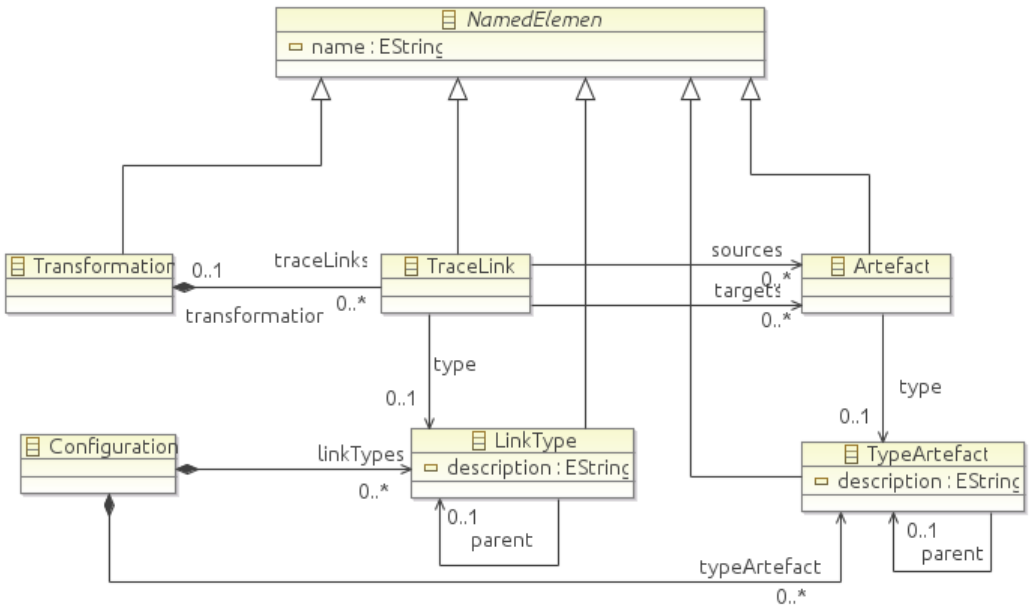

Figura 4.1: Esquema de traceability propuesto

Configuración del esquema

Esta parte refleja la configuración necesaria para lograr un esquema adaptable a la infinidad de escenarios en los que puede aplicarse el mismo. Dicha configuración consiste en la definición de dos jerarquías, una que define la semántica de los artefactos que se van a ***tracear***, y la otra para determinar la semántica de las tracelinks que pueden llegar a crearse. Ambas jerarquías están determinadas mediante las clases *LinkType* y *TypeArtefact* respectivamente, éstas comparten la siguiente estructura: un nombre (*name*) que las identifica, un texto que posibilita describirlas (*description*) y una relación (*parent*) que asocia una clasificación con su padre, gracias a dicha relación se logra la estructura jerárquica.

Un ejemplo de una configuración de tipos de tracelinks puede ser la jerarquía propuesta en la sección 2.3.1, que se puede ver siguiendo los diagramas 2.1, 2.2 y 2.3.

4.2.1. Lo que no ofrece

A continuación se lista un conjunto de aspectos y/o funcionalidades que el esquema propuesto actual no ofrece o tiene en cuenta:

- ***funcionalidad de versionado***: que permita navegar por los *tracelinks* y refleje las modificaciones en el tiempo que se fueron realizando sobre los artefactos relacionados, a lo largo de la ejecución de todas las tareas que forman parte de la ingeniería de un software.
- ***métodos de detección o información de errores***: que determine o informe cuándo un *tracelink* es inválido.

4.3. El prototipo

En la presente sección se explica y presenta el prototipo desarrollado que materializa la idea detrás del *esquema de traceability* propuesto en la sección anterior 4.2.

4.3.1. La herramienta

La idea es lograr una herramienta que dada la definición de una transformación acompañada con el conjunto de modelos de entrada/origen y de resultado/destino, retorne y muestre de una forma amigable el mapa de *tracelinks* que se generaron implícitamente productos de su ejecución.

El modelo del mapa de *tracelinks* (o meta-modelo) que usa la herramienta se encuentra definido según el *esquema de traceability* propuesto y presentado en 4.2. Por otro lado, el prototipo de la herramienta propuesta trabaja sólo sobre transformaciones Query/View/Transformation (QVT), pero en la sección 4.4 de más adelante se presentan algunas ideas de cómo implementar algo similar para transformaciones Atlas Transformation Language (ATL) por ejemplo.

Para lograr el mapa de tracelinks, dada una definición de una transformación cualquiera QVT, primero se debe ejecutar dicha transformación con su modelo origen con el fin de obtener el modelo resultado. Como resultado de esta transformación, QVT genera implícitamente información de *tracelinks* en un modelo definido por el meta-modelo de *tracelinks* de QVT que se muestra en la imagen 4.2. Lo que sigue es tomar este modelo de *tracelinks* y transformarlo al nuestro (visto el meta-modelo en la imagen 4.1), para ésto como parte del prototipo se incluye una definición de una transformación (*QvtoTrace_To_Trace*) que realiza esta tarea.

Arquitectura

El prototipo se encuentra definido por dos módulos

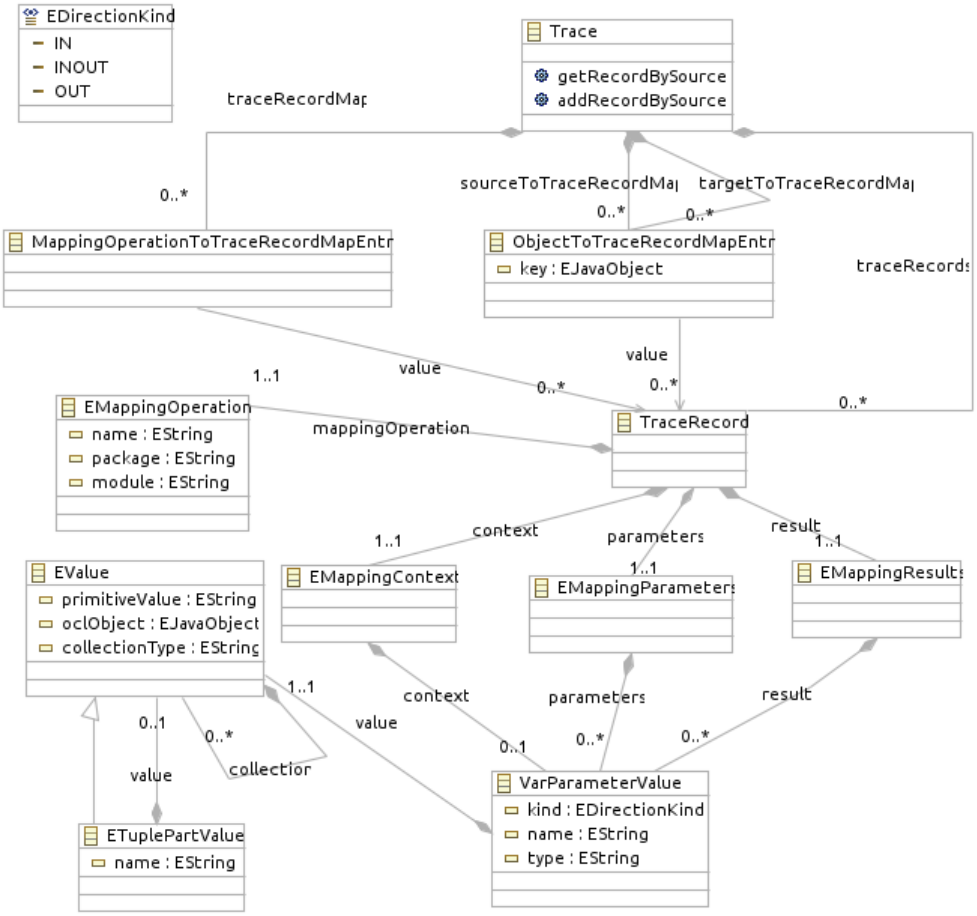

Figura 4.2: Modelo de tracelinks de QVT

4.4. Traceability en ATL

La información de *traceability* forma parte del lenguaje ATL, ésta ayuda en la interacción de cada regla de transformación cuando necesita tomar la salida de alguna otra regla por medio del mecanismo interno de *traceability* implícito. Sin embargo, actualmente ATL ofrece un acceso muy limitado a dicha información de *traceability* sólo por medio del método *resolveTemp()*. Además, ésta información es eliminada ni bien la ejecución de la transformación termina. Dada las carencias enumeradas antes, para lograr obtener alguna información de *traceability* en esta tecnología es necesario implementar de dicha funcionalidad.

Existen varias propuestas de implementación, una de ellas, la que se presenta en [9] en donde propone considerar tanto a los programas de transformación como a la información de *traceability* como modelos, entonces mediante la definición de una nueva transformación modificar los programas para que generen la información de *traceability* como una salida más al modelo resultado original. Con este enfoque, la generación de código de *traceability* está claramente separada de la lógica de la transformación y la misma se puede agregar después de que un programa haya sido escrito. Por lo cual, es posible añadir el soporte para nuevos formatos o modificar la granularidad o rango de las tracelinks sin interferir con la lógica del programa.

Otras propuestas hacen uso del mecanismo de *traceability* implícito de ATL, como copiar toda la información de tracelinks durante la ejecución de la transformación, o directamente persistirla y dejarla accesible en un archivo aparte, pero para todas estas implementaciones se requiere de modificaciones del motor de ejecución de ATL.

Capítulo 5

Descripción de tecnologías

En el presente capítulo se realiza una breve introducción sobre cada una de las tecnologías que fueron utilizadas para la implementación del prototipo que hace uso del esquema propuesto de *traceability*.

5.1. Eclipse

Empezamos con el proyecto base sobre el cual el prototipo va a funcionar/ejecutarse, en la presente sección daremos una introducción a Eclipse en la que se presenta el proyecto y se detalla en breve la plataforma con sus principales componentes y/o funcionalidades.

5.1.1. El proyecto

Según se presenta en [19, 17] Eclipse es un proyecto de desarrollo de software de código abierto, cuyo propósito es proveer una plataforma de herramientas altamente integradas para la construcción, implementación y administración de software a lo largo de todo su ciclo de vida.

El proyecto núcleo es un framework genérico para la integración de herramientas conjunto con un entorno de desarrollo Java que ya se incluye para usarlo. Otros proyectos extienden el framework núcleo para soportar distintos tipos de herramientas y ambientes de desarrollo específicos. Estos proyectos en Eclipse están implementados en Java y pueden ser ejecutados en muchos sistemas operativos.

