Extracción de modelos en una modernización basada en ADM

Javier Luis Cánovas Izquierdo and Jesús García Molina

Universidad de Murcia {jlcanovas,jmolina}@um.es

Abstract. La Modernización Dirigida por Modelos ha emergido recientemente como una nueva área de investigación centrada en la automatización basada en modelos de actividades de evolución de software. Esta área ha sido impulsada por la iniciativa ADM de OMG que ha propuesto estándares para favorecer la interoperabilidad entre herramientas, como un beneficio clave para el éxito de la modernización con modelos.
En este artículo presentamos un proceso para la extracción de modelos conformes al metamodelo KDM, una de las especificaciones estándar incluidas en ADM. En nuestra propuesta primero se usa el lenguaje Gra2MoL para extraer modelos conformes al metamodelo ASTM, también incluido en ADM, y en un segundo paso, estos modelos se transforman en modelos KDM de más alto nivel mediante transformaciones modelo a modelo en RubyTL. El proceso se ilustra con un caso práctico de extracción de modelos de una plataforma Java, en concreto del framework Struts, enmarcado dentro de un proyecto de migración de Struts a JSF.

1 Introducción

La Modernización de Software Dirigida por Modelos aplica los principios y técnicas del Desarrollo Dirigido por Modelos a la automatización de actividades relacionadas con la evolución o modernización de aplicaciones existentes. El interés por esta disciplina surgió principalmente a partir de que OMG lanzase en 2004 la iniciativa ADM [1] (*Architecture Driven Modernization*), destinada a definir un conjunto de siete metamodelos estándares que faciliten la interoperabilidad de las herramientas de modernización. Hasta el momento sólo se han publicado los metamodelos KDM (*Knowledge Discovery Metamodel*), ASTM (*Abstract Syntax Tree Metamodel*) y SMM (*Software Metrics Metamodel*) pero no se han publicado experiencias de casos de estudio. Los primeros dos metamodelos se complementan y constituyen la parte central de ADM ya que los modelos KDM y ASTM son básicos en cualquiera de los posibles escenarios de modernización y son el punto de partida para realizar actividades como *refactoring*, obtención de métricas o análisis de propiedades.

El objetivo de este trabajo es mostrar cómo obtener modelos ASTM y KDM a partir del código fuente de una aplicación. Para ello se propone un proceso de dos etapas basado en una cadena de transformaciones. Primero se extrae un modelo ASTM a partir del código fuente y después se transforma el modelo obtenido en un modelo KDM. El proceso se ilustra con un caso práctico relacionado con

una migración de la plataforma Struts a la plataforma JSF (Struts2JSF). En la actualidad existe poca información sobre la utilización de ADM, por lo que este trabajo es una de las primeras contribuciones que muestra cómo aplicarlo. Otra contribución es mostrar la utilidad de Gra2MoL para la extracción de modelos, presentándose dos aspectos novedosos: las reglas *skip* para extraer expresiones y su aplicación para la extracción automática de metamodelos a partir de gramáticas de sintaxis abstracta.

El documento está organizado de la siguiente manera. En primer lugar se describe la iniciativa ADM y los metamodelos KDM y ASTM. A continuación, la sección 3 presenta el proceso propuesto. En la sección 4 se expone el trabajo relacionado y finalmente, se presentan las conclusiones y el trabajo futuro.

2 ADM

La iniciativa ADM de OMG propone el uso de estándares (metamodelos MOF) en modernización como forma de favorecer el intercambio de metadatos entre herramientas y con ello la interoperabilidad, lo cual es un factor clave para el éxito de la modernización basada en modelos. La primera especificación aparecida fue la que describe el metamodelo KDM y actualmente se encuentra en desarrollo la especificación del metamodelo ASTM, de la cual se dispone una versión preliminar (versión FTF) desde noviembre de 2008 y la aparición de la versión final está prevista en los próximos meses. Además se encuentran en desarrollo cinco especificaciones más, como por ejemplo la que describe el metamodelo SMM. Las especificaciones KDM y ASTM se complementan para modelar la sintaxis y semántica de los sistemas software. Mientras que ASTM permite representar la sintaxis y semántica básica (análisis de ámbitos, referencias y tipos) del código fuente, KDM permite crear modelos semánticos de alto nivel.

