

Implementación de un motor de transformaciones con soporte MOF 2.0 QVT

Padrón Lorenzo, J., García Luna, J.D., Sánchez Rebull, E.V., Estévez García, A.

Open Canarias SL, Santa Cruz de Tenerife, España

<http://www.opencanarias.com>

info@opencanarias.com

Resumen

El objetivo de este trabajo es describir la implementación de un motor de transformaciones de modelos que, desarrollado sobre la plataforma Eclipse y EMF, sea capaz de soportar los aspectos más importantes del futuro estándar QVT [17] de MDA [16]. El motor se basa en un lenguaje de transformaciones de bajo nivel llamado ATC, cuyas definiciones de transformación se basan en modelos serializables en XMI. Los lenguajes de alto nivel deberán compilarse hacia ATC para ser ejecutados por el motor. La inclusión simultánea de prestaciones tan potentes como trazabilidad y compilación incremental de modelos, conlleva unos retos de diseño y arquitectura del motor que trataremos de identificar y compatibilizar. Aprovechando la experiencia que nos ha aportado el desarrollo del framework BOA 1 [4] en su aplicación a proyectos reales, intentaremos dar solución a sus limitaciones a partir de un enfoque basado en la manipulación directa de objetos en memoria, en lugar del procesamiento de ficheros con XSLT.

1. Introducción

En este artículo describiremos las bases de funcionamiento de un motor de transformaciones de modelos. Este motor se basa en la plataforma Eclipse y el proyecto EMF [12] como entorno de desarrollo en Java para describir, inspeccionar y modificar modelos PIM y PSM.

Empezaremos con una presentación de MDA y sus características, y proseguiremos con una puesta al día de la situación del estándar QVT. Tras ello entraremos a discutir la tecnología adoptada, los aspectos más innovadores de nuestra aproximación, el porqué del lenguaje ATC, y las dificultades encontradas y previstas, junto con las decisiones de diseño y arquitectura que dirigen el desarrollo del motor.

1.1. MDA, PIM's, PSM's y transformaciones

Nos situamos en la infraestructura de MDA, con su arquitectura piramidal para la definición formal de lenguajes de modelado, estructurada en cuatro capas o niveles, donde el nivel superior o M3 se corresponde con el lenguaje madre, llamado MOF, MetaObject Facility.

Los modelos a manipular se concentran en la capa M1. Mediante el uso de lenguajes y la definiciones de reglas de transformación, los modelos originales, resultantes de la fase de análisis, se convierten a componentes finales desplegados en las distintas plataformas de ejecución, en un proceso donde se persigue el mayor grado de automatización posible. En nuestro trabajo, la transformación de PIM's a PSM's se producirá en sucesivas etapas de refinamiento que irán enriqueciendo los modelos hasta dotarlos con toda la información deseada.

El motor de transformaciones termina sus servicios con la obtención de estos modelos ricos en detalle. Más allá entra en escena otro proceso alternativo independiente del motor que recoge la información de los modelos finales y genera todo un arsenal de ficheros destino que constituyen nuestro sistema de software final.

Típicamente, estos ficheros contendrán código fuente, información de despliegue, esquemas de bases de datos, y cualquier otro elemento que forme parte del sistema a construir.

El motor incluye soporte para interactuar con un lenguaje de transformaciones de bajo nivel, llamado ATC, que funciona como una capa aislante con respecto al resto de posibles lenguajes de transformaciones a utilizar.

Este motor se integra en un proyecto denominado BOA 2, que incluye además la posibilidad de crear y editar nuevos modelos y metamodelos,

mediante editores de diagramas para la generación manual de modelos EMF y su edición.

2. Antecedentes

Mientras los actuales estándares de OMG, tales como MOF y UML, nos proveen de una base bien establecida para las definiciones de PIM's y PSM's, no existe tal base para las transformaciones de PIM's a PSM's.

En 2002, con el fin de solventar esta circunstancia, OMG inició un proceso de estandarización proponiendo un RFP (Request for Proposal) sobre MOF 2.0 QVT (Queries/Views/Transformations). Este proceso de recopilación de propuestas, podrá eventualmente llevar a OMG hacia un estándar para la definición de transformaciones de modelos MOF, interesante no sólo para transformaciones de PIM's a PSM's, sino también para definir vistas sobre modelos y sincronización entre ellos.

Ya sea por necesidad o a consecuencia de la propuesta de OMG, se han realizado múltiples aproximaciones y proposiciones a la transformación de modelos. Podemos encontrar numerosas publicaciones donde se abordan diferentes propuestas y soluciones: GReAT [1], UMLX [6], ATOM [8], VIATRA [3], BOTL [9] o ATL [21], así como muchas otras, son ejemplos de ello.

Algunas aproximaciones han sido desarrolladas en código abierto (AndroMDA [7], JET, FUUT-je, GMT [13], ...), enriqueciendo las diferentes posibilidades que existen para solucionar el problema de las transformaciones. También existen herramientas MDA comerciales, algunas de ellas ya conocidas, que han desarrollado soluciones propias, tales como ArcStyler [14], XDE [18], Codagen Architect [10], OptimalJ [11], etc.