La comunidad Eclipse tiene más de 200 proyectos, los cuales pueden conceptualmente organizarse dentro de las siguientes 7 categorías:

1. Enterprise Development

2. Embedded and Device Development
3. Rich Client Platform (RCP)
4. Rich Internet Applications (RIA)
5. Application Frameworks
6. Application Lifecycle Management (ALM)
7. Service Oriented Architecture (SOA)

Eclipse hace uso de la Eclipse Public License (EPL), dicha licencia comercial permite a las organizaciones incluir software Eclipse en sus productos comerciales, mientras que al mismo tiempo les solicita en retorno un aporte a la comunidad con algo del producto derivado comercializado.

5.1.2. La plataforma Eclipse

La plataforma Eclipse es un framework para la construcción de IDEs, el mismo ha sido descripto como "un ambiente para cualquier cosa y nada en particular". La plataforma define simplemente la estructura básica del IDE, luego mediante la definición de herramientas específicas que amplían y se conectan al framework terminan definiendo un IDE particular de forma colectiva.

Arquitectura de plugins

En Eclipse la unidad básica de funcionamiento, o sencillamente un componente, es llamado plugin-in. Tanto la plataforma misma de Eclipse como las herramientas que la extienden están compuestas por éstos plug-ins. Una herramienta sencilla puede consistir de un simple plug-in, pero las más complejas por lo general están divididas en varios de éstos.

Un plug-in incluye todo lo necesario para la ejecución del mismo, ésto puede ser: código Java, imágenes, textos, etc. También incluye un archivo manifiesto (plugin.xml), en el que se declaran las interconexiones con otros plug-ins.

Durante el arranque la plataforma Eclipse descubre todos los plug-ins disponibles, sin embargo éstos son sólo activados cuando es necesaria su ejecución con el fin de no ralentizar el arranque.

Workspace

Las herramientas integradas en Eclipse trabajan con archivos y carpetas ordinarias, pero también disponen de una Interfaz de programación de aplicaciones o Application Programming Interface (API) de alto nivel que define los siguientes componentes: recursos (resources), proyectos y un espacio de trabajo (workspace).

Resource Un recurso es la representación que da Eclipse de un archivo y/o una carpeta, a los que provee de capacidades adicionales como detectores de cambios (change listeners), marcadores como las listas por hacer (to-do list) y/o mensajes de errores, y un registro de historia de cambios.

Project Un proyecto es un recurso especial de tipo carpeta que es asignada por el usuario a una carpeta del sistema de archivos, es la que contiene todos los recursos del proyecto y la que define el tipo del mismo.

Workspace El workspace es el espacio de trabajo o contenedor virtual en el que se encuentran todos los proyectos del usuario.

Workbench

El Workbench es la ventana principal que se le presenta al usuario cuando ejecuta la plataforma Eclipse, se encuentra implementada usando Standard Widget Toolkit (SWT) y JFace. Esta ventana principal, que se puede apreciar en la imagen 5.1 está compuesta de vistas, editores y perspectivas.

Editores Éstos permiten al usuario abrir, editar y guardar distintos tipos de objectos.

Vistas proveen información de algún objeto sobre el que el usuario se encuentra trabajando en el Workbench. Por ejemplo, una vista puede asistir un editor dando información sobre el documento que se está editando.

Perspectivas Una perspectiva es sencillamente una agrupación de vistas y editores de manera que en su conjunto dan apoyo en una actividad completa.

5.1.3. Resumen y más información

Para cerrar y en resumen, la plataforma Eclipse proporciona un núcleo de elementos básicos y un conjunto de APIs genéricas, como el workspace

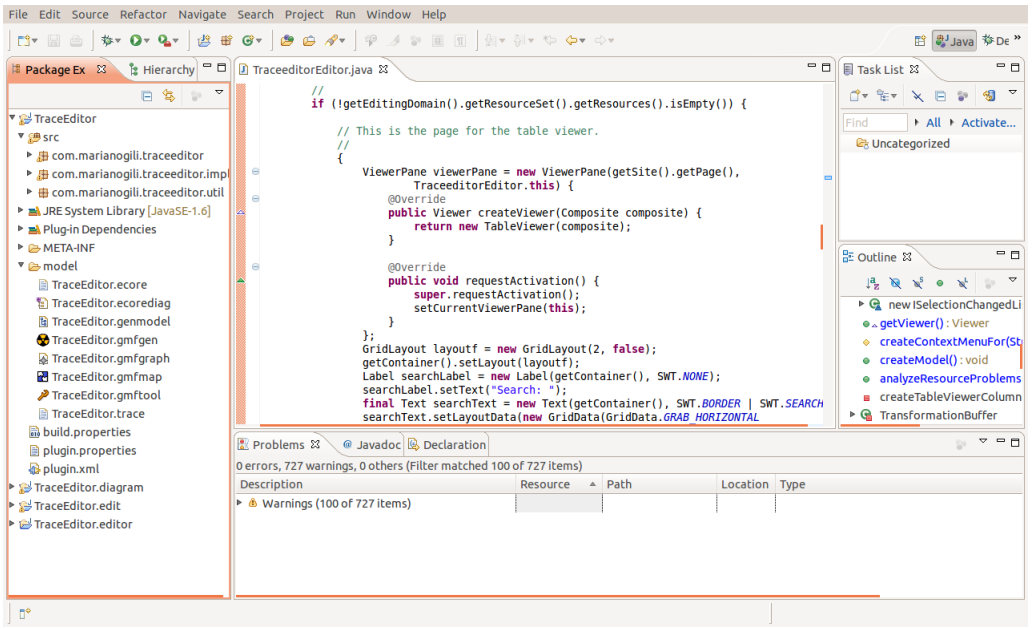

Figura 5.1: Elcipse Workbench

y el workbench, y varios puntos de extensión a través del cual se pueden integrar nuevas funcionalidades. A través de estos puntos de extensión, las herramientas escritas como plug-ins independientes pueden extender la plataforma Eclipse.

Para obtener más información sobre el proyecto Eclipse ver [17, www.eclipse.org], y para información técnica consultar [18, Eclipse Platform Technical Overview].

5.2. Eclipse Modeling Framework

A continuación proseguimos con una de las tecnologías del proyecto Eclipse usada para la implementación de una parte del prototipo, el framework EMF [15].

5.2.1. El framework EMF

El EMF es un framework de modelado y generación de código para herramientas de construcción y otras aplicaciones basadas en un modelo de datos estructurados. A partir de una especificación de un modelo descrito en XML Metadata Interchange (XMI), EMF provee herramientas y un entorno de ejecución para producir un conjunto de clases Java para el modelo y un

conjunto de clases adaptadoras que permiten la visualización y la edición de dicho modelo.

EMF consiste de tres partes fundamentales:

EMF El framework EMF base incluye un meta-modelo (Ecore) para la descripción de modelos y soporte para un entorno de ejecución para dichos modelos que incluye las funcionalidades de notificación de cambios, persistencia por defecto vía serialización XMI y una librería muy eficiente para la manipulación de objectos EMF genérica.

EMF.Edit El framework EMF.Edit incluye clases genéricas reusables para la construcción de editores de modelos EMF. Este provee:

- Clases proveedoras de contenido y etiquetas, acceso a propiedades orígenes y otras clases convenientes que permiten a los modelos EMF ser visualizados mediante entornos visuales estándares de escritorio (vía JFace) y/o hojas de propiedades.
- Un framework de comandos que incluye un conjunto de clases que implementan comandos para la construcción de editores con soporte íntegro de deshacer y rehacer (undo/redo).

EMF.Codegen La funcionalidad de generación de código EMF es capaz de generar todo lo necesario para la construcción de un editor de un modelo EMF completo. Ésta incluye una Interfaz de Usuario Gráfica o Graphical User Interface (GUI) desde la cual las opciones de generación pueden ser especificadas y los generadores son invocados. Dicha funcionalidad de generación aprovecha el componente de Eclipse Java Development Tool (*JDT*).

Tres niveles de generación de código son soportados:

Model Proporciona la implementación de interfaces y clases Java para todas las clases del modelo, además la implementación de una clase fábrica y el paquete.

Adapters Genera la implementación de las clases, llamadas ItemProviders, que se adaptan a las clases del modelo para su edición y visualización.

Editor Produce un editor estructurado apropiadamente que se ajusta al estilo recomendado para los editores de modelo EMF de Eclipse y sirve como punto de partida para comenzar con la personalización.

5.2.2. El (Meta) modelo Ecore

Ecore es el modelo usado para representar modelos en EMF. Ecore es en sí mismo un modelo EMF, por lo cual es su propio meta-modelo.

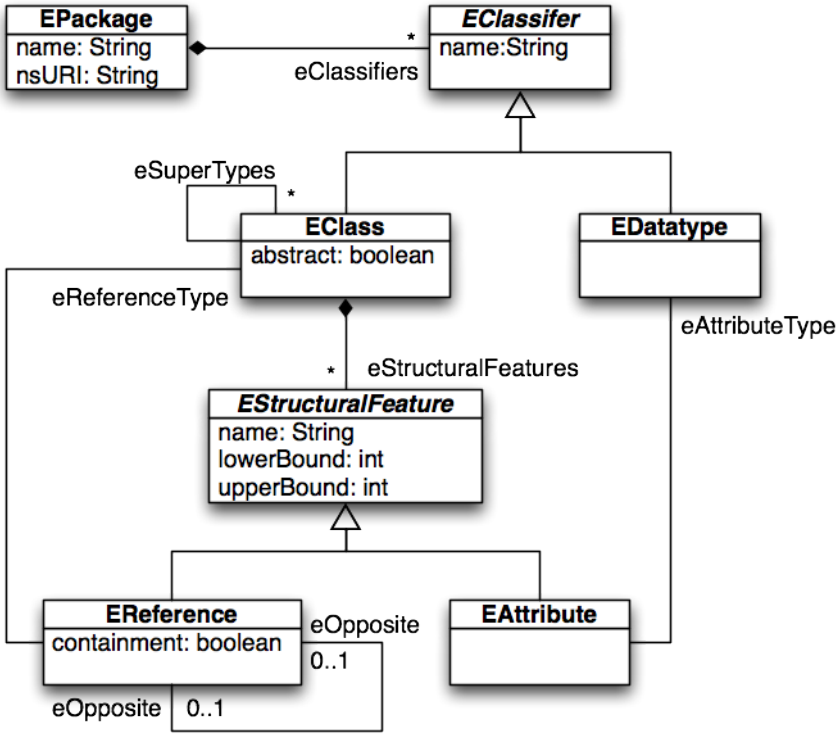

Figura 5.2: Modelo Ecore simplificado

Un modelo simplificado de Ecore se puede observar en la imagen 5.2, en particular lo conforman las siguientes cuatro clases Ecore principales:

EClass Usada para representar una clase en el modelo. Tiene un nombre, atributos y referencias.

EAttribute Para representar en el modelo un atributo. Tiene un nombre y un tipo.

EReference Para representar una relación entre clases. Tiene un nombre, una marca que indica si es una composición, y el tipo de clase referenciada u objetivo que es otra clase.