KDM define un metamodelo para representar cualquier tipo de elemento software, las relaciones entre elementos y los entornos en los que se ejecutan. El metamodelo de KDM es muy grande y se ha organizado en cuatro capas de abstracción que a su vez están formadas por varios dominios o vistas arquitecturales del sistema. Cada dominio está formado por un único paquete y contiene las metaclases para los conceptos propios del aspecto considerado. Esta organización favorece la modularidad y separación de aspectos.

La capa de *infraestructura* proporciona los elementos básicos para la construcción de los modelos KDM, como las entidades, relaciones entre elementos y trazabilidad. La capa de *elementos de programa* permite representar los artefactos software del sistema a nivel de implementación. La capa de *recursos runtime* representa elementos de mayor nivel de abstracción ya que están ligados a la plataforma de ejecución, como por ejemplo la interfaz de usuario. Finalmente la capa de *abstracciones* trabaja a un nivel de abstracción todavía mayor, representando el conocimiento específico del dominio, como las reglas de negocio o el conocimiento arquitectural.

El uso de KDM no requiere considerar todos los modelos sino aquellos que sean de interés para el usuario. Por otra parte, se han definido niveles de compa-

tibilidad como una forma de especificar y asegurar el nivel de interoperabilidad entre herramientas KDM. Dado un determinado nivel de compatibilidad, una herramienta KDM debe proporcionar: (1) capacidad para analizar artefactos software de una aplicación existente y exportar su representación como modelos KDM correspondientes al nivel de compatibilidad soportado y (2) capacidad para importar los modelos del mismo nivel de compatibilidad soportado. Se han identificado dos niveles de compatibilidad. El nivel 0 indica el soporte de los paquetes contenidos en la capa de *infraestructura* y de *elementos de programa* y es el nivel básico que ofrece la base de interoperabilidad entre las herramientas KDM. El nivel 1 se define individualmente para cada paquete contenido en las capas de *recursos runtime* y *abstracciones*.Además, la compatibilidad a nivel 1 implica ser compatible también a nivel 0. Adicionalmente, se dice que una herramienta KDM es compatible a nivel 2 cuando es compatible a nivel 1 con todos los paquetes de las capas de *recursos runtime* y *abstracciones*.

ASTM permite modelar el código fuente de un sistema software en forma de modelos de Árbol de Sintaxis Abstracta (*Abstract Syntax Tree*, AST). Un modelo ASTM representa las sentencias del código fuente y refleja la estructura gramatical de un lenguaje de programación. Sin embargo, los modelos ASTM no son exactamente ASTs ya que también incluyen información semántica básica que produce referencias cruzadas entre elementos del árbol. Estas características los convierten en grafos de sintaxis abstracta.

La especificación de ASTM define un metamodelo base llamado *Generic Abstract Syntax Metamodel* (GASTM) y propone la creación de metamodelos específicos que lo extiendan llamados *Specialized Abstract Syntax Metamodels* (SASTMs). El GASTM es un metamodelo que contiene elementos comunes a numerosos lenguajes de programación, mientras que los SASTMs son extensiones para capturar las necesidades de lenguajes específicos, como por ejemplo el concepto de paquete en Java. La Figura 1 muestra la correspondencia de estos metamodelos con la arquitectura de cuatro capas de metamodelado del OMG. En el nivel M0 se encuentra el código fuente, que es representado mediante modelos GAST y SAST que son conformes a los metamodelos GASTM y SASTM.

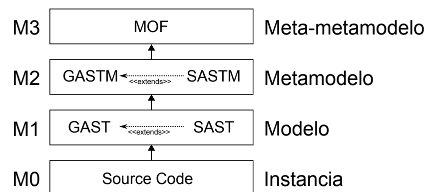

Fig. 1. Niveles de abstracción en ASTM.

Al igual que ocurre con KDM, en ASTM también se han definido niveles de compatibilidad. El nivel 0 asegura la compatilibilidad con los elementos sintácticos del metamodelo ASTM, tanto del GASTM como de los posi-

bles SASTM. Por otro lado, el nivel 1 indica el soporte de las características semánticas del metamodelo, además de las sintácticas.

Desde la perspectiva de ADM, los modelos ASTM son la principal fuente para obtener la información necesaria para crear modelos KDM, que a su vez son la base para realizar las actividades ligadas a la mayoría de escenarios de modernización y para obtener otros modelos ADM.

3 Extracción de modelos ADM

En esta sección presentaremos el proceso que hemos definido para extraer modelos KDM en una modernización basada en ADM. Este proceso ha sido aplicado en una migración entre plataformas Java, en concreto de Struts a JSF [2]. Como muestra la Figura 2, el proceso se basa en una cadena de dos transformaciones. En un primer paso se extraen los modelos ASTM a partir del código fuente, para lo que se aplica una transformación código a modelo implementada en Gra2MoL [3]. Este lenguaje permite definir transformaciones código-a-modelo especificando las relaciones (*mappings*) entre los elementos de la gramática y los elementos del metamodelo. En el segundo paso, los modelos ASTM obtenidos se transforman en modelos KDM mediante una transformación ASTM-KDM implementada en RubyTL [4].

A continuación detallaremos los dos pasos del proceso, pero antes de ello explicaremos cómo se ha obtenido un metamodelo de GASTM para la última versión de la especificación, puesto que no existía. La descripción del proceso se ilustra con ejemplos extraídos de la migración Struts2JSF.

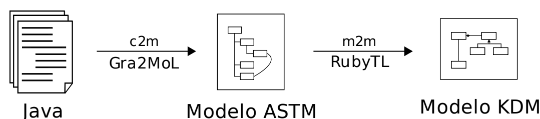

Fig. 2. Extración de modelos ASTM y KDM de un sistema Java

3.1 Generación del metamodelo ASTM

Gra2MoL puede utilizarse para generar automáticamente un metamodelo a partir de una gramática de la sintaxis abstracta, lo cual se ha aprovechado para obtener una versión actualizada del metamodelo del GASTM. La especificación ASTM contiene la descripción de la sintaxis abstracta de GASTM en dos formatos principales: de forma textual, mediante una gramática expresada en una variante de la notación BNF; y de forma gráfica, utilizando diagramas UML para expresar el metamodelo. La Figura 3 muestra ambas representaciones de una parte del metamodelo GASTM.

Para aplicar Gra2MoL se definió una meta-gramática que reconociera la variación de BNF que describe el metamodelo ASTM y se establecieron las correspondencias entre los elementos de dicha meta-gramática y los elementos del

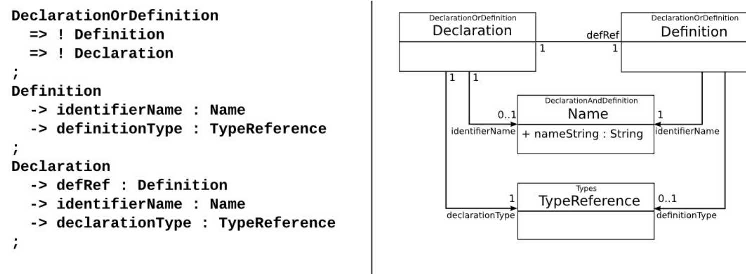

Fig. 3. Descripción del metamodelo ASTM de forma textual y gráfica según el documento de especificación .

meta-metamodelo Ecore. El motor de transformación Gra2MoL toma la especificación textual del metamodelo GASTM, que es conforme a la meta-gramática definida, y genera un metamodelo que es conforme al meta-metamodelo Ecore. La ventaja de utilizar este proceso es la posibilidad de obtener automáticamente la última versión del metamodelo a partir de la especificación.

