DSLs para la extracción de modelos en modernización

Javier Luis Cánovas Izquierdo, Óscar Sánchez Ramón,
Jesús Sánchez Cuadrado, y Jesús García Molina

Departamento de Informática y Sistemas
Universidad de Murcia {jlcanovas,osanchez,jesusc,jmolina}@um.es

Resumen. La Modernización Dirigida por Modelos ha emergido recientemente como una nueva área de investigación dedicada a la automatización basada en modelos de procesos de evolución de software. En los próximos años se necesitará un gran esfuerzo para encontrar principios, técnicas y métodos para esta nueva área y será crucial la experiencia adquirida en el desarrollo de casos de estudio reales de modernización.
En este artículo presentamos dos DSLs para la extracción de modelos: Gra2MoL para la extracción a partir de código fuente conforme a una gramática y H2MoL para el caso de texto que no está bien formado, como por ejemplo HTML. El diseño de estos lenguajes es fruto de la experiencia adquirida en un proyecto de migración de aplicaciones de la plataforma Struts a JSF, en el que se han aplicado las técnicas basadas en modelos.

Palabras clave: Modernización dirigida por modelos, Gra2MoL, H2MoL, extracción de modelos

1 Introducción

Las técnicas de Desarrollo de Software Dirigido por Modelos (DSDM) no son solamente útiles para la creación de nuevos sistemas software sino también para la modernización de sistemas existentes. De este modo, la Modernización de Software Dirigida por Modelos (MSDM) ha emergido recientemente como un nuevo paradigma para abordar la automatización basada en modelos de las actividades de evolución del software. En esta dirección el OMG ha propuesto varios estándares de modernización dentro de la iniciativa ADM [1]. En los próximos años se requerirá un gran esfuerzo para encontrar principios, técnicas y métodos para esta nueva área, y la experiencia adquirida en el desarrollo de casos de estudio de modernización reales será crucial.

De acuerdo con los objetivos de un proyecto de modernización, se pueden distinguir varios escenarios [2], como la migración de plataformas o la conversión de lenguajes. Además, con frecuencia un problema de modernización abarca más de un escenario. Las características de un escenario de modernización determinan las tareas necesarias para llevar cabo el proyecto.

En este artículo presentamos un caso de estudio de Modernización Dirigida por Modelos para un escenario de migración de plataformas: la migración de un sistema web basado en Struts a un sistema *Java Server Faces* (JSF). Nos referiremos a este caso de estudio como Struts2JSF. Este proyecto ha servido para experimentar en las tres principales etapas de un proyecto de modernización dirigido por modelos: extracción de los modelos de los artefactos fuente, rediseño del sistema y generación del nuevo sistema.

Sin embargo, y por cuestiones de espacio, en este artículo nos centraremos en las aproximaciones que hemos ideado para solucionar los retos aparecidos durante la extracción de modelos, que es la etapa que actualmente requiere de una mayor labor de investigación. En concreto, presentaremos Gra2MoL como lenguaje para extraer modelos a partir de código conforme a una gramática y H2MoL como lenguaje para extraer modelos en base a texto que

no está bien formado, como por ejemplo HTML. En la migración Struts2JSF utilizaremos Gra2MoL para extraer modelo del código Java, H2MoL para código JSP y el *framework* EMF (*Eclipse Modeling Framework*) para ficheros XML.

El documento está organizado de la siguiente manera. En primer lugar se describe el problema y los metamodelos de las plataformas. A continuación la sección 4 presenta las dos contribuciones principales del trabajo: dos lenguajes específicos del dominio dirigidos a la extracción de modelos a partir de los ficheros del sistema origen (por ejemplo código fuente Java o JSP). En la sección 5 se expone el trabajo relacionado y finalmente, se presentan las conclusiones y el trabajo futuro.

2 Descripción del problema

Struts [3] es un *framework* para desarrollar aplicaciones web en Java que apareció en el año 2000, y que esta basado en la arquitectura Modelo-Vista-Controlador (MVC). Por otra parte, *Java Server Faces* (JSF) [4] apareció en el año 2004, con el objetivo de simplificar el desarrollo de interfaces web Java. Este *framework* también utiliza una arquitectura MVC y se ha convertido en un estándar de facto.