EDataType Representa el tipo de un atributo. El mismo puede ser un tipo primitivo (int, float) o un tipo objeto.

5.2.3. Beneficios y más información

Además de incrementar la productividad gracias a la generación automática de código, el uso del framework EMF provee otros beneficios: la notificación de cambios, una funcionalidad de persistencia (como la serialización XMI entre otras), y una API genérica reflexiva muy eficiente que sirve para la manipulación de objetos EMF. Sin embargo, uno de los beneficios más importantes de este framework es que EMF provee las bases para la interoperabilidad con otras herramientas y aplicaciones que se basan en él.

Más información sobre el framework EMF se puede obtener en [15] y [19].

5.3. Graphical Modeling Framework

En la presente sección analizamos el framework Graphical Modeling Framework (GMF), el mismo fue usado para el desarrollo de la parte gráfica del editor de tracelinks del prototipo.

5.3.1. El framework GMF

GMF es un framework para la construcción de editores gráficos de modelado para la plataforma Eclipse, por ejemplo editores UML, ECore, de procesos de negocios, diagramas de flujo, etc. Este framework está compuesto por un componente de generación (GMF Tooling) y un entorno de ejecución (GMF Runtime) que ayudan en el desarrollo de editores gráficos que se basan en EMF y Graphical Editing Framework (*GEF*).

GMF Tooling incluye por un lado editores para crear y/o modificar los modelos que describen los aspectos de notación, semántica y utilidad de un editor gráfico, y por otro, un generador que da como resultado la implementación del editor definido.

GMF Runtime Los plug-ins generados con GFM-Tooling dependen de este componente que es el entorno de ejecución sobre el que va a correr el editor.

5.3.2. Arquitectura

En el diagrama de componentes que podemos ver en la figura 5.3 se muestran las dependencias existentes entre el editor gráfico generado, el entorno GMF Runtime, EMF, Graphical Editing Framework (*GEF*) y la plataforma Eclipse. Como se puede ver, el editor gráfico GMF depende del componente GMF runtime, y a su vez hace uso directo de EMF, Graphical Editing Framework (*GEF*) y la plataforma Eclipse.

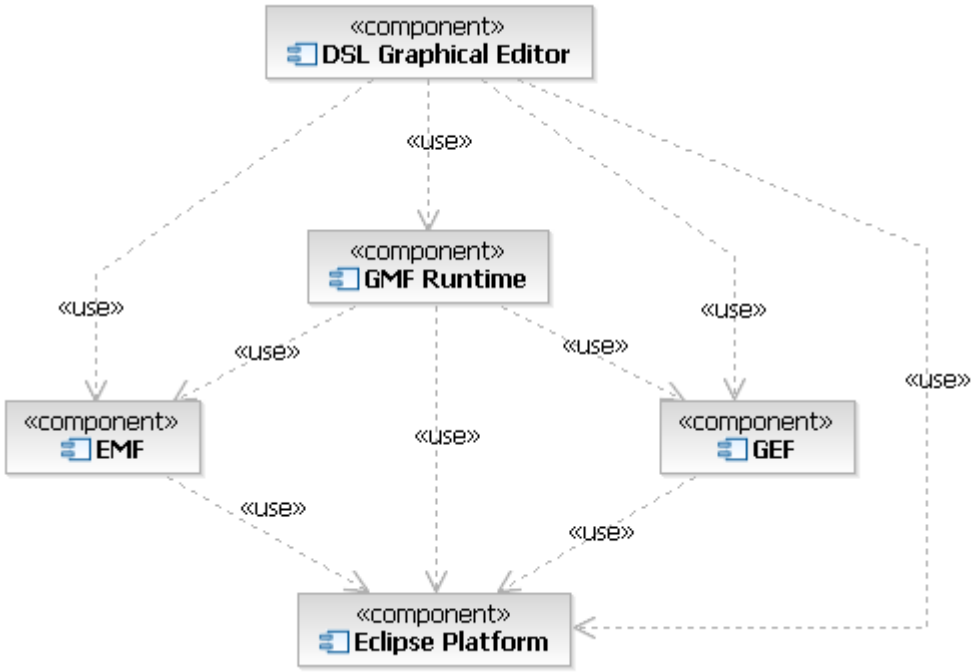

Figura 5.3: Arquitectura GMF

5.3.3. Modelos y flujo de trabajo

En el diagrama 5.4 se encuentran los principales componentes y modelos usados durante el desarrollo de un editor gráfico mediante el framework GMF. Entre ellos se encuentra el concepto de **modelo de definición gráfica**, el cual contiene toda la información relacionada con los elementos gráficos que formarán parte del editor (figuras, nodos, enlaces, etc), pero que no tiene ninguna conexión directa o es dependiente con ninguno de los componentes del modelo de dominio para el cual ofrecerá la representación y/o edición. También tenemos el **modelo de definición de herramientas** que es opcional y usado para el diseño de la paleta, el menú y las barras de herramientas.

Tanto la definición gráfica como la de las herramientas pueden funcionar para modelos variados de dominios, éste es uno de los objetivos de GMF, lograr que estas definiciones sean reusables para distintos dominios que se puedan presentar. Lo anterior se logra gracias al uso de un **modelo de asignación o mapeo**, que se encuentra separado y vincula las definiciones gráficas y las herramientas con los correspondientes modelos de dominios seleccionados.

Una vez fueron definidas los mapeos o asignaciones, GMF dispone de un **modelo de generación** el cual posibilita la definición de los detalles para la implementación de la siguiente y última fase, la generación del editor.

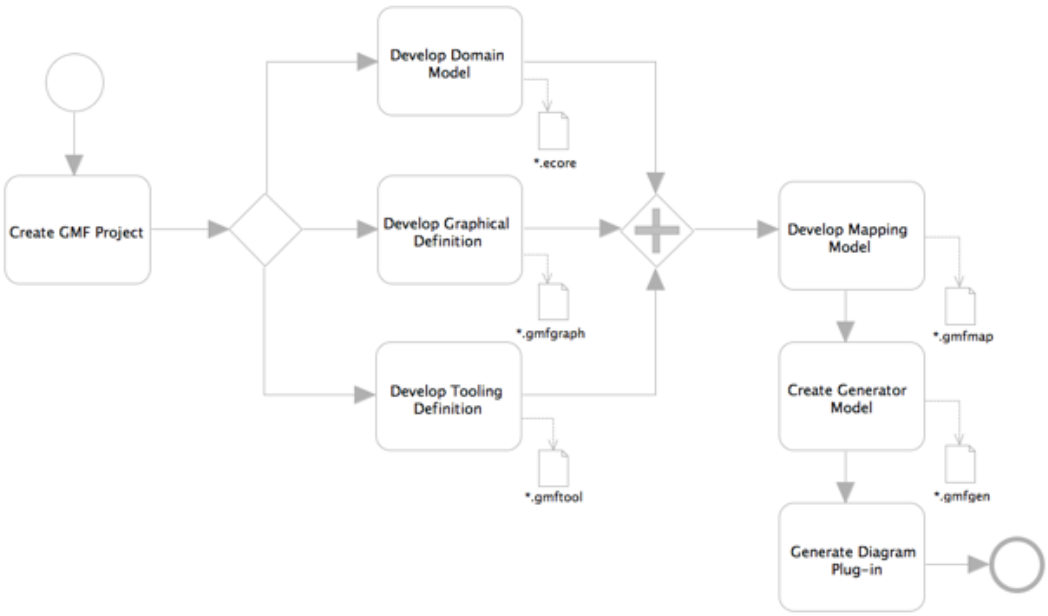

Figura 5.4: Flujo de trabajo de GMF

Flujo de trabajo

1. Creación del modelo del dominio, en este modelo se define la información no gráfica que gestiona el editor.

2. Creación del modelo de definición gráfica, en el que se definen los elementos gráficos que se mostrarán/presentarán en el editor.

3. Creación del modelo de asignación gráfica, que es el que define la correspondencia entre los elementos del modelo del dominio con los elementos gráficos del modelo de definición gráfica.

4. Generación del editor gráfico.

5. Mejorar el editor gráfico por medio de la edición del código del plug-in generado.

5.3.4. Más información

Más información sobre el framework GMF se puede obtener en [20], [21] y [22].

5.4. Atlas Transformation Language

A continuación se da una introducción a la tecnología ATL, que no solo refiere a un lenguaje de transformación de modelos como su nombre puede confundir, sino que además trata de un conjunto de herramientas de desarrollo construidas para ser ejecutadas sobre la Plataforma Eclipse (introducida en 5.1.2).

5.4.1. ¿Qué es ATL?

En el campo de MDE, ATL nos ofrece un medio para especificar la forma de producir un conjunto de modelos resultados/destinos a partir de un conjunto de modelos fuentes.

El lenguaje ATL es un híbrido de la programación declarativa e imperativa, dado que aunque el estilo declarativo es el más conveniente para la definición de las transformaciones, ATL también provee la posibilidad de construcciones imperativas con el fin de facilitar la especificación de algunos mapeos que en forma declarativa pueden llegar a resultar muy complejos de expresar.

Por otro lado, el IDE ATL provee un conjunto de herramientas estándar con el fin de facilitar el desarrollo de las transformaciones mediante este lenguaje como el resaltado de sintaxis, el auto-completado de código, un depurador, entre otras.

5.4.2. Conceptos de ATL

Un modelo fuente se transforma en un modelo destino gracias a la definición de una transformación escrita en ATL, la cual también es un modelo. Los modelos fuente, destino y la definición de la transformación, cada uno conforman a sus respectivos meta-modelos y, a su vez, todos los meta-modelos

se ajustan a MetaObject Facility (*MOF*) o Ecore. Esta relación se puede observar en la imagen 5.5.

Una transformación ATL es unidireccional, o sea que trabaja sobre un modelo fuente de solo lectura y produce un modelo destino de solo escritura. Durante la ejecución de una transformación, el modelo fuente puede ser navegado pero no cambiado, en cambio el modelo destino no puede ser navegado.