2.1. MOF 2.0 QVT: búsqueda de un estándar.

El planteamiento QVT se basa principalmente en la definición de un lenguaje para las consultas (Queries) sobre los modelos MOF, la búsqueda de un estándar para generar vistas (Views) que revelen aspectos específicos de los sistemas modelados, y finalmente, la definición de un lenguaje para la descripción de transformaciones (Transformations) de modelos MOF.

En su propuesta para la estandarización de las transformaciones, OMG especifica una serie de requisitos obligatorios y opcionales, además de

unos puntos a discutir por los proponentes. Dentro de los requisitos obligatorios se encuentran:

- Se debe definir un lenguaje para las queries, que nos permita filtrar y seleccionar elementos de los modelos. (Parece que va a ser una extensión de OCL 2.0).
- Se debe definir un lenguaje para las transformaciones.
- Ambos lenguajes deberán estar definidos con meta-modelos MOF versión 2.0.
- El lenguaje de transformaciones:
 - Debe servir para generar vistas de un meta-modelo.
 - Debe ser capaz de expresar toda la información necesaria para generar automáticamente la transformación de un modelo origen a uno destino.
 - Debe soportar cambios incrementales en un modelo origen que se vean reflejados en el modelo destino.
- Todos los mecanismos que se utilicen deben operar sobre modelos instancias de meta-modelos MOF versión 2.0.

Requerimientos Opcionales

- Bidireccionalidad en las transformaciones. Dos aproximaciones:
 - Las transformaciones se definen simétricamente. Una única transformación que nos sirva para las dos direcciones.
 - Se realizan dos definiciones de la transformación, donde una es la inversa de la otra.
- Posibilidad de trazabilidad (debugger).
- Reutilización y extensibilidad, entre lo que se podría incluir:
 - Herencia (Extensibilidad)
 - Modularización- Reutilizar transformaciones más simples en transformaciones más complejas.
 - Soporte de plantillas o patrones de transformación.
- Transformaciones transaccionales en las que partes de la definición de una transformación sean capaces de realizar acciones de "commit" o "rollback" durante la ejecución.
- Uso de parámetros adicionales, no contenidos en el modelo origen, como entradas a la definición de la transformación, para generar el modelo destino. Adicionalmente, las propues-

tas podrían permitir la definición de valores por defecto para esos parámetros.

- Permitir definiciones de transformaciones in-situ, para la actualización de modelos donde el modelo destino sea el mismo que el origen.

Desde el punto de vista de los usuarios, se detecta la necesidad de soportar una serie de características funcionales, que comprenden la especificación de una transformación bi-direccional, trazable, fácil de usar y de entender, personalizable y funcional.

Varios contratiempos han frenado y prorrogado el desarrollo de la especificación y han evitado que ya se disponga de un estándar. La duda sobre si la aproximación debe ser de una naturaleza declarativa, imperativa o un híbrido entre ambas, el uso o no de OCL 2.0 como lenguaje para efectuar consultas, la necesidad o no de trazabilidad, la necesidad de interoperabilidad, si la aparición de transformaciones en cajas negras limitará el intercambio de modelos de transformación, la atomicidad de las transformaciones y la capacidad de roll-back, la existencia o no de una concordancia con la definición de mappings en CWM, unido a las necesidades que los usuarios van a requerir de la especificación son el núcleo de discusiones que han impedido la consolidación de un criterio unificado sobre la esencia de un estándar definitivo.

Si observamos la evolución que han ido adoptando las propuestas presentadas, vemos que en 2002, tras la aparición de la solicitud de OMG, se recogieron 8 propuestas de diversas organizaciones y empresas, que a lo largo de estos años han ido convergiendo hasta llegar a una única propuesta, expuesta por el grupo QVT-Merge [19], que aún a prácticamente todos los proponentes iniciales, y está apoyada por un grupo importante de entidades, entre ellas, varias universidades. La diversidad de propuestas iniciales junto con las posteriores convergencias y revisiones de las propuestas han sido otro factor que ha prolongado la consecución del estándar.

La especificación QVT propuesta tiene una naturaleza híbrida entre declarativa/imperativa, con la sección declarativa dividida en una arquitectura de dos niveles: Relations y Core, con sus lenguajes correspondientes. Existen dos mecanismos para invocar una implementación imperativa de las transformaciones: un lenguaje estándar (Operational Mappings), y soporte para enlazar

con código no estándar (Black Box MOF Operation). En cuanto al lenguaje para las consultas (queries), tal como solicita OMG, la propuesta apoya el uso de OCL 2.0, con ligeras extensiones.

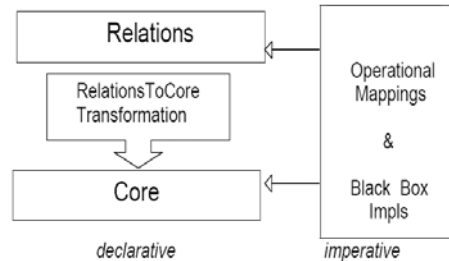


Figura 1. Relaciones entre metamodelos de los lenguajes QVT

En definitiva, nos encontramos ante la descripción de tres lenguajes para especificar transformaciones. Estos lenguajes soportan la composición, especialización de reglas, escalabilidad, modularización, reutilización, etc.