La migración Struts2JSF es un ejemplo de escenario de migración de plataformas. Este escenario normalmente se combina con un escenario de conversión de lenguajes, aunque en este caso no se requiere debido a que las dos plataformas están basadas en el lenguaje Java. En general, un proceso de modernización requiere la creación de metamodelos que representen la arquitectura existente y la arquitectura destino. Dicho proceso consta de tres pasos principales: extracción de modelos a partir de los artefactos del sistema existente (ingeniería inversa), transformación de los modelos del sistema existente para generar modelos del sistema destino (rediseño) y generación de los artefactos del sistema destino (ingeniería directa).

El proceso de modernización aplicado se muestra en la Figura 1. En primer lugar se extraen tres modelos conforme al metamodelo de la plataforma Struts a partir del código fuente Struts. Cada modelo Struts se transformará en su correspondiente modelo JSF. El último paso consiste en la generación de los ficheros fuente de la aplicación JSF usando transformaciones modelo-código. El código fuente con la lógica de negocio no requiere ningún tipo de transformación sino que se migra sin cambios.

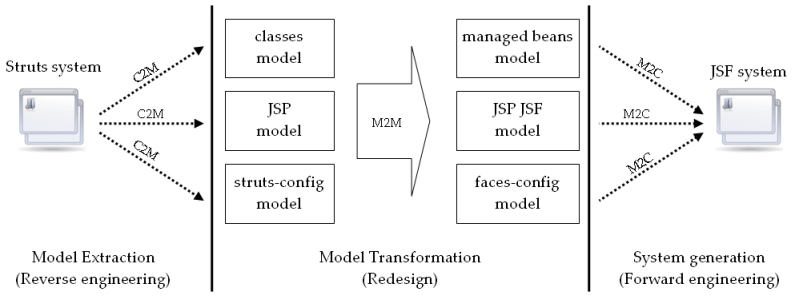

Fig. 1. Proceso de modernización aplicado en el caso de estudio Struts2JSF

Puede observarse que tanto los artefactos de Struts como los de JSF se organizan en las mismas tres partes: clases Java, código JSP y ficheros de configuración XML.

3 Metamodelos de las plataformas

Todo proyecto de modernización basada en modelos necesita de metamodelos que describan las plataformas involucradas. En este caso de estudio no estamos interesados en modelar completamente las plataformas Struts y JSF, dado que esto implicaría tratar con demasiados elementos y variantes de codificación. En cambio nos hemos centrado en un conjunto de características básicas como validación de formularios, navegación entre páginas, manejo de *beans* e internacionalización, que es suficiente para el objetivo de este caso de estudio. Además, dado que no se ha realizado un análisis semántico, el código reconocido se restringe a las convenciones de programación más utilizadas. A pesar de que no se necesitan metamodelos de Struts y JSF completos, el diseño es todavía complejo porque se debe representar el lenguaje Java y la arquitectura MVC.

Para mejorar la modularidad, los metamodelos de Struts y JSF se han organizado en tres paquetes, uno por cada una de las partes mencionadas en la sección anterior: JSP, *beans* y configuración. Tanto Struts como JSF, aunque usan librerías de etiquetas específicas, se apoyan sobre la tecnología JSP. Por otra parte, los *beans* de Struts y JSF comparten conceptos Java comunes y extienden las metaclases definidas en los metamodelos comunes para especializarlas. Por ejemplo, es útil distinguir específicamente un método Struts *validate*, es decir, crear una metaclase `validateMethod` que hereda de la metaclase `JavaMethod` del metamodelo Java. Por lo tanto, se han desarrollado dos metamodelos reutilizables para JSP y Java que se comparten por ambos metamodelos, como puede observarse en la Figura 2. Esto ha permitido que los metamodelos sean más compactos y fáciles de mantener además de promover su reutilización[1].

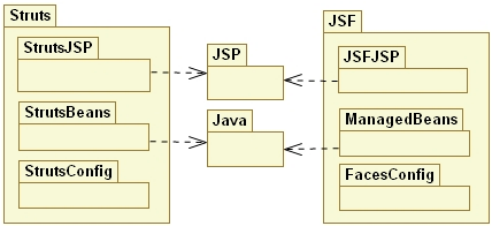

Fig. 2. Organización de paquetes de metamodelos.