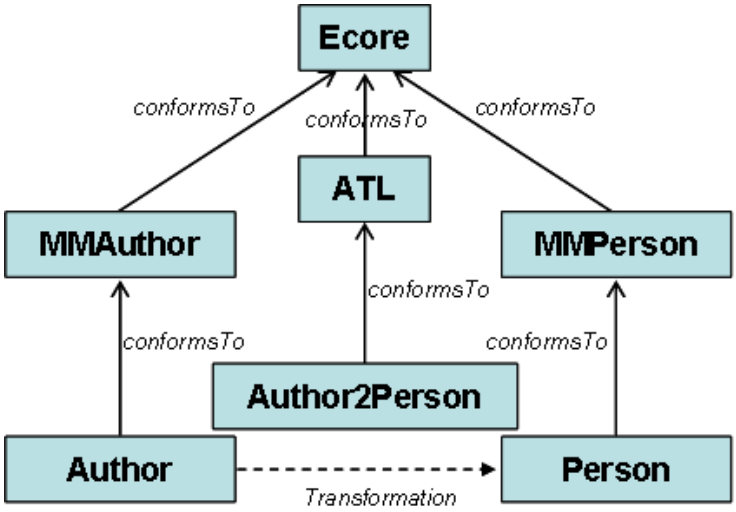

Figura 5.5: Modelos, transformaciones y sus meta-modelos

5.4.3. El lenguaje ATL

En este lenguaje de transformaciones modelo a modelo o Model-to-Model Transformation (MMT), las operaciones de transformación son especificadas mediante módulos ATL. Cada módulo ATL permite a un desarrollador especificar la forma de producir un conjunto de modelos resultados desde un conjunto de modelos fuentes/orígenes.

Además de módulos, este lenguaje permite crear programas que transforman modelos en tipos de datos primitivos (como booleanos, enteros o cadenas), lo cual se logra mediante la especificación de unidades llamadas ATL queries.

Por último ATL ofrece la posibilidad de desarrollar librerías independientes que pueden ser importadas a lo largo de diferentes tipos de unidades ATLs. Ésto nos da una forma conveniente de factorizar el código que va a ser usado por muchas unidades ATL.

5.4.4. Más información

Más información sobre la tecnología ATL se puede encontrar en la sección de documentación de la página del proyecto [23, www.eclipse.org/atl/].

5.5. QVT

Por último en este capítulo dedicado a las tecnologías que forman parte de la presente tesis, se introduce el estándar para transformaciones especificado por la Object Management Group (*OMG*), el lenguaje QVT.

5.5.1. Introducción a QVT

QVT como ya se dijo antes es el estándar de Object Management Group (*OMG*) para la definición de transformaciones. La especificación de QVT es híbrida, como en el caso de ATL, relacional (o declarativa) y operacional (o imperativa). Comprende tres diferentes lenguajes MMT: dos lenguajes declarativos llamados Relations y Core, y un tercer lenguaje, de naturaleza imperativa, llamado Operational Mappings. Puede observarse la relación de los meta-modelos de cada uno de los lenguajes en la figura 5.6.

La naturaleza híbrida de este estándar fue introducida para abarcar distintos tipos de usuarios con diferentes necesidades, requisitos y hábitos.

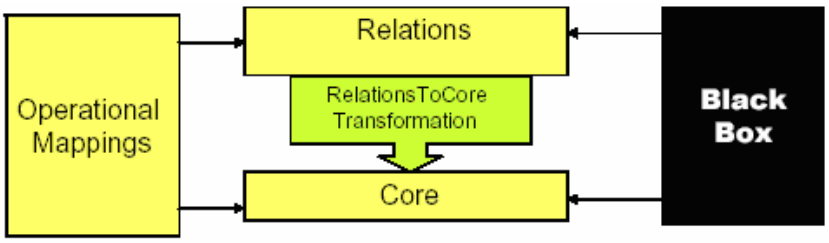

Figura 5.6: Relación entre los meta-modelos QVT

5.5.2. Lenguajes QVT

QVT declarativo

Relations Es un lenguaje amigable para el usuario, que soporta pattern matching complejo y la creación de templates para objetos. Tiene creación de tracelinks implícita. También incluye la propagación de cambios, ya que provee un mecanismo para identificar elementos del modelo destino.

Core Es un lenguaje pequeño con un soporte de pattern matching acotado. Tiene *traceability* explícito y las tracelinks pueden crearse y borrarse como cualquier otro objeto. Es igual de poderoso semánticamente que el lenguaje Relations pero trabaja a un nivel más bajo de abstracción. Esta propuesta absolutamente minimal lleva a que el lenguaje Core sea usado como un "assembler" de los lenguajes de transformación.

QVT imperativo

Operational Mappings Este lenguaje se especificó como una forma estándar para proveer implementaciones imperativas de transformaciones unidireccionales. También como el lenguaje Relations, dispone de creación de tracelinks implícita sobre el mismo modelo de tracelinks.

Implementaciones Black-box Estas implementaciones de caja negra, permite escribir transformaciones en otro lenguaje distinto a QVT. Una implementación de este tipo no tiene relación explicita con el lenguaje Relations, aunque una caja negra podría implementar una Relation, la misma debe ser responsable de mantener las tracelinks entre los elementos relacionados.

5.5.3. Más información

Para obtener más información introductoria consultar [25]. Para más detalles técnicos y ejemplos se puede leer el documento de especificación MetaObject Facility (*MOF*) Query/View/Transformation [24].

Capítulo 6

Trabajos relacionados

En este capítulo se realiza una breve introducción de un conjunto de trabajos que tienen alguna relación con el tema en cuestión de la presente tesis, encontrados a lo largo del aprendizaje y la investigación.

6.1. Un motor de traceability de transformación de modelos en la Ingeniería de Software

En [14] se describe un motor de *traceability* al que llamaron ETraceTool, que funciona como un plug-in de Eclipse programado mediante el paradigma orientado a aspectos con el fin de mantener aislada la generación de las tracelinks del código que pertenece a la transformación. El mismo trabaja sobre transformaciones escritas en Java usando la API EMF [15]. A continuación se listan sus principales características:

- El código de generación de tracelinks no es intrusivo en el código de la transformación;
- La generación de tracelinks tiene que ser activada explícitamente por el diseñador de la transformación;
- Los modelos de las tracelinks se encuentran aislados de los modelos origen y destino que forman parte de la transformación;
- Los modelos de las tracelinks pueden ser usados a diferentes niveles de granularidad.

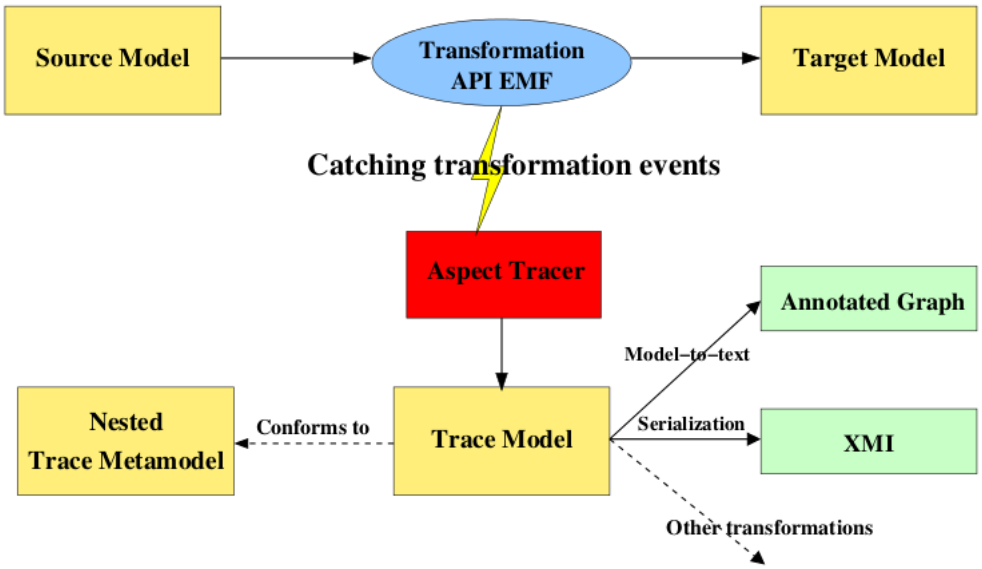

Figura 6.1: Arquitectura de la herramienta ETraceTool

La arquitectura se puede apreciar en el dibujo 6.1 y se explica de la siguiente manera, durante la transformación el plug-in captura un conjunto de eventos previamente identificados y clasificados gracias a la programación orientada a aspectos, luego el Aspect Tracer genera un modelo de tracelinks que conforma al meta-modelo de tracelinks anidado que se muestra en el dibujo 6.2. Al final, el modelo de tracelinks generado puede ser serializado en un archivo XMI o transformado a cualquier otro lenguaje mediante una transformación modelo-texto.

El fundamento del diseño del meta-modelo de tracelinks anidado propuesto es para el caso en el que se presente una operación de transformación que llama o hace uso de otra transformación. En este caso el enlace compuesto permite separar las operaciones de bajo nivel (creación, eliminación, etc) de las operaciones de alto nivel (como una operación de refactorización).

6.2. Un Framework de Traceability dirigido por modelos para el desarrollo de Software Product Line (SPL)

El framework presentado en [16] provee una plataforma abierta y flexible para crear enlaces de tracelinks entre distintos artefactos del desarrollo de una Línea de Producto de Software o Software Product Line (SPL). Pero,

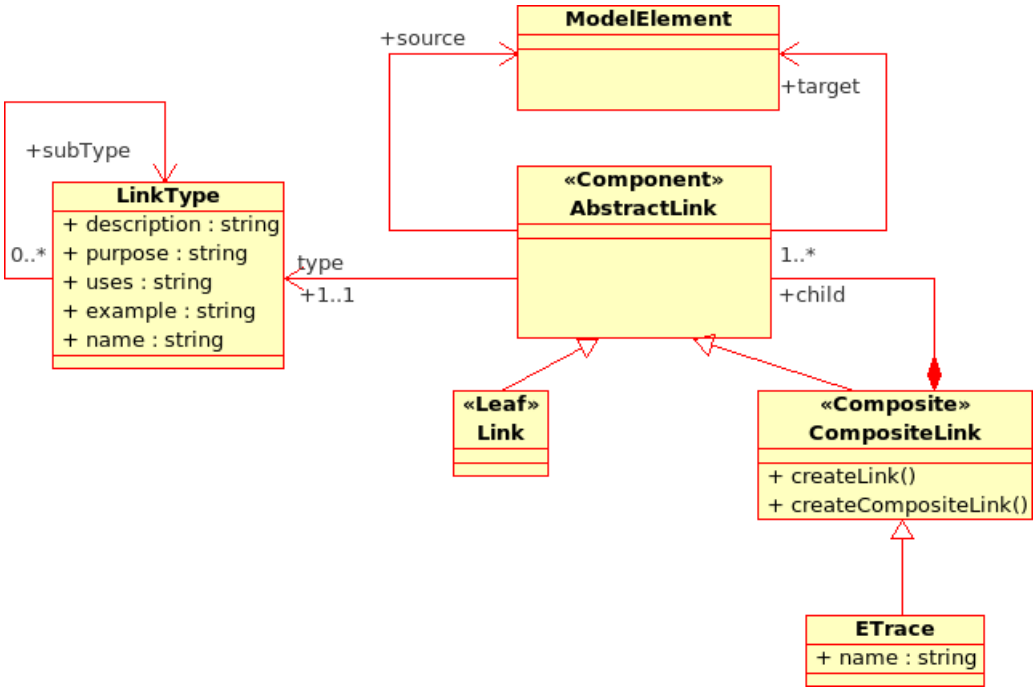

Figura 6.2: Meta-modelo de tracelinks anidado

dado que el diseño del framework es genérico, éste también puede aplicarse o usarse en otras áreas del desarrollo SPL. El mismo ha sido diseñado e implementado basado en el uso de técnicas dirigidas por modelos. El meta-modelo de *traceability* descripto en el dibujo 6.3 permite definir distintos tipos de enlaces de tracelinks entre los artefactos.