3.2 Extracción de modelos ASTM

Como se indicó anteriormente, el metamodelo ASTM es resultado de extender GASTM con metamodelos SASTM específicos de los lenguajes de programación involcurados. Por ello, en la fase de extracción de modelos ASTM es necesario definir el metamodelo SASTM que represente los conceptos no incluidos en GASTM. Por ejemplo, en el caso de Struts2JSF, se definió un metamodelo SASTM para representar conceptos Java como los paquetes o los *imports*.

La extensión del metamodelo GASTM con metamodelos SAST tiene sus implicaciones en las transformaciones Gra2MoL si se quiere conseguir modularidad, y dos posibles soluciones serían: (1) componer transformaciones Gra2MoL, lo cual exigiría incorporarle un mecanismo similar a las fases de RubyTL [5], de modo que cada una de las transformaciones Gra2MoL para los SASTM se podría componer con la transformación común definida para GASTM; o (2) generar modelos GASTM y SASTM separados y componerlos mediante una transformación de modelos RubyTL. Sin embargo, en nuestro caso se optó por una sola definición de transformación Gra2MoL dado que las metaclases del ASTM se han integrado en el GASTM obtenido.

La transformación Gra2MoL para una extracción de modelos ASTM puede organizarse en varias partes según las principales categorías de elementos gramaticales del lenguaje de programación. En el caso de la migración Struts2JSF se han distinguido cuatro grupos de reglas para los elementos: declaraciones, sentencias, tipos y expresiones.

Para algunos tipos de elementos gramaticales fue necesario resolver las referencias entre elementos del código fuente ya que son explícitas en los modelos, como sucede con la referencia entre un identificador de variable y su declaración

o entre la llamada a un método y su definición en el código. Este problema es muy común en las transformaciones código-a-modelo y requiere de un proceso complejo para identificar las correspondencias entre elementos. Gra2MoL integra un lenguaje de consultas que ha sido especialmente diseñado para resolver referencias, como ilustra la siguiente regla que trata con llamadas a métodos con receptor `this` y se encarga de localizar la declaración del método.

```
rule 'mapMethodCall'
  from primary{THIS.exists}/identifierSuffix exp
  to FunctionCallExpression
  queries
    params     : /exp/identifierSuffix/arguments//expList/#expressic
    container  : //#classOrInterfaceDecl//primary{this.check(exp)};
    locatedMet : //#classBodyDecl//methodDecl{Id.eq(exp.Id)};
  mappings
    calledFunction = new IdentifierReference;
    calledFunction.name = exp.Id;
    calledFunction.refersTo = locatedMet;
    actualParams = params;
end_rule
```

La sección de *queries* de la regla extrae la información necesaria para crear un elemento `FunctionCallExpression`, en concreto, la información relativa a los parámetros (consulta `params`) y el método invocado. El atributo `calledFunction` del elemento `FunctionCallExpression` representa la referencia a la declaración del método invocado. El valor de dicho atributo es un elemento `IdentifierReference` el cual tiene una referencia llamada `refersTo` a la declaración del método. Para resolver esta referencia ha sido necesaria la definición de dos consultas. En primer lugar, la consulta `container` localiza la definición de la clase que contiene la invocación al método y, en segundo lugar, la consulta `locatedMet` parte del resultado de la anterior para localizar el método a partir de su identificador. Finalmente, en la sección de *mappings* se utiliza el resultado de las consultas para especificar el valor de los atributos del elemento del metamodelo.

Cabe destacar que Gra2MoL facilita la obtención de modelos ASTM con un nivel de compatibilidad 1 gracias al lenguaje de consultas que integra.

En cuanto al manejo de expresiones con Gra2MoL, el uso de reglas normales como la anterior provoca un problema que motivó la definición de un nuevo tipo de regla. Normalmente las reglas gramaticales utilizadas para el reconocimiento de las expresiones en un lenguaje de programación se definen de forma encadenada de manera que cada regla introduce un nuevo operador a la expresión. Por ejemplo, la Figura 4 muestra las reglas gramaticales que reconocen los operadores AND y OR en una expresión junto con el árbol de análisis que se obtiene para la expresión `exp1 && exp2`.

Las correspondencias entre elementos de la gramática y del metamodelo son prácticamente directas, por ejemplo, un elemento `condOrExp` se corresponde con un elemento `BinaryExpression` de tipo `Or`. Sin embargo, las reglas normales de Gra2MoL no son apropiadas para manejar expresiones debido a que, en algunos casos, reconocer un elemento gramatical de tipo expresión no supone crear un elemento del metamodelo, por ejemplo, reconocer un elemento `condOrExp` de la Figura 4 no implica la creación de un elemento `BinaryExpression` de tipo `Or`.

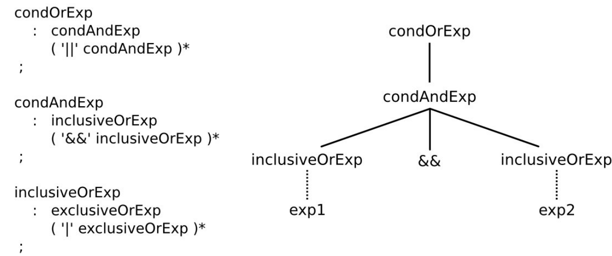

Fig. 4. Reglas gramaticales para reconocer los operadores AND y OR en una expresión y el correspondiente árbol de análisis para la expresión `exp1 && exp2`.

En general, estos casos son aquellos en los que el elemento de la gramática no contiene el operador correspondiente.

Por tanto, para tratar con expresiones, las reglas Gra2MoL deben disponer de algún mecanismo que permita decidir si se crea el elemento correspondiente del metamodelo según una determinada condición, en el caso del ejemplo, la existencia del operador. Gra2MoL incorpora un tipo especial de regla, denominada `skip_rule`, que ofrece esa funcionalidad, permitiendo retrasar la creación del elemento del metamodelo establecido en la parte `to` hasta realizar comprobaciones en los elementos de la gramática. De esta forma, dependiendo del resultado de dichas comprobaciones, puede transferirse la ejecución a la regla adecuada. La estructura de las reglas `skip` es muy parecida a las reglas normales, solamente se sustituye la parte de *mappings* por una sección *do* donde se indica la regla a la que transferir la ejecución por medio de la sentencia `skip`. Por ejemplo, estas son las reglas que tratan con expresiones condicionales OR y AND.

```
skip_rule 'skipOr'
  from condOrExp{!OR.exists} exp
  to Expression
  queries
    next : /exp/#condAndExp;
  do
    skip next;
end_rule
```

```
rule 'mapOrExpression'
  from condOrExp{OR.exists} exp
  to BinaryExpression
  queries
    exp1 : /exp//#condAndExp[0];
    exp2 : /exp//#condAndExp[1];
  mappings
    leftOperand = exp1;
    operator = new Or;
    rightOperand = exp2;
end_rule
```

```
skip_rule 'skipAnd'
  from condAndExp{!AND.exists} exp
  ...
end_rule
```

```
rule 'mapAndExpression'
  from condAndExp{AND.exists} exp
  ...
end_rule
```

La regla `skipOr` es de tipo `skip` y el filtro de la parte *from* comprueba que el elemento gramatical no contiene el token `OR`. A continuación localiza el siguiente elemento del árbol de expresiones por medio de la consulta `next` y transfiere la ejecución por medio de la sentencia `skip next`. La regla a ejecutar sería aquella cuya parte *from* conformase con `condAndExp` y la parte *to* sea conforme a la metaclase `Expression`. En este caso, las reglas candidatas son `skipAnd` y `mapAndExpression`, que tratan con los árbol de expresiones AND. La regla `mapOrExpression` es de tipo normal y se encarga de tratar el caso en el que

exista el token `OR`, para ello, crea un elemento de tipo `BinaryExpression` y establece sus atributos `leftOperand` y `rightOperand` con los operandos izquierdo y derecho de la expresión, respectivamente. También establece el operador de la expresion a `Or`. Por simplicidad, el ejemplo solamente ilustra la regla que trata con expresiones con dos operandos.

3.3 Extracción de modelos KDM

Los modelos KDM son generados a partir de la información de los modelos ASTM mediante transformaciones modelo a modelo en las que el metamodelo origen es ASTM y el destino es KDM. Para la definición de estas transformaciones se utilizó el lenguaje RubyTL que se integra con Gra2MoL en el entorno AGE [6]. Este entorno está siendo extendido con funcionalidad orientada a la modernización.

En una primera etapa, se define una transformación para conseguir un nivel 0 de compatibilidad, es decir, que el modelo sólo contenga elementos de las capas de infraestructura y elementos de programa. A continuación se definirían transformaciones para los dominios de las capas de *abstracciones* y *recursos runtime*, con el fin de conseguir el nivel 1 de compatibilidad. El mecanismo de fases de RubyTL [5] favorece la composición de la transformación para el nivel 0 con las transformaciones de nivel 1, y llegado el caso conseguir el nivel 2 si existe una transformación para cada uno de los dominios. En este momento AGE ofrece compatibilidad a nivel 0, y actualmente estamos definiendo una transformación para conseguir el nivel 1 para los paquetes `Platform`, `Data` y `UI`.

Aunque el metamodelo KDM contiene un gran número de referencias entre los elementos, la representación del código Java se vio facilitada al haber utilizado el metamodelo ASTM, donde la mayoría de las referencias se encuentran resueltas.

La transformación modelo a modelo para el nivel 0 de compatibilidad se encarga principalmente de la organización de los elementos del metamodelo y de la creación de las acciones asociadas a las sentencias y condiciones. Con respecto al primer aspecto, se definieron las reglas necesarias para construir los elementos principales que representan un sistema Java, como los conceptos de clase o método. Para el segundo aspecto, dado que el metamodelo KDM no contiene elementos específicos para representar sentencias y expresiones de los lenguajes de programación, como por ejemplo, sentencias de asignación o expresiones booleanas, para cada sentencia se generaron los elementos de tipo `Action` correspondientes. Estos elementos representan, de forma abstracta, el comportamiento que va a realizar la sentencia. Por ejemplo, la Figura 5 muestra las acciones asociadas a la sentencia de asignación $a = a + 1$.

4 Trabajo relacionado

La extracción de modelos se implementa normalmente mediante *parsers ad-hoc* [7]. Se podría pensar en utilizar alguna de las herramientas de generación de modelos a partir de lenguajes textuales específicos de dominio, como xText [8]

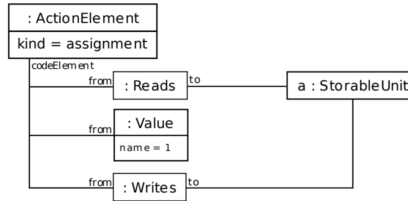

Fig. 5. Acciones KDM asociadas a la sentencia de asignación $a = a + 1$.

o TCS [9], o en el uso de herramientas para transformación de programas. Sin embargo en [3] se explican los inconvenientes de usar estos enfoques. La definición de Gra2Mol, un lenguaje específicamente diseñado para abordar la extracción de modelos, es una respuesta a la necesidad de tener un medio eficaz para extraer modelos a partir de código fuente de un lenguaje de programación.

Modisco [10] es un framework creado para extraer modelos en el contexto de ADM, y que forma parte del proyecto GMT de Eclipse. Modisco ofrece un soporte para construir *discoverers*, esto es, extractores de modelos. En este sentido, Gra2MoL puede integrarse en Modisco como un medio para implementar *discoverers*.

Hasta ahora no se han publicado procesos para la aplicación de ADM y, aunque varias empresas están colaborando en la especificación de KDM y ASTM, todavía hay muy pocos ejemplos de uso que ilustren cómo crear modelos ASTM y KDM, especialmente de ASTM, cuya versión final todava no se ha publicado. Las aplicaciones más interesantes han sido llevadas a cabo por la empresa Software Revolution [11] pero los modelos KDM y ASTM no son accesibles. La empresa KDM Analytics [12] también ha publicado un conjunto de ejemplos pero son muy simples.

5 Conclusiones y trabajo futuro

En este artículo hemos presentado la que quizá pueda ser la primera propuesta de un proceso para ADM, en concreto, para un aspecto básico como es la obtención de los modelos KDM a partir de modelos ASTM del código fuente. Hemos discutido cómo llevar a cabo la extracción de modelos ASTM mediante transformaciones Gra2MoL y la conversión de éstos en modelos KDM mediante transformaciones modelo a modelo con RubyTL. Las dos etapas se han ilustrado con ejemplos de un caso real como es la migración de Struts2JSF, un proyecto de modernización que se había abordado previamente sin considerar ADM [2].

El proceso aprovecha las ventajas de Gra2MoL en la extracción de modelos y la utilidad del mecanismo de fases de RubyTL para componer transformaciones. Se ha discutido la conveniencia de las reglas *skip* para manejar expresiones en Gra2MoL y se ha mostrado cómo generar automáticamente metamodelos a partir de la gramática de la sintaxis abstracta.

En cuanto al uso de ASTM y KDM, hay que destacar que ASTM ha facilitado la extracción de modelos AST del código fuente, dado que las correspondencias entre elementos de la gramática y metamodelo han sido directas exceptuando casos muy puntuales. Además, los modelos ASTM han permitido reducir el nivel de complejidad de la extracción de modelos KDM. De esta forma, actuando como modelos intermedios entre el código y el metamodelo KDM, la información necesaria para los modelos KDM se encontraba organizada y fácilmente localizable en los modelos ASTM, principalmente por incorporar referencias semánticas. Por otra parte, aunque el metamodelo KDM representa en un formato común de intercambio el código de la plataforma Java, es necesaria una correcta alineación entre ambos metamodelos, dado que KDM no cubre toda la expresividad del código a nivel procedural.

Los modelos KDM obtenidos han demostrado ser muy limitados por el hecho de tener un nivel 0 de compatibilidad. En una migración de plataformas es necesario disponer de información semántica detallada de los elementos del código, por lo que se necesita un nivel 1 de compatibilidad que contemple los paquetes que representan la plataforma, los datos y la interfaz gráfica.

Como trabajo futuro se está trabajando en extraer modelos KDM para los mencionados paquetes del nivel 1 de compatibilidad y continuar con el resto de fases de migración del proyecto Struts2JSF. También se está extendiendo Gra2MoL para proporcionar un mecanismo de modularidad que permita componer las transformaciones para la extracción de modelos ASTM.

References

1. ADM. http://adm.omg.org
2. J. Cánovas, O. Sánchez, J. S. Cuadrado, J. García Molina. "DSLs para la extracción de modelos en modernización". V Taller DSDM08.
3. J. Cánovas and J. García Molina. "A Domain Specific Language for Extracting Models in Software Modernization". ECMDA-09.
4. J. S. Cuadrado, J. G. Molina and M. M. Tortosa, "Rubytl: A practical, extensible transformation language" in ECMDA-FA, L. N. in Computer Science, vol. 4066/2006, pp. 158, 172 (2006).
5. J. S. Cuadrado and J. G. Molina. "Modularization of model transformations through a phasing mechanism". Software and Systems modeling (2008).
6. J. S. Cuadrado, J. G. Molina, "AGE". Demostración en XII JISBD07.
7. L. F. Andrade, J. Gouveia, M. Antunes, M. El-Ramly and G. Koutsoukos, "Forms2Net - Migrating Oracle Forms to Microsoft .NET", GTTSE. (2006).
8. S. Efftinge, "openarchitectureware 4.1 xtext language reference", http://www.eclipse.org/gmt/oaw/doc/4.1/r80 xtextReference.pdf (2006).
9. F. Jouault, J. Bézivin, and I. Kurtev, "TCS: a dsl for the specification of textual concrete syntaxes in model engineering", in GPCE, pp. 249-254 (2006).
10. MoDisco. http://www.eclipse.org/gmt/modisco/
11. Software Revolution. https://rhs.softwarerevolution.com/auto-docs/index.html
12. Hatha Systems. KDM analytics. http://hathasystems.com/