4 Extracción de modelos

El primer paso en un proyecto de MSDM consiste en la extracción de modelos a partir de los artefactos fuente existentes, tales como código fuente o ficheros de configuración XML. Estos modelos son conformes a los metamodelos de la plataforma origen y representan la información del sistema existente que es necesaria para crear el nuevo sistema.

La extracción de estos modelos requiere establecer un puente entre diferentes espacios tecnológicos, en particular entre la tecnología del DSDM (*modelware*) y las tecnologías empleadas para describir el texto de los ficheros origen, normalmente gramáticas (*grammarware*) y esquemas XML. En un escenario de migración de plataformas, este puente es unidireccional y consiste en la creación de un analizador sintáctico que construye los modelos a partir de los artefactos fuente. Sin embargo, un sistema también puede estar formado por ficheros cuyo contenido no está bien formado (por ejemplo, HTML), donde las aproximaciones anteriores no pueden ser aplicadas.

[1] Los metamodelos se pueden descargar de `http://gts.inf.um.es/age`

4.1 Extracción de modelos a partir de código conforme a una gramática

La mayoría de los escenarios de modernización requieren tratar con código fuente cuya estructura puede ser expresada por una gramática. Por lo tanto, es posible utilizar herramientas que proporcionen un puente entre los espacios tecnológicos *grammarware* y *modelware* para llevar a cabo la extracción de modelos. Se han propuesto varias aproximaciones para establecer un puente entre las gramáticas y los modelos. xText [5] y los trabajos de Wimmer et al. [6] y Kunert [7] son los ejemplos más destacables. Estos trabajos están basados en la generación automática de un metamodelo a partir de la gramática, y de un analizador sintáctico para la extracción de modelos.

La utilidad de estas aproximaciones está restringida por varias limitaciones que surgen en situaciones prácticas. Desde nuestra experiencia, hemos identificado cuatro problemas principales:

– El bajo nivel de los metamodelos generados frecuentemente obliga a aplicar transformaciones modelo a modelo para obtener metamodelos de más alto nivel como un AST o un metamodelo KDM [8]. Esta idea aparece reflejada en la Figura 3, en la que se contrasta un metamodelo generado con el metamodelo deseado.

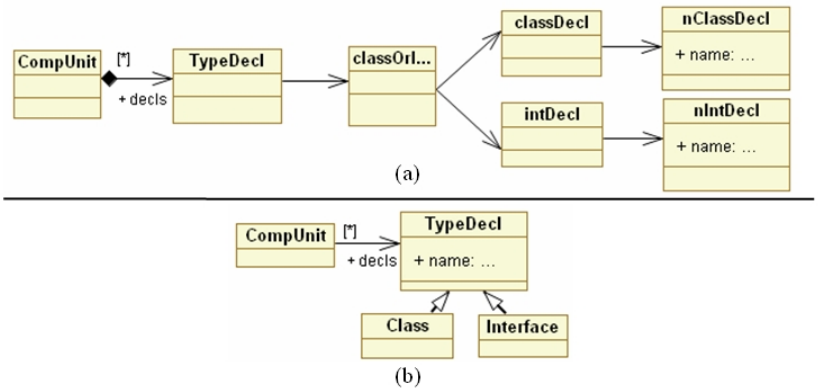

Fig. 3. (a) Metamodelo generado a partir de la gramática de Java usando xText. (b) Metamodelo de alto nivel deseado.

– Los modelos extraídos se almacenan normalmente en ficheros XMI. En proyectos de gran tamaño que contienen cientos de ficheros fuente, se puede producir la duplicación de información dado que los artefactos software están representados en los ficheros fuente y en los modelos. Esto hace que esta aproximación sea ineficiente en cuanto a tiempo y memoria.
– Determinada información derivada del análisis sintáctico tal como nombres de ficheros, líneas, columnas, etc. es importante en el proceso de modernización, principalmente para permitir la trazabilidad entre el código y el modelo. Esta información se pierde debido a que los metamodelos generados no la incluyen, y por lo tanto no se propaga a los modelos.
– Existe un catálogo considerable de definiciones de gramáticas para generadores de analizadores sintácticos como ANTLR [9] o JavaCC [10]. Además, los artefactos software se programan normalmente con lenguajes de programación con bastante difusión, como COBOL, C, o Java, y crear la gramática de un lenguaje de programación desde cero es una tarea difícil y que conlleva mucho tiempo. Las aproximaciones de modernización dirigida por modelos deberían permitir la reutilización de las definiciones de gramáticas para algún generador de analizadores sintácticos existentes.

Con estos problemas en mente, hemos definido un DSL para la extracción de modelos denominado Gra2MoL [11], cuyas dos características más relevantes son: (i) incorpora un potente lenguaje de consultas para recorrer el árbol sintáctico y recuperar la información del código fuente, (ii) la gramática es utilizada para poder referenciar a los elementos de ésta en las consultas al árbol sintáctico. Gra2MoL nos permite especificar explícitamente las relaciones entre los elementos de la gramática origen y los elementos de un metamodelo destino, y trata el código fuente como un modelo usando la definición de gramática subyacente como si se tratara de un metamodelo. La Figura 4 ilustra el esquema de Gra2MoL.

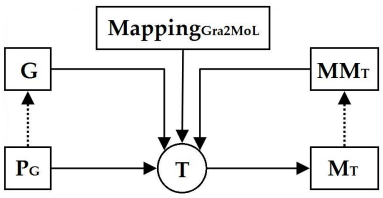

Fig. 4. Esquema de Gra2MoL. El código fuente P_G es conforme a la gramática G y el modelo M_T es conforme a el metamodelo MM_T. T denota al processo de transformación y $Mapping_{Gra2MoL}$ son las correspondencias expresadas con Gra2MoL entre G y MM_T.

La entrada de una transformación Gra2MoL es código fuente junto con la definición de la gramática a la que conforma y una definición de transformación (*mapping*). En primer lugar el código fuente se analiza sintácticamente para construir un árbol sintáctico y la definición de la transformación trata con dicho árbol, utilizando la gramática para tipar los nodos.

En los árboles sintácticos, las referencias entre elementos se establecen implícitamente por medio de identificadores. Por otra parte, los modelos son grafos donde las referencias entre elementos son explícitas. Transformar una referencia basada en identificadores en una referencia explícita implica buscar el nodo identificado que puede estar fuera del ámbito de la regla que realiza la transformación. Por tanto, las transformaciones gramática a modelo realizan un uso intensivo de consultas sobre todo el árbol sintáctico para recuperar información que está fuera del ámbito de la regla actual. En [12] este tipo de transformaciones se denomina transformaciones global a local. Si se usase la notación punto para escribir estas consultas, se necesitaría una larga cadena de navegación. Por esta razón hemos desarrollado un lenguaje de consultas inspirado en Xpath [13] para extraer información que está dispersa a lo largo del código fuente y para permitir la navegación del árbol sintáctico sin necesidad de especificar cada paso de navegación, con lo que se evita especificar largas expresiones de navegación.

En el caso de estudio Struts2JSF, Gra2MoL nos ha permitido extraer modelos del código fuente Java del sistema Struts. Para ello hemos implementado una definición de transformación entre los elementos de la gramática Java y los elementos del metamodelo *StrutsBeans*. Por ejemplo, el siguiente código muestra una regla que extrae un elemento `ValidateMethod` del metamodelo a partir de un elemento `methodDeclaration` de la gramática de Java.

```
rule 'validateMethod'
  from methodDeclaration{Identifier.eq("validate")} cbd
  to StrutsBeans::ValidateMethod
  queries
    md : /cbd//#methodDeclaration{Identifier.exists};
    t  : /md/#type;
    fp : /cbd//formalParameters///#formalParameterDecls;
    st : /cbd//#blockStatement;
  mappings
    name = new JavaSimplified::Name;
```

```
    name.identifier = md.Identifier;
    name.strValue = md.Identifier;
    parameters = fp;
    statements = st;
    returnType = t;
end_rule
```

Una regla Gra2Mol tiene cuatro secciones. La sección *from* especifica los símbolos no terminales de la gramática origen (en el ejemplo es `methodDeclaration` y filtra los métodos llamados `validate`) y declara una variable que será ligada a un nodo del árbol cuando se aplique la regla (en el ejemplo es `cbd`). Esta variable puede ser usada en cualquier expresión dentro de la regla. La sección *to* especifica la metaclase del elemento del metamodelo destino (en el ejemplo es `ValidateMethod`). La sección de consultas (*queries*) contiene un conjunto de expresiones de consulta que sirven para asignar a variables determinados elementos del código fuente. Finalmente, la sección *mappings* contiene un conjunto de *bindings* para asignar valores a las propiedades de los elementos del modelo destino en base a las variables definidas en la sección de consultas. Estos *bindings* son un tipo especial de asignación utilizada en lenguajes de transformación de modelos como RubyTL [14] y ATL [15].

Atendiendo a la parte de las consultas, una consulta se compone de un conjunto de operadores de consulta. Hay tres tipos de operadores de consulta: `/`, `//` y `///`. El operador `/` devuelve los hijos inmediatos de un nodo y es similar al uso de la notación punto. Por otra parte, los operadores `//` y `///` permiten la navegación de todos hijos del nodo (directos e indirectos) recuperando todos los nodos de un tipo dado. Estos dos operadores permiten ignorar nodos superfluos intermedios, facilitando la definición de la consulta, dado que se especifica qué tipo de nodos deben ser encontrados pero no cómo alcanzarlos.

El operador `///` difiere ligeramente del operador `//`. Mientras que el operador `///` busca recursivamente en el árbol sintáctico, el operador `//` solo selecciona aquellos nodos cuya profundidad es menor o igual que la profundidad del primer nodo seleccionado. De este modo el operador `///` sólo se utiliza para extraer información de estructuras gramaticales recursivas. Por ejemplo, en la consulta `/cbd//formalParameters///#formalParameterDecls` de la regla `ValidateMethod`, una vez que se ha seleccionado un elemento `formalParameters`, se recupera la lista de parámetros del método.

Los operadores de consulta pueden incluir expresiones de filtro opcionales y una expresión de tipo índice tales que solo aquellos nodos que satisfagan dichas expresiones serán seleccionados. En el ejemplo anterior, la consulta `/cbd//#methodDeclaration{Identifier.exists}` define una expresión de filtro de modo que solo se seleccionarán aquellos nodos de tipo `methodDeclaration` que tiene una hoja del árbol de tipo `Identifier`.

En cuanto a la extracción de modelos a partir de ficheros XML, un sistema Struts involucra un fichero de configuración XML que es conforme a un esquema XML. Actualmente existen herramientas para construir un puente entre los espacios tecnológicos XML y *modelware*, tales como las existentes en el proyectos EMF de la plataforma Eclipse. En este caso de estudio hemos utilizado EMF para extraer modelos de los ficheros de configuración. Estos modelos, aunque reflejan la estructura del esquema XML, han resultado adecuados dado que la definición de la transformación de los ficheros de configuración es prácticamente directa.

4.2 Extracción de modelos a partir de texto no bien formado

En esta sección abordaremos el problema de la extracción de modelos a partir de texto que no es conforme a ningún formalismo (como una gramática o un esquema XML), por lo que son necesarios reconocedores dedicados. En aplicaciones web el ejemplo prototípico son las páginas HTML y las plantillas como JSP o ASP. Los navegadores web tienen reconocedores dedicados para tratar con las diferentes variantes de HTML. Los lenguajes de plantillas no

consideran el contenido de la plantilla sino que simplemente se encargan de ejecutar el código scriptlet añadiendo el resultado a la salida estándar. Por ejemplo, el siguiente fragmento de código JSP es una página HTML que contiene etiquetas HTML, etiquetas Struts y código *scriptlet*.

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean"%>
<html></body>
  Data for: <%= session.getAttribute("loggedUser") %>
  </html:form action="/showData">
    <p>
    <bean:message key="user.name"/><html:text property="name"/>
  </html:form>
</body></html>
```

Claramente esto no es un documento válido por dos razones: (i) no se permite código *scriptlet* en un XML y (ii) algunas etiquetas como `<p>` no están cerradas. Es importante destacar que incluso aunque es una mala práctica escribir código como éste, la mayoría de los sistemas existentes están construidos con este estilo.

Para tratar con el primer problema hemos creado un pequeño preprocesador basado en el uso de expresiones regulares que sustituye código *scriptlet* por comentarios XML. Posteriormente un paso de postprocesamiento extraerá el código *scriptlet* de los comentarios. Daremos más detalles más adelante. Para el segundo problema no es posible usar una plantilla XSLT, un analizador XML, el *framework* EMF o Gra2MoL porque el código HTML puede ser inconsistente, es decir, no es un XML válido o no es conforme a la gramática HTML. Por esta razón habría que crear un analizador dedicado.

Sin embargo, crear este analizar sintáctico consume tiempo y es propenso a errores debido a la variedad de casos que debe manejar. Por tanto hemos definido la siguiente aproximación. Primero hemos utilizado una de las API existente en Ruby (HTree [16]) para convertir texto HTML en texto XHTML que puede ser procesado por un analizador de XML.

Una vez que hemos sido capaces de hacer que el HTML sea conforme a XML, el siguiente paso es extraer modelos conformes al metamodelo Struts-JSP en base al contenido del XML. En este punto podríamos utilizar EMF para extraer modelos del XHTML y a partir de ellos aplicar un lenguaje de transformación de modelos. Sin embargo, los únicos elementos que deseamos hacer corresponder entre plataformas son las etiquetas específicas de Struts y JSF, mientras que el resto de las etiquetas HTML se traducirán a una etiqueta genérica (`GenericHTMLTag`). Por lo tanto un mecanismo para establecer correspondencias genéricas entre patrones del nombre de la etiqueta HTML aumentaría la legibilidad, productividad y la facilidad de mantenimiento que la aplicación de un lenguaje de transformación de modelos, donde es necesario escribir reglas con diferentes filtros para tratar cada tipo de etiqueta.

Por esta razón, para abordar el problema de los artefactos XML no bien formados, hemos ideado un lenguaje de transformación específico del dominio denominado H2MoL implementado como un DSL embebido en Ruby. Este lenguaje nos permite especificar qué partes de la página JSP serán ignoradas (que no son conformes a XML) con el fin de realizar la etapa de preprocesamiento. Internamente se utiliza HTree para transformar el código HTML en XHTML que posteriormente es procesado con un analizador sintáctico XML. Además, proporciona construcciones para especificar como transformar etiquetas XML en elementos del metamodelo destino. Es importante destacar que este DSL no está acoplado con nuestro metamodelo JSP sino que puede ser utilizado con cualquier metamodelo destino. El único requisito es que el metamodelo destino tenga estructura de árbol.

A continuación se muestra un fragmento de código H2MoL utilizado para establecer la correspondencia entre páginas JSP con etiquetas Struts a nuestro metamodelo JSP de JSF.

```
sub /<%@.+%>/, 'JSP::Directive', 'directives'
sub /<%.+%>/,  'JSP::ScriptLet', 'tags'
pre 'html', 'Struts::View::HTMLTag', 'tags'
map 'bean:message', 'Struts::View::MessageTag', 'tags'
generic 'JSP::GenericHTMLTag', 'tags'
```

En su estado actual, este DSL proporciona cuatro tipos de sentencias, que se utilizan para reconocer nodos XML y realizar correspondencias uno a uno con una metaclase destino. Para cada correspondencia el resultado se almacena en la propiedad especificada en el último parámetro de la sentencia (por ejemplo, un elemento `GenericHTMLTag` se almacena en la propiedad `tags` del elemento padre). Los tipos de sentencias son los siguientes:

- *sub*. Toma una expresión regular con el fin de reconocer un fragmento de texto XML no legal. También se puede utilizar una metaclase para especificar un elemento del modelo que se creará cuando el texto se reconozca. El motor de ejecución es el encargado de sustituir la porción de texto con un comentario XML como se ha explicado antes y de establecer las correspondencias necesarias.
- *map*. Reconoce cualquier etiqueta XML cuyo prefijo y nombre concuerden con el identificador o expresión regular dados.
- *pre*. Reconoce cualquier etiqueta XML prefijada por la etiqueta indicada o que concuerda con la expresión regular dada.
- *generic*. Cualquier nodo XML que no forme parte de una de las correspondencias anteriores es tomado por esta sentencia. Es importante destacar que solo es posible definir una sentencia de este tipo.

En cuanto a la ejecución de las sentencias, conviene mencionar que una vez que una sentencia se ejecuta con un nodo XML el resto de sentencias no se ejecutan. De hecho, el orden de prioridad de las sentencias es el siguiente: *sub*, *map*, *pre*, *generic*.

El uso de esta aproximación aporta varias ventajas sobre la creación de un inyector y el uso de un lenguaje de transformación de modelos general [12]:

- La operación *sub* nos permite aplicar esta estrategia para JSP, ASP y cualquier otro motor de ejecución de plantillas de un modo genérico.
- La sentencia *generic* hace posible la realización de correspondencias genéricas. Además tratar con prioridades es más sencillo que escribir filtros en reglas de transformación para asegurar una estrategia de aplicación de reglas apropiada.
- Este DSL se ejecuta directamente sobre el texto HTML sin apoyarse en ficheros XMI intermedios con lo que se obtiene una mejora del rendimiento.

5 Trabajo relacionado

La extracción de modelos a partir de código fuente es un área que todavía no tiene la madurez deseada, aunque ya se han presentado varias propuestas interesantes. Entre ellas destacamos el *framework* MoDisco y las propuestas para conectar las técnicas propias de las gramáticas (*grammarware*) con las técnicas del DSDM (*modelware*), como son la herramienta xText incluida en el *toolkit openArchitectureWare* y los trabajos de Wimmer y Kunert.

MoDisco (*Model Discovery*) [17] es una aproximación extensible para ofrecer soporte a la extracción de modelos de sistemas existentes que ha sido definida como parte del proyecto GMT [18] de Eclipse. Los componentes del *framework* MoDisco son: un metamodelo basado en KDM [8], un mecanismo de extensión del metamodelo, varias facilidades de manipulación de modelos, y una metodología para diseñar extensiones. Actualmente, debido a que está en desarrollo sólo ofrece la infraestructura para manejar y crear modelos. MoDisco define el

concepto de *discoverer*, que es un componente software capaz de analizar un sistema existente (por ejemplo código Java), y extraer un modelo mediante la infraestructura proporcionada. La finalidad de MoDisco no es cubrir todos los aspectos de un proceso de modernización sino únicamente ofrecer la infraestructura necesaria para la construcción de *discoverers*. De esta forma, Gra2MoL puede ser utilizado para la implementación de los *discoverers*.

xText [5] es un lenguaje que pertenece al *framework openArchitectureWare* [19] y permite construir DSLs textuales en la plataforma Eclipse. En este lenguaje, la sintaxis concreta textual del DSL se especifica por medio de un lenguaje de definición de gramáticas similar a EBNF. A partir de la gramática especificada se genera automáticamente el metamodelo del DSL, un analizador sintáctico para reconocer la sintaxis del DSL y para instanciar el metamodelo, y un editor específico del DSL. Wimmer [6] et. al. han propuesto un *framework* genérico para conectar el *grammarware* y el *modelware*. El funcionamiento de este *framework* se resume en dos etapas. En una primera etapa, se aplican un conjunto básico de reglas para generar un metamodelo no refinado a partir de una gramática EBNF tales como las generadas por xText. En la segunda etapa se aplican varias heurísticas para mejorar dichos metamodelos. De forma similar, en Kunert [7] se opta por añadir anotaciones a la gramática con el objetivo de guiar el proceso de generación. Es importante destacar que no existen herramientas que soporten estas dos últimas aproximaciones.

Las principales desventajas de xText y los trabajos de Wimmer et. al y Kunert se han comentado en la sección 4.1. Gra2MoL carece de estas deficiencias dado que permite crear directamente modelos que conformen a un metamodelo cualquiera como puede ser KDM o un metamodelo AST. Para ello, se especifica la definición de una transformación de una gramática a un metamodelo. Dado que Gra2MoL ha sido diseñado explícitamente para trabajar con este tipo de transformaciones, es más fácil escribir estas transformaciones que cuando se utiliza un lenguaje de transformación de modelos común. Gra2MoL también nos permite acceder a información tales como la línea y columna donde se localiza un nodo en el código fuente. Además, puesto que Gra2MoL no utiliza un lenguaje especial para definir la gramática, sino ANTLR [9], se promueve la reutilización de gramáticas existentes.

6 Conclusiones y trabajo futuro

En los próximos años, la realización de casos de estudio será crucial para que la Modernización de Software Dirigida por Modelos madure como disciplina. La experimentación con los casos de estudio ayudará a identificar qué cuestiones deben ser resueltas, así como probar las aproximaciones propuestas.

En este artículo hemos presentado parte de los resultados de un proyecto de migración de un sistema Struts a un sistema JSF, en concreto, el trabajo relacionado con la fase de extracción de modelos. Los retos que hemos identificado relacionadas con esta fase son: (i) necesidad de diseñar metamodelos modulares para promover la separación de conceptos y la reutilización y (ii) utilización de técnicas eficientes y efectivas de extracción de modelos de artefactos fuente.

Hemos presentado una aproximación para cada uno de estos retos. Los metamodelos han sido diseñados de una forma modular de modo que compartan conceptos comunes, facilitando así su mantenimiento y reutilización. Hemos analizado diversas soluciones para la extracción de modelos a partir de código conforme a una gramática y hemos justificado el uso de Gra2MoL frente a otras alternativas como xText. Para la extracción de código conforme a un esquema XML hemos recurrido al *framework* EMF, y para la extracción de modelos de texto no bien formado hemos creado un DSL llamado H2MoL con la finalidad de tratar con código HTML no bien formado.

En cuanto al trabajo futuro, continuaremos trabajando en Gra2MoL. Actualmente estamos estudiando la adecuación de la estructura de reglas (que actualmente es dirigida por el

origen), ya que la experiencia adquirida en este caso de estudio nos ha inducido a cuestionar la necesidad de una aproximación dirigida por el destino. También seguiremos mejorando H2MoL para que sea posible incluir condiciones de filtrado más complejas.

Finalmente en este caso de estudio hemos trabajado con metamodelos *ad-hoc* para representar las plataformas, pero una alternativa para afrontar la fase de metamodelado es el uso de KDM. Sin embargo, debido al reducido número de casos de estudio prácticos con KDM, todavía no están claras las ventajas de éste. Como línea de trabajo futuro, planeamos estudiar esta alternativa en profundidad.

Agradecimientos

Este trabajo ha sido parcialmente financiado por los proyectos TIC-INF 06/01-0001 de la Consejería de Educación y Cultura (CARM) y 05645/PI/07 de la Fundación Séneca. Javier Luis Cánovas Izquierdo dispone de una beca FPI de la Fundación Séneca y Jesús Sánchez Cuadrado dispone de una beca FPU del Ministerio de Educación y Ciencia.

Referencias

1. ADM. http://adm.omg.org
2. ADM Task Force. *Architecture-driven modernization scenarios.* http://adm.omg.org/ADMTF_Scenario_White_Paper(pdf).pdf
3. Struts. http://struts.apache.org
4. JSR 127 Java Server Faces (JSF) Specification. http://jcp.org/en/jsr/detail?id=127
5. Efftinge, Sven. *openArchitectureWare 4.1 Xtext language reference.* 2006 http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf
6. Wimmer, Manuel and Kramler, Gerhard. *Bridging Grammarware and Modelware.* Satellite Events at the MoDELS 2005 Conference. 2006. 159–168
7. Andreas Kunert. *Semi-automatic Generation of Metamodels and Models from Grammars and Programs.* Fifth Intl. Workshop on Graph Transformation and Visual Modeling Techniques. 2006. Electronic Notes in Theorical Computer Science
8. ADM Task Force. *Knowledge Discovery Meta-Model (KDM).* 2007 http://www.omg.org/spec/KDM/1.0/
9. Parr, Terence. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. 2007
10. Java Compiler Compiler. https://javacc.dev.java.net
11. J. Cánovas Izquierdo, J. Sánchez Cuadrado and J. García Molina. *Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization.* 2nd Workshop on Model-Driven Software Evolution. 12th European Conference on Software Maintenance and Reengineering. 2008
12. J. van Wijngaarden and E. Visser. *Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems.* Technical Report UU-CS-2003-048. Department of Information and Computing Sciences, Utrecht University. 2003
13. Xpath. www.w3.org/TR/xpath
14. J. Sánchez Cuadrado and J. García Molina. *A plugin-based language to experiment with model transformation.* 9th International Conference in Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science. Springer (2006) 336-350.
15. F. Jouault and I. Kurtev. *Transforming Models with ATL.* 2005.
16. HTree. http://a-k-r.org/htree/
17. MoDisco project. http://www.eclipse.org/gmt/modisco
18. GMT project. http://www.eclipse.org/gmt/
19. Open Architecture Ware. http://www.eclipse.org/gmt/oaw