Utilidad de las transformaciones modelo-modelo en la generación automática de código

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

Facultad de Informática

Universidad de Murcia

jlcanovas@um.es, osanchez@um.es, jesusc@um.es, jmolina@um.es

Resumen

Aunque las transformaciones modelo-modelo son un elemento clave de MDA, todavía hay dudas sobre su utilidad en escenarios reales. De hecho, algunos paradigmas de *Desarrollo de Software Dirigido por Modelos*, e incluso algunas de las herramientas MDA más populares, plantean un enfoque generativo basado en transformaciones modelo-código.

En este artículo se analiza la utilidad de las transformaciones modelo-modelo a partir de un problema real de una empresa, en concreto la generación automática de código para la integración de software existente. El problema se ha resuelto desde dos perspectivas diferentes que se contrastan, una que genera el código directamente a partir de un modelo y otra que utiliza transformaciones modelo-modelo para mejorar la legibilidad y el mantenimiento.

1. Introducción

Las transformaciones modelo-modelo son un elemento clave de la propuesta MDA [1], que plantea el conocido esquema de desarrollo en el que un modelo independiente de la plataforma (PIM) es transformado en un modelo específico de la plataforma (PSM), y esta primera transformación modelo-modelo puede ir seguida de otras (PSM-PSM) antes de la definitiva transformación modelo-código.

No obstante, dentro del paradigma del *Desarrollo de Software Dirigido por Modelos* (DSDM) hay otras visiones que sólo contemplan transformaciones modelo-código, como son el *Desarrollo Específico del Dominio* [2] basado en el uso de lenguajes específicos del dominio y el *Desarrollo Centrado en la Arquitectura* [3], e incluso herramientas MDA como AndroMDA [4] o ArcStyler [5] soportan un enfoque generativo PIM-código.

Con la aparición de MDA surgió un interés académico e industrial por comprender la naturaleza de las transformaciones modelo-modelo, y por identificar las características deseables de los lenguajes de transformación de modelos, pero estos dos problemas todavía requieren grandes esfuerzos de investigación. Como se señala en [3], las transformaciones modelo-modelo todavía no se comprenden bien y no está clara su utilidad en escenarios reales de *Desarrollo Dirigido por Modelos*, pero a pesar de ello se considera que este tipo de transformaciones serán un mecanismo importante para reducir algunas de las diferencias semánticas entre abstracciones dentro del DSDM. Se han publicado muy pocos trabajos que analicen con detalle este papel de las transformaciones modelo-modelo.

En este trabajo se contrasta la diferencia entre utilizar o no transformaciones modelo-modelo en el contexto de una aplicación práctica del DSDM para resolver un problema real de la empresa Sinergia Tecnológica (IT Deusto). En concreto, se aborda la generación automática de *wrappers* para acceder desde código Java a una lógica de negocio ya existente e implementada en PL/SQL. La generación automática se realizó siguiendo dos en-

foques. Primero se siguió una aproximación basada en una transformación modelo-código, utilizando la herramienta xText de OpenArchitectureWare [6] para generar el metamodelo de PL/SQL, y el lenguaje MOFScript [7] para la generación de código. Después se ideó una transformación modelo-modelo como paso intermedio con el fin de simplificar la transformación modelo-código y mejorar el mantenimiento, entre otras ventajas Se ha utilizado el lenguaje RubyTL [8, 9] como lenguaje de transformación de modelos y se ha aprovechado su mecanismo de fases para organizar las transformaciones [10].

La organización del trabajo es la siguiente. Tras esta introducción, se describe el problema planteado por la empresa y se enumeran las restricciones a tener en cuenta en la generación de los *wrappers*. En el tercer apartado, se presenta la solución basada en una aproximación que sólo utiliza generación modelo-código, y a continuación se presenta la alternativa que añade una transformación modelo-modelo intermedia. En el quinto apartado se comparan ambos enfoques y finalmente se presentan las conclusiones.

2. Descripción del problema

En la actualidad, la empresa Sinergia Tecnológica dispone de una gran cantidad de lógica de negocio implementada en código PL/SQL, la cual necesita integrar en sus aplicaciones sobre la plataforma Java. Para integrar este código, los desarrolladores de la empresa crean de forma manual adaptadores (*wrappers*) que actúan de puente entre el código PL/SQL y las nuevas aplicaciones Java. La creación de estos *wrappers* es una tarea repetitiva que sigue unos determinados patrones y convenciones de acuerdo a la estructura del código PL/SQL.