Esta propuesta del grupo QVT-Merge, que tiene muchas posibilidades de evolucionar en lo que será el futuro estándar QVT reclamado por OMG, nos ha servido de referencia durante el desarrollo de nuestro trabajo. El motor, que se independiza en principio de cualquier lenguaje de transformación externo, ha sido preparado para compatibilizar la gramática de uno de los lenguajes de la última propuesta de estandarización de QVT. Más adelante veremos cómo.

2.2. Antecedentes BOA 1

BOA 1 fue diseñada como una herramienta para reducir drásticamente el tiempo empleado en el desarrollo de aplicaciones, pudiendo diseñar en poco tiempo un modelo que, después de ser compilado por BOA 1, se convierte en una aplicación de tres capas preparada para ser desplegada sin necesidad de realizar ningún cambio o retoque.

En BOA 1 se destacan dos grandes conceptos: la aplicación, integrada en la plataforma Eclipse a través de un plug-in; y los cartuchos que contienen las plantillas de transformación, librerías y código auxiliar necesario. El plug-in de BOA 1 permite crear proyectos específicos e iniciar el proceso de compilación del modelo, para lo que se descomprime el correspondiente cartucho, y se aplican las correspondientes plantillas de transformación a dicho modelo, obteniendo como

resultado las clases y las páginas web que formarán la aplicación.

El modelo UML que alimente BOA 1 debe estar creado con cualquier herramienta de modelado que soporte el estándar XMI, y tiene que usar un profile UML concreto, con el que BOA 1 podrá distinguir cada uno de los diferentes entes que pueden intervenir en el modelado de la aplicación. En este modelo podemos definir toda la aplicación: con un diagrama de clases se crea el modelo conceptual, que simbolizará el conjunto de entidades de negocio y su representación en la base de datos; la interfaz web sigue un patrón MVC2; para definir el comportamiento dinámico de la aplicación se usan diagramas de secuencias o una combinación de diagramas de estados y del lenguaje ASL (Action Specification Language) [5], que se convierte en código fuente.

Los cartuchos de BOA 1 mencionados contienen un fichero que hace las veces de descriptor de despliegue, indicando dónde han de almacenarse los distintos ficheros generados. Las plantillas de transformación están escritas en lenguaje XSLT porque es el motor de transformación estándar para ficheros XML. Inicialmente, el fichero XMI con el modelo de entrada pasa por una etapa de filtrado. Se obtiene una versión simplificada del modelo a partir de la cual, se identifican las distintas unidades de generación presentes, se resuelven dependencias y, BOA 1 escoge las plantillas del cartucho a aplicar, según sea el estereotipo que identifique a la unidad de generación que se esté estudiando en ese momento, obteniendo como resultado final el conjunto de objetos, páginas web y demás artefactos que formarán el 100% de la aplicación modelada.

2.3. Ventajas e inconvenientes BOA 1

La indiscutible ventaja de BOA 1 es que hemos conseguido crear una herramienta que se ajusta en gran medida a las directrices de MDA, que consigue, de forma rápida, sencilla y precisa, crear aplicaciones completas partiendo de modelos UML.

Las desventajas también son importantes: la creación de un nuevo cartucho obliga a volver a definir todas las plantillas de transformación, ya que éstas son apenas reutilizables. Como el lenguaje XSLT es bastante duro y oscuro, también son difíciles de mantener. Con este tipo de planti-

llas no es posible realizar compilaciones incrementales del modelo, por lo que si realizamos algún pequeño cambio en el modelo origen, debemos generar por completo el código de nuevo.

3. Base tecnológica

Nuestro motor de transformaciones se implementa como un surtido de plug-ins para la plataforma Eclipse, igual que BOA 1. No obstante, en esta ocasión nos basaremos en el proyecto Eclipse Modeling Framework (EMF).

3.1. Eclipse y EMF

Eclipse es un entorno de desarrollo que en sí mismo contiene poca funcionalidad como IDE, pero que, mediante su potente y flexible sistema de plug-ins, es capaz de extender su funcionalidad de forma ilimitada sin incurrir en penalizaciones significativas de rendimiento.

EMF es un framework que ostenta toda la funcionalidad necesaria para el tratamiento de modelos al estilo MOF. Está concebido en torno a un metamodelo de la capa superior M3 llamado *Ecore*, que sustituye eficazmente al lenguaje subconjunto de MOF 2.0, EMOF (Essential MOF), porque EMF soporta la lectura y escritura transparentes de serializaciones EMOF y la conversión de modelos EMOF a Ecore.

EMF distingue e identifica metamodelos mediante el concepto de espacios de nombres determinados por URIs. El formato de almacenamiento de modelos usado por EMF sigue el estándar XMI, lo que garantiza la interoperabilidad con herramientas y aplicaciones externas que también soporten este formato. También resuelve referencias cruzadas a elementos de modelos externos mediante el soporte de objetos proxy y 'lazy initialization'.

Con todas estas características de manipulación de modelos en Eclipse y EMF, y otras prestaciones como la facilidad de crear plug-ins personalizados que se integran de forma transparente con el entorno, nos decantamos por Eclipse y EMF como entorno de desarrollo, gracias también a la experiencia en Eclipse adquirida por Open Canarias con el desarrollo de BOA 1. Otras alternativas de interés incluyen MDR [15], que funciona directamente con MOF.

3.2. Manipulación de elementos en EMF

Ecore incorpora el mecanismo de reflectividad que lo equipara a EMOF. Los elementos de modelo del nivel M1 pueden ser manipulados dentro de EMF a través de la interfaz básica de soporte a la reflectividad que heredan de EObject, el tipo raíz del metamodelo de Ecore.

Cada metamodelo en EMF está compuesto por uno o varios paquetes (EPackage) que son los que contienen los metaelementos clasificadores de referencia (EClassifier) para inspeccionar los modelos y sus elementos.

Existen dos formas de tratar metamodelos y modelos en EMF: estática y dinámica. Sin entrar mucho en detalle, la aproximación estática se basa en la disponibilidad de un plug-in que contiene interfaz e implementación Java para cada EPackage gestionado. Se dispone también de una EFactory para la creación de instancias de los distintos tipos de datos dispuestos en el EPackage. La aproximación dinámica se basa en la inspección ciega de los modelos tomando como referencia unos EPackage's cargados a partir de un fichero XMI, o contruidos en memoria en tiempo de ejecución, pero no registrados en Eclipse como plug-ins. Asimismo, los objetos creados de esta forma son EObject's, sin mayor especialización.

La aproximación dinámica es ligeramente más ineficiente que la estática, puesto que las propiedades de los distintos elementos de modelo se almacenan en listas genéricas, en vez de figurar directamente como campos de las clases que los contienen. Pero a su vez, es más flexible y desacoplada, y actualmente es el modo en que opera el motor. Así, no es necesario disponer del plug-in del EPackage para poder manipular modelos subordinados a él en nuestro entorno.

Dejando aparte la inmutabilidad, distinguimos tres acciones fundamentales de manipulación de objetos de modelo, y por ende, de las transformaciones:

- creación de objetos y vinculación a modelos,
- modificación del estado de los objetos,
- desvinculación de objetos del modelo.

Naturalmente, EMF ofrece soporte a los tres. La creación de objetos se maneja automáticamente con una EFactory (también disponible en la aproximación dinámica) por cada EPackage, encargada de la instanciación de elementos de modelo. Una vez creado, este objeto habrá de ser

vinculado al modelo, añadiéndolo a alguna propiedad de alguno de los elementos contenidos en él. Para la destrucción y desvinculación de elementos, basta con dereferenciarlos de los contenedores (listas, etc.) de los elementos del modelo donde estén ubicados. Quedarán desligados del modelo si, tras esta acción, ningún elemento los referencia. EMF mantiene de forma transparente la consistencia entre referencias opuestas, lo que simplifica mucho esta tarea. Para modificar propiedades, se invoca el método 'eSet()' de la interfaz EObject desde el objeto poseedor de la propiedad a modificar, la cual se identifica mediante un elemento del metamodelo, y su valor se sustituye por el nuevo valor proporcionado.

4. Planteamiento de la arquitectura

Nuestro nuevo motor deja atrás el uso de plantillas XSLT de BOA 1 para la conversión de ficheros XMI, y se centra en una secuencia operativa que parte de una representación eficiente en memoria, proporcionada por EMF, de los modelos XMI. El motor de transformaciones tiene acceso a un repositorio de metamodelos con el cual puede validar las reglas de transformación y las instancias de modelos que intervienen en la ejecución.

4.1. Características soportadas

En cada etapa de transformaciones los modelos irán recorriendo distintos dominios, cada uno de ellos regido por uno o varios metamodelos, y se fusionarán o expandirán en varios conjuntos de modelos intermedios y finales.

Tiene que ser posible dar cabida en este motor a la resolución correcta y razonablemente eficiente de cualquier definición de transformaciones de modelos concebibles, desde las más simples hasta las más complejas: operar sobre los modelos de entrada y salida y hacer cambios precisos, crear o eliminar objetos, modificar sus propiedades, crear nuevos modelos a partir de otros existentes, convertir un modelo a varios, cada uno perteneciente a una plataforma concreta, o simplemente chequear la consistencia en relación a los contenidos de modelos definidos en dominios distintos, etc. Al terminar, el nuevo contenido de los modelos modificados será salvado de nuevo a un fichero XMI.

La capacidad de automatización y reutilización de las transformaciones se consigue mediante la ejecución repetida del mismo conjunto de reglas. Esta automatización no impedirá la posible introducción manual de parámetros de configuración o de control de la ejecución de la transformación. Esta información paramétrica adicional podrá ser serializada en forma de valores por defecto para usos posteriores.