Las principales funcionalidades ofrecidas por el framework son las siguientes:

1. Creación y mantenimiento de los enlaces de tracelinks de artefactos existentes (modelos UML, código fuente, etc);

2. Almacenamiento de los enlaces de tracelinks mediante el uso de un repositorio;

3. Búsqueda de enlaces de tracelinks específicos usando consultas de tracelinks predefinidas o personalizadas;

4. Visualización flexible de los resultados de las consultas de tracelinks por medio de diferentes tipos de vistas, como vista de árbol, grafo, tabla, etc.

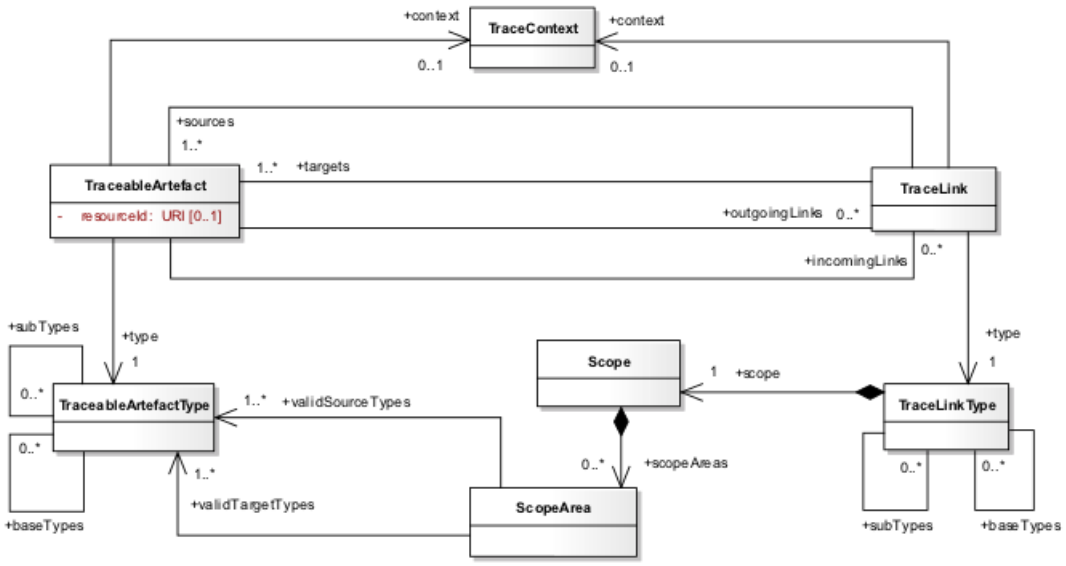

Figura 6.3: Meta-modelo de traceability

6.2.1. Meta-modelo de traceability

Los elementos principales del meta-modelo son los siguientes:

- Un TraceableArtefact representa un artefacto que juega un rol en el ciclo del desarrollo. La granularidad del artefacto es arbitraria, puede ser un requerimiento, un diagrama UML, un elemento de dicho diagrama, una clase o un método de dicha clase. El artefacto es identificado mediante la propiedad resourceId.
- Un TraceLink es la abstracción de una transición de un artefacto a otro.
- Cada TraceableArtefact tiene asignada una instancia de TraceableArtefactType, éstos se pueden encontrar agrupados de forma jerárquica.
- Análogo a los tipos de los artefactos los TraceLinks también tienen un tipo (TraceLinkType), teniendo en cuenta que la semántica de una relación entre dos artefactos puede variar.
- Información adicional de artefactos y enlaces puede ser modelada mediante un contexto, el cual se encuentra representado por la clase TraceContext.
- Las restricciones que determinan el conjunto de artefactos válidos, sobre los cuales los tipos de enlaces también son válidos se encuentran modeladas mediante los elementos ScopeArea y Scope.

6.2.2. Arquitectura

Como se muestra en el dibujo 6.4, la arquitectura ha sido definida en término de cuatro módulos principales. Cada uno de los cuales implementa una de las funcionalidades principales del framework, se detallan a continuación:

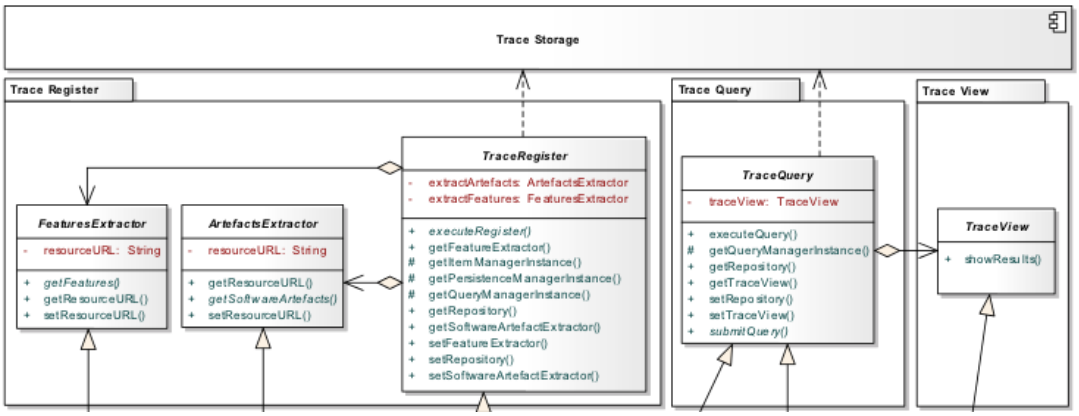

Figura 6.4: Arquitectura del Framework de Traceability

1. Trace Register: este módulo provee mecanismos para crear y mantener (actualizar, eliminar y buscar) los enlaces de tracelinks;

2. Trace Storage: define los mecanismos de almacenamiento para persistir dichos enlaces;

3. Trace Query: este modulo permite crear y ejecutar consultas para buscar enlaces de tracelinks específicos que se encuentran previamente almacenados;

4. Trace View: usado específicamente para la representación visual de los enlaces resultados de una consulta realizada.

6.3. Integración de herramientas Case

En [11] se presenta el problema real que sufre cualquier proceso de desarrollo actual, en el que como resultado del conjunto de actividades que lo conforman, se van generando una variedad muy amplia de artefactos de software (documentos de textos, hojas de cálculo, resultado de pruebas, modelos, gráficos, etc). Estos artefactos, aunque en esencia se encuentran relacionados

lógicamente, al ser creados y manipulados por herramientas muy distintas que no fueron pensadas para interactuar (editores de textos, editores de modelo UML, etc), las relaciones lógicas nombradas se pierden, o mejor dicho no existen o pasan desapercibidas en la práctica. En otras palabras, nos presenta el problema de la imposibilidad de *traceability* que encontramos entre la mayoría de las herramientas Computer Aided Software Engineering (CASE) actuales.

Como solución a este problema, se propone un ambiente de integración para las herramientas CASE al que llamaron TiE - Tool Integration Environment, el cual basa su integración en la creación de enlaces de tracelinks entre los artefactos de las distintas herramientas.

6.4. Framework genérico de extracción de datos de traceability

En [13] proponen un framework genérico de *traceability* que toma como entrada una transformación de modelo y le aumenta arbitrariamente su funcionalidad con un mecanismo de *traceability*. En el dibujo 6.5 se muestra un panorama de alto nivel de la arquitectura propuesta. Ésta se basa en una interfase genérica que provee un punto de conexión para cualquier motor de transformación de modelos, mediante una API que se ofrece al ingeniero que conecta su motor de transformación con el motor de *traceability* (oAW connector, QVT connector). Como resultado el motor de transformación incluye la funcionalidad de *traceability*. El modelo de datos que usa el framework es el lenguaje específico de dominio para *traceability* al que llaman Trace-DSL.

Trace-DSL que se detalla en el dibujo 6.6, tiene como elemento raíz TraceModel. Artefact representa cualquier producto traceable generado durante el proceso de desarrollo, esto puede ser un requerimiento o una clase o un componente, como el método que pertenece a una clase. Todo artefacto es identificado unívocamente mediante un identificador único universal (UUID). TraceLink es una abstracción de una transición de un artefacto a otro dirigida por la relación desde-hacia entre artefactos origen y destino. TraceLink puede ser de uno de las siguientes cuatro instancias: CreateTraceLink, QueryTraceLink, UpdateTraceLink y DeleteTraceLink.

Para la asignación de tipos a los artefactos y a los enlaces se usa el concepto de faceta, donde Trace-DSL asigna un conjunto de facetas (Facet) a cada uno de los mismos. Un ejemplo de faceta se da en el dibujo 6.7. Además de lograr una solución simple al tipado de artefactos y enlaces, se obtiene un mecanismo fácilmente extensible y configurable al contexto donde se necesite

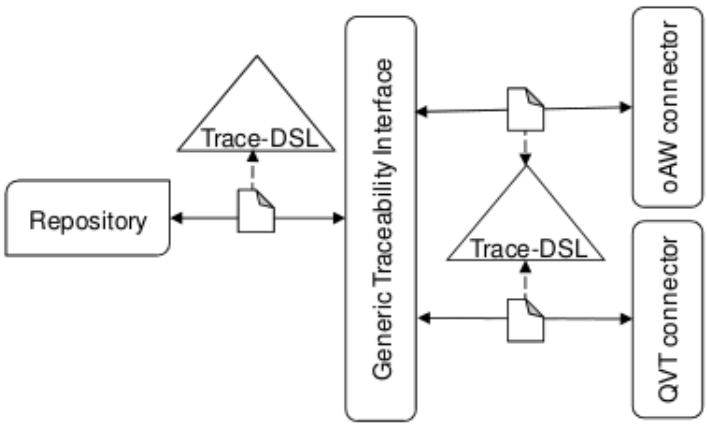

Figura 6.5: Resumen de la arquitectura del Framework Genérico de Traceability

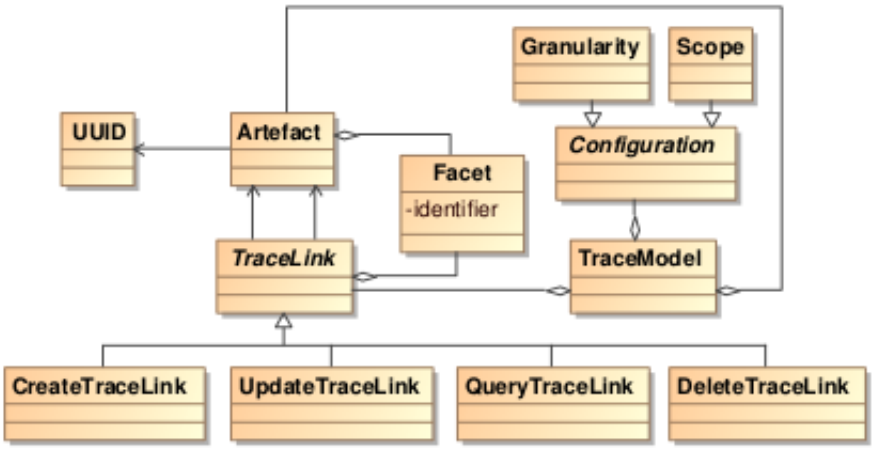

Figura 6.6: Lenguaje específico de dominio para traceability

aplicar *traceability*.

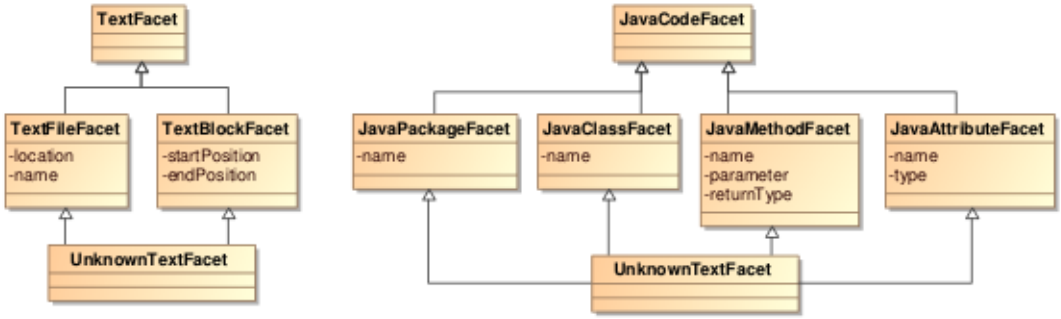

Figura 6.7: Faceta para traceability de código fuente

La configuración necesaria para hacer uso del framework implica:

1. Seleccionar las facetas requeridas para el escenario
2. Configurar la granularidad (Granularity) y el alcance (Scope)

La configuración de la granularidad consiste en la especificación de qué tipos (definidos por las facetas) de artefactos y enlaces serán trazados para un escenario de *traceability* particular. En cambio la configuración del alcance implica restringir los datos de *traceability* a una combinación específica de valores. En otras palabras la primera solo chequea la existencia de facetas, mientras que la segunda adicionalmente examina las propiedades especificas de las facetas. Por ejemplo en el caso de TextFileFacet, puede ser necesario trazar solo archivos de textos con cierto nombre.

6.5. Traceability local y global

En la propuesta presentada en [8] usan la idea de separación del proceso de *traceability* en los siguientes niveles, *traceability* en lo pequeño y *traceability* en lo grande refiriéndose a los mismos como *traceability* local y *traceability* global respectivamente.

6.5.1. Meta-modelo de Traceability Local

Este meta-modelo toma las tracelinks de la entrada y la salida de una única transformación. El meta-modelo está basado en el meta-modelo de tracelinks presentado en [9] y se muestra en el dibujo 6.8.

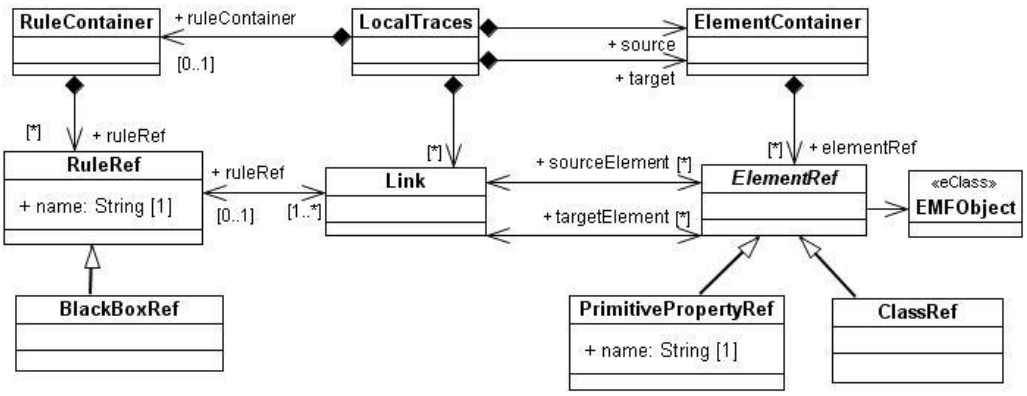

Figura 6.8: Meta-modelo de tracelinks Local

El meta-modelo de tracelinks local contiene dos conceptos principales Link y ElementRef en donde se expresa que uno o más elementos orígenes son enlazados a uno o más elementos objetivos. ElementRef es una clase abstracta que representa elementos que pueden ser traceados: instancias de clases o valores de propiedades. Los valores de las propiedades son traceados usando PrimitivePropertyRef el cual apunta a la instancia contenedora de la propiedad y tiene el nombre de la misma. Este tratamiento especial para los tipos primitivos de Java se debe a que no existen instancias de ellos en el modelo. Por otro lado las propiedades que son tipadas mediante una clase normal, son traceadas mediante ClassRef.

Para almacenar la información sobre las reglas de transformación aplicadas así como el caso particular de las cajas negras, se hace uso de los conceptos RuleRef y BlackBoxRef. En ambos casos, dado que pueden dar como resultado de su ejecución varios enlaces, se define la relación como uno a varios entre RuleRef y Link. RuleRef y BlackBoxRef son opcionales, en el caso de la primera solo se usa para realizar una depuración de las transformaciones, y la segunda si nos encontramos en la situación en la que ciertas partes del sistema no pueden ser pública su implementación.

ElementRef tiene una referencia al objeto real de los modelos origen y destinos. Como estos modelos están implementados mediante EMF, la referencia EMFObject es un EObject del meta-modelo Ecore. La clase LocalTraces representa la raiz del modelo de tracelinks local y tiene un RuleContainer que se usa como contenedor de las reglas y dos ElementContainer usados para agrupar los ElementRef origen y destino respectivamente. Separar los elemento orígenes y destinos permite reducir los costos de búsquedas de elementos de entrada o salida.

6.5.2. Meta-modelo de Traceability Global

Este meta-modelo enlaza tracelinks locales de acuerdo a la cadena que define la transformación. Un modelo de tracelinks global es el punto de entrada principal en el cual todas los modelos de tracelinks locales se encuentran, y describe qué modelo origen/objetivo de una transformación es el modelo objetivo/origen de la siguiente/previa transformación. En el dibujo 6.9 se puede observar el meta-modelo.

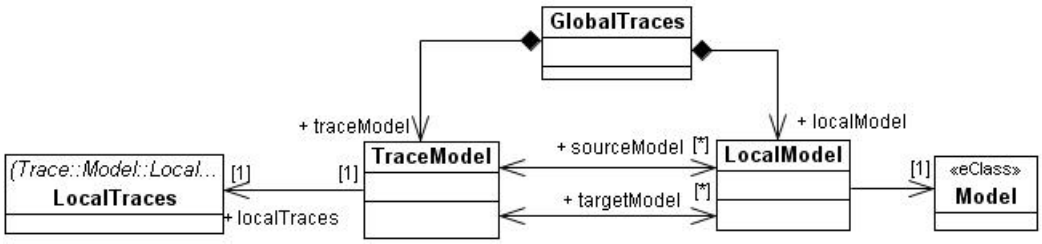

Figura 6.9: Meta-modelo de tracelinks Global

Este meta-modelo se engloban todas las tracelinks locales, los modelos de una cadena de transformación y la forma en que éstos se encuentran enlazados mediante TraceModel y LocalModel. Los modelos pueden ser compartidos entre distintas transformaciones, es decir uno puede ser producto de una transformación y también ser consumido por otra transformación.

Introducir este nivel global de tracelinks permite la navegación entre los modelos transformados y sus modelos de tracelinks locales, dando una mejor separación de lo que es en verdad la compleja información de *traceability*, lo que permite además una mejor flexibilidad para la creación de las tracelinks y la explotación de las mismas. No utilizar esta idea de tracelinks globales tiene como consecuencia disponer de todas las tracelinks en un único modelo de toda una cadena de transformación, lo cual desencadenaría en el colapso para la creación y la consulta de dicho modelo.

6.5.3. ¿Cómo trabaja el framework?

Una de los principales objetivos de recolectar las tracelinks es dar luego la posibilidad a un usuario de inspeccionarlas realizando distintas consultas, una puede ser por ejemplo obtener los elementos relacionados a uno seleccionado. Este meta-modelo permite desde el modelo de tracelinks global navegar hacia los modelos de tracelinks locales y/o hacia a los modelos envueltos en cada una de las transformaciones. También desde el modelo de tracelinks local se puede navegar entre los elementos del modelo parte de la transformación.

En el dibujo 6.10 se muestra un ejemplo de un modelo de tracelinks locales y globales producto de una cadena de transformación. En el ejemplo se representan los enlaces (Link) entre los elementos sin tener en cuenta las instancias de RuleRef para no sobrecargarlo.

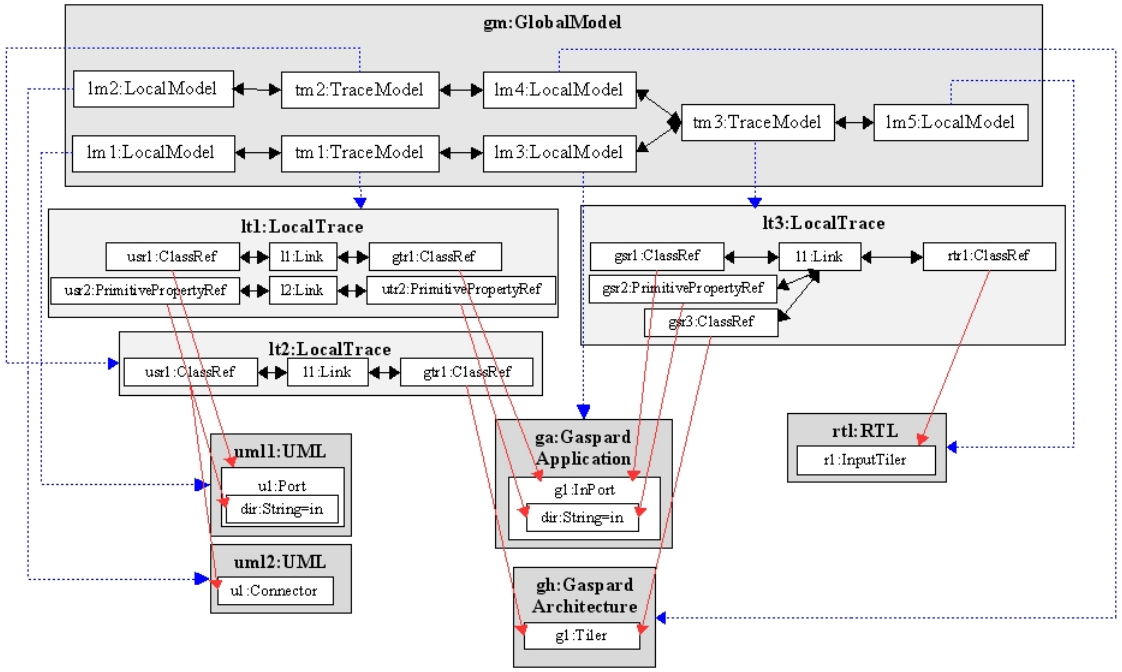

Figura 6.10: Ejemplo de un modelo de tracelinks local y global

Conclusión

Anexos

I: QvtoTrace_To_Trace

A continuación la definición completa de la transformación QVT Qvto-Trace_To_Trace implementada para el prototipo desarrollado en la presente Tesis:

```
/*
 * La definición de la presente transformación
 * QvtoTrace_To_Trace forma parte del prototipo
 * desarrollado para la tesis del alumno Mariano Gabriel
 * Gili.
 *
 * Esta transformación toma como entrada las trazas generadas
 * por la tecnología QVT que conforman el meta-modelo Trace,
 * y genera un modelo para el prototipo de edición de trazas
 * que conforma el meta-modelo propuesto TraceEditor.
 */

/*
 * Definición de los meta-modelos.
 */
modeltype TraceEditor uses 'http://traceeditor/1.0';
modeltype QvtoTrace uses
    'http:///www.eclipse.org/m2m/qvt/operational/trace.ecore'
        ;

/*
 * Declaración de la transformación.
 */
transformation QvtoTrace_To_Trace(in qvto : QvtoTrace,
    out TraceEditor);

/*
 * Punto de entrada donde se inicia la transformación por los
 * objectos raiz de tipo Trace del modelo de entrada mediante
 * la invocación del mapeo trace2TraceEditor().
```

```
 */
main() {
    qvto.rootObjects()[QvtoTrace::Trace]->map
        trace2TraceEditor();
}

////////////////////////////////////////////////////////////
////////////////////// Mappings ////////////////////////////
////////////////////////////////////////////////////////////

/*
 * trace2TraceEditor() toma una instancia de Trace y retorna
 * una de TraceEditor. Genera la configuración
 * (_configuracion) mediante el mapeo trace2Configuration() y
 * el dashboard mediante trace2Dashboard().
 */
mapping QvtoTrace::Trace::trace2TraceEditor() :
    TraceEditor::TraceEditor {
    _configuration := self.map trace2Configuration();
    dashboard :=  self.map trace2Dashboard();
}

/*
 * trace2Configuration() genera una configuración del editor
 * TraceEditor representada por el modelo de
 * TraceEditor::Configuration en la que se definen los tipos
 * de las trazas y se obtienen los tipos de artefactos que se
 * encuentran en las trazas del modelo QvtoTrace::Trace.
 */
mapping QvtoTrace::Trace::trace2Configuration() :
    TraceEditor::Configuration {
    linkTypes := self.linkTypes();
    typeArtefacts := self.typesByName()->collect(n |
        object TraceEditor::TypeArtefact {
            name := n;
            //TODO: solucionar prototipo de editores para que
            // muestren la descripción si la tiene e
            // inicializarla con alguna información.
            //description := "";
    });
}

/*
 * trace2Dashboard() de las trazas del modelo de entrada
 * QvtoTrace::Trace obtiene el listado de transformaciones
 * (transformations) y los listados de los artefactos origen
 * y destino (sourceArtefacts y targetArtefacts
```

```
 * respectivamente) mediante la inspección de las instancias
 * TraceRecord de dicho modelo. Las trazas que representan
 * los enlaces explícitos (tracelinks) se inicializan como un
 * conjunto vacío.
 */
mapping QvtoTrace::Trace::trace2Dashboard() :
    TraceEditor::Dashboard {
    transformations := self.transformationsByName()->collect(
        t |
        object TraceEditor::Transformation {
            name := t;
            traceLinks := self.traceRecords
                [mappingOperation.name=t]-> map
                    traceRecord2TraceLink();
    });
    // dado que las trazas de QVT son todas producto de
    // transformaciones, la colección traceLinks se
    // inicializa vacía.
    traceLinks := OrderedSet{};
    sourceArtefacts := self.traceRecords._context._context
        [name="self"].map
            varParameterValue2Artefact("Source");
    targetArtefacts := self.traceRecords._result._result.map
        varParameterValue2Artefact("Target");
}

/*
 * traceRecord2TraceLink() convierte una traza TraceRecord en
 * su correspondiente de tipo TraceLink, el nombre (name) es
 * la representación en string de la traza origen, el tipo de
 * la traza es 'implícita',la transformación
 * (_transformation) es del resultado el artefacto que lo
 * contiene, los artefactos origen y destino (sources y
 * targets) son el contexto (_context) y los resultados
 * (result) respectivamente.
 */
mapping QvtoTrace::TraceRecord::traceRecord2TraceLink() :
    TraceEditor::TraceLink {
    name := self.repr().toTraceLinkName();
    type := self.varParameterValueLinkType('Implicit');
    _transformation := result.container().oclAsType(
        TraceEditor::Transformation);
    sources := self._context._context[name="self"].map
        varParameterValue2Artefact("Source");
    targets := self._result._result->map
        varParameterValue2Artefact("Target")->flatten();
}
```

```
/*
 * varParameterValue2Artefact() transforma los parámetros
 * VarParameterValue de las trazas del modelo de entrada en
 * artefactos de tipo Artefact, deriva la transformación en
 * varParameterValueSimple2Artefact() y
 * varParameterValueCollection2Artefact() dependiendo de si
 * el parámetro representa un valor simple o una colección
 * respectivamente.
 */
mapping QvtoTrace::VarParameterValue::
    varParameterValue2Artefact(in prefix : String)
    : OrderedSet(TraceEditor::Artefact)
    disjuncts QvtoTrace::VarParameterValue::
        varParameterValueSimple2Artefact,
        QvtoTrace::VarParameterValue::
            varParameterValueCollection2Artefact {}

mapping QvtoTrace::VarParameterValue::
    varParameterValueSimple2Artefact(in prefix : String)
    : OrderedSet(TraceEditor::Artefact)
    when { self.value.collectionType = null } {
        init {
            result := object TraceEditor::Artefact {
                name := self.value.repr().toArtefactName(
                    prefix);
                type := self.varParameterValueTypeArtefact(
                    self.type.prefix(prefix));
            }->asOrderedSet();
        }
}

mapping QvtoTrace::VarParameterValue::
    varParameterValueCollection2Artefact(in prefix : String)
    : OrderedSet(TraceEditor::Artefact)
    when { self.value.collectionType <> null } {
        init {
            var typeName := self.type.substringAfter("(")
                .substringBefore(")");
            result := self.value.collection->collect(e |
                object TraceEditor::Artefact {
                    name := e.repr().toArtefactName(prefix);
                    type := self
                        .varParameterValueTypeArtefact(
                            typeName.prefix(prefix));
            })->asOrderedSet();
        }
```

```
}

////////////////////////////////////////////////////////////
////////////////////// Constructors ////////////////////////
////////////////////////////////////////////////////////////

constructor TraceEditor::LinkType::LinkType (aName : String,
    aParent : TraceEditor::LinkType) {
    name := aName;
    parent := aParent;
}

////////////////////////////////////////////////////////////
/////////////////////////// Helpers ////////////////////////
////////////////////////////////////////////////////////////

/*
 * linkTypes() crea un conjunto de tipos de enlaces por
 * defecto.
 */
helper QvtoTrace::Trace::linkTypes() : OrderedSet(
    TraceEditor::LinkType) {

    var Explicit := new TraceEditor::LinkType ('Explicit',
        null);
    var Implicit := new TraceEditor::LinkType ('Implicit',
        null);

    var ModelModel := new TraceEditor::LinkType(
        'Model-Model', Explicit);
    var ModelArtefact := new TraceEditor::LinkType (
        'Model-Artefact', Explicit);
    var Query := new TraceEditor::LinkType ('Query',
        Implicit);
    var M2M := new TraceEditor::LinkType ('M2M', Implicit);
    var M2T := new TraceEditor::LinkType ('M2T', Implicit);
    var Composition := new TraceEditor::LinkType (
        'Composition', Implicit);
    var Update := new TraceEditor::LinkType ('Update',
        Implicit);
    var Creation := new TraceEditor::LinkType ('Creation',
        Implicit);
    var Delete := new TraceEditor::LinkType ('Delete',
        Implicit);

    var Static := new TraceEditor::LinkType ('Static',
        ModelModel);
```

```
    var Dinamic := new TraceEditor::LinkType ('Dinamic',
        ModelModel);
    var Satisfies := new TraceEditor::LinkType ('Satisfies',
        ModelArtefact);
    var AllocatedTo := new TraceEditor::LinkType (
        'Allocated-To', ModelArtefact);
    var Performs := new TraceEditor::LinkType ('Performs',
        ModelArtefact);
    var Supports := new TraceEditor::LinkType ('Supports',
        ModelArtefact);
    var Explains := new TraceEditor::LinkType ('Explains',
        ModelArtefact);

    var Dependency := new TraceEditor::LinkType (
        'Dependency', Static);
    var ConsistentWith := new TraceEditor::LinkType (
        'ConsistentWith', Static);
    var Calls := new TraceEditor::LinkType ('Calls',
        Dinamic);
    var Generates := new TraceEditor::LinkType ('Generates',
        Dinamic);
    var Notifies := new TraceEditor::LinkType ('Notifies',
        Dinamic);

    return OrderedSet {
        Implicit, Explicit, ModelModel, ModelArtefact, Query,
        M2M, M2T, Composition, Update, Creation, Delete,
        Static, Dinamic, Satisfies, AllocatedTo, Performs,
        Supports, Explains, Dependency, ConsistentWith,
        Calls, Generates, Notifies
    };
}

/*
 * prefix() agrega el string parámetro pre como prefijo.
 */
helper String::prefix(in pre : String) : String {
    return pre + "_" + self;
}

/*
 * toArtefactName() define el formato del nombre de los
 * artefactos con el string parámetro pre como prefijo y
 * luego tomando sólo la parte del número identificador de lo
 * retornado por la función repr().
 */
helper String::toArtefactName(in pre : String) : String {
```

```
    var temp_name = self.substringAfter(
        "org.eclipse.m2m.internal.qvt.oml.trace.impl.
            EValueImpl@")
        .substringBefore("_(");
    return pre + "_" + temp_name;
}

/*
 * toTraceLinkName() define el formato del nombre de las
 * trazas tomando sólo la parte del número identificador de
 * lo retornado por la función repr() y como prefijo "Trace_"
 */
helper String::toTraceLinkName() : String {
    var temp_name = self.substringAfter(
        "org.eclipse.m2m.internal.qvt.oml.trace.impl.
            TraceRecordImpl@");
    return "Trace_" + temp_name;
}

////////////////////////////////////////////////////////////
////////////////////////// Querys //////////////////////////
////////////////////////////////////////////////////////////

/*
 * varParameterValueTypeArtefact() retorna del modelo ya
 * convertido que representa la configuración, el tipo del
 * artefacto que conforma a TypeArtefact identificado por el
 * parámetro typeName.
 */
query QvtoTrace::VarParameterValue::
    varParameterValueTypeArtefact(in typeName : String)
    : TraceEditor::TypeArtefact {
    return self.container().container().container()
        .oclAsType(QvtoTrace::Trace).resolveoneIn(
            QvtoTrace::Trace::trace2Configuration,
            TraceEditor::Configuration
    ).typeArtefacts![name = typeName];
}

/*
 * varParameterValueLinkType() retorna del modelo ya
 * convertido que representa la configuración, el tipo de
 * traza que conforma a LinkType identificado por el
 * parámetro typeName.
 */
query QvtoTrace::TraceRecord::varParameterValueLinkType(
    in typeName : String)
```

```
    : TraceEditor::LinkType {
    return self.container().oclAsType(QvtoTrace::Trace)
        .resolveoneIn(
            QvtoTrace::Trace::trace2Configuration,
            TraceEditor::Configuration
    ).linkTypes![name = typeName];
}

/*
 * typesByName() retorna el conjunto de nombres de los
 * distintos tipos de artefactos que se encuentran en el
 * conjunto de trazas del modelo de entrada Trace. Este
 * conjunto es el tipo de los orígenes, obtenidos del
 * contexto del modelo Trace, unión los tipos de los
 * resultados, obtenidos de los resultados.
 */
query QvtoTrace::Trace::typesByName() : OrderedSet(String) {
    return self.traceRecords._context._context[name="self"]
        .type.prefix("Source")->asSet()->union(
            self.traceRecords._result._result[name="result"
                and value.collectionType = null].type.prefix(
                    "Target")->asSet()->union(
                        self.traceRecords._result._result[
                            name="result" and
                            value.collectionType <> null]
        .type.substringAfter("(").substringBefore(")")
        .prefix("Target")->asSet())
    )->asOrderedSet();
}

/*
 * transformationsByName() retorna el conjunto de nombres de
 * las transformaciones del conjunto de trazas que se
 * encuentran en el modelo de entrada Trace.
 */
query QvtoTrace::Trace::transformationsByName() :
    OrderedSet(String) {
    return self.traceRecords.mappingOperation.name->
        asOrderedSet();
}
```

Glosario

GEF - Graphical Editing Framework Es un framework Eclipse para el desarrollo de editores gráficos y vistas del Eclipse Workbench UI. 37, 38

JDT - Java Development Tool Es un proyecto que proporciona herramientas tipo plug-ins de Eclipse que asisten en el desarrollo de aplicaciones Java. Incluye la creación de proyectos Java, una perspectiva para el Workbench Eclipse, así como también un conjunto de vistas, editores, asistentes, constructores y herramientas de refactorización y fusionado de código. Estas herramientas transforman a Eclipse en un IDE. 35

MDA - Arquitectura Dirigida por Modelos en inglés Model-Driven Architecture (MDA), es una visión o propuesta particular definida por el OMG de MDD que se basa en el uso de estándares de OMG. VI, 59

MDD - Desarrollo Dirigido por Modelos en inglés Model-Driven Development (MDD), es un paradigma de desarrollo de software que utiliza modelos como artefactos principales del proceso de desarrollo. Por lo general, la implementación se genera o deriva automáticamente a partir de los modelos. VI, 1, 2, 58

MDE - Ingeniería de Software Dirigida por Modelos en inglés Model-Driven Engineering (MDE), es un paradigma de ingeniería de software en el cual los modelos juegan el rol principal en todas las actividades del ciclo de vida de la misma. VI, VII, 8, 9, 13, 40, 58

MOF - MetaObject Facility es un lenguaje estándar del OMG para MDE, en particular es un meta-meta lenguaje que permite definir meta-modelos en la capa M2, como por ejemplo el meta-modelo Lenguaje Unificado de Modelado (*UML* de Unified Modeling Language) que describe al lenguaje Lenguaje Unificado de Modelado (*UML* de Unified Modeling

Language). Como su nombre lo indica, MOF se basa en el paradigma Orientado a Objetos. 14, 41, 43, 59

OCL - Object Constraint Language es un lenguaje declarativo para describir reglas que se aplican a meta-modelos MOF y a los modelos Lenguaje Unificado de Modelado (*UML* de Unified Modeling Language). Fue desarrollado por IBM y en la actualidad es parte del estándar Lenguaje Unificado de Modelado (*UML* de Unified Modeling Language). OCL es un lenguaje de texto que permite definir restricciones y consultas sobre expresiones de objetos de cualquier modelo o meta-modelo MOF. OCL es componente clave del estándar OMG para la transformación de los modelos QVT. Otros lenguajes de transformación de modelos como ATL también están construidos utilizando OCL. 7

OMG - Object Management Group Consorcio internacional de la industrial informática sin fines de lucro y de membresía abierta responsable de la definición de varios estándares de modelado como Lenguaje Unificado de Modelado (*UML* de Unified Modeling Language), MetaObject Facility (*MOF*) y MDA. VI, 42, 58, 59

UML - Unified Modeling Language es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad y es también el estándar oficial, respaldado por el OMG. 5, 8, 10, 14, 37, 46, 47, 49, 58, 59

Siglas

ALM Application Lifecycle Management. 32

API Interfaz de programación de aplicaciones o Application Programming Interface. 33, 37, 44, 49

ATL Atlas Transformation Language. 28, 30, 40–42, 59

CASE Computer Aided Software Engineering. 49

EMF Eclipse Modeling Framework. 7, 14, 34–38, 44, 52

EPL Eclipse Public License. 32

EVL Epsilon Validation Language. 7

GMF Graphical Modeling Framework. 37–40

GUI Interfaz de Usuario Gráfica o Graphical User Interface. 35

IDE Entornos de Desarrollo Integrados o Integrated Development Environment. 17, 24, 25, 32, 40, 58

IEEE Institute of Electrical and Electronics Engineers. 1

MMT Model-to-Model Transformation. 41, 42

QVT Query/View/Transformation. 28, 42, 43, 49, 59

RCP Rich Client Platform. 32

RIA Rich Internet Applications. 32

SOA Service Oriented Architecture. 32

SPL Línea de Producto de Software o Software Product Line. 45, 46

SWT Standard Widget Toolkit. 33

TEAP Traceability Elicitation and Analysis Process. 8

XMI XML Metadata Interchange. 34, 35, 37, 45

Bibliografía

[1] IEEE Standard Glossary of Software Engineering Terminology. Number Std 610.12-1990, IEEE (1990).

[2] R. Brcina and M. Riebisch: Defining a Traceability Link Semantics for Design Decision Support. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[3] B. Grammel and K. Voigt: Foundations for a Generic Traceability Framework in Model-Driven Software Engineering. In: ECMDA Traceability Workshop (ECMDA-TW) 2009 Proceedings.

[4] Glossary of Center of Excellence for Software Traceability (CoEST) http://www.coest.org/index.php/traceability/glossary.

[5] Center of Excellence for Traceability - Problem Statements and Grand Challenges. In: Center of Excellence of Traceability Technical Report (COET-GCT-06-01-0.9) September 10, 2006.

[6] Gotel, O.C.Z., Finkelstein, A.C.W., "An Analysis of the Requirements Traceability Problem", International Conference on Requirements Engineering, ICRE'94, Los Alamitos, California, Abril, 1994, pp 94-101.

[7] N. Drivalos, R. F. Paige, K. J. Fernandes, D. S. Kolovos: Towards Rigorously Defined Model-to-Model Traceability. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[8] F. Glitia, A. Etien and C. Dumoulin: Fine Grained Traceability for an MDE Approach of Embedded System Conception. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[9] F. Jouault: Loosely Coupled Traceability for ATL, In: Proceedings of the European Conference on MDA Traceability Workshop, Nurnberg, Germany (2005).

[10] R. Paige, G. Olsen, D. Kolovos, S. Zschaler, C. Power: Building Model-Driven Engineering Traceability Classifications, In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[11] F. Klar, S. Rose, A. Schurr: TiE - A Tool Integration Environment, In: ECMDA Traceability Workshop (ECMDA-TW) 2009 Proceedings.

[12] S. B. Abid, G. Botterweck: Resolving Product Derivation Tasks using Traceability in Software Product Lines, en: ECMDA Traceability Workshop (ECMDA-TW) 2009 Proceedings.

[13] B. Grammel, S. Kastenholz: A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development, en: Proceedings of the 6°ECMFA Traceability Workshop (ECMFA-TW), 15 de junio de 2010, Paris, Francia.

[14] B. Amar, H. Leblanc, B. Coulette: A Traceability Engine Dedicated to Model Transformation for Software Engineering. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[15] Eclipse Modeling Framework Project (EMF) http://www.eclipse.org/modeling/emf/.

[16] A. Sousa, U. Kulesza, A. Rummler, N. Anquetil, R. Mitschke, A. Moreira, V. Amaral, J. Araújo: A Model-Driven Traceability Framework to Software Product Line Development. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceedings.

[17] Eclipse Project http://www.eclipse.org.

[18] Eclipse Platform Technical Overview. Object Technology International, Inc., February 2003, http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[19] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose: Eclipse Modeling Framework - A Developer's Guide. Addison Wesley.

[20] Graphical Modeling Project http://www.eclipse.org/modeling/gmp/.

[21] Frederic Plante, IBM: Introducing the GMF Runtime, January 2006 http://www.eclipse.org/articles/Article-Introducing-GMF/article.html

[22] Graphical Modeling Framework - Tutorial - Part 1 http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1

[23] ATL Eclipse Project http://www.eclipse.org/atl/.

[24] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1, January 2011 - Version 1.0, April 2008. http://www.omg.org/spec/QVT/index.htm.

[25] C. Pons, R. Giandini y G. Pérez: Desarrollo de Software Dirigido por Modelos. Conceptos teóricos y su aplicación práctica. Facultad de Informática, Universidad Nacional de La Plata. Octubre de 2008.

[26] Pierre F. Tiako: Designing Software-Intensive Systems, Methods and Principles. Langston University, USA. 2008.