Los *wrappers* son clases Java con métodos que ocultan el acceso a los procedimientos y funciones PL/SQL. Para cada paquete se crea una clase Java que contiene un método por cada función o procedimiento PL/SQL. La correspondencia entre una función o procedimiento PL/SQL y un método Java es diferente, dependiendo de una serie de convenciones sobre nombres, tipos y localización de métodos que ha establecido la empresa para las clases adaptadoras.

Las técnicas y herramientas del DSDM permiten la generación automática de los *wrappers* descritos. En los dos próximos apartados se describen las dos aproximaciones planteadas para resolver el problema, y en éste se discute cómo se obtiene un modelo a partir del código PL/SQL y se enumeran las convenciones establecidas por la empresa.

Para aplicar las técnicas del DSDM es necesario partir de un modelo expresado en un determinado lenguaje de modelado o lenguaje específico del dominio. Por tanto, en este caso es necesario convertir el código PL/SQL en un modelo conforme a cierto metamodelo (lo que en [11] se denomina un cambio de espacio tecnológico). Para este fin se ha utilizado la herramienta xText incluida en el framework OpenArchitectureWare [6].

xText permite generar un metamodelo a partir de una gramática expresada en un DSL propio muy parecido a BNF. Además, genera un editor textual que es capaz de instanciar un modelo conforme al metamodelo creado, a partir de una especificación textual conforme a la gramática. La Figura 1 muestra el metamodelo generado para la gramática de PL/SQL No es un objetivo de este artículo explicar en detalle la obtención de un modelo a partir del código PL/SQL, sino que el interés se centra en el proceso de generación de código a partir del modelo obtenido utilizando xText.

A continuación, se presenta un listado de las restricciones a tener en cuenta para la generación del código de los *wrappers*, incluyendo las convenciones más relevantes de la empresa.

- Para cada procedimiento o función PL/SQL se debe generar un método Java que encapsule su acceso. Estos métodos deben generarse en el mismo orden en que aparecen sus correspondientes funciones o procedimientos en el código PL/SQL. La Figura 2 muestra un ejemplo de método generado.
- Para cada procedimiento o función

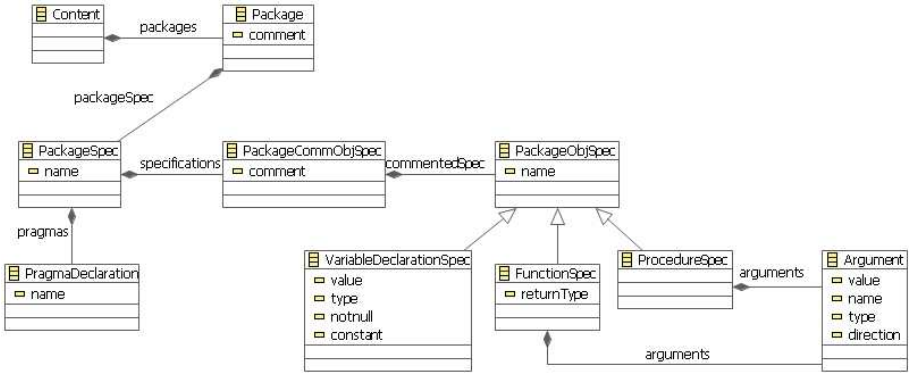

Figura 1: Metamodelo generado por xText a partir de la gramática PL/SQL

PL/SQL se debe generar una constante Java de tipo cadena que contiene el código nativo PL/SQL para realizar la llamada, según especifica el API JDBC. Esta constante será utilizada por el método Java para invocar al procedimiento o función. La Figura 2 muestra un ejemplo de este tipo de constantes.

- Para cada uno de los argumentos de salida (argumentos OUT) de los procedimientos o funciones se debe generar una constante (constantes OUT). Su nombre será `PAR_<nombreVariable>` y tendrá como valor el nombre de la variable. Estas constantes deben reutilizarse si existe alguna coincidencia en el nombre de los argumentos de los procedimientos o funciones. Se situarán antes de las variables de instancia y de las constantes de las cadenas de llamada. Además, si es una función, deberá generarse una constante con el nombre `PAR_RETURN_OUT`, como ilustra la Figura 2.