Seguimos una aproximación elaboracionista, donde el proceso de transformación se subdivide en varios pasos más o menos secuenciales, y dejamos atrás la alternativa traslacionista, utilizada en BOA 1, donde la conversión tiene lugar de un solo golpe, la información original se convierte directamente en los artefactos resultantes finales.

Para cada etapa de refinamiento intermedia, las reglas de transformaciones introducidas describen el tratamiento que el motor ha de dar a los modelos origen y destino. Los artefactos obtenidos tras la ejecución de cada etapa, serán los modelos destino resultantes, que a su vez podrán servir como modelos de entrada en subsiguientes etapas dentro de la cadena de transformaciones. Esta aproximación es indispensable para dar cabida a prestaciones tales como actividades transaccionales o compilación incremental, ausentes en BOA 1 y muy importantes para una operativa flexible desde el punto de vista del usuario.

A diferencia de BOA 1, las competencias del motor de transformaciones de modelos se centran en este caso con la obtención de un PSM con todo lujo de detalles. El proceso posterior de conversión de estos modelos diagramáticos finales, ricos en detalle, a ficheros tipo código fuente, etc., se ejecuta aparte.

A pesar de que un formato textual puede ser considerado también un modelo, para esta etapa se planea una aproximación distinta en la cual no participa el motor, y que se basa en el uso de metamodelos de los lenguajes de programación finales y plantillas JET, una por cada metamodelo, responsables de la generación del código fuente final o cualquier otro formato necesario.

4.2. Soporte a características QVT

Las características QVT que trataremos aquí con cierto detalle son la trazabilidad, compilación incremental y transformaciones con carácter transaccional, y en menor grado, la bidireccionali-

dad. Intentaremos dar soporte integrado a todas ellas desde el motor, lo cual nos impone unos requisitos de diseño y arquitectura de nuestro entorno que deben encajar convenientemente.

Todas estas características tienen un nexo común. Los fundamentos de soporte se concentran en la capacidad para registrar los cambios acaecidos en los modelos durante la ejecución de transformaciones. Podemos gestionar transacciones y grupos de transacciones si somos capaces de registrar un histórico de modificaciones sufridas por los modelos y hacer back-up de los valores antiguos modificados. Los cambios incrementales manuales en origen o destino tras una transformación también pueden reconciliarse de forma óptima a partir de la información registrada, aunque ello requiere un tratamiento inteligente de dicha información, junto con un análisis profundo del contenido semántico de la definición de transformación y sus implicaciones al ser ejecutada.

El caso de la bidireccionalidad también tiene relación con estos registros. Tal bidireccionalidad no es posible por sí sola si la transformación no tiene una naturaleza bidireccional y se pierden datos irrecuperables al ejecutarla en una dirección. Por ejemplo, cuando un string en origen se modifica para eliminar sus espacios y sustituir vocales acentuadas por vocales normales. Sólo podemos devolver los modelos de origen a su estado original al ejecutar la transformación recíproca, si disponemos de los datos originales.

Entonces, cada cambio que vayan sufriendo los modelos durante las transformaciones deberá ser registrado para su posterior consulta, o para habilitar actividades de 'rollback', etc. En la propuesta del grupo QVT-Merge, y en relación, sobre todo con los lenguajes declarativos, se emplean unas entidades de registro denominadas clases *trace*, basadas en vínculos.

Afortunadamente, EMF dispone de notificaciones asociados a los posibles cambios de contenido en los modelos. Tan sólo hace falta registrar en el entorno unos observadores que reciben automáticamente las notificaciones con la información de modificación y las registran convenientemente.

Esta información de consulta sirve para implementar una trazabilidad capaz de recuperar información de los elementos que hayan participado en uno o varios de estos bloques transformados. Al mismo tiempo nos permite almacenar estados previos de los objetos participantes que

nos ayudarán en el soporte a actividades transaccionales. También respalda la forma cómo responderá el motor ante posibles cambios incrementales introducidos manualmente en los modelos, origen o destino, después de haber efectuado una transformación. Esta información permitirá acelerar la identificación y actualización de las zonas afectadas por los cambios a efectuar.

4.3. ATC, un lenguaje intermediario

Conviene que el motor tenga la mínima dependencia posible con respecto a los lenguajes de transformación para los que se quiera dar disponibilidad al arquitecto transformador. Intentamos minimizar el impacto de posibles actualizaciones de su gramática y facilitar la labor de incorporación de lenguajes nuevos al conjunto sin tener que efectuar cambios internos en el motor.

Para ello colocamos un lenguaje de transformaciones propio como intermediario y aislante entre el motor y el resto de lenguajes. Este lenguaje, bautizado con el nombre de Código de Transformación Atómica, Atomic Transformation Code (en adelante, ATC), es el único con quien se entiende el motor de transformaciones. Cualquier lenguaje de transformaciones que se quiera usar, deberá poder traducir su gramática al formato ATC.

4.4. Características del lenguaje ATC

ATC es un lenguaje imperativo neutro, creado para el tratamiento automático y eficiente de transformaciones por parte del motor a bajo nivel. Sus instrucciones representan una especie de byte-code con un mínimo grado de abstracción, por eso decimos que es un lenguaje atómico. ATC no requiere altos grados de modularidad y reutilización, pues no está pensado para ser utilizado en explotación por los arquitectos. Para eso ya existen varias opciones, como los lenguajes descritos en la última propuesta QVT, que han sido diseñados pensando en la comodidad del usuario.

Naturalmente, ATC está dotado de una expresividad que incluye en su núcleo los mecanismos fundamentales de modificación de modelos antes descritos, que ya están resueltos dentro del motor para el framework EMF. El resto de funcionalidad se construye a partir de su set de funcionalidad semántica. Las acciones programadas en ATC

están muy alineadas con el framework EMF, para conseguir el máximo grado de eficiencia computacional. En realidad, la porción de ATC relacionada con EMF es mínima. En suma, ATC es un lenguaje bastante arisco y “verbose”, poco o nada apto para ser tratado directamente por el usuario.

Por encima de ATC se canalizan los lenguajes de transformaciones externos que deban ser soportados en nuestro sistema. Para ellos, hay que compilar sus definiciones de transformaciones al formato ATC.

Una definición de transformación en ATC consiste en una jerarquía de objetos semánticos, cada uno con su propio estado, el cual comprende identificadores y otros componentes que les dirige a las partes concretas de los modelos donde actuar. Estas definiciones se representan en EMF como modelos, derivados del metamodelo ATC. Durante la ejecución se identifica el objeto raíz que representa la transformación, y se invoca su operación *main*, que contiene un cuerpo con semántica ATC, junto con la semántica de invocación a operaciones definidas dentro de la transformación, que a su vez pueden invocar otras en sucesión.

Todas las clases con contenido semántico ATC constan de un método con idéntica signatura: *IExpression act(TransformationContext tc)*

El parámetro *tc* contiene información sobre la operación en curso, vínculos entre identificadores y valores concretos accesibles en esta operación, parámetros, e información adicional necesaria durante la ejecución de la transformación.

Una vez el lenguaje esté estable se prevé que a partir de un modelo ATC se pueda generar código Java equivalente, cuya ejecución será bastante más eficiente que la profusión de objetos e invocación de métodos ATC actuales. La invocación de este código Java se alinearía con la de las *black-box operations*, lo cual no implica que el código tenga que estar inaccesible.

Una actividad muy común en la transformación o inspección de modelos es el recorrido del contenido de una lista. Por ejemplo, en UML encontramos la propiedad ‘ownedAttribute’ dentro de la categoría ‘Class’. Esta propiedad tiene multiplicidad mayor que uno, y suele implementarse como un contenedor más o menos genérico. En multitud de ocasiones hay que explorar todos los elementos de estos contenedores, por ejemplo, para comprobar si alguno coincide en nombre con

un string dado. Esta funcionalidad se recoge en una clase ATC llamada *ForEach*, muy semejante a la expresión *ForEachExp* que aparece en el lenguaje *OperationalMappings*.

Esta clase recibe un identificador para obtener la lista a recorrer, un identificador que servirá para vincular cada uno de los objetos extraídos, y finalmente un bloque semántico con las acciones que se ejecutarán sobre estos objetos, el cual podrá estar compuesto de porciones semánticas secuenciales, o incluso contener la invocación de una nueva operación.

```
- Entering: main
Assign: mySet
ForEach: mySet->p
  - Entering: encapAttrsInPackClasses
  EStrFeatGet: ownM=p.ownedMember
  ForEach: ownM->classMember
  if: true
  EStrFeatGet: a=c.ownedAttribute
  ForEach: a->b
    - Entering: privatizeAttribute
    center
    SetEEnum
    EStrFeatGet: aName = b.name
    Assign: attrName
    ToString: getName = getProp
    EStrFeatGet: atType = b.type
    ...
```

Figura 2. Ejemplo de salida por consola de una ejecución ATC

4.5. Planteamiento de la compilación a ATC

La obtención del modelo de definición ATC a aplicar al motor de transformaciones es un proceso automatizado de compilación hacia ATC de los lenguajes abstractos mediante el uso del generador SableCC [20]. La compilación se produce a nivel de definiciones de transformaciones y tiene lugar una única vez por cada definición. Tras efectuar la traducción, el modelo ATC resultante se almacena para su posterior invocación desde el motor.

Para la aproximación declarativa, QVT describe las transformaciones mediante el lenguaje Relacional. Para obtener el código en lenguaje Core equivalente a la transformación definida en lenguaje Relacional, usaremos un compilador generado a partir de la gramática detallada en el documento presentado por el grupo QVT-Merge. Este compilador respetará las reglas propuestas en este documento para obtener el código equivalente en lenguaje Core. Este código obtenido sirve como entrada a otro compilador generado también

a partir de la gramática Core especificada en el mismo documento, y cuyo cometido es obtener el modelo equivalente en ATC, compuesto por un conjunto jerárquico de objetos en memoria, que posteriormente EMF se encargará de volcar a un fichero XML.

La aproximación imperativa describe las transformaciones en el lenguaje *Operational Mappings*. Para obtener el código equivalente en ATC se usa un compilador generado a partir de la gramática del lenguaje *Operational Mappings*.

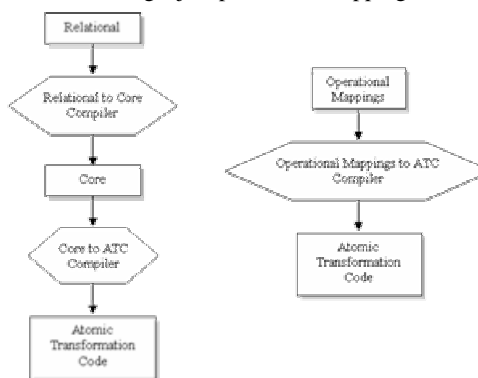


Figura 3. Esquema de los pasos para llegar al código de transformación atómica

Para compilar un lenguaje a ATC es imprescindible disponer de su gramática, en BNF o similares. Es un trabajo duro, más aún si el lenguaje origen es declarativo. Afortunadamente, el lenguaje Core explicita muchas de las actividades y decisiones condicionales a tomar según cada circunstancia. Además, tarde o temprano, el soporte a los lenguajes declarativos ha de ser contemplado. Nosotros hemos decidido ubicarlo en el contexto de la compilación, y dejar intacto el motor de transformaciones, que seguirá entendiéndose únicamente con ATC. Más adelante respaldaremos esta decisión.

4.6. Validación y depuración

Nos proponemos también incorporar dos factores de gran interés en el motor como son, por una parte, la validación de la consistencia sintáctica de las reglas de transformación, a partir de la gramática de cada lenguaje soportado y la capacidad para analizar su contenido, lo cual va implícito en el paso de traducción a ATC.

Y por otra parte, trataremos de ofrecer soporte para la depuración durante las ejecuciones de transformaciones. Por ejemplo, presentando información puntual de los elementos que van evaluándose, proporcionando una lista de consulta de todos los elementos tratados, identificando las expresiones concretas responsables de las distintas acciones que el motor va realizando, etc. De nuevo, la plataforma Eclipse nos marcará las pautas de compilación y depuración a seguir, gracias a unas prestaciones como son las *naturalezas* de los proyectos y la potencia de los *builders*, que servirán de envoltura a nuestro compilador para su integración en el IDE.

Para los casos declarativos, será difícil identificar las correspondencias entre las sentencias relacionales originales y el código atómico generado por ellas, que nos permitan sincronizar los puntos de interrupción activados en el código original con el conjunto de instrucciones ATC asociadas.

Tendremos varios ficheros fuente con definiciones de transformaciones (declarativas o imperativas). El builder se encargará de detectar ficheros modificados e informar a nuestro compilador de los cambios para provocar una recompilación mínima. La naturaleza nos permite también tratar el tema de la depuración: puntos de interrupción, ejecución paso a paso, etc.

4.7. Ventajas aportadas por ATC

ATC contiene toda la semántica de cualquier lenguaje de programación, decisiones, bucles, tipos de datos propios, etc. Pero, a parte de su inestimable labor de aislante, ¿qué sentido tiene esforzarnos en crear un nuevo lenguaje ATC, cuando ya disponemos de un Operational Mappings imperativo? ¿Y qué pasa con la integración de lenguajes de corte declarativo?

Como respuesta a la primera pregunta, nombraremos algunos ejemplos de funcionalidad de alto nivel no disponible directamente en ATC, y que se obtiene a través de la ejecución de múltiples unidades semánticas: la herencia de transformaciones, la redefinición de operaciones de una transformación en otra, la herencia de mappings, etc. Todas estas propiedades, relacionadas con la reutilización y modularización de las definiciones de transformaciones, se pierden en gran medida en ATC, que por su condición de lenguaje de bajo

nivel no necesita cuidar tales detalles y se dedica a reproducir tal funcionalidad paso a paso de forma explícita: chequeo de tipos, decisión del mapping a invocar en función de éstos, etc.

Respecto a la incorporación de lenguajes declarativos, será ésta una tarea ardua, pero ineludible de todas formas. Se puede programar un motor intérprete de expresiones declarativas, como un compilador Prolog, pero su funcionamiento interno continuará siendo imperativo en su núcleo. Por lo tanto, deberá recibir instrucciones sobre cómo analizar las declaraciones y tomar decisiones de ejecución. Pensar en diseñar ATC como lenguaje declarativo es ineficiente, puesto que los lenguajes imperativos serían convertidos a una nomenclatura declarativa, para después reconvertir estas declaraciones a un conjunto de acciones ejecutables dentro del motor, equivalentes a las sentencias originales, y posiblemente con poco parecido. ATC se coloca en el punto final de la cadena de ejecución, a bajo nivel, donde las instrucciones son imperativas. Y por ello, la labor de traducción tanto de los lenguajes declarativos como de los imperativos debe producirse durante el proceso de compilación de sus definiciones a ATC.

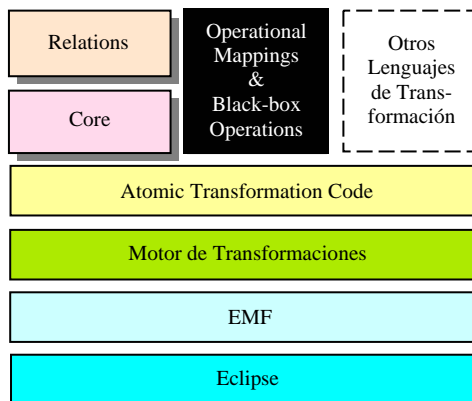


Figura 4. Esquema completo de la arquitectura de nuestro entorno de transformaciones

Una vez resuelta la conversión de la gramática de un lenguaje a ATC, este lenguaje pasa a engrosar la lista de lenguajes soportados en BOA 2. Cualquier definición de transformaciones escrita en este lenguaje podrá ser compilada a ATC. Si se estructura bien, si se le dota de la potencia necesaria para permitir la traducción de cualquier lenguaje abstracto, si su implementación es eficiente, podríamos pensar en la posibilidad de estandarizar

ATC como un lenguaje byte-code, que facilitaría la integración del conjunto de lenguajes que pugnan por sobrevivir en el entorno QVT. Los interesados en emplear lenguajes propietarios para definir transformaciones, podrían ellos mismos crearse un compilador para generar ficheros XMI que representaran los modelos equivalentes en ATC. De esta manera, las definiciones podrían exportarse a cualquier herramienta de transformaciones de modelos compatible con ATC.

5. Conclusiones

El objetivo de este artículo era la exposición de la implementación de un motor de transformaciones que diera soporte a prestaciones QVT. Este motor constituye la base de la nueva versión del framework BOA, como continuidad a su ciclo de vida, mejorando aquellos aspectos limitantes detectados en la primera versión tras la utilización en proyectos reales de desarrollo. Con una aproximación en algunos casos alejada de las premisas de la versión anterior, ésta será más acorde con los principios MDA, al estar basado en un meta-metalenguaje equivalente a EMOF, Ecore de EMF. Esperamos finalmente combinar todos estos ingredientes en un entorno de transformaciones integrado en Eclipse que nos dé acceso a modelos, metamodelos y reglas para aplicar transformaciones de forma sencilla, flexible y potente.

Hemos visto la dificultad de traducir fragmentos declarativos a sentencias imperativas, y otras dificultades y retos tecnológicos aún no del todo resueltos, como la implementación y optimización del soporte a las características del estándar QVT que hemos discutido.

Referencias

- [1] Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A.: The Design of a Simple Language for Graph Transformations, Journal in Software and System Modeling, in review, 2005.
http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2005_The_Design.pdf
- [2] Budinsky, F., Steinberg, D., Merks, Ed., Ellersick, R., Grose, T.J. Eclipse Modeling Framework: A Developer's Guide, Addison Wesley, 2003
- [3] Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D. Visual Automated Transformations for Formal Verification and Validation of UML Models.
<http://csdl.computer.org/dl/proceedings/ase/2002/1736/00/17360267.pdf>
- [4] Padrón, J., Estévez, A., Roda, J.L., García, F. BOA, un framework de alta productividad.
<http://www.dsic.upv.es/workshops/dsdm04/files/06-Padron.pdf>
- [5] Wilkie, I. et al. UML Action Specification Language (ASL) Reference Guide.
<http://www.kc.com/download/index.html>
- [6] Willink, E. D. UMLX: A graphical transformation language for MDA..
<http://www.softmetaware.com/oopsla2003/willink.pdf>
- [7] AndromDA, <http://www.andromda.org/>
- [8] ATOM: A Tool for Multi-Paradigm modeling.
<http://atom3.cs.mcgill.ca/>
- [9] BOTL: The Bidirectional Object oriented Transformation Language,
<http://www4.in.tum.de/~marschal/botl/>
- [10] Codagen Architect,
<http://www.codagen.com/products/architect/default.htm>
- [11] Compuware OptimalJ,
<http://www.compuware.com/products/optimalj/>
- [12] EMF: Eclipse Modeling Framework.
<http://www.eclipse.org/emf>
- [13] GMT: Generative Model Transformer,
<http://www.eclipse.org/gmt/>
- [14] iO Arc Styler, <http://www.arcstyler.com/>
- [15] Metadata Repository (MDR).
<http://mdr.netbeans.org/>
- [16] Model Driven Architecture (MDA).
<http://www.omg.org/mda>
- [17] MOF 2.0 Query/Views/Transformations RFP, de OMG. <http://doc.omg.org/ad/2002-4-10>
- [18] Rational XDE, <http://www-136.ibm.com/developerworks/rational/products/xde>
- [19] Revised submission for MOF 2.0 Q/V/T RFP, del grupo QVT-Merge.
<http://www.omg.org/cgi-bin/doc?ad/2005-03-02>
- [20] SableCC. <http://sablecc.org>
- [21] The Atlas Transformation Language (ATL).
<http://modelware.inria.fr/rubrique12.html>