PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 09-19

Estado del Arte de Lenguajes y Herramientas de Transformación de Modelos

Horacio López, Fernando Varesi, Marcelo Viñolo,
Daniel Calegari, Carlos Luna

2009

López, Horacio; Varesi, Fernando; Viñolo, Marcelo; Calegari, Daniel; Luna, Carlos
Estado del arte de lenguajes y herramientas de transformación de modelos
ISSN 0797-6410
Reporte Técnico RT 09-19
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, 2009

Estado del Arte de Lenguajes y Herramientas de Transformación de Modelos

Horacio López, Fernando Varesi, Marcelo Viñolo, Daniel Calegari, Carlos Luna
Instituto de Computación, Facultad de Ingeniería, UDeLaR

Noviembre 2009

Resumen

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) es un enfoque de ingeniería de software basado en el modelado de un sistema como la principal actividad del desarrollo. La construcción del sistema es guiada por transformaciones de dichos modelos, terminando en la generación automática de código. Este documento presenta un relevamiento del estado del arte de lenguajes y herramientas empleados para MDD. Se describen las principales características que distinguen a los lenguajes y a las herramientas dentro de un marco teórico basado en el análisis de dominio de los diferentes enfoques de transformación de modelos. Por último, se presentan y analizan los lenguajes más populares que posibilitan transformaciones de modelos, así como las herramientas que los implementan.

Índice

1. Introducción

El Desarrollo de Software Guiado por Modelos (Model-Driven Development, MDD) es un enfoque de ingeniería de software basado en el modelado de un sistema como la principal actividad del desarrollo. Uno de los objetivos de este enfoque es posiblitar la resolución de problemas de integración mediante la separación de la especificación de interoperabilidad, de las plataformas tecnológicas que la implementan. La utilización de modelos independientes de plataforma (PIM) es fundamental para llevar a cabo este objetivo [33].

La construcción de los sistemas en este paradigma es guiada por transformaciones de dichos modelos, terminando en la generación automática de código. Su éxito depende fuertemente de la disponibilidad de lenguajes y herramientas apropiados para realizar las transformaciones entre modelos y validar la corrección en cada caso.

La mayoría de las técnicas de MDD utilizan el Unified Modeling Language (UML) como lenguaje de modelado, considerado el estándar de facto a nivel académico e industrial. La validación de una transformación incluye la verificación de la corrección sintáctica de la misma y de los modelos producidos, así como la corrección semántica de la transformación, es decir, la preservación de la corrección del modelo resultante con respecto al correspondiente modelo original. Muy pocas propuestas se ocupan de este último punto. Adicionalmente, la mayoría de las técnicas de transformación se enfocan en la transformación de la estructura, dejando de lado su comportamiento.

El InCo-UdelaR (Instituto de Computación-Universidad de la República) presenta en el 2009 un proyecto cuyo objetivo principal, enmarcado en un proyecto de investigación más general, es realizar un estudio del estado del arte sobre lenguajes y herramientas existentes para la especificación y verificación de transformaciones de modelos de software, en particular de modelos de comportamiento. Se pretende contribuir a la especificación formal de transformaciones de modelos especificados en UML, con el objetivo de hacer posible la verificación de la corrección sintáctica y semántica de dichas transformaciones. Se propone el desarrollo y análisis completo de un caso de estudio como parte del trabajo.

Este documento constituye el primer paso hacia el objetivo del proyecto. El mismo resume el conocimiento adquirido sobre el estado del arte de los lenguajes y las herramientas de transformación de modelos relevados en la primera fase del proyecto.

En la sección 2 de este documento se detallan los criterios empleados para analizar los diferentes enfoques de transformaciones. Se introducen diferentes características de los enfoques presentadas en la bibliografía y se complementan con criterios para analizar las herramientas que implementan dichos enfoques.

A continuación, la sección 3 presenta el caso de estudio con el que se ejemplificarán los enfoques en este documento. Este caso de estudio es el más común en el ámbito de MDA y consiste en la transformación de un modelo de clases UML a un modelo relacional [11]. Este ejemplo de transformación fue seleccionado para la aceptación de los trabajos publicados en un workshop realizado en 2005 dedicado al tratamiento de las cuestiones prácticas que se presentan en la transformación de modelos. En la referencia citada se detallan tanto el caso de estudio como los trabajos presentados.

En la sección 4 se analizan los enfoques relevados, agrupados en categorías generales, empleando los criterios considerados en la sección 2. Para cada enfoque se presenta, cuando este lo permite, un ejemplo de transformación relacionado al caso de estudio introducido en la sección 3.

Finalmente, la sección 5 exhibe las conclusiones de este trabajo.

2. Criterios para analizar enfoques de transformación de modelos

Los criterios analizados en esta sección se basan en [16] y [37]. Toda la bibliografía relacionada a la clasificación y el análisis de enfoques de transformación se basan fuertemente en estas publicaciones. Estos trabajos constituyen el framework más genérico desarrollado hasta el momento para analizar las

transformaciones sin profundizar en un área de aplicación específica. Las imágenes presentadas en esta sección pertenecen a dichos trabajos.

Tomaremos la definición de transformación de modelos planteada en [34], en donde se define una transformación como la generación automática de un modelo destino desde un modelo origen, de acuerdo a una definición de transformación. Una definición de transformación es un conjunto de reglas de transformación que juntas describen cómo un modelo en el lenguaje origen puede ser transformado en un modelo en el lenguaje destino. Una regla de transformación es una descripción de cómo una o más construcciones en el lenguaje fuente pueden ser transformadas en una o más construcciones en el lenguaje destino.

Los trabajos citados en el inicio de esta sección refieren en su análisis a aproximaciones o enfoques de transformación en lugar de considerar lenguajes de transformación. El primer motivo de esta distinción es que el área de investigación es relativamente reciente, por lo que aún se trabaja activamente en la conformación de estándares y los distintos trabajos realizados en el área apuntan en diferentes direcciones. El segundo motivo es que algunos de estos enfoques no implican necesariamente la utilización de un lenguaje de transformación, como se verá más adelante.

El propio concepto de lenguaje de transformación de modelos no cuenta aún con una definición académica consensuada. En nuestro trabajo, aproximamos la definición de lenguaje de transformación como aquel que permite especificar metamodelos para las instancias de los modelos de origen y destino de la transformación utilizando una representación formal. Además, debe permitir definir reglas de transformación que serán ejecutadas por un motor.

Los elementos básicos de toda transformación de modelos se pueden apreciar en la Figura 1. Aquí se puede ver como una transformación se define en base a metamodelos de origen (Source Metamodel) y destino (Target Metamodel). Esta definición de transformación es tomada por el motor de transformaciones (Transformation Engine) el cual genera un modelo destino (instancia de Target Metamodel) a partir del modelo origen (instancia de Source Metamodel). Este esquema se puede extender naturalmente para transformaciones que operan sobre varios metamodelos de entrada y salida.

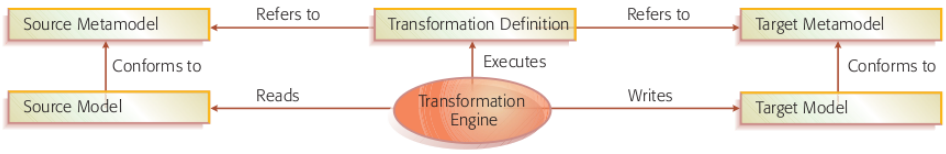


Figura 1: Conceptos básicos de transformación de modelos

2.1. Características de los enfoques de transformación

En [16] se aplica análisis de dominio sobre los diferentes enfoques de transformación de modelos. De esta forma se obtuvo un diagrama de características (Figura 2), que refleja los aspectos comunes y variables que caracterizan al conjunto de instancias de un concepto (en este caso, aproximaciones a la transformación de modelos).

El diagrama presentado define la cardinalidad de los conceptos mediante el extremo inferior de los conectores. Un círculo oscuro especifica que el atributo especificado es parte indispensable del enfoque analizado, mientras que un círculo claro indica que el atributo se presenta en forma opcional. Si el extremo inferior es un cuadrado, indica que elemento inferior corresponde a una agrupación de atributos. La relación entre los atributos de un mismo nivel esta dada por el arco formado por los distintos conectores: un arco sombreado indicará que los atributos no son excluyentes entre sí (permitiendo enfoques híbridos), mientras que un arco no sombreado indica que solo uno de los atributos inferiores podrá estar presente.

Extenderemos y usaremos este diagrama de características como framework para analizar algunos de los principales lenguajes de transformación. Aquellas características marcadas como opcionales no estarán

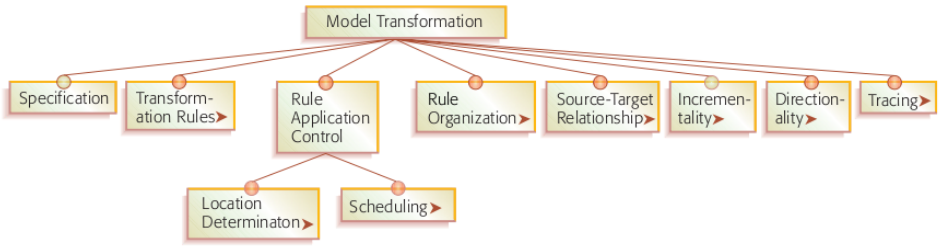


Figura 2: Diagrama principal de características

necesariamente presentes en los lenguajes a analizar.

Las siguientes subsecciones detallan los aspectos más relevantes de estas características generales.

2.1.1. Especificación (opcional)

Esta característica refiere a los mecanismos de especificación provistos para definir la transformación como podrían ser precondiciones y postcondiciones expresadas en OCL. Por lo general estas especificaciones describen relaciones y no son ejecutables.

Resulta de interés conocer si es posible especificar transformaciones de alto nivel (transformaciones sobre transformaciones) o si la especificación posee propiedades matemáticas que faciliten algún tipo de prueba.

2.1.2. Reglas de transformación

Una *regla de transformación* es la mínima unidad de transformación. El ejemplo más común son reglas con un lado izquierdo (LHS) y derecho (RHS). Tambien son reglas de transformación las funciones o los procedimientos que implementan un paso de la transformación. En sección 4.1.2 de este documento se presentarán los templates, que pueden considerarse una degeneración de una regla de transformación.

A continuación se describen las características de las reglas de transformación ilustradas en la Figura 3:

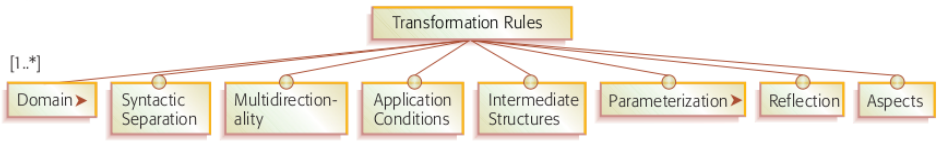


Figura 3: Diagrama de características de las reglas de transformación

- Las reglas pueden tomar como entrada uno o más *dominios*. Los modelos que se pueden describir están determinados por el lenguaje asociado a la especificación de las reglas. Los dominios en una transformación pueden ser de entrada y/o salida. Cuando los dominios de entrada y salida son descriptos por un mismo metamodelo hablamos de *transformaciones endógenas*. En caso contrario se trata de *transformaciones exógenas*. La determinación del tipo de dominio (entrada y/o salida) es estática y opcionalmente dinámica. El dominio debe contar además con metavariables (contenedores de elementos del modelo) y opcionalmente con patterns (segmentos con una o más variables) y expresiones lógicas (ejecutables o no).

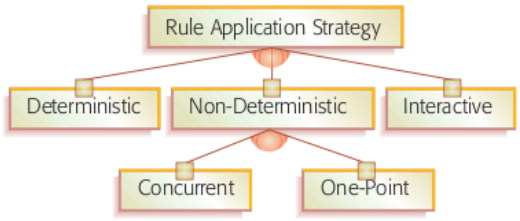


Figura 4: Determinación de ubicación

- En algunos enfoques puede existir una *separación sintáctica* de las reglas (ej: lado izquierdo y derecho).
- Las reglas pueden ser *multidireccionales*, es decir pueden ejecutarse en diferentes direcciones.
- En algunos casos pueden definirse *precondiciones* (o guardas) para aplicar la regla (ej: when en QVT [38])
- La ejecución de una regla podrá hacer uso de *estructuras intermedias* que no forman parte de los modelos de origen y destino. Un ejemplo de este tipo de estructuras son los traceability links, que a diferencia de la mayoría de las estructuras intermedias son persistidos luego de la transformación.
- En algunos enfoques es posible la *parametrización* de las reglas, volviéndolas más reusables.
- Otra característica opcional es el soporte para *reflection* y *aspects*. En particular, reflection permite una mayor navegabilidad de las propiedades de los elementos a transformar y potencialmente también de las reglas que los afectan. La utilización de aspects brinda los beneficios de mayor modularidad y facilidad para separar distintas visiones sobre las transformaciones.

2.1.3. Aplicación de las reglas

Esta característica busca analizar los mecanismos brindados para especificar la selección y ejecución de reglas y determinar los elementos del modelo origen afectados. A continuación se profundizará en cada uno de estos aspectos.

- La *determinación de ubicación* (Rule Application Strategy en la Figura 4) consiste en determinar los lugares del modelo en donde las reglas deben aplicarse. La estrategia para realizar esta tarea puede ser determinista (ej: recorrer una representación interna), no determinista (ej: aplicación concurrente en todos los elementos que cumplan con cierto criterio) o interactiva con el usuario.

- La *planificación* (Figura 5) determina el orden en que son aplicadas las reglas. Los diferentes mecanismos de planificación pueden ser agrupados en las siguientes categorías:

  - La *forma* de la planificación es implícita si el usuario no tiene ningún tipo de control explícito sobre el algoritmo de planificación de la herramienta. La única forma de incidir es diseñando patrones y lógica para asegurar cierto orden (ej: chequear precondiciones para ejecutar una regla). En cambio, es explícita si existen construcciones dedicadas para especificar el orden de ejecución, las cuales pueden ser externas, separadas de la lógica de las reglas, o internas.
  - La *selección de las reglas* a ejecutar puede especificarse por una condición explícita, por una condición no determinista, por un mecanismo de resolución de conflictos o mediante interacción con el usuario.

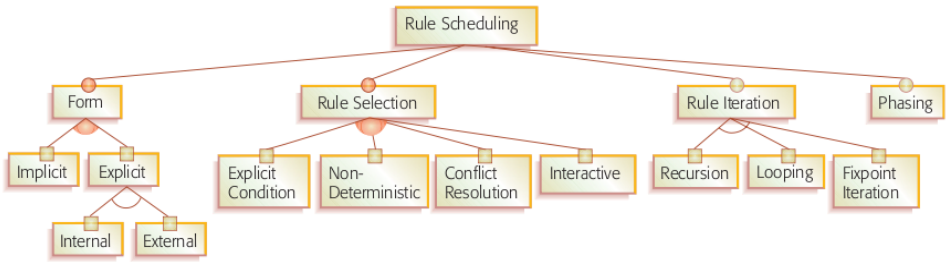


Figura 5: Planificación

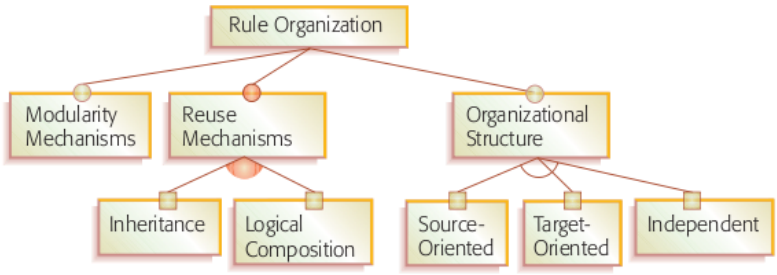


Figura 6: Organización de reglas

- La *iteración de reglas* provista por el enfoque puede contener recursión, ciclos o FIXPOINT ITERATIONS (la regla se aplica repetidamente hasta no detectar cambios).
- Las reglas podrían organizarse en fases. Estas tienen un propósito específico y sólo ciertas reglas pueden ejecutarse en cada una de ellas.

2.1.4. Organización de Reglas

Este ítem comprende los aspectos de composición y estructuración, como modularización y reuso. Se proponen tres áreas de variación (Figura 6):

- mecanismos de modularización: las reglas se agrupan en módulos que pueden importar a otros módulos para acceder a su contenido.
- mecanismos de reuso: definición de una regla basada en otras reglas, utilizando mecanismos de herencia entre reglas y/o módulos y composición lógica de las mismas.
- estructura organizacional: las reglas pueden organizarse según la estructura del lenguaje de origen, del lenguaje destino o usando una organización independiente de ambos.

2.1.5. Relación origen-destino

Este ítem describe la relación entre los modelos de origen y destino de la transformación (Figura 7).

En algunos enfoques (ej: ATL [10]) se debe crear un nuevo modelo objetivo que debe ser diferente del modelo origen. Sin importar si los metamodelos de origen y destino coinciden, el motor de transformaciones trabaja con instancias diferentes de los mismos.

En otros enfoques (ej: VIATRA [49]) solo se permite actualizaciones *in-place*: el modelo origen y destino deben ser una misma instancia. Las actualizaciones podrían ser destructivas o únicamente permitir extensiones del modelo.

También existen enfoques (ej: QVT) que permiten tanto actualizaciones al modelo origen como la creación de nuevos modelos.

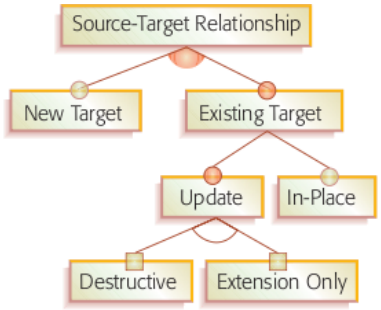


Figura 7: Relación entre modelo origen y modelo destino

2.1.6. Incrementalidad (opcional)

La incrementalidad (Figura 8) puede ser descrita por tres características:

- *Incrementalidad del objetivo* (o propagación de cambios): Capacidad de actualizar los modelos objetivos existentes teniendo en cuenta cambios en los modelos de origen.
- *Incrementalidad del origen*: Minimización de la cantidad de información del modelo origen que debe ser reanalizada tras una modificación en el mismo.
- *Preservación de los cambios de usuario en el objetivo*: capacidad de cambiar el objetivo en base a cambios del origen, pero preservando cambios manuales del usuario en el destino.

2.1.7. Direccionalidad

Se refiere a si las transformaciones pueden ser ejecutadas en forma unidireccional o multidireccional. La multidireccionalidad puede ser lograda mediante reglas multidireccionales o definiendo reglas unidireccionales complementarias.

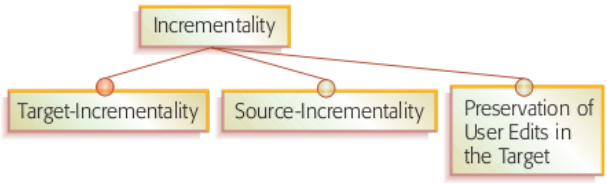


Figura 8: Incrementalidad

2.1.8. Trazabilidad

Se refiere a los mecanismos para registrar la ejecución de la transformación. La forma más común de realizarlo es mediante traceability links, que conectan elementos del modelo origen y de destino. Esta información puede ser útil para realizar un análisis de impacto y para realizar debugging. Algunos enfoques proveen un mecanismo dedicado para trazabilidad. Aunque este no existiera, podría ser creado como un elemento más en el modelo destino.

2.2. Características de las herramientas de transformación

En esta sección complementamos las características para categorizar enfoques de transformación con [37] y algunos criterios propios. Estas características están relacionadas con las herramientas que implementan los diferentes enfoques para transformación de modelos. A continuación describiremos brevemente aquellas que consideramos de interés para analizar una herramienta:

- *Areas de aplicación*: esta característica pretende describir la clase de problemas que la herramienta puede resolver, los casos de estudio en que fue aplicada y las aplicaciones industriales.
- *Sugerencia de aplicación de transformaciones*: se relaciona con la interactividad con el usuario descrita en la sección previa y describe en que forma la herramienta asiste al usuario para descubrir aquellas reglas de transformación que pueden ser de interés en una determinado modelo a transformar.
- *Tipo de licencia de distribución*: la herramienta puede ser software propietario o distribuírse con algún tipo de licencia de código abierto.
- *Historia de la herramienta*: comprende el origen de la misma (por ejemplo: académico o emprendimiento privado), su antigüedad y la continuidad de su soporte.
- *Soporte brindado al usuario*: refiere al tipo de documentación brindada: manuales, tutoriales, ejemplos, foros y todo tipo de soporte disponible.
- *Comunidad de usuarios y aceptación*: refiere a la cantidad de usuarios activos de la herramienta y la aceptación obtenida por la misma dentro de la comunidad.
- *Mecanismos ofrecidos para garantizar correctitud de las transformaciones*: refiere a las garantías ofrecidas por la herramienta de que el resultado producido es correcto sintáctica y semánticamente.
- *Manejo de modelos incompletos o inconsistentes*: refiere a como maneja la herramienta situaciones en las cuales el modelo no está especificado por completo o contiene inconsistencias.
- *Mecanismos ofrecidos para verificación y validación de transformaciones*: a diferencia de la correctitud, esto se refiere a los mecanismos ofrecidos para verificar que una transformación desarrollada presenta el comportamiento deseado por el programador. Un ejemplo de esto podría ser asistir al usuario en la generación de casos de prueba.
- *Escalabilidad de transformaciones y de modelos*: refiere a la capacidad de la herramienta para manejar complejidad y tamaño creciente ya sea en las transformaciones o en los modelos a transformar.

3. Caso de estudio: transformación de modelo de clases UML a modelo relacional

A continuación se describirá en forma general, el caso de estudio de transformación de modelo de clases UML a modelo relacional. Este caso de estudio es el ejemplo más utilizado para presentar las características de un lenguaje de transformación. A pesar de su relativa simplicidad, este ejemplo permite mostrar las características más relevantes de las transformaciones de modelos [11].

Para fijar ideas, se presentarán ejemplos de metamodelos origen y destino para dicha transformación, y luego se mostrarán ejemplos de instancias de estos metamodelos.

En la Figura 9 se presenta un metamodelo que representa los diagramas de clases UML a transformar.

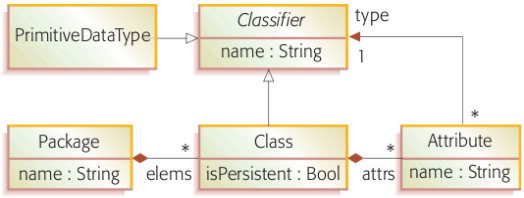


Figura 9: Metamodelo de clases UML origen

De acuerdo con este metamodelo las clases (*Class)* poseen un nombre (atributo *name*) y atributos (asociación con la clase *Attribute*). Las mismas pueden ser declaradas como persistentes o no persistentes (atributo *isPersistent*). Los atributos poseen un clasificador (*Classifier*) que corresponde al tipo de los mismos, el cual puede ser un tipo primitivo (*PrimitiveDataType*) o una clase. Las clases por lo tanto pueden estar relacionadas a través de sus atributos, las llamadas asociaciones en UML. En la figura 10 se puede observar una instancia del metamodelo origen.

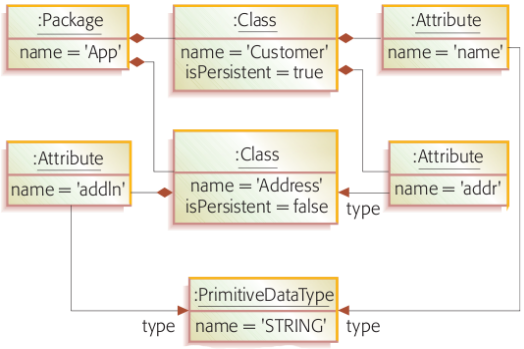


Figura 10: Modelo de clases de UML

El modelo destino respetará un metamodelo que describa a los modelos relacionales. Dicho metamodelo puede observarse en la Figura 11.

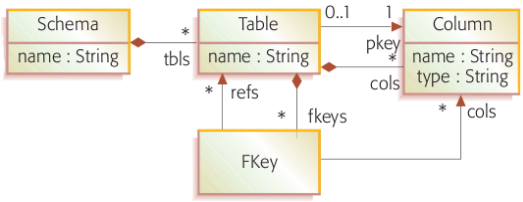


Figura 11: Metamodelo relacional destino

Todo esquema destino contará con un número de tablas. Cada tabla está compuesta de columnas (asociación con *Column*), de las cuales alguna será la clave primaria de la misma. Una tabla puede estar asociada a cero o más de una claves foráneas (foreign key). Cada clave foránea puede referirse a un conjunto de columnas de una tabla que constituyen la clave.

Luego de haber presentado los metamodelos que intervendran en la transformación, veamos algunas de las tareas básicas que deberá realizar dicha transformación:

- Las clases marcadas como persistentes serán transformadas en tablas
- Las columnas de las tablas serán derivadas a partir de los atributos y las asociaciones de las clases
- Cada atributo de un tipo primitivo es transformado a una columna. Si el atributo considerado el primario, la columna resultado será clave primaria de la tabla
- Cada atributo cuyo tipo no sea primitivo, se resolverá y se crearán las estructuras necesarias en el modelo destino que lo representen.

En la Figura 12 puede observarse el modelo resultante tras aplicar la transformación al modelo de la Figura 10.

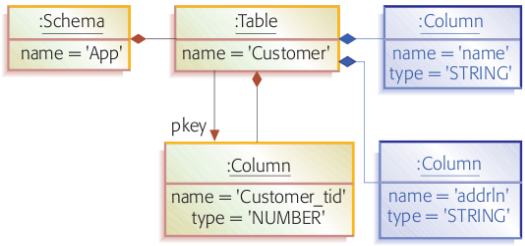


Figura 12: Modelo relacional destino

Si el lector desea profundizar en los detalles del ejemplo o en un ejemplo completo de trasformación para el caso de estudio presentado, puede consultar [11] y [32].

4. Estado del arte: lenguajes y herramientas

Las siguientes categorías para enfoques de transformación están basadas en los trabajos mencionados en la Sección 2. Es importante señalar que agrupan enfoques de transformación que no necesariamente son lenguajes de transformación de modelos.

En [16] se realiza una primer distinción entre transformaciones modelo-modelo y modelo-texto. Mientras que las primeras producen una instancia del metamodelo objetivo, las segundas producen solamente cadenas de texto. Las transformaciones texto-modelo comprenden tareas de parsing y tecnologías de ingeniería inversa.

A continuación se detallarán las diferentes categorías mencionando los aspectos más relevantes de la mismas. Para cada categoría se presentarán ejemplos de lenguajes junto con algunas de las herramientas que implementan dichos lenguajes. Este documento no pretende brindar una presentación exhaustiva de todos los enfoques existentes, sino una referencia de los principales enfoques empleados en la actualidad.

4.1. Aproximaciones Modelo-Texto (M2T)

Los enfoques M2T no son de especial interés en el contexto del proyecto, pero dada su aplicación en problemas reales serán brevemente mencionados en esta sección.

4.1.1. Enfoques visitor-based

En este enfoque se utiliza un mecanismo de visita para recorrer la representación interna del modelo y escribir la salida en el texto correspondiente. Como ejemplo de una aplicación de este enfoque se presentará Stratego/XT.

Stratego/XT provee un lenguaje (Stratego) y un kit de herramientas (XT) para transformación de programas[1]. Algunas aplicaciones de transformación de programas con Stratego son la compilación, la generación de documentación, el análisis de código y la ingeniería inversa [50].

El lenguaje Stratego asume que los programas se pueden representar como árboles sintácticos abstractos. Para esto se emplean parsers que los generan a partir de una definición formal de la sintaxis de los mismos.

Para definir la transformación se definen reglas para transformar los términos de los árboles. Dichas reglas de reescritura de términos tienen un lado izquierdo y un lado derecho. Se deben definir además estrategias para controlar la aplicación de reglas. Una estrategia es un plan para lograr una transformación aplicando un conjunto de reglas.

Herramientas

El conjunto de herramientas XT se distribuye bajo la licencia GNU LGPL. Este conjunto contiene los parsers mencionados y el motor para realizar las transformaciones definidas por las reglas.

Esta herramienta se encuentra ampliamente documentada en el sitio oficial [4] con ejemplos de su utilización.

4.1.2. Enfoques template-based

Un *template* consiste en un texto objetivo que contiene segmentos de metacódigo. El metacódigo accede a información del modelo origen, realiza una selección de código y una expansión iterativa. Se puede profundizar este enfoque en [14].

El lado izquierdo de las reglas de transformación ejecuta la lógica para acceder al modelo origen, mientras que el lado derecho combina strings no tipados con lógica ejecutable para realizar la selección

[1] Por programa se entiende un objeto estructurado con semántica. Es precisamente la estructura la que permite realizar transformaciones sobre el mismo.

de código y la expansión iterativa. No existe una separación sintáctica entre el lado izquierdo y derecho. Los templates pueden ser considerados como una degeneración de las reglas de transformación.

La lógica del lado izquierdo puede ser código (ej: código Java) que accede a una API (ej: JMI) provista por la representación interna del modelo a acceder.

En comparación con las transformaciones visitor-based (sección 4.1.1), la estructura de los templates se asemeja más al código a ser generado que al código de origen. Se tiende a un proceso de desarrollo iterativo al facilitar la derivación de templates en base a ejemplos. Dado que se está trabajando con texto, los patrones del template son no tipados y pueden producir errores sintácticos y/o semánticos en el texto generado. Sin embargo, esta independencia entre los templates y el lenguaje objetivo permite generar diferentes artefactos textuales (ej: documentación).

Este enfoque esta implementado por la mayoría de las herramientas MDA como AndroMDA, OptimalJ, JET y otros.

4.2. Aproximaciones Modelo a Modelo (M2M)

Este es el conjunto de enfoques más expresivo para la transformación de modelos. Todo modelo que pueda ser descrito por un metamodelo puede utilizarse como entrada a una transformación. Incluso las transformaciones pueden ser expresadas como modelos que pueden ser transformados para generar nuevas transformaciones.

A continuación se describirán las diferentes categorías de aproximaciones para realizar transformaciones de modelo a modelo, presentando algunos lenguajes de ejemplos y las herramientas más importantes de los mismos.

4.2.1. Enfoques de Manipulación Directa

Esta aproximación ofrece una representación interna de los modelos y una API para utilizarla. Las implementaciones que aplican este enfoque son usualmente desarrolladas como un framework orientado a objetos, el cual puede proveer infraestructura para organizar las transformaciones.

SiTra es un ejemplo muy sencillo y básico de un framework que emplea el enfoque de manipulación directa. Fue desarrollado por la University of Birmingham [40] y se enfoca en el desarrollo de transformaciones simples en Java o C#.

Básicamente, SiTra define una estructura de clases para escribir transformaciones en Java (o C#). Provee también un pequeño framework para la ejecución de dichas transformaciones. La librería consiste en dos interfaces y una clase que implementa un algoritmo de transformación. Los metamodelos de origen y destino se podrán definir mediante clases Java. En otras palabras, todo el metamodelo debe poder representarse en el paradigma de OO.

En [9] se muestra un ejemplo utilizando este framework.

4.2.2. Enfoques Operacionales

Este enfoque es similar al enfoque de manipulación directa (sección 4.2.1), ofrece soporte dedicado a transformacion de modelos. Una típica solucion que utiliza este enfoque, es proveer la extensión del formalismo de metamodelado para agregar facilidades para expresar computaciones.

Kermeta (Kernel Metamodeling) es un lenguaje de metamodelado clasificado en la categoría operacional. Es un proyecto de IRISA (Institut National de Recherche en Informatique et en Automatique) y la Universidad de Rennes que comenzó en 2005. Es un lenguaje definido con el propósito de servir a todas las posibles manipulaciones de modelos. Ha sido diseñado para permitir tanto la definición de

metamodelos y modelos como la definición de consultas, vistas, transformaciones y acciones sobre los modelos [44].

Kermeta fue diseñado para ser compatible con el lenguaje de metamodelado EMOF[2] y con Ecore[3] para Eclipse. Dentro de sus características principales, Kermeta es un lenguaje imperativo con soporte para las estructuras de control tradicionales. Es orientado a objetos con soporte para paquetes, clases, operaciones y métodos, herencia múltiple y binding dinámico. Estos rasgos pueden ser utilizados para encapsular transformaciones permitiendo la modularización de las mismas. La composición de transformaciones puede ser realizada por medio de llamadas a operaciones o sobrecarga de métodos.

Las reglas de transformación en Kermeta pueden ser interpretadas como las operaciones, ya que el comportamiento de la transformación es especificado por las mismas. La *planificación* de reglas es afectada por la naturaleza imperativa del lenguaje y es creada por el desarrollador, por lo tanto es explícita. No existe ningún mecanismo para *selección de reglas*. Sin embargo las reglas pueden ser *recursivas*, utilizando recursión sobre funciones.

Es un lenguaje orientado a modelos y aspectos, que permite entrelazar elementos provenientes de varias fuentes (Ecore, Kermeta, OCL, etc) para crear herramientas sobre metamodelos existentes y asi extenderlos. Permite diseño por contratos de software, definiendo pre y post condiciones en las operaciones e invariantes sobre las clases. Es un lenguaje fuertemente tipado y soporta expresiones lambda de primera clase, que resultan útiles para especificar funciones OCL. Además brinda soporte para *aspects* y *reflection* [25].

Kermeta es un lenguaje robusto con manejo de errores. Al ser estáticamente tipado, se puede chequear la correctitud del código en tiempo de compilación. Para comportamientos inesperados en tiempo de ejecución el lenguaje provee manejo de excepciones [39].

En las transformaciones de modelos es de interés mantener mapeos entre objetos del modelo origen y destino. Kermeta no cuenta con soporte dedicado para trazas, pero brinda facilidades de diseño de soluciones reusables para manejar información de trazas. Tampoco cuenta con soporte para incrementalidad ni multidireccionalidad por defecto, pero ambos pueden ser simulados [51].

Para ejemplificar la sintaxis del lenguaje, mostraremos un fragmento del ejemplo de transformaciones de diagrama de clases de UML a esquema relacional. Para su mayor compresión el lector podrá consultar [39]. Las figuras 13 y 14 muestran el metamodelo del esquema relacional expresado en texto Kermeta y el código de la transformación respectivamente.

[2]EMOF (Essential MOF) es un subconjunto de MOF que adquiere las facilidades de los lenguajes de programación orientados a objetos y XML. Reutiliza la infraestructura de UML2.

[3]Ecore es un metamodelo que permite describir modelos y forma parte del framework EMF [29].

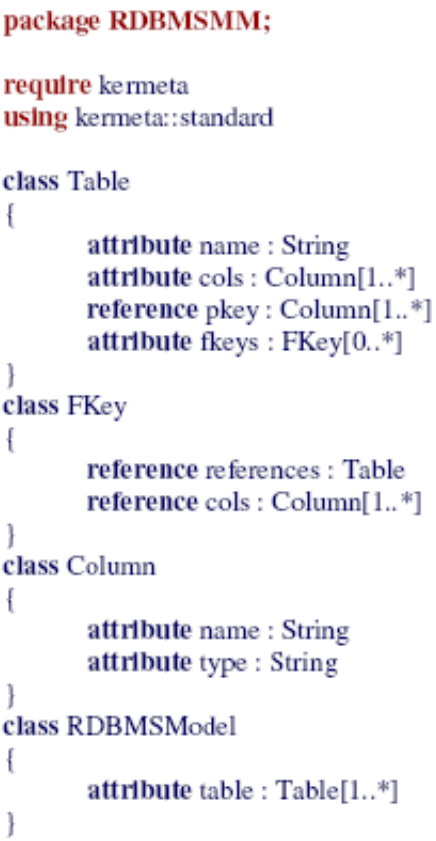


Figura 13: Metamodelo del esquema relacional expresado en texto Kermeta

```
package Class2RDBMS;

require kermeta     // The kermerta standard library
require "trace.kmt"                 // The trace framework
require "../metamodels/ClassMM.ecore"  // Input metamodel in ecore
require "../metamodels/RDBMSMM.kmt"    // Output metamodel in kermeta

[...]

class Class2RDBMS
{
  /** The trace of the transformation */
  reference class2table : Trace<Class, Table>
  /** Set of keys of the output model */
  reference fkeys : Collection<FKey>

  operation transform(inputModel : ClassModel) : RDBMSModel is do
    // Initialize the trace
    class2table := Trace<Class, Table>.new
    class2table.create
    fkeys := Set<FKey>.new
    result := RDBMSModel.new
    // Create tables
    getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
      var table : Table init Table.new
      table.name := c.name
      class2table.storeTrace(c, table)
      result.table.add(table)
    }
    // Create columns
    getAllClasses(inputModel).select{ c | c.is_persistent }.each{ c |
      createColumns(class2table.getTargetElem(c), c, "")
    }
    // Create foreign keys
    fkeys.each{ k | k.createFKeyColumns }

  end

  [...]
}
```

Figura 14: Transformación Kermeta

Herramientas

La herramienta utilizada para trabajar con el lenguaje Kermeta es un plugin basado en el IDE Eclipse. Es un proyecto con licencia open source EPL (Eclipse Public License) integrado totalmente en Eclipse. La plataforma incluye librerías para cargar y almacenar modelos desde EMF (Eclipse Modelling Framework [29]) e importar metamodelos Ecore.

La herramienta brinda facilidades como sintaxis coloreada y capacidades para autocompletado de código. Su versión actual es la 1.3.0, testeada bajo Eclipse 3.4. Es un proyecto vigente que cuenta con actualizaciones en el año 2009. Cuenta con manual de usuario, ejemplos de desarrollo en su website oficial [30], soporte a usuarios a través de lista de distribución de correos y blog.

4.2.3. Enfoques Relacionales

Esta categoría agrupa aproximaciones declarativas donde el concepto principal son las relaciones matemáticas. La idea básica es establecer el tipo de los elementos de origen-destino y especificarlos usando restricciones.

Para implementar este tipo de transformaciones se utiliza programación lógica. Una característica a destacar es que los lenguajes que siguen aproximaciones basadas en programación lógica naturalmente soportan bidireccionalidad [26].

Tefkat [5] es una implementación de un lenguaje de transformación de modelos declarativo. Fue desarrollado por Distributed Systems Technology Centre en Australia y la última actualización de su código fue realizada en Setiembre de 2008. Los ejemplos de Tefkat presentados en esta sección fueron extraído de [35], en donde se examinan las características de Tefkat con mayor profundidad.

Los metamodelos comprendidos por Tefkat son aquellos expresados por MOF, UML2 y XML Schema. Las transformaciones en Tefkat pueden aceptar varios modelos de entrada y producir varios modelos de salida. Se pueden expresar restricciones sobre la existencia de objetos y/o relaciones entre estos tanto en los modelos de entrada como en los de salida. Sin embargo al tratarse de transformaciones constructivas, no es posible especificar la inexistencia de objetos. Tampoco es posible realizar transformaciones in-place. Las restricciones se pueden parametrizar mediante la utilización de patterns. Podemos ver un ejemplo de restricción en la Figura 15.

```
RULE constraint_no_reflexive_relations_on_non_persistent_classes
  FORALL Class C
  WHERE C.is_persistent = false
    AND ClassHasReference(C, C, _)
    AND println("Found a non-persistent class in relation (by association
or attribute) with itself: ", C)
  SET FALSE;
```

Figura 15: Restricción: las clases no pueden estar asociadas a si mismas

La sintaxis del lenguaje se asemeja a SQL y permite expresar reglas unidireccionales. Las precondiciones de una regla se especifican mediante la cláusula WHERE. Es posible reutilizar las reglas mediante dos mecanismos provistos por Tefkat: EXTEND y OVERRIDE. Otra característica importante es la posibilidad de utilizar reflection para acceder a meta-atributos y especificar los mismos mediante arreglos de caracteres. Además brinda un superobjeto ANYTYPE que permite que una regla sea aplicable a todos los objetos sin importar su tipo específico. En la Figura 16, vemos un ejemplo de declaración de reglas en donde se crea una tabla en el modelo destino y se almacena un link de trazabilidad para cada clase del modelo origen.

```
RULE ClassAndTable(C, T)
  FORALL Class C {
           is_persistent: true;
           name: N;
         }
  MAKE Table T {
         name: N;
       }
  LINKING ClsToTbl WITH class = C, table = T;
```

Figura 16: Definición de una regla

No existen mecanismos explícitos en las transformaciones para especificar el orden de aplicación de las reglas ni la forma en que se recorre el modelo origen. Las reglas pueden obtener información sobre los elementos de origen y sobre los *tracking extents*. Los tracking extents son elementos de trazabilidad generados durante la ejecución de las reglas de transformación y son especificados por el desarrollador (ver declaración de clases de tracking en la Figura 17). Constituyen el mecanismo dedicado de trazabilidad brindado por Tefkat.

```
CLASS ClsToTbl {
  Class class;
  Table table;
};

CLASS AttrToCol {
  Class class;
  Attribute attr;
  Column col;
};
```

Figura 17: Declaración de clases de tracking

Si bien Tefkat no ofrece soporte nativo para la propagación de cambios, en [31] se describe como incorporarlo.

Herramientas

Los creadores de Tefkat desarrollaron un plug-in de Eclipse basado en EMF y AntLR. Posee un editor que resalta la sintaxis y un debugger que se integra a Eclipse, resultando una interfaz intuitiva y simple de usar.

Se dispone de tutoriales y ejemplos que facilitan el aprendizaje de esta herramienta. Existe además una comunidad reducida de usuarios que cuenta con un foro.

El parser de esta herramienta emite warnings si detecta una utilización incorrecta de las variables. Sin embargo no es capaz de detectar violaciones de restricciones de forma indirecta por parte de las reglas (por más detalles consultar [35]).

4.2.4. Enfoques Basados en Transformaciones de Grafos

Estos enfoque para transformación de modelos se basan en el trabajo teórico sobre transformación de grafos. Operan sobre grafos tipados, con atributos y etiquetas los cuales pueden ser pensados como representaciones formales de diagramas de clases simplificados.

Las reglas de transformación consisten en patrones LHS y RHS. Los patrones de grafos pueden ser traducidos en la sintaxis concreta de sus respectivos modelos de origen y destino. Los patrones LHS son reconocidos en el modelo a transformar y reemplazados por el patrón RHS.

MOLA (MOdel transformation LAnguage) [21] es clasificado como un lenguaje basado en grafos. Fue desarrollado por la Universidad de Latvia como un lenguaje procedural gráfico, proyecto que comenzó en el año 2003 y que cuenta con actualizaciones recientes. Se destaca por utilizar definiciones de patrones gráficos avanzados y estructuras de control tomadas de los lenguajes tradicionales de programación estructurada.

La idea básica para una transformación en MOLA es la utilización de patrones (pattern matching), a los cuales se les puede incluir restricciones OCL. En MOLA, se deben especificar reglas que definen cómo los elementos reconocidos deben ser transformados.

Como en la mayoría de los lenguajes de transformación, en MOLA se definen metamodelos de origen y destino para especificar la sintaxis abstracta de los modelos. Ambos metamodelos pueden coincidir y el lenguaje de metamodelado utilizado es EMOF. Los metamodelos deben ser especificados en un mismo diagrama de clases y pueden incluir clases, atributos, asociaciones, enumerados, paquetes y herencia simple. Además, asociaciones especiales que mapean clases entre los metamodelos de origen y destino pueden ser agregadas. El rol de estas asociaciones es similar al de las relaciones en otros lenguajes de transformación; estructurar la transformación y documentar la *trazabilidad* (traceability links). Clases y asociaciones temporales pueden ser definidas para almacenar datos intermedios (*estructuras intermedias*).

La Figura 18 muestra un fragmento del metamodelo MOLA para el ejemplo de transformación de diagrama de clases UML a esquema relacional. En la figura, el metamodelo origen se encuentra en la parte superior y el metamodelo destino en la parte inferior. Las líneas punteadas identifican las asociaciones que mapean elementos del metamodelo origen al metamodelo destino.

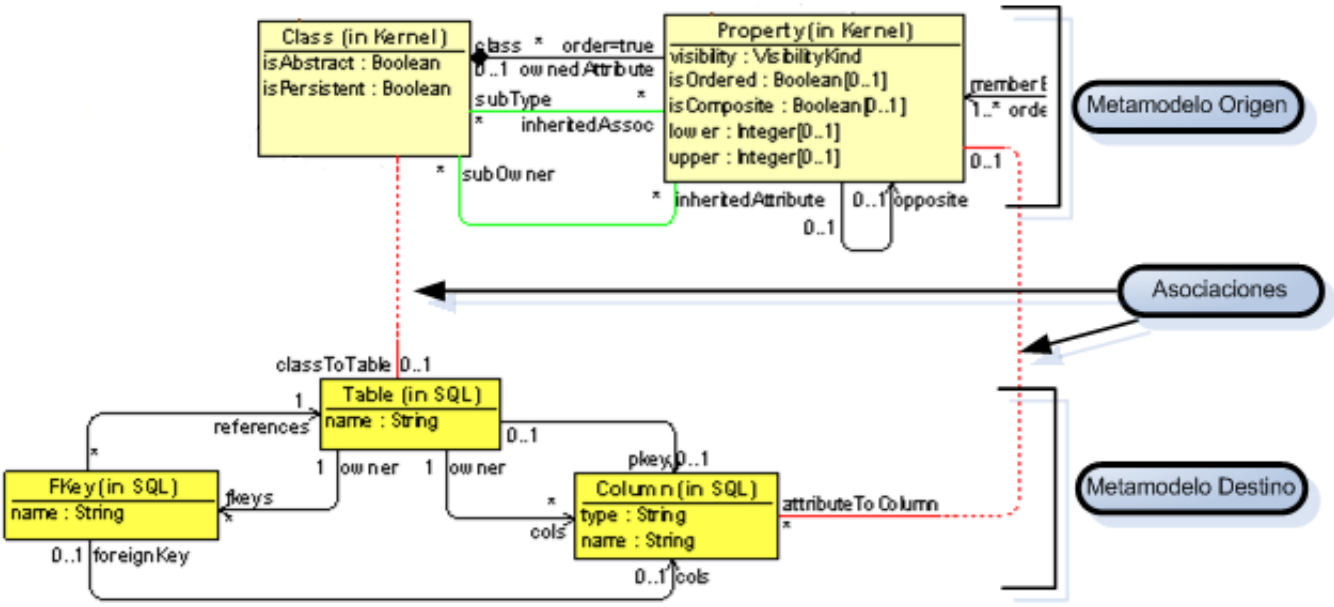


Figura 18: Fragmento de la especificación de los metamodelos del caso de estudio en MOLA

Una transformación MOLA consiste de un metamodelo y un conjunto de procedimientos o diagramas (similares a los diagramas de actividad de UML) de los cuales uno será el principal. El algoritmo de la transformación es descrito por los procedimientos MOLA. Como se observa en la Figura 19, un procedimiento MOLA es una secuencia de declaraciones gráficas, unidas por flechas, que comienza con un símbolo de inicio UML y termina con un símbolo de fin UML. Para mayor comprensión de los constructores del

lenguaje y del ejemplo de dicha figura el lector puede consultar [20].

La selección de reglas para su ejecución (planificación) se realiza mediante una condición explícita y la iteración sobre las mismas mediante loops. El loop está constituido por una o más reglas; a una de esas reglas se le denomina cabezal del loop; esta es la regla principal que contiene la variable sobre la cual se iterará (variable del loop). Las reglas MOLA no solo realizan reconocimiento de patrones sino que también realizan acciones, como creación y eliminación de instancias o modificación del valor de un atributo de una instancia (asignación).

MOLA permite realizar llamadas a subprogramas. Un subprograma puede ser un procedimiento MOLA así como también un programa externo (ej: programa en C++).

El lenguaje permite la agrupación de diagramas en paquetes, esto aporta legibilidad y permite crear transformaciones en forma incremental y reusables. Asismismo, MOLA permite especificar transformaciones bidireccionales, aunque este tipo de transformaciones deben ser desarrolladas específicamente pensando en esta propiedad.

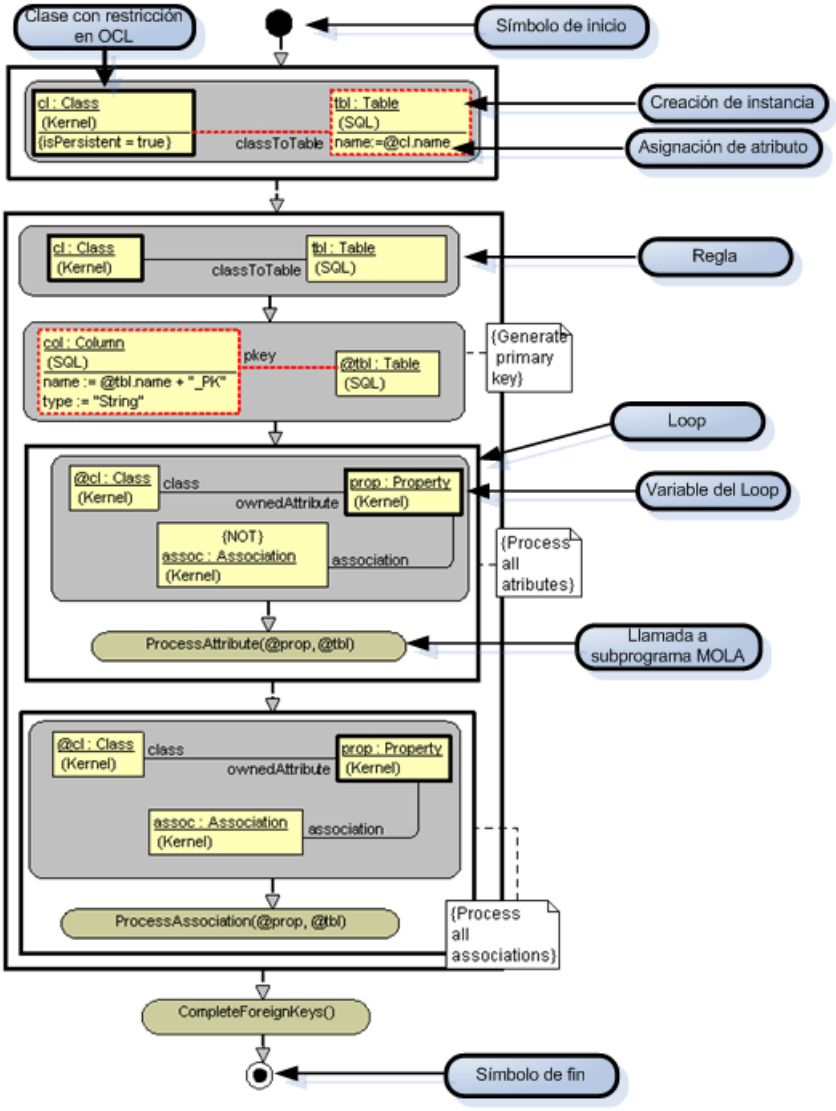


Figura 19: Procedimiento MOLA

Herramientas

MOLA Tool [19] es la herramienta utilizada para trabajar con el lenguaje MOLA. La misma es freeware y fue desarrollada por la Universidad de Latvia. La versión actual de esta herramienta (MOLA2 Tool) implementa la versión final del lenguaje de transformación MOLA2.

MOLA Tool consiste en dos partes, el entorno de desarrollo de transformaciones (TDE) y el entorno de ejecución de transformaciones (TEE).

TDE cuenta con:

- Editores gráficos de metamodelos y procedimientos MOLA desarrollados en base del METAclipse framework (plugin de la plataforma de metamodelado de Eclipse). Los editores cuentan con reconocimiento de sintaxis, los que proveen correctores y asistentes para el ingreso de contenido. Herramientas de integridad también son soportadas, donde los cambios son automáticamente dis-

tribuidos sobre todos los elementos refererenciados en un programa MOLA y warnings son mostrados si los cambios violan la integridad de los modelos.

- Compiladores de transformaciones que construyen ejecutables (dll o jar) para varios repositorios basados en metamodelos "en memoria" (ej: mii_rep de la Universidad de Latvia, JGraLab de la Universidad de Koblenz-Landau, EMF de Eclipse Community). Estos compiladores chequean la correctitud sintáctica y son capaces de marcar los errores sobre los diagramas MOLA.

- Importación/Exportación desde varios entornos de modelado, análisis y refactoring de transformaciones, etc.

TEE consiste de un repositorio basado en metamodelo "en memoria" y el ejecutable de la transformación. Actualmente son soportados mii_rep, JGralab y EMF. El repositorio debe contener el respectivo metamodelo (que puede ser creado con MOLA Tool). En TEE se brindan las siguientes facilidades para ejecutar transformaciones compiladas:

- Ejecutor de transformaciones, construido como un plugin de Eclipse que permite la ejecución de la transformación de forma sencilla.

- Buscador en repositorio, construido como una aplicación Java que permite recorrer el modelo en una vista tabular.

- Usar las transformaciones compiladas (dll o jar) directamente en cualquier herramienta de modelado como un plugin, como por ejemplo en Rational Software Architect y en Enterprise Architect.

MOLA cuenta con manual de usuario y ejemplos de desarrollo en su website oficial [18].

VIATRA (VIsual Automated model TRAnsformations) es una herramienta para la transformación de modelos que forma parte del framework VIATRA2 [49], implementado en Java e integrado en Eclipse. Este proyecto tuvo sus inicios a principios del 2004 y sus últimas actualizaciones datan de fines del 2008. Su objetivo principal es proveer soporte al ciclo completo de transformación de modelos, incluyendo especificación, diseño, ejecución, validación y mantenimiento de transformaciones.

VIATRA está formado por lenguajes textuales para definir modelos, metamodelos y transformaciones. Estos lenguajes no están basados en los estándares MOF ni QVT. Cuenta con un lenguaje de generación de código basado en templates, llamado VTTL (Viatra Textual Template Language). El lenguaje de metamodelado es denominado VTML (Viatra Textual Metamodeling Language), y el lenguaje para especificar transformaciones se denomina VTCL (Viatra Textual Command Language). VTCL es un lenguaje basado en grafos, utiliza métodos formales como transformación de grafos (GT [45]) y máquinas de estados abstractos (ASM [12]) para manipular modelos basados en grafos y realizar tareas de verificación, validación y seguridad.

La naturaleza de VIATRA es declarativa, aunque también son soportadas secciones de código imperativo. El concepto básico para definir una transformación es utilizar reconocimiento de patrones. La manipulación de los modelos es llevada a cabo mediante reglas a las cuales se les especifican *precondiciones*, *post-condiciones* y acciones. Cuando un patrón es reconocido y cumple con la precondición, las reglas de la transformación cambian los elementos del modelo de acuerdo a los patrones especificados en las post-condiciones y a acciones adicionales. *Estructuras intermedias* son utilizadas para prevenir múltiples ejecuciones de una regla para el mismo elemento de entrada. VIATRA no cuenta con soporte para reglas *bidireccionales*, sólo son soportadas reglas *unidireccionales*.

Las figuras 20 y 21 muestran el ejemplo de transformación de modelo de clases UML a esquema relacional. En la Figura 20 se observa la definición del metamodelo para el esquema relacional definido en VTML. En la Figura 21 se muestra un extracto del código de la transformación especificado en VTML. Si el lector desea profundizar en la definición de este ejemplo puede consultar [48].

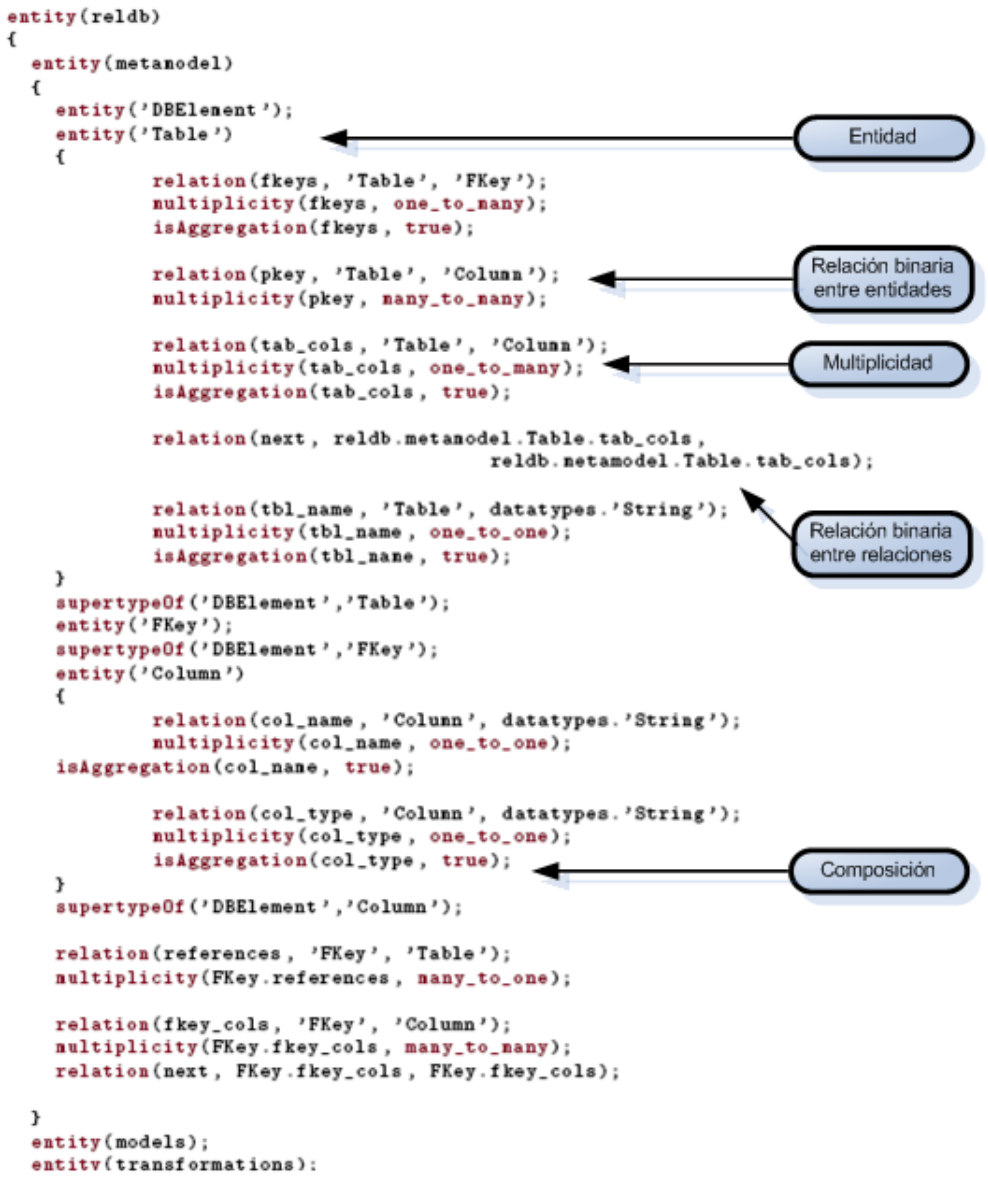


Figura 20: Definición de metamodelo en VIATRA

Como se comentó con anterioridad, VIATRA ofrece manipulación de los modelos mediante máquinas de estados finitos. Los usuarios pueden crear dichas máquinas, lo que ofrece un mecanismo de planificación de reglas de *condición explícita* (*planificación*). También soporta *iteración de punto fijo* sobre sus reglas *(fixpoint iterarion)* y *aplicación concurrente (concurrent aplication)*, ya que un regla puede ser aplicada concurrentemente a todos los patrones reconocidos en el modelo origen.

VIATRA permite agrupar reglas en módulos, donde un módulo puede importar otro para acceder a su contenido. Así el lenguaje provee mecanismos de modularidad para la *organización de las reglas* y *reusabilidad* por medio de *herencia*. La información de *trazabilidad* puede ser manipulada como cualquier otro elemento en el destino, aunque no se provee de un mecanismo dedicado.

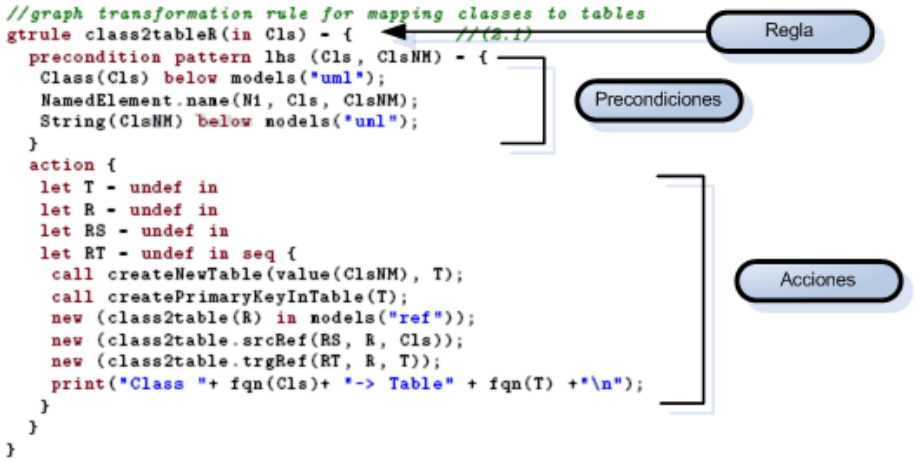


Figura 21: Transformación en VTML

En VIATRA los modelos de origen y destino son siempre el mismo, sólo son soportadas actualizaciones *in-place*. Transformaciones incrementales son soportadas utilizando reglas de transformación con condiciones negativas, que se pueden corresponder sólo a nuevos elementos en el modelo *(incrementalidad)*. También cuenta con soporte para transformaciones de alto orden, así se pueden crear meta transformaciones, en las cuales los modelos de origen y destino son transformaciones en si mismas.

Herramientas

VIATRA es una herramienta gratuita que utiliza las bondades del IDE Eclipse (3.0, última versión). Existe una guía de usuario con especificaciones de los lenguajes (VTML, VTCL, VTTL), semántica semiformal de estos lenguajes y ejemplos. Se brinda soporte a usuarios por medio de una lista de distribución de correo, newsgroup, wiki y se pueden realizar reportes de errores.

VIATRA ha sido utilizado en varios proyectos [47], como DECOS EU FP6 IP en el campo de sistemas embebidos dependientes, SENSORIA EU FP6 IP en el campo de computación orientada a servicios, DIANA EU FP6 STREP en el campo de la aviación electrónica, HIDENETS EU FP6 IP en el campo de las redes vehiculares y MOGENTES EU FP7 ICT STREP en el campo de generación de test basados en modelos.

AToM3 (A Tool for Multi-Formalism Modeling and Meta-Modeling) es una herramienta gratuita escrita en Python. Fue desarrollada bajo un proyecto de investigación por la Universidad de McGill en Montreal y la colaboración del Prof. Juan de Lara de la Universidad Autónoma de Madrid.

En esta herramienta se incluye un formalismo básico similar al modelo entidad relación extendido con el cual se realiza la definición de los diferentes metamodelos en un entorno gráfico, que posteriormente se convierte a código Phyton (el estándar MOF no es soportado por la herramienta). Permite conversiones entre modelos utilizando *gramática de grafos*, un mecanismo también programado en el lenguaje Phyton. Es posible definir una secuencia de gramáticas de grafos que se pueden aplicar al modelo (ej: transformar un modelo en otro formalismo y luego aplicarle una optimización).

Una transformación se define por medio de reglas que poseen una parte LHS donde se definen de manera grafica las precondiciones necesarias para disparar la regla y una parte RHS que contiene el modelo que reemplazará el que equipare el lado izquierdo de la regla. Dichas reglas son unidireccionales. Para este tipo de reglas se deben definir condiciones y acciones a ejecutar cuando la regla se active. Las condiciones y acciones pueden ser expresadas en lenguaje Phyton u OCL. Dado que el lado izquierdo de una regla puede ser reconocido en varios subgrafos del modelo, la selección de reglas para la planificación puede ser

resuelta por medio de selección interactiva (el usuario selecciona la regla que se aplicará) o resolviendo los conflictos mediante un mecanismo de prioridades. Las reglas se aplican en forma concurrente en todas las detecciones. La iteración de reglas se realiza mediante fixpoint iterations.

En AToM3 los modelos de origen y destino son siempre el mismo, solo son soportadas actualizaciones in-place. Algunos de los metamodelos que actualmente se encuentran disponibles son: entidad/relacion, GPSS, autómatas finitos determinísticos, automatas finitos no determinísticos, redes de Petri y diagramas de flujo de datos. Existen ejemplos y manual de usuario para la herramienta.

4.2.5. Enfoques Híbridos

El enfoque híbrido basicamente combina distintas características que proveen los enfoques anteriores [7, 51].

Los lenguajes puramente declarativos por lo general se limitan a escenarios donde los metamodelos de entrada y de salida poseen una estructura similar y consecuentemente permiten realizar un mapeo.

Si bien los lenguajes puramente imperativos permiten manejar un mayor número de escenarios, poseen un nivel de abstracción menor. Esto resulta en un mayor esfuerzo por parte del usuario para resolver cuestiones como la trazabilidad.

Los lenguajes híbridos combinan ambos enfoques para sumar sus ventajas.

QVT (Query/View/Transformation) es un estándar para la transformación de modelos definido por la OMG. Depende de los estándares OCL 2.0 y MOF 2.0 para la especificación de los metamodelos y modelos. En QVT solo son soportadas transformaciones de modelo a modelo, donde un modelo es una entidad que se ajusta a un metamodelo MOF. Como toda transformación QVT es un modelo, debe ajustarse a un metamodelo MOF. Esto significa que la sintaxis abstracta de QVT se ajusta a un metamodelo MOF 2.0.

La especificación QVT cuenta con una naturaleza híbrida (declarativa/imperativa). La parte declarativa es separada en una arquitectura de dos capas en distintos niveles de abstracción. Estas capas son:

- Relations, un lenguaje amigable al usuario que soporta reconocimiento de patrones complejos, creación de objetos mediante templates y creación implícita de trazas entre elementos de los modelos origen/destino.
- Core, un lenguaje definido usando extensiones mínimas de EMOF y OCL. Las trazas son explícitamente definidas como modelos MOF y sus instancias son creadas y eliminadas de igual forma que para otros objetos.

Se puede realizar una analogía entre estos dos lenguajes y la arquitectura Java, donde el lenguaje Core juega el papel de Java Byte Code, el lenguaje Relations simboliza al lenguaje Java, y la transformación entre Relation y Core puede representarse como la especificación del compilador Java que produce Byte Code [38].

La parte imperativa está formada por dos componentes que extienden a los lenguajes Relations y Core:

- El lenguaje Operational Mapping, cuya sintaxis provee constructores comúnmente encontrados en lenguajes de programación imperativos.
- La implementación BlackBox, para invocar facilidades de transformaciones expresadas en otros lenguajes (XSLT, XQuery). Es útil para integrar librerías y transformaciones no realizadas con QVT.

Siguiendo con la analogía de la arquitectura Java, la habilidad de invocar Operational Mapping y Black-Box es equivalente a realizar llamadas a Java Native Interface (JNI) [38].

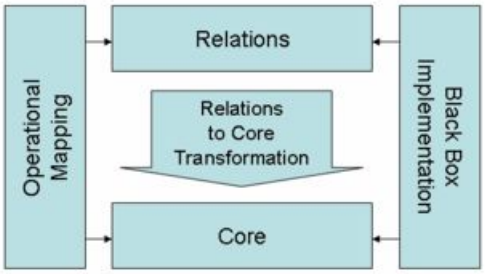


Figura 22: Arquitectura QVT

QVT Relations (QVT-R) es un lenguaje clasificado como relacional. Brinda una especificación *declarativa* de las relaciones entre los modelos MOF, soporta reconocimiento de patrones y puede ser representado mediante una declaración textual o gráfica. En este lenguaje una transformación necesita declarar parámetros para manejar los modelos involucrados en la transformación. Estos parámetros están tipados sobre los metamodelos apropiados.

Las transformaciones son descritas a través de:

- la enumeración de los metamodelos participantes,
- un conjunto de reglas que especifican la relación existente entre términos de los metamodelos,
- un conjunto de dominios por regla que se ajusta al conjunto de términos para los que se expresan relaciones,
- un conjunto de patrones que cumplen con la estructura de los términos y a los que se aplican operaciones OCL [42].

Una relación (regla) puede incluir condiciones de aplicación *(precondiciones)*, cláusulas WHEN y/o cláusulas WHERE. Las relaciones pueden ser top-level (siempre son ejecutadas) o pueden ser relaciones "comunes" (sólo pueden ser invocadas desde las cláusulas anteriormente mencionadas). Existen definiciones KEY utilizadas para definir claves que luego determinarán si las instancias de los objetos deben ser creadas o actualizadas durante la ejecución de la transformación.

En QVT-R las reglas pueden ser de transformación o de validación. El dominio de destino determina si la regla es de transformación o validación, dependiendo si se encuentra marcado como ENFORCE o CHECKONLY. La marca CHECKONLY comprueba si existe una correspondencia válida entre los modelos, mientras que la marca ENFORCE fuerza a que los términos de los respectivos modelos cumplan la relación descrita en la regla [42].

El lenguaje provee un mecanismo para *organizar reglas* en módulos. Un módulo puede importar a otro módulo para acceder a su contenido. Las *organización de reglas* es orientada al objetivo *(target-oriented)*: existe una regla por cada tipo de elemento en el modelo destino. La anidación de las reglas corresponde a la herencia existente en el modelo origen.

La estrategia de *aplicación de reglas* es determinista. La *planificación* de reglas puede ser implícita o explícita. El orden de ejecución de las reglas top-level no puede ser especificado por el usuario, lo que lo hace una ejecución implícita. Sin embargo una regla puede invocar a otras reglas, esto habilita al desarrollador a especificar el orden de ejecución de estas reglas.

QVT-R cuenta con soporte para *bidireccionalidad*, ya que reglas multidireccionales pueden ser especificadas. Al momento de la invocación, el usuario debe especificar la dirección de la ejecución de la transformación. Una transformación puede ser ejecutada en la dirección de un solo parámetro. El modelo especificado por este parámetro será el modelo destino de la transformación. En casos especiales, el modelo destino puede ser uno de los modelos de origen, por lo que el lenguaje también soporta actualizaciones *in-place*.

El lenguaje posee la habilidad de generar actualizaciones en modelos de destino existentes basadas en cambios realizados en los modelos de origen. Esta propagación de cambios hace que QVT-R provea incrementalidad. Además cuenta con creación ímplicita de declaraciones de clases para trazas y sus instancias para almacenar información durante la ejecución de la transformación. De esta forma traceability links son creados automáticamente.

Las figuras 23 y 24 ejemplifican la codificación en QVT-R del ejemplo de transformación entre diagrama de clases UML y modelo relacional. En la Figura 23 se puede apreciar un extracto del lenguaje QVT-R para definir una transformación en modo textual. La Figura 24 muestra la declaración de la relación ClassToTable expresada en el modo gráfico del lenguaje QVT-R. Para una mayor comprensión del ejemplo y de la especificación del lenguaje, el lector puede consultar [38].

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
key Table (name, schema);
key Column (name, owner);   // owner:Table opposite column:Column
key Key (name, owner); // key of class ëKeyí;
                       // owner:Table opposite key:Key

top relation PackageToSchema   // map each package to a schema
{
   pn: String;

   checkonly domain uml p:Package {name=pn};
   enforce domain rdbms s:Schema {name=pn};
}

top relation ClassToTable   // map each persistent class to a table
{
   cn, prefix: String;

   checkonly domain uml c:Class {namespace=p:Package {},
                                 kind='Persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
          column=cl:Column {name=cn+'_tid', type='NUMBER'},
          key=k:Key {name=cn+'_pk', column=cl}};
    when {
       PackageToSchema(p, s);
    }
    where {
       prefix = '';
       AttributeToColumn(c, t, prefix);
    }
 }
```


Figura 23: Transformación QVT-R expresada mediante sintaxis textual

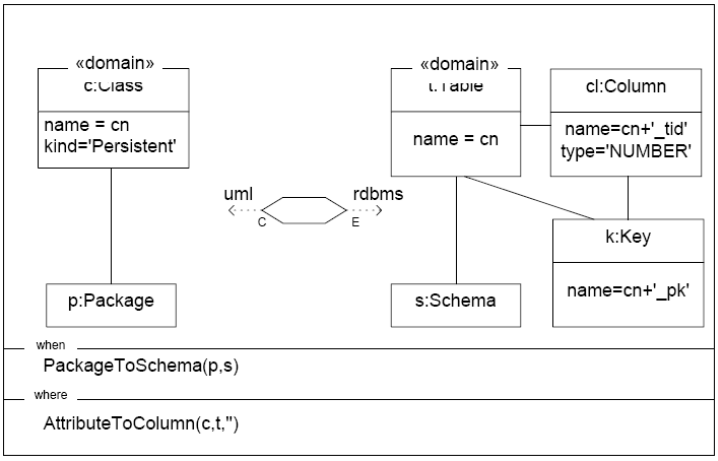


Figura 24: Transformación QVT-R expresada mediante sintaxis gráfica

Herramientas

Medini QVT [28] es una herramienta desarrollada por ikv++ technologies [27] bajo licencia EPL. Al momento de escribir el documento se encuentra disponible la versión 1.6.0. La herramienta está integrada en el entorno de desarrollo Eclipse, permite la ejecución de transformaciones QVT expresadas con la sintaxis textual del lenguaje Relations. Cuenta con un editor de código con asistencia y debugger para relaciones. Soporta el manejo de trazas, actualizaciones incrementales y transformaciones bidireccionales. Provee ejemplos, tutorial, lista de FAQs, foro de discusión, y video de iniciación en el uso.

ModelMorf [36] es una herramienta propietaria desarrollada por Tata Consultancy Services Ltd. Al momento de escribir este documento se encuentra en su versión Beta1. Cuenta con soporte para realizar transformaciones in-place, operaciones BlackBox, trazas, herencia de transformaciones, sobrecarga de relaciones y ejecución incremental. Una version trial puede ser descargada desde el sitio oficial.

Eclipse M2M [24] es una herramienta oficial de Eclipse open source que se encuentra en fase de desarrollo.

QVT Core (QVT-C) es un lenguaje que brinda una especificación declarativa de las relaciones entre los modelos MOF. Soporta reconocimiento de patrones sobre un conjunto de variables, permitiendo evaluar condiciones sobre las mismas en un conjunto de modelos.

QVT-C es un lenguaje tan potente como QVT Relations y su semántica puede ser definida de forma más simple. A diferencia del lenguaje Relations que implícitamente crea clases y objetos para mantener trazas y almacenar lo ocurrido durante la transformación, en QVT-C las trazas deben ser definidas explícitamente. Estas clases son definidas utilizando MOF.

En QVT-C una transformación es especificada como un conjunto de mappings que declaran restricciones sobre los elementos de los modelos y el modelo de trazas. Una transformación puede ser ejecutada en dos modos: CHECKING o ENFORCE. En el modo CHECKING, la ejecución de la transformación valida las restricciones sobre los modelos involucrados y el modelo de trazas. En modo ENFORCE, la transformación es ejecutada en una dirección particular determinada al seleccionar el modelo objetivo.

Una transformación contiene mappings. Un mapping contiene uno o más dominios asociados al tipo de los modelos definidos en la transformación y puede contener operaciones con expresiones OCL dentro del cuerpo. A los dominios no se les asigna un nombre, son identificados por el mapping y el tipo del modelo asociado. QVT-C permite realizar refinamiento de mappings para especializarlos en mappings más específicos (similar al concepto herencia entre clases). También cuenta con la posibilidad de componer

patrones. Un mapping especifica patrones para especificar los objetos a los cuales afectará. Un patrón es especificado por un conjunto de variables, predicados y asignaciones, pudiendo además depender de otros patrones.

La Figura 25 ejemplifica la definición de una transformación de modelo de clases de UML a esquema relacional. Para mayor detalle de la especificación del lenguaje, y la explicación del ejemplo, el lector puede consultar [38].

```
module UmlRdbmsTransformation imports SimpleUML, SimpleRDBMS {

    transformation umlRdbms {
        uml imports SimpleUML;
        rdbms imports SimpleRDBMS;
    }

    -- Package and Schema mapping
    class PackageToSchema {
        composite classesToTables : Set(ClassToTable) opposites owner;
        composite primitivesToNames : Set(PrimitiveToName) opposites owner;
        name : String;
        -- uml
        umlPackage : Package;
        -- rdbms
        schema : Schema;
    }

    map packageToSchema in umlRdbms {
        uml () {
            p:Package
        }
        rdbms () {
            s:Schema
        }
        where () {
            p2s:PackageToSchema|
            p2s.umlPackage = p;
            p2s.schema = s;
        }
        map {
            where () {
                p2s.name := p.name;
                p2s.name := s.name;
                p.name := p2s.name;
                s.name := p2s.name;
            }
        }
    }
```

Figura 25: Transformación expresada en QVT-C

Como se mencionó con anterioridad, QVT Core y QVT Relations cuentan con la misma expresividad. Esto implica que una transformación definida en el lenguaje Relations pude ser traducida con semántica equivalente al lenguaje Core. La Figura 26 muestra la traducción de la relación ClassToTable expresada en QVT Relation a QVT Core.

```
relation ClassToTable
{
  checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent',
name=cn}
  checkonly domain rdbms t:Table {schema=s:Schema {}, name=cn}
  when {
    PackageToSchema(p, s);
  }
  where {
    AttributeToColumn(c, t);
  }
}
```

Se traduce en:

```
map ClassToTable in umlRdbms
{
  check uml (p:Package) {
    c: Class|
    c.namespace = p;
    c.kind = 'Persistent';
  }
  check rdbms (s:Schema) {
    t:Table|
    t.schema = s;
  }
  where (v1: TPackageToSchema| v1.p = p; v1.s = s;) {
    realize v2: TClassToTable, cn:String |
    v2.p := p;
    v2.s := s;
    v2.c := c;
    v2.t := t;
    c.name = cn;
    t.name = cn;
  }
}
```

Figura 26: Traduccion de QVT-R a QVT-C

Herramientas

OptimalJ es una herramienta desarrollada por Compuware en el año 2001 para el lenguaje Java y fue descontinuada en el año 2008. Algunas de sus características principales son: desarrollo orientado a modelos, proceso orientado al desarrollo, transformación de patrones, diseñador de interfaz de usuario y desarrollo integrado.

QVT Operational (QVT-O) es un lenguaje imperativo clasificado como operacional. También provee mecanismos para extender el lenguaje QVT Relations y de esta forma se convierte en un lenguaje híbrido, combinando la naturaleza declarativa de QVT Relations con su naturaleza imperativa. Las expresiones imperativas en QVT-O utilizan características funcionales en OCL y construcciones utilizadas en los clásicos lenguajes de propósito general (if-then-else, loops, variables, etc.). Los tipos de datos de los modelos son definidos por metamodelos MOF.

Una transformación en QVT-O es expresada unidireccionalmente. Se codifica expresando en la firma de la transformación cuáles son los modelos de origen y objetivo explícitamente (parámetros) y define un punto de entrada para la ejecución (main) que representa el código inicial a ser ejecutado para realizar la transformación. La transformación en si es una entidad instanciable con propiedades y operaciones. Utiliza para su definición datos intermedios almacenados en nuevas clases y propiedades intermedias que son creadas temporalmente durante la ejecución de la transformación.

En QVT-O se pueden definir mapeos entre objetos del modelo origen y del modelo destino mediante Mapping Operations (reglas). Es posible restringir los mapeos mediante precondiciones (cláusulas WHEN) y postcondiciones (cláusulas WHERE). Estas reglas cuentan con un cuerpo en donde se pueden especificar acciones, como por ejemplo creación de instancias.

El lenguaje provee tres mecanismos de reúso de reglas: herencia, merge y disyunción. Una regla puede heredar de otra regla (herencia), puede incluir una lista de reglas que la complementan (merge), o puede seleccionar una regla de un conjunto de reglas disjuntas (disyunción).

Los dominios de una regla son especificados por una dirección que restringe los posibles cambios. Estos

dominios pueden ser declarados como de entrada, de entrada/salida o de salida. El orden de ejecución de las reglas sigue el flujo del código de la transformación. QVT-O también permite definir helpers: operaciones sobre elementos del modelo origen que usualmente son utilizadas para realizar consultas.

El lenguaje provee un mecanismo de librerías para expresar definiciones que pueden ser reusadas en otras transformaciones. Una librería puede definir tipos específicos y operaciones, incluso en lenguajes diferentes a QVT (librerías BlackBox). Puede ser importada mediante las facilidades access o extension. Estos mecanismos permiten ejecutar transformaciones en parelelo, componer transformaciones y optimizar el resultado de las composiciones.

QVT-O no brinda soporte para la propagación de cambios: los modelos de salida creados con anterioridad no pueden ser actualizados [51]. Si bien QVT-O no soporta incrementalidad, si permite la ejecución de transformaciones in-place y cuenta con habilidad para invocar operaciones de alto nivel en modelos. Además cuenta con creación implícita de declaraciones de clases para trazas y de sus instancias.

La Figura 27 muestra un segmento de la codificación en QVT-O del ejemplo de transformación entre diagrama de clases UML y modelo relacional. Para una mayor comprensión del ejemplo y de la especificación del lenguaje, el lector puede consultar [38].

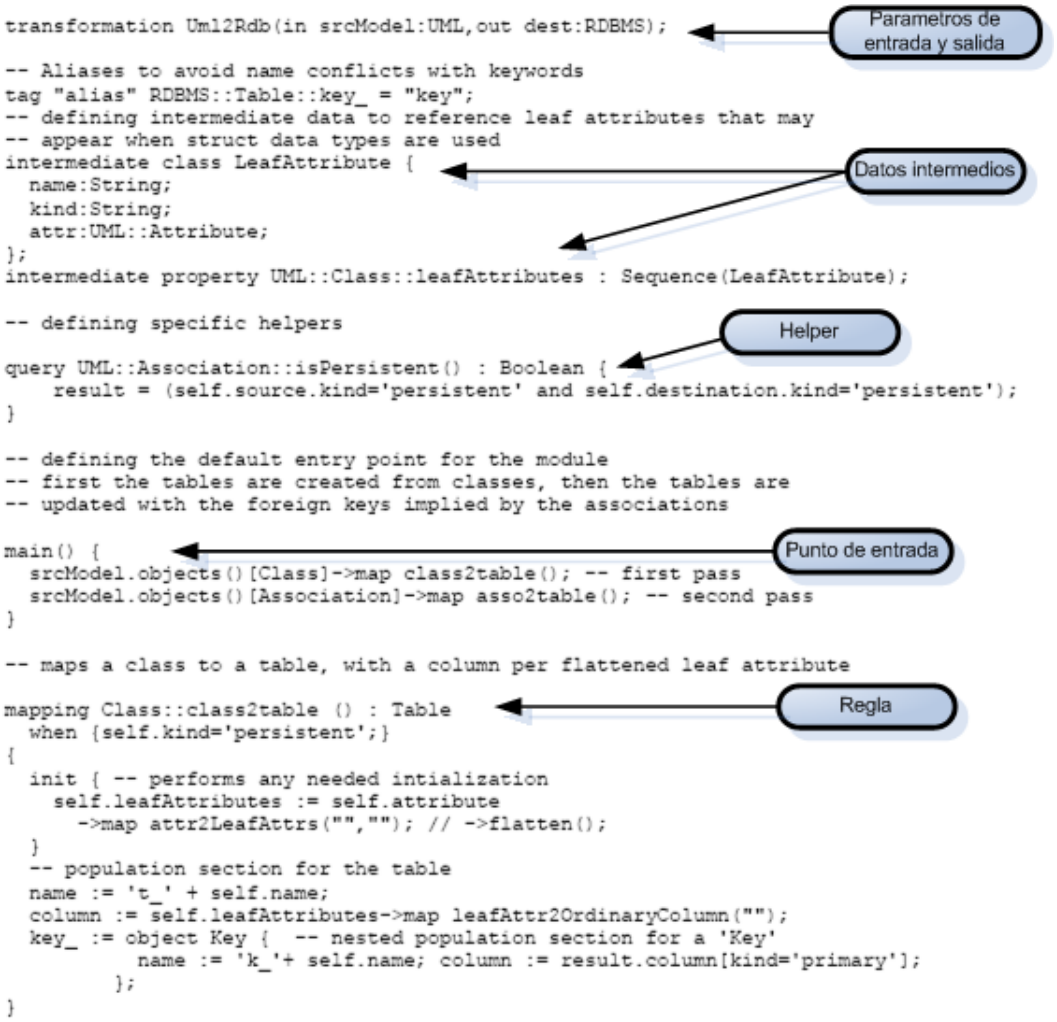


Figura 27: Transformación expresada en QVT-O

Herramientas

Borland Together [13] es una herramienta propietaria desarrollada por la empresa Borland como un conjunto de plugins disponibles para la plataforma Eclipse 3.5. Su última versión es Borland Together 2008 R2 Worldwide, realizada en Agosto del 2009. Utiliza QVT-O como lenguaje para realizar transformaciones de modelos y cuenta con soporte para sintaxis resaltada, validación y autocompletado de código OCL 2.0. Cuenta además con otras funcionalidades como la generación de documentación, herramientas para la creación de DSLs, modelado de procesos de negocio, de datos y diagramas UML. Provee de soporte a usuarios mediante ejemplos, un tutorial y lista de FAQs.

SmartQVT [43] es una herramienta open source dedicada a la transformación de modelos que utiliza el lenguaje QVT-O. Se distribuye como un plugin para Eclipse que ejecuta sobre EMF y cuenta con licencia EPL. Es desarrollada por France Telecom R&D y ha sido parcialmente financiada por el proyecto IST Modelware. La versión actual, al momento de escribir este documento es la 0.2.2, realizada en Agosto del 2008: esta versión es compatible con Eclipse 3.3 y EMF 2.3.0. Provee soporte a usuarios por medio

de lista de correos, grupo de noticias, foro de discusión y manual.

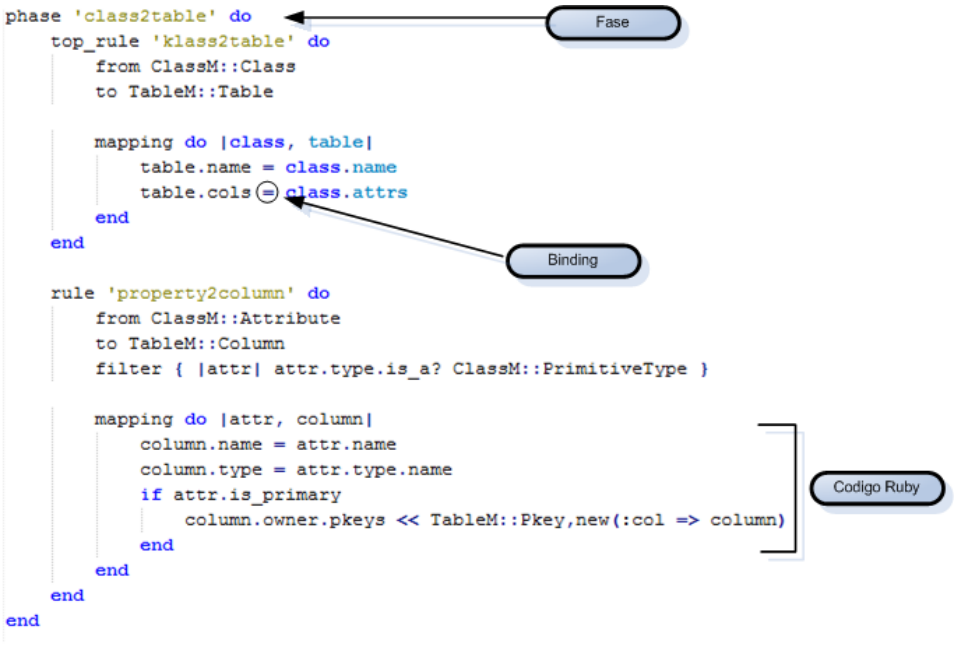
Eclipse M2M Operational QVT [24] es una herramienta open source oficial de Eclipse que implementa la especificación de QVT-O. La versión actual, al momento de escribir este documento, es la 2.0.0 realizada en Junio del 2009, compatible con Eclipse 3.5 y EMF 2.5.0. Entre sus principales características se encuentran: el resaltado de código, el marcado de errores, la asistencia para autocompletado de código y la existencia de un debugger. Provee soporte a usuarios por medio de grupo de noticias, foro de discusión, ejemplos y documentación.

RubyTL fue creado como un lenguaje específico de dominio (DSL) embebido en el lenguaje de programación dinámico Ruby. Posee una sintaxis clara y consistente, y una organización similar a la de ATL. A pesar de su relación con Ruby, requiere conocimientos mínimos del lenguaje para utilizarlo.

Como lenguaje híbrido, RubyTL posee una parte declarativa y una imperativa. La parte declarativa está basada en reglas y bindings (similares a los proporcionados por ATL). Un binding es una clase especial que indica "qué necesita ser transformado en qué", en lugar de "cómo debe ser algo transformado en algo". La parte imperativa viene dada por el propio lenguaje Ruby.

La definición de meta-modelos se realiza mediante el lenguaje de meta-modelado Ecore. También puede especificarse mediante EMOF y convertir la especificación a Ecore Estas funcionalidades se discutirán más adelante al presentar la herramienta AGE.

Las transformaciones pueden ser ejecutadas solamente de forma unidireccional, por lo que deben definirse reglas específicas si se desea ejecutar en ambos sentidos. Para mostrar un ejemplo[4] sencillo de definición de transformación que ilustre las características básicas de RubyTL se ha considerado nuevamente el problema clásico de transformación de un modelo de clases a un esquema relacional.




```
phase 'class2table' do
    top_rule 'klass2table' do
        from ClassM::Class
        to TableM::Table

        mapping do |class, table|
            table.name = class.name
            table.cols = class.attrs
        end
    end

    rule 'property2column' do
        from ClassM::Attribute
        to TableM::Column
        filter { |attr| attr.type.is_a? ClassM::PrimitiveType }

        mapping do |attr, column|
            column.name = attr.name
            column.type = attr.type.name
            if attr.is_primary
                column.owner.pkeys << TableM::Pkey.new(:col => column)
            end
        end
    end
end
```

Figura 28: Ejemplo de RubyTL

Uno de los constructores del lenguaje que se muestra en la Figura 28, es PHASE ("fase" en Inglés). Una fase es una agrupación de reglas de transformación que tratan con una parte del modelo origen.

[4] [15, 22, 23] presentan varios ejemplos

La utilización de fases es útil para manejar complejidad y aumentar legibilidad en el código de las transformaciones. Se puede organizar una definición de transformación en varias fases y así modularizar el código. Esto también provee una forma de reutilización del código de las transformaciones.

Siguiendo con el ejemplo, se puede observar que una definición de transformación consiste en un conjunto de reglas de transformación. Cada regla tiene un nombre y está formada de las siguientes partes:

- FROM especifica la metaclase del elemento origen
- TO especifica la metaclase de los elementos de destino
- FILTER especifica la condición o filtro que deben satisfacer los elementos origen (con este constructor se pueden diseñar patrones de ejecución de las reglas, planificación)
- MAPPING especifica las relaciones entre los elementos de los modelos origen y destino. Dicha tarea se puede realizar de forma declarativa mediante un conjunto de bindings (regla klass2table), o bien de forma imperativa utilizando instrucciones Ruby (regla property2column).

Toda trasformación en RubyTL debe tener un "punto de entrada" (en el ejemplo klass2table) que inicie la transformación. El algoritmo de aplicación de reglas ejecuta cada regla "punto de entrada", aplicándose la regla a todas las instancias de la metaclase especificada en su parte FROM (en el ejemplo a todas las instancias de ClassM::Class). Por lo tanto el motor de ejecución se comporta como un bucle en el que se ejecuta cada punto de entrada para cada instancia del modelo origen y crea las instancias correspondientes del modelo destino.

RubyTL es un lenguaje extensible, está desarrollado sobre un núcleo básico que provee mecanismos para extenderlo. Su mecanismo de extensión está basado en plugins desarrollados en Ruby que permiten añadir nuevas características al núcleo. Las extensiones se pueden clasificar en tres categorías:

- relacionadas con el algoritmo de transformación
- relativas a la creación de nuevas reglas y a la gestión del ciclo de ejecución de las reglas
- extensiones de la sintaxis del lenguaje.

Las extensiones relacionadas con el algoritmo de transformación (aplicación de reglas) no permiten determinar los lugares del modelo en donde las reglas deben aplicarse (determinación de la ubicación). El algoritmo ejecuta cada punto de entrada para cada instancia de la metaclase que cumpla el filtro de la regla. Es aquí donde puede extenderse el comportamiento de la operación encargada de ejecutar la regla sobre la instancia particular. En [15] se pueden ver más en detalle los mecanismos de extensión del lenguaje.

RubyTL tiene implementadas varias funcionalidades como plugins del lenguaje. Uno de ellos es la regla copy_rule que permite que una regla sea ejecutada más de una vez para un mismo elemento origen. Otros ejemplos son: la separación en fases del lenguaje, plugins que permiten definir varios puntos de entrada en una transformación mediante el uso de la regla top, establecer mappings muchos-a-uno o especificar reglas transitorias, es decir, reglas que crean elementos que sólo existen durante la ejecución de la transformación (estructuras intermedias).

Otra característica interesante de este lenguaje es que permite la composición de transformaciones. Esto puede aportar a la reusabilidad del código.

Como se ha comentado en principio, RubyTL fue diseñado como un DSL embebido en el lenguaje Ruby y por ello no es posible manipular una definición de transformación como un modelo. Sin embargo, el lenguaje cuenta con una versión simplificada llamada RubyTL-M que difiere de la versión original en su diseño ([22]). En RubyTL-M se ha partido de un metamodelo que describe su sintaxis abstracta y por lo tanto, una definición de transformación puede ser manipulada como un modelo.

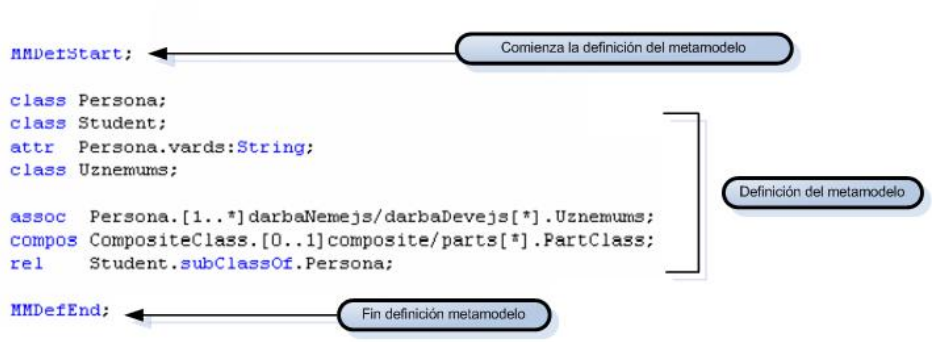


Figura 29: Metamodelado en Lx

Herramientas

La herramienta utilizada para trabajar con el lenguaje RubyTL es un plugin basado en el IDE Eclipse llamado AGE (Agile Generative Enviroment [1]). Es un proyecto iniciado en 2005 y que sigue en desarrollo (al momento de escribir este documento, su última versión es de Febrero de 2009). Es un proyecto con licencia GNU General Public License (GPL) version 3, y su estado de desarrollo es 3 - Alpha.

AGE integra un editor que resalta la sintaxis y provee asistencia de contenido para RubyTL. Brinda la posibilidad de configuración del plugin, como también un motor de plantillas para las transformaciones modelo-código y un editor de metamodelos. Además, cuenta con un lenguaje de validación similar a OCL. El soporte brindado consta de un manual de usuario, ejemplos de desarrollo en su website oficial, soporte a usuarios a través de lista de distribución de correos, blog, ejercicios de aprendizaje, etc.

Lx Family es una familia de lenguajes que se complementan entre sí. Para ser más exactos son cinco lenguajes llamados: L0, L0+ (basado en L0), L1, L2 y L3. Esta familia constituye integralmente un lenguaje de propósito general fuertemente tipado, que se considera interesante por sus funcionalidades para definir transformaciones de modelos. Desarrollado por el Institute of Mathematics and Computer Science, University of Latvia, su última actualización es de Octubre de 2007. Este lenguaje podría ser considerado un "assembler" de los lenguajes de transformación.

Se utiliza metamodelado basado en MOF, el cual se define en archivos de texto con una sintaxis propia de lenguaje L0 [6]. Es un lenguaje básico y simple, cuya sintaxis puede observarse en la Figura 29.

Los constructores del lenguaje específicos para el nivel de metamodelado permiten dinámicamente crear, borrar y modificar elementos del metamodelo. Además brinda la posiblidad de iterar a través del metamodelo.

La definición de trasformaciones en Lx Family se realiza siguiendo una estructura muy similar a la de otros lenguajes de propósito general. Los diseñadores del lenguaje se enfocaron en diseñar los constructores del lenguaje de forma tan simple como sea posible. Básicamente proveen al programador la habilidad de:

- Iterar en los modelos, ya sea recorriendo los links u objetos del modelo.
- Creación y eliminación de links y/o objetos del modelo.
- Leer o escribir valores de atributos de los objetos del modelo

Al ser un lenguaje de propósito general, el estilo en la definición de una trasformación es muy similar a los mismos y forma imperativa. En la Figura 30 se puede observar la estructura de un programa en Lx.

Lx permite modularización del código de las transformaciones, permitiendo a los programas invocar subprogramas del mismo lenguaje L0.

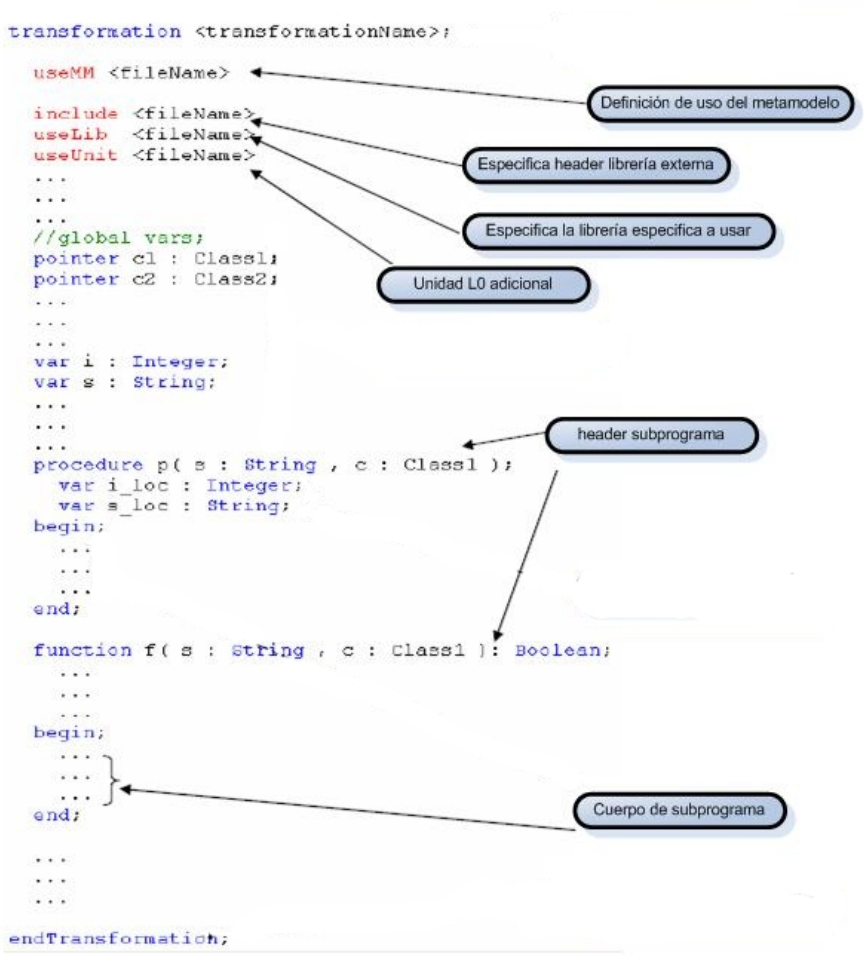


Figura 30: Programa en Lx

Otro aspecto a destacar de la familia Lx es su compilador. Lx posee sentencias especiales para el compilador denominadas *pragma*. También llamadas pseudo-instrucciones, las instrucciones pragma no cambian el significado del programa sino que simplemente envían información al compilador para invocar plugins. Estos constructores están basados en el proyecto MozartDev [**?**].

Un uso del lenguaje Lx se puede ver en MOLA, el cual genera código C# o Java, usando como representación intermedia Lx.

Epsilon Transformation Language (ETL) fue construído sobre Epsilon Object Language (EOL) y forma parte del proyecto Epsilon ([3]) iniciado en Octubre de 2005. Epsilon es una plataforma que permite construir tareas para manipular modelos en diferentes formas.

ETL permite definir transformaciones que toman varios modelos de entrada y producen varios modelos de salida. Tanto los modelos de entrada como los de salida pueden ser consultados por las reglas de transformación. Además es posible especificar para cada modelo si sus contenidos deben ser preservados de forma de evitar sobreescrituras [7].

En la Figura 31 se puede ver la sintaxis de una regla de transformación. Las reglas de transformación se se agrupan en módulos y se identifican con un nombre único dentro de los mismos. Al tratarse de un enfoque híbrido, las reglas pueden ser declarativas con un cuerpo imperativo (en la figura se representa como un conjunto de sentencias al final). Las reglas pueden ser reusadas mediante el mecanismo de herencia múltiple *extends* y la aplicación de *overriding*. Además se pueden definir guardas para la ejecución de las mismas expresadas en EOL.

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
 transform <sourceParameterName>:<sourceParameterType>
 to (<rightParameterName>:<rightParameterType>
 (, <rightParameterName>:<rightParameterType>)*
 (extends (<ruleName>,)*<ruleName>)? {

 (guard (:expression)|({statement+}))?

 statement+
}
```

Figura 31: Sintaxis de las reglas de transformación

Un ejemplo de regla aplicado al caso de estudio referenciado en este documento puede verse en la Figura 32 (la especificación completa se encuentra disponible en [3]). En este ejemplo se emplea una guarda y la operación definida por el usuario *toDbType*.

```
-- Transforms a single-valued attribute
-- to a column
rule SingleValuedAttribute2Column
  transform a : OO!Attribute
  to c : DB!Column {

  guard : not a.isMany

  c.name := a.name;
  c.table ::= a.owner;
  c.type := a.type.name.toDbType();
}
```

Figura 32: Ejemplo de regla de transformación

EOL ofrece mecanismos para controlar el orden de ejecución mediante bloques PRE y POST, y la posibilidad de estereotipar las reglas como LAZY.

Finalmente, al construirse sobre EOL hereda las siguientes características (puede verse la lista completa en [3]):

- construcciones típicas de los lenguajes de programación como iteraciones y variables,
- maneja un subconjunto de operaciones de OCL,
- permite la creación e invocación de objetos Java,
- brinda la posibilidad de interacción con el usuario durante la ejecución de la transformación.

Herramientas

ETL se distribuye dentro de Epsilon como un conjunto de plug-ins de Eclipse. Sin embargo, es posible utilizar ETL dentro de una aplicación Java sin soporte Eclipse.

El soporte brindado al usuario se basa en el libro de referencia de Epsilon [7] y en un foro de usuarios.

ATL (ATLAS Transformation Language) [2] es un lenguaje de transformación bajo los lineamientos del QVT RFP [8]. Desarrollado como parte de la plataforma AMMA (ATLAS Model Management Architecture), es clasificado como un lenguaje de transformación híbrido, que combina constructores tanto declarativos como imperativos.

La sintaxis abstracta de ATL es especificada mediante un metamodelo MOF, y provee de lenguajes en modo textual y gráfico para representar las reglas de transformación del lenguaje. Es un proyecto con constantes actualizaciones, foros, ejemplos, reportes de bugs, etc.

ATL aplica un patrón de transformación de modelos que se puede observar en la Figura 33.

La transformación está basada en su definición mma2mmb.atl escrita en ATL, donde la propia transformación puede ser vista como un modelo. Los modelos origen, destino y la transformación están basados en sus metamodelos MMa, MMb y ATL, respectivamente. Estos metamodelos respetan la especificación MOF. Un patrón define una estructura que separa sintácticamente la definición del *lado izquierdo* de la transformación del *lado derecho* de la misma, sin posibilidad de actualizaciones in-place[5].

En ATL se puede definir una transformación de un conjunto de modelos origen a un conjunto de modelos destino. Los metamodelos de cada modelo origen y de cada modelo destino deben ser especificados y deben estar disponibles para que la transformación pueda ser ejecutada. ATL soporta cualquier modelo que pueda ser definido usando metamodelado MOF, incluyendo meta-metamodelos.

[5] En [10] se plantea una idea para simular la substitucion in-place, copiando el modelo origen a uno nuevo, que será el modelo destino, para luego realizar las transformaciones en este último.

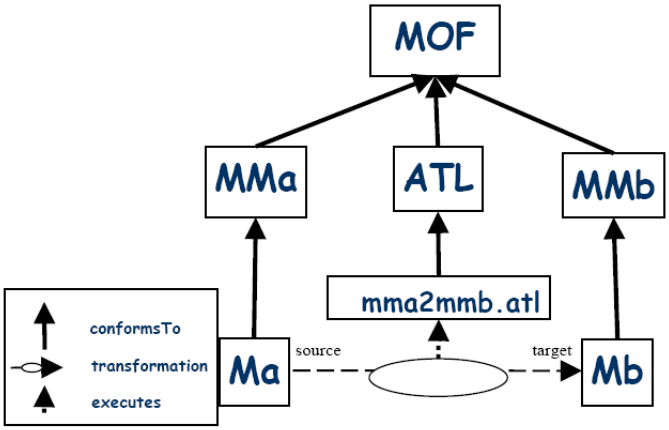


Figura 33: Esquema de enfoque de ATL

Las definiciones de transformaciones en ATL se organizan en módulos. Un módulo contiene una sección de cabezal obligatoria, una sección de imports, un número de *helpers* (el concepto helper se define en [46]) y reglas de transformación. La sección del cabezal da el nombre al módulo de transformación y declara los modelos de origen y los modelos de destino (Figura 34).

Un helper puede ser especificado en base a un tipo de OCL o a un tipo que proviene del metamodelo origen, dado que el modelo objetivo no es navegable. Los helpers pueden ser usados para definir operaciones que realicen navegaciones convenientes sobre el modelo origen o para definir pseudo atributos que faciliten la definición de las transformaciones.

La *regla de transformación* es el constructor básico en ATL para expresar la lógica de la trasformación. Las transformaciones en ATL son unidireccionales: operan sobre modelos origen de sólo lectura (read only) y producen un modelo objetivo de sólo escritura (write only). Durante la ejecución de una transformación los modelos origen pueden ser navegados, pero no se permiten cambios sobre ellos; en cambio los modelos objetivo no pueden ser navegados. Una transformación bidireccional puede ser implementada como una pareja de transformaciones, una para cada dirección.

Adicionalmente como se ha mencionado, ATL ofrece soporte para definiciones de estilo imperativo como otra característica del lenguaje. Las definiciones de forma imperativa pueden ser utilizadas dentro de bloques DO, o completamente separadas en helper rules.

La reglas del estilo declarativo son llamadas *matched rules*. Una matched rule está compuesta por un patrón origen y un patrón destino. El patrón origen especifica un conjunto de tipos origen (que provienen del metamodelo origen o del conjunto de tipos disponible en OCL) y una guarda (definida como una expresión booleana en OCL). El patrón de origen es evaluado para un conjunto de elementos en el modelo origen que cumple cierta restricción indicada en la regla de transformación. El patrón destino está compuesto por un conjunto de elementos, donde cada uno de estos elementos especifica un tipo destino (definido en el metamodelo destino) y un conjunto de *bindings*. Un binding especifíca una expresión cuyo valor se utilizará para inicializar el valor de la expresión que se encuentra a su izquierda. En la Figura 34 antes presentada, puede verse observarse un ejemplo.

La semántica de la ejecución de las reglas es en forma resumida la siguiente: dada una parte reconocida del modelo origen, mediante los patrones de origen, son creadas sus correspondientes partes en el modelo destino. Automáticamente se crean traceability links internos al motor de transformación. Estos links relacionan tres componentes: la regla, los elementos reconocidos y los nuevos elementos creados. Luego se evalúan y resuelven los bindings, para utilizar sus valores sobre los correspondientes elementos en el modelo

Figura 34: Ejemplo de transformación ATL

de destino. La estrategia de ubicación de reglas dentro del modelo a transformar es determinista para las reglas definidas de forma imperativa, y no determinista para las reglas definidas de forma declarativa (matched rules). La aplicación de reglas de forma interactiva no es soportada.

Con respecto a la planificación de la ejecución de reglas, ATL mezcla un enfoque implícito con uno explícito. Es decir, para las reglas definidas de forma declarativa es implícita, ya que no se tiene control sobre la ejecución de las mismas. Por otro lado, para reglas llamadas de forma impertiva (en particular para las denominadas, *called rules*) o cuando se utilizan estructuras de control, se define la ejecución de forma explícita. La selección de reglas que se ejecutarán se realiza mediante las guardas definidas en las reglas. Además, el motor soporta recursión y la separación en reglas por fases para ejecución concurrente.

La discusión sobre ATL presentada en este documento está basada fundamentalmente en [32], donde se puede encontrar ejemplos más extensos y un análisis más profundo del lenguaje ATL.

Herramientas

La actual herramienta de desarrollo para ATL está basada en el IDE Eclipse. Este incluye un editor que provee resaltado de sintaxis, vista de los modelos donde se está trabajando, reporte de errores, etc. El IDE presenta una interfaz similar a la de otros lenguajes para la utilización del debugger de ATL. En el sitio del proyecto se pueden encontrar tutoriales, documentación, ejemplos, manuales de usuario, etc.

YATL (Yet Another Transformation Language) YATL es clasificado como un lenguaje híbrido diseñado en base a la propuesta QVT de la OMG. Es descrito por un metamodelo MOF que especifica su sintaxis abstracta y una sintaxis concreta textual.

Una transformación expresada en este lenguaje describe un mapeo entre metamodelos origen y destino expresados en MOF. El motor de transformaciones utiliza este mapeo para generar un modelo destino a partir de un modelo origen, donde ambos metamodelos pueden coincidir. YATL utiliza los conceptos de repositorio y warehouse para almacenar las instancias de los modelos origen y destino. Estos conceptos son implementados utilizando KMF-Studio, una herramienta incluida en el KMF (Kent Modelling Framework).

Cada transformación consiste de una o más reglas de transformación, donde cada regla está formada por dos partes: LHS y RHS.

- La LHS de una regla describe los patrones que deben cumplir los elementos del modelo origen para que se pueda aplicar la regla. Esta parte es especificada usando expresiones escritas en OCL o en código nativo (Java, C#, scripts). Esta aproximación permite crear filtros que incluyan información de modelado y propiedades dependientes de la plataforma, lo que aumenta la potencia del lenguaje [41]. La naturaleza declarativa de YATL proviene de la descripción de la parte izquierda de las reglas y de las expresiones OCL.

- La RHS especifica operaciones de creación, borrado y modificación del valor de una propiedad, sentencias de declaración, iteración y condicionales. Este conjunto de operaciones y declaraciones son ejecutadas secuencialmente en el orden especificado para crear la instancia del modelo destino. Estos constructores utilizan expresiones OCL para especificar operaciones básicas (ej: la suma de valores enteros). La naturaleza imperativa de YATL proviene de estos rasgos de la especificación de la parte derecha de las reglas. Una regla es invocada explícitamente utilizando su nombre y sus parámetros. El proceso para aplicar estas reglas debe ser determinístico.

Un programa YATL consiste de una o más unidades de traducción separadas en archivos. Cuando un programa es procesado, todas las unidades de traducción son procesadas, donde además cada unidad puede depender de otra. Estas unidades pueden contener consultas, vistas y transformaciones bajo un determinado namespace. El uso de namespaces facilita la resolución de colisiones de nombres en transformaciones de gran escala, la organización de los programas y el reúso de transformaciones. Así la reusabilidad de código es soportada en YATL, pero aún no cuenta reglas abstractas, virtuales o sobrecargadas.

YATL no cuenta con soporte para multidireccionalidad, sólo transformaciones unidireccionales son soportadas. Permite la definición de transformaciones en donde el modelo origen es idéntico al modelo destino (actualizaciones in-place). Así, una transformación YATL puede ser utilizada para cambiar valores de una propiedad o añadir o eliminar objetos del modelo que se está transformando. En estos casos particulares, para evitar comportamientos no deseados el motor de transformación solo aplica las reglas de transformación a los elementos que originalmente formaban parte del modelo origen, omitiendo los nuevos objetos creados.

Este lenguaje no chequea implícitamente si la instancia del modelo origen o la instancia del modelo de destino generada están bien formadas. El mecanismo de consultas que usa construcciones del estándar OCL provisto por el lenguaje puede ser utilizado por el desarrollador para revisar las pre y post condiciones asociadas a la transformación. Esto puede ser realizado antes y después de ejecutada la transformación.

El lenguaje no provee soporte para incrementalidad automática. No puede detectar automáticamente si las instancias de los modelos de origen o destino sufren cambios incrementales. Es tarea del programador mantener las trazas de los cambios. La versión actual de YATL soporta solamente trazabilidad explícita, la información de trazas es almacenada utilizado un operador llamado "track".

Herramientas

YATL Studio es una herramienta utilizada para transformar modelos de origen en modelos de destino utilizando transformaciones descritas en YATL. Forma parte de KMF (Kent Modellig Framework) desarrollado por la Universidad de Kent. La versión actual de la herramienta al momento de redactar este documento es la 1.0, desarrollada en marzo de 2004. YATL Studio puede ser descargado gratuitamente desde [17].

5. Conclusiones

Este documento presentó el marco teórico empleado por numerosos trabajos para analizar y clasificar enfoques de transformación de modelos. El análisis desarrollado permite extraer conclusiones sobre las virtudes de cada de uno de éstos y cómo la combinación de los mismos produce herramientas eficientes para el desarrollo de software guiado por modelos. Finalmente, se expusieron características de los enfoques que pueden servir de guía para la elección de un enfoque en un contexto dado. En otras palabras, dada las características de una transformación a desarrollar, es posible determinar si se requiere por ejemplo de un enfoque que permita la multidireccionalidad de la aplicación de las reglas.

Si bien la lista de lenguajes analizada no es completa, brinda un panorama extenso de los principales trabajos en esta área y de las aplicaciones actuales. Se pueden apreciar como los lenguajes incorporan elementos comunes como mecanismos de trazabilidad que seguramente sean estandarizados en un futuro próximo.

Se presentó QVT como una especificación estándar de transformación de modelos. Esta especificación es el resultado de varios años de trabajo y cooperación del OMG y constituye una referencia para el desarrollo de la mayoría de los lenguajes de transformación de modelos.

Los lenguajes declarativos resultan una elección natural para escenarios donde los metamodelos de entrada y de salida poseen una estructura similar y consecuentemente permiten realizar un mapeo. Si bien los lenguajes imperativos permiten manejar un mayor número de escenarios, poseen un nivel de abstracción menor. Los lenguajes híbridos combinan ambos enfoques para sumar sus ventajas. Resulta claro que los lenguajes híbridos juegan un rol dominante en los trabajos recientes sobre transformación de modelos, por lo que es probable que futuros trabajos continúen en esta línea.

Como se mencionó, ATL se basa en la especificación de QVT y sigue la línea de los enfoques híbridos. Actualmente es uno de los más utlizados y difundidos, contando con una gran cantidad de publicaciones relacionadas. Dentro del mismo enfoque, RubyTL es un lenguaje similar a ATL que a nuestro juicio presentará una mayor aceptación en la comunidad de desarrolladores. Al basarse en el lenguaje de propósito general Ruby, puede ser fácilmente incorporado por la comunidad.

La mayoría de las herramientas de transformación de modelos han sido desarrolladas como plug-in del IDE Eclipse. Al desarrollarse sobre esta plataforma con varios años de desarrollo y uso, se asegura cierta robustez para las heramientas. En general, las herramientas disponibles ofrecen características básicas como el reconocimiento de sintaxis y un debugger. Sin embargo, las herramientas cuentan con pocos años de desarrollo y aún existen funcionalidades deseables sin implementar como la detección de errores en la codificación de transformación.

Para culminar esta sección se presentarán en forma resumida las principales características de los lenguajes analizados. Para cada uno de ellos se indicarán los atributos explicitados en este documento, dejando en blanco aquellos para los cuales no se encontró información concluyente. El símbolo de tick indica que el lenguaje posee el atributo, mientras que la cruz indica que no lo posee.

			Kermeta	Tefkat	Mola	VIATRA	Atom3	QVT Relations	QVT Operational	RubyTL	Epsilon	ATL	YATL
Reglas de transformación	Separación sintáctica		✗	✓	✗	✗	✓			✓	✓	✓	✓
	Multidireccionalidad		✗[1]	✗		✗	✗	✓	✗	✗[1]		✗	
	Precondiciones		✓	✓		✓	✓	✓	✓		✓	✓	
	Parámetros			✓				✓		✓		✓	✓
	Estructuras intermedias		✓		✓	✓			✓	✓		✓	
	Reflection		✓	✓						✓		✓	
	Aspects		✓										
Relación orígen/destino	Nuevo destino		✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
	In-place			✗		✓	✓	✓	✓			✗	✓
Determinación de la ubicación	Determinista		✗		✓			✓	✓			✓	✓
	No determinista		✗			✓	✓			✓		✓	
	Interactiva		✗									✗	
Planificación de reglas	Forma	Implícita	✗	✓				✓				✓	
		Explícita	✓	✗	✓	✓[3]		✓	✓		✓	✓	✓
	Selección	Condición explicita	✗		✓							✓	
		No determinista	✗				✓						
		Resolución de conflictos	✗										
		Interactiva	✗				✓						
	Iteración de reglas	Recursión	✓									✓	
		Looping	✓		✓				✓	✓	✓		✓
		Fixpoint				✓	✓			✓[4]			
	Fases		✗							✓			
Organización de reglas	Mecanismos para modularización		✓		✓	✓		✓		✓	✓	✓	
	Mecanismos para reuso		✓	✓	✓	✓			✓	✓	✓	✓	✓
	Estructura organizacional	Orientada al origen		✓	✓							✓	
		Orientada al destino						✓					
		Independiente	✓										
Incrementalidad			✗[1]	✓[2]				✓	✗				✗
Multidireccionalidad			✗[1]	✗	✗	✗	✗	✓	✗	✗		✗	✗
Trazabilidad	Soporte Explícito		✓			✓							✓
	Soporte Implícito		✗	✓	✓			✓	✓			✓	

Figura 35: Tabla de comparación

Notas

1. Puede ser simulado
2. Desarrollado por proyecto externo
3. Planificación explícita externa mediante máquina de estados finita.
4. Mediante reglas copy_rule

Referencias

[1] AGE Trac. http://gts.inf.um.es/trac/age. Último acceso, Noviembre 2009.

[2] ATL Project. http://www.eclipse.org/m2m/atl/. Último acceso, Noviembre 2009.

[3] Epsilon Transformation Language. http://www.eclipse.org/gmt/epsilon/doc/etl/. Último acceso, Noviembre 2009.

[4] Stratego Program Transformation Language. http://strategoxt.org/. Último acceso, Noviembre 2009.

[5] Tefkat: The EMF Transformation Engine. http://tefkat.sourceforge.net/. Último acceso, Noviembre 2009.

[6] The Base Transformation Language L0. http://lx.mii.lu.lv/L0_CurrVers.pdf. Último acceso, Noviembre 2009.

[7] The Epsilon Book. http://www.eclipse.org/gmt/epsilon/doc/book/. Último acceso, Noviembre 2009.

[8] Object Management Group: OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP. http://www.omg.org/spec/QVT/1.0/PDF/, Octubre 2002. Último acceso, Noviembre 2009.

[9] D.H. Akehurst, B. Bordbar, MJ Evans, WGJ Howells, and K.D. McDonald-Maier. SiTra: Simple transformations in java. *Lecture Notes in Computer Science*, 4199:351, 2006.

[10] J. Bezivin, G. Dupe, F. Jouault, G. Pitette, and J.E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. 2003.

[11] J. Bezivin, B. Rumpe, A. Schurr, and L. Tratt. Model transformations in practice workshop. *Lecture Notes in Computer Science*, 3844:120, 2006.

[12] E. Börger and R.F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Verlag, 2003.

[13] Borland. Borland Together. http://www.borland.com/us/products/together/index.html. Último acceso, Noviembre 2009.

[14] C.C. Cleaveland and J.C. Cleaveland. *Program Generators with XML and Java with CD-ROM*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[15] J.S. Cuadrado, J.J.G. Molina, and M.M. Tortosa. RubyTL: un Lenguaje de Transformación de Modelos Extensible.

[16] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[17] Universidad de Kent. YATL Studio. http://www.cs.kent.ac.uk/projects/kmf/downloadLocation.html. Último acceso, Noviembre 2009.

[18] Universidad de Latvia. MOLA Home Page. http://mola.mii.lu.lv/index.html. Último acceso, Noviembre 2009.

[19] Universidad de Latvia. MOLA Tool. http://mola.mii.lu.lv/tool_description.html. Último acceso, Noviembre 2009.

[20] Universidad de Latvia. MOLA Tutorial. http://mola.mii.lu.lv/MOLA_Tutorial.htm. Último acceso, Noviembre 2009.

[21] Universidad de Latvia. The MOLA Language Reference Manual. http://mola.mii.lu.lv/mola2fin_refmanual.pdf. Último acceso, Noviembre 2009.

[22] M.S.C. de Tenerife. RubyTL a ATC: un caso real de transformación de transformaciones Jesús Sánchez Cuadrado Eduardo Victor Sánchez Rebull Jesús J. García Molina Antonio Estévez García. *Actas de Talleres de Ingeniería del Software y Bases de Datos*, 1(6):71, 2007.

[23] O. Díaz, A. Irastorza, J. Sánchez Cuadrado, and L.M. Alonso. From page-centric to portlet-centric Web development: Easing the transition using MDD. *Information and Software Technology*, 50(12):1210–1231, 2008.

[24] Eclipse. Eclipse M2M. http://www.eclipse.org/m2m/. Último acceso, Noviembre 2009.

[25] F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher, and V. Mahé. Kermeta Language, Reference Manual. *Internet: http://www. kermeta. org/docs/KerMeta-Manual. pdf. IRISA*, 2006.

[26] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. *Lecture Notes in Computer Science*, pages 90–105, 2002.

[27] ikv++ technologies. ikv++ technologies Home Page. http://www.ikv.de/. Último acceso, Noviembre 2009.

[28] ikv++ technologies. Medini QVT. http://projects.ikv.de/qvt/. Último acceso, Noviembre 2009.

[29] E.F. Inc. Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/. Último acceso, Noviembre 2009.

[30] IRISA. Kermeta. http://www.kermeta.org/. Último acceso, Noviembre 2009.

[31] AM Jimenez. *Change Propagation in the MDA: A Model Merging Approach*. PhD thesis, MS Thesis. University of Queesland, 2005.

[32] F. Jouault and I. Kurtev. Transforming models with ATL. *Lecture Notes in Computer Science*, 3844:128, 2006.

[33] S. Kent. Model driven engineering. *Lecture notes in computer science*, pages 286–298, 2002.

[34] A.G. Kleppe, J. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[35] M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. *Lecture Notes In Computer Science*, 3844:139, 2006.

[36] Tata Consultancy Services Ltd. Model Morf. http://www.tcs-trddc.com/ModelMorf/ModelMorf.htm. Último acceso, Noviembre 2009.

[37] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations. (04101), 2005.

[38] OMG MOF. 2.0 Query/View/Transformation Specification (2007).

[39] P.A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.M. Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, 2005.

[40] The University of Birmingham. Simple Transformer (SiTra). http://www.cs.bham.ac.uk/~bxb/SiTra.html. Último acceso, Noviembre 2009.

[41] O. Patrascoiu. YATL: Yet Another Transformation Language-Reference Manual Version 1.0. 2004.

[42] P. Queralt, L. Hoyos, A. Boronat, JÁ Carsí, and I. Ramos. Un motor de transformación de modelos con soporte para el lenguaje QVT relations. *Ill Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones (DSDM'06). Espana*, 2006.

[43] France Telecom R&D. SmartQVT. http://smartqvt.elibel.tm.fr/. Último acceso, Noviembre 2009.

[44] AM Reina, J. Torres, and M. Toro. Hacia Lenguajes de Metamodelado Orientados a Aspectos.

[45] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc. River Edge, NJ, USA, 1997.

[46] OMG Uml. 2.0 OCL Specification. *OMG Adopted Specification (ptc/03-10-14)*, 2003.

[47] VIATRA. VIATRA Known Uses. http://wiki.eclipse.org/VIATRA2_Known_Uses. Último acceso, Noviembre 2009.

[48] VIATRA. VIATRA User Guide. http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/viatratut2.pdf. Último acceso, Noviembre 2009.

[49] VIATRA. VIATRA2 Home Page. http://www.eclipse.org/gmt/VIATRA2/. Último acceso, Noviembre 2009.

[50] Eelco Visser. Program transformation with stratego/xt. rules, strategies, tools, and systems in stratego/xt 0.9. Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004.

[51] TU Wien. Language Jungle-An Evaluation and Extension of Existing Approaches. 2008.