- No se generarán métodos para funciones o procedimientos que contengan tipos diferentes a los siguientes: `VARCHAR2`, `NUMBER`, `BOOLEAN`, `CLOB`, `XML`. Tampoco se generarán para aquellos que tengan algún argumento de tipo `INOUT`.

- Se utiliza el sufijo `%TYPE` en los nombres de las variables, argumentos y tipos devueltos por las funciones, para indicar que el tipo concreto de la variable o argumento se infiere de la primera letra del nombre. Esta letra será `v` para `VARCHAR2`, `n` para `NUMBER`, `b` para `BOOLEAN`, `c` para `CLOB` y `x` para `XML`. Por ejemplo, la variable con nombre `bTieneSaldo%TYPE` es de tipo `BOOLEAN`.

- Las funciones con nombre `existe` se deben sobrecargar para que tengan una versión que acepte y devuelva una conexión.

- Las funciones con nombre `tieneHijos` deben devolver un objeto del tipo `RespuestaTieneHijos`.

- El código PL/SQL está implícitamente dividido en varias *zonas*, de manera que la correspondencia entre funciones o procedimientos PL/SQL y métodos Java variará en función de la zona en la que se encuentren. Hasta ahora, Sinergia ha identificado cinco zonas: (0) desde el principio hasta el primer procedimiento o función, (1) desde la definición del primer procedimiento o función hasta la primera función denominada `existe`, (2) desde esta primera hasta la última función `existe`, (3) desde la última función `existe` hasta la primera función cuyo nombre comience

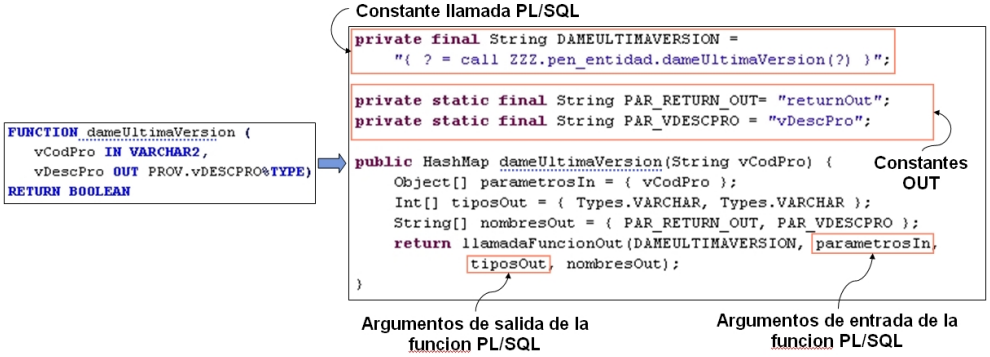

Figura 2: Ejemplo de código PL/SQL generado

por `tiene`, y (4) desde esta función hasta el final.

- Existen diferentes tipos de correspondencias entre procedimiento o función PL/SQL y método Java en función de la zona donde se encuentren, si tienen o no argumentos de salida, si no tienen argumentos, si la función se llama `existe` o `tieneHijos`, etc.

3. Aproximación modelo a código

En un principio se decidió aplicar un enfoque generativo modelo-código, puesto que los requisitos eran sencillos y no parecía justificada la complejidad de añadir una etapa previa de transformación modelo-modelo. El modelo intermedio que se habría utilizado tendría un nivel semántico similar al código Java, por tanto sería un paso innecesario. Se eligió MOFScript como lenguaje de transformación modelo-código. Se trata de un lenguaje cuyo uso está muy extendido, basado en reglas y con una sintaxis muy sencilla [7]. Las reglas MOFScript se aplican sobre el modelo PL/SQL obtenido a través de xText para obtener el código de los *wrappers*.

Existe una regla principal que se aplica para cada especificación de un paquete PL/SQL, encargada de analizar las especificaciones de todos los elementos principales (variables, funciones y procedimientos) existentes en el paquete y filtrar aquellas especificaciones que deben transformarse, descartando aquellos elementos que no es necesario transformar, como por ejemplo los procedimientos y funciones con algún argumento de tipo `INOUT`. A continuación, las agrupa en colecciones según las zonas de código PL/SQL señaladas anteriormente. Finalmente, invoca en orden a las reglas de transformación para la generación del código de cada uno de los elementos de las zonas.

Antes de generar el código de los métodos es necesario un paso previo para la generación de las cadenas para las constantes OUT y las cadenas de llamada PL/SQL. Las cadenas para las constantes OUT deben reutilizarse para aquellos argumentos cuyo nombre coincida. El control de dicha unicidad en el nombre se lleva a cabo por medio de una colección global. Para las cadenas de llamada también se utilizó una colección global para controlar las posibles colisiones de nombres (dado que los procedimientos y funciones pueden estar sobrecargados). En ambos casos se hace necesario utilizar un conjunto de funciones auxiliares para la gestión de las colecciones y para conseguir un nombrado consistente de los elementos.

Tras el paso anterior, se recorren una a una las colecciones para generar el código de los métodos del *wrapper*. Como anteriormente se ha comentado, los elementos de cada

colección corresponden a una zona determinada del *wrapper* y son recorridas en orden. Se aprovecha el polimorfismo soportado por MOFScript para discernir entre procedimientos y funciones y así ejecutar la regla apropiada.

Es importante destacar que el metamodelo de la gramática PL/SQL no incorpora las restricciones del problema relacionadas con la distribución del código en diferentes zonas y los distintos tipos de funciones (normales, `existe` y `tieneHijos`). Esto provoca la complejidad del código MOFScript, tanto por aumentar el tamaño de la transformación como por complicar su lógica con muchas sentencias condicionales, lo que reduce su extensibilidad y dificulta su mantenimiento. Por ejemplo, en el metamodelo no se refleja que existen tipos de función especiales como `existe` y `tieneHijos`, por lo que las reglas de transformación deben considerar este hecho a través de sentencias condicionales.

Entre los aspectos que añaden complejidad al código MOFScript (por no estar contemplados explícitamente en el metamodelo PL/SQL) se pueden destacar el hecho de que un procedimiento o función tenga argumentos OUT, o que se deba reutilizar la conexión. Para el primer caso, antes de generar el código para un procedimiento o función, se debe discriminar la regla de transformación a utilizar comprobando si alguno de sus argumentos es de tipo `OUT`. En el segundo caso, la reutilización de la conexión se indica mediante un parámetro en la regla, el cual modifica su comportamiento. Sin embargo, la reutilización de la conexión por los métodos estándar y la sobrecarga de las funciones `existe` son requisitos que reflejan que la incorporación del parámetro en la regla no está directamente ligado a un requisito del problema sino a varios. Esta situación, además de reducir la mantenibilidad de la solución, dificulta su legibilidad.

La correspondencia entre los tipos de datos PL/SQL y los tipos Java da lugar a un problema de *scattering* (la misma funcionalidad está distribuida en varias reglas de transformación) que ha sido resuelto con la definición de un metamodelo adicional. Esta correspondencia determina la forma de inicializar las variables y cómo convertir el resultado de una llamada PL/SQL a un tipo Java. Por ejemplo, el resultado de una función PL/SQL booleana es una cadena conteniendo el valor `true` o `false`, y por tanto debe generarse automáticamente el código para convertir esta cadena al tipo Java `boolean`. La generación del código para la conversión de tipos es necesaria para los argumentos de entrada, de salida y para el resultado de las funciones, y por tanto está distribuida en varias reglas de transformación, lo cual provoca problemas de *scattering*.

Para abordar este problema se diseñó un metamodelo que describe el conjunto de tipos existentes y sus correspondientes usos. La finalidad de este metamodelo es disponer de un modelo que recopile los tipos que se pueden utilizar y para cada uno de ellos indicar su forma de inicialización, correspondencia de tipos, etc. De esta forma, la transformación se puede parametrizar con este modelo y se independiza de los tipos permitidos. Con esta solución, los problemas de *scattering* relativos al manejo de tipos de datos desaparecen, puesto que toda la información se encuentra recogida en un sólo lugar, esto es, el modelo de tipos. Además, se facilitan futuras modificaciones sin necesidad de alterar el código de transformación ya implementado.

En la Figura 3 se muestra gráficamente el metamodelo definido. Para cada tipo a considerar existe una metaclase `TypeMapping` que incluye un atributo para identificar el tipo PL/SQL (`source`), así como la forma de detectar el tipo a partir del nombre de la variable (según los requisitos del problema) con el atributo `inferenceRule`. Ambos utilizan una expresión regular para llevar a cabo la identificación. El tipo Java correspondiente se especifica con los atributos `targetIn` y `targetOut` (dependiendo de cuando el tipo se utiliza en un argumento de entrada, o como argumento de salida o valor de retorno). Por otra parte, la referencia a `Converter` permite independizar el metamodelo del código que se debe generar para convertir el tipo PL/SQL al correspondiente tipo Java. Para cada subclase de `Converter` se definen en el código MOF-

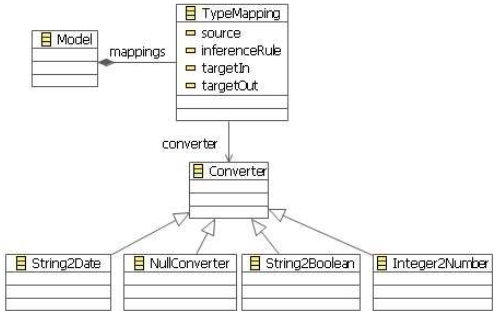

Figura 3: Metamodelo para describir la correspondencia entre tipos PL/SQL y tipos Java.

Script un conjunto de reglas de transformación encargadas de generar el código necesario para cada tipo. Por ejemplo, para la metaclase `String2Boolean` se definen las reglas que generan el código para la conversión de cadena SQL (`Varchar`) a booleano.

4. Aproximación modelo a modelo

Como se ha señalado, originalmente se optó por realizar el proceso de transformación en una única etapa: modelo a código. Con la evolución del proyecto los requisitos aumentaron y resultaron estar muy ligados a las convenciones de la empresa, con lo que el código de transformación se modificó en muchas ocasiones. Entonces, se valoró la posibilidad de comprobar el beneficio obtenido en el caso de combinar una transformación modelo-modelo con otra modelo-código.

La solución del problema utilizando una aproximación modelo a modelo se basa en separar la generación de código en dos pasos: en el primer paso se crea un modelo intermedio mediante una transformación modelo-modelo, y en el segundo paso se genera código para este modelo intermedio utilizando un lenguaje de transformación modelo-código. En nuestro caso, hemos utilizado RubyTL como lenguaje de transformación modelo-modelo y MOFScript como lenguaje de transformación modelo-código.

Un aspecto importante para la aproximación modelo a modelo es el diseño del metamodelo intermedio. En nuestro caso, se diseñó un metamodelo que satisficiese tres objetivos: reflejar de manera explícita los requisitos impuestos por la empresa, que simplificase la generación de código, y que permitiese añadir nuevos elementos fácilmente. La Figura 4 muestra el metamodelo que se ha diseñado.

Puesto que el metamodelo origen está dividido implícitamente en varias zonas según las convenciones de la empresa, se ha definido una metaclase abstracta llamada `Zone` y cada subtipo representa un tipo concreto de zona, que contiene únicamente cierto tipo de elementos. De esta forma, la división en zonas del código PL/SQL es ahora explícita.

Cada uno de los diferentes tipos de variables de instancia son un subtipo de `Constant`. Por ejemplo, `CallConstant` representa una constante cuyo valor corresponde a la cadena de llamada PL/SQL, como se explicó en la sección 2.

Se ha definido una jerarquía de funciones y procedimientos, cuyo elemento raíz es `JavaCall`. Esta jerarquía representa los diferentes tipos de correspondencias entre funciones y procedimientos PL/SQL y métodos Java. Por ejemplo, `NormalFunction` representa una función PL/SQL que tiene cero o más argumentos, mientras que `StandardFunction` representa una función sin argumentos que solamente puede aparecer en una zona de métodos standard (Zona 3).

Estas jerarquías permiten aprovechar el polimorfismo propio de un metamodelo orientado a objetos para facilitar la posterior transformación modelo-código, ya que cada subtipo tendrá una correspondencia diferente con código Java. Nótese que éste es un metamodelo dedicado a facilitar la posterior transformación modelo-código, y por tanto debe reflejar explícitamente todos los conceptos con los que debe tratar el generador de código. Además, si en el futuro la empresa identificara nuevos tipos de procedimientos o funciones PL/SQL, simplemente habría que añadir un nuevo tipo en el lugar adecuado de la jerarquía (y crear un nuevo subtipo de `Zone` si fuese necesario).

Por último, obsérvese que en el meta-

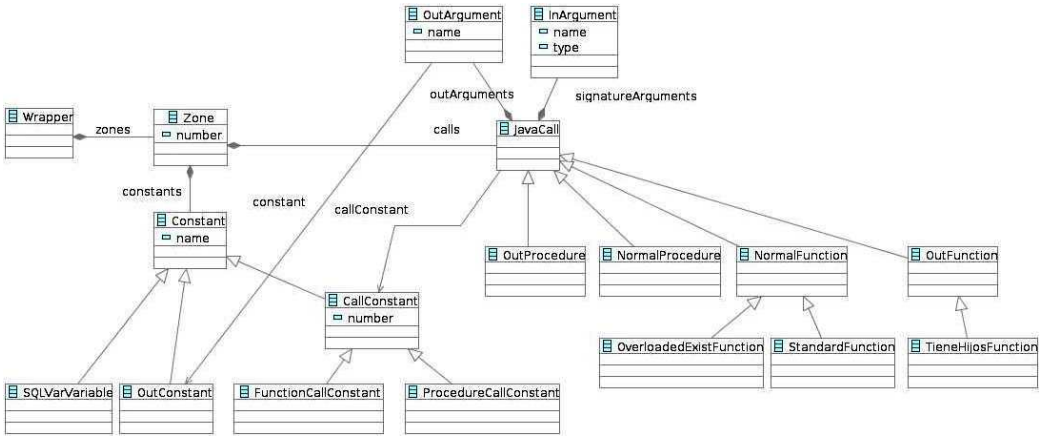

Figura 4: Metamodelo intermedio para facilitar la generación de código.

modelo, `JavaCall` tiene una referencia hacia `CallConstant`. Esta referencia refleja explícitamente un requisito impuesto por la empresa, como es que para cada método Java que adapta un procedimiento o función PL/SQL debe existir una constante Java de tipo cadena con el código nativo PL/SQL para realizar la llamada. Al estar ligados los procedimientos y funciones con las cadenas de llamada se facilita el mantenimiento y se reduce la complejidad de la transformación modelo-código.

La implementación de la transformación modelo-modelo en RubyTL, que transforma un modelo conforme al metamodelo de la Figura 1 en un modelo conforme al metamodelo de la Figura 4, utiliza el concepto de fase[10]. Una fase es una agrupación de reglas de transformación que tratan con una parte del modelo origen. El orden de ejecución de las fases es secuencial, y las reglas de una fase concreta pueden añadir o modificar elementos del modelo destino.

Así, para este caso, la correspondencia entre fase y zona es directa. Para cada posible zona se ha definido una fase, que contiene las reglas necesarias para transformar los elementos que pueden aparecer en tal zona (normalmente procedimientos y funciones PL/SQL). Una ventaja de esta aproximación es que en el futuro, si se identificaran nuevas zonas, simplemente habría que añadir las correspondientes fases, sin necesidad de modificar las ya existentes, incrementando así la extensibilidad de la transformación y favoreciendo su mantenimiento.

La Figura 5 ilustra cómo una fase agrupa conjuntos de reglas relacionadas, las cuales crean determinados tipo de elementos, sin colisionar con reglas pertenecientes a otras fases. Por ejemplo, las funciones PL/SQL de la zona 1 se transforman en elementos del tipo `NormalFunction`, mientras que las de la zona 2 se transforman en dos elementos, `NormalFunction` y `OverloadedExistFunction`. De esta forma, creando las reglas de transformación adecuadas, es posible agrupar en una fase todos los requisitos que se imponen sobre una determinada zona.

Por último, la responsabilidad de filtrar y descartar aquellos elementos que no se van a tener en cuenta en la generación de código posterior recae también en la transformación modelo-modelo. Esto simplifica enormemente la transformación modelo-código, ya que ésta sólo debe tratar con elementos que es necesario transformar a código. Es importante tener en cuenta que un lenguaje de transformación modelo-modelo dispone de construcciones específicamente diseñadas para "filtrar" elementos, como por ejemplo los filtros o pa-

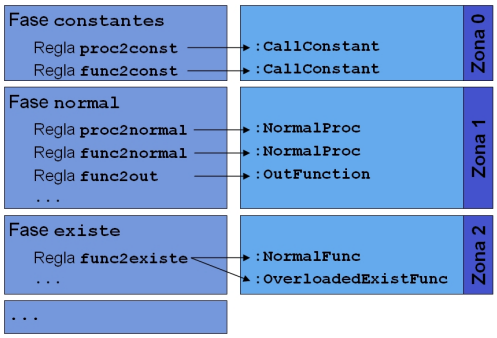

Figura 5: Esquema de la correspondencia entre el concepto de fase y zona. Las reglas de cada fase crean elementos en función del tipo de zona que corresponde.

trones de entrada de las reglas. Por tanto, esta tarea es más adecuada realizarla a nivel de la transformación modelo-modelo.

Una vez obtenido el modelo intermedio tal y como se ha explicado, la transformación modelo-código no tiene que utilizar las estrategias explicadas en la sección 3 (basadas en sentencias condicionales) para decidir cuál es la correspondencia textual de cada elemento el modelo. Ahora, las reglas MOFScript pueden hacer uso del polimorfismo y cada elemento del modelo tiene una correspondencia única con un fragmento de texto, incrementando de esta forma la legibilidad, extensibilidad y facilidad de mantenimiento de la transformación modelo-código.

Por último, el hecho de que las relaciones entre los elementos que hay que generar sean explícitas (por ejemplo, entre `JavaCall` y `CallConstant`) permite evitar una gran cantidad de funciones auxiliares que son sólo necesarias para conseguir un nombrado consistente del mismo elemento en lugares diferentes del texto.

5. Comparación de las aproximaciones

En esta sección se compararán las dos aproximaciones, para lo cual se considerarán las siguientes dimensiones: tiempo destinado a implementar las transformaciones, modularidad, legibilidad, facilidad de mantenimiento y extensibilidad.

Para la primera versión de la implementación modelo a código se invirtieron en torno a 50 horas, pero tras numerosas modificaciones y añadidos, la calidad del código se degradó enormemente y fue necesaria una reescritura. Se estima una inversión de 30 horas en la reescritura. El tiempo estimado en la aproximación modelo a modelo, para diseñar el metamodelo intermedio, implementar la transformación modelo-modelo y la transformación modelo-código, asciende a 50 horas. Es destacable que se dedicó muy poco tiempo a implementar la transformación modelo-código en esta aproximación, ya que la correspondencia era directa. Tanto para la reescritura como para la aproximación modelo a modelo se partió del conocimiento del problema obtenido en la primera versión implementada.

La aproximación que trata el problema únicamente con transformaciones modelo-código presentaba en un principio problemas de *scattering* y *tangling*, que reducen la modularidad. El primero de ellos se refiere al hecho de que encontramos numerosos fragmentos de código similar diseminados a lo largo de la plantilla. La solución adoptada para solventar el *scattering* ha sido factorizar el comportamiento común mediante funciones auxiliares. Por ejemplo, la generación de los parámetros de entrada PL/SQL aparece tanto en las reglas para generar funciones como procedimientos, y por tanto ha sido factorizada en una función auxiliar. El *tangling* aparece porque la misma regla de transformación, por ejemplo la usada para transformar funciones PL/SQL a texto, tiene que tratar al mismo tiempo con diferentes tipos de funciones (`normal`, `existe`, `standard`, etc.) dependiendo del valor de un parámetro. Es decir, una misma regla de transformación implementa distinta funcionalidad en base a un parámetro externo.

En la aproximación modelo a modelo, el *tangling* no aparece en la transformación modelo-modelo porque cada fase de la transformación trata con un tipo concreto de funciones

y procedimientos PL/SQL, mientras que el *scattering* se aborda utilizando las técnicas explicadas en [10]. La transformación modelo-código ya no presenta problemas de *tangling* puesto que existe una regla diferente para cada tipo de función o procedimiento PL/SQL (véase el metamodelo de la Figura 4). Los problemas de *scattering* se han solventado con funciones auxiliares, igual que en el primer caso.

Tanto en el enfoque directo modelo-código, como en el que se apoya en transformaciones modelo-modelo, la conversión de tipos PL/SQL a tipos Java provocaba numerosos casos de *scattering*. En la aproximación modelo a código, la medida adoptada para solucionar el problema fue utilizar un modelo para parametrizar sendas transformaciones modelo-código, tal y como se explicó en la sección 3. Esta parametrización puede efectuarse tanto a nivel de transformación modelo-modelo como a nivel de modelo-código. No hay ventajas de uno respecto del otro puesto que la factorización no se encuentra en el código de transformación, sino en el modelo.

En lo que se refiere a legibilidad, las transformaciones modelo-modelo y modelo-código de la aproximación que utiliza el metamodelo intermedio son notablemente más sencillas y fáciles de comprender que la transformación modelo-código de la otra aproximación. El metamodelo intermedio reduce la distancia conceptual entre el metamodelo origen (PL/SQL) y el metamodelo destino (*wrapper* Java) lo que facilita la escritura de las transformaciones. Es importante destacar el papel que cumple el metamodelo intermedio como esquema conceptual que ayuda a comprender la naturaleza del problema.

El hecho de que el grado de modularidad en la aproximación modelo a modelo sea mayor, hace que la legibilidad también lo sea. Esto es debido a que no existe una gran distancia conceptual entre el metamodelo origen y el intermedio. El uso de funciones auxiliares para factorizar código como solución al *scattering* produce una proliferación de las mismas en las transformaciones modelo-código (en ambas aproximaciones), con lo que se reduce la legibilidad en cierta medida.

El alto nivel de modularidad y legibilidad que se consigue con el uso de un modelo intermedio en el enfoque basado en transformaciones modelo-modelo, favorece la extensibilidad y mantenimiento del código de transformación. En la aproximación modelo a código, el hecho de que muchos requisitos estén implementados en forma de sentencias condicionales provoca que la facilidad de mantenimiento del código sea menor que con la aproximación modelo a modelo.

Como resultado de la experiencia, se ha observado que si la distancia entre los conceptos del modelo origen y el código destino es reducida, y se prevén pocos cambios en los requisitos, las transformaciones directas modelo-código pueden ser la alternativa más rápida y eficaz. Si por el contrario, la distancia conceptual es grande, o los cambios en los requisitos pueden ser sustanciales, es conveniente optar por el enfoque combinado de transformaciones modelo-modelo y modelo-código, diseñando un metamodelo intermedio que represente en cierta medida los requisitos que debe cumplir el código final y sirva para reducir la distancia conceptual.

6. Conclusiones

Este artículo ha mostrado cómo resolver un problema de generación automática de código surgido en la empresa Sinergia Tecnológica, en concreto la generación de *wrappers* Java para acceder a código PL/SQL, que puede ser contemplado como un caso particular del problema de integrar código existente en nuevas aplicaciones. Se ha abordado el problema desde dos perspectivas diferentes: generación directa modelo-código y generación a través de una transformación modelo-modelo[1].

El análisis del problema y de las dos soluciones propuestas ha mostrado que el uso de transformaciones modelo-modelo es un buen mecanismo para abordar algunos problemas de generación de código. En particular, cuando la distancia conceptual entre el modelo origen

[1] Las transformaciones pueden descargarse en http://gts.inf.um.es/downloads

y el código a generar es grande, o se prevén cambios frecuentes en los requisitos, utilizar un paso intermedio de transformación modelo-modelo puede incrementar la legibilidad, modularidad y la facilidad de mantenimiento de la solución.

El resultado de esta experiencia sirve además como ejemplo del uso de transformaciones modelo-modelo para resolver un problema real no trivial, así como un punto de partida para razonar sobre la utilidad de dichas transformaciones en escenarios reales de *Desarrollo Dirigido por Modelos*.

En la actualidad, se continúa la colaboración con la empresa Sinergia Tecnológica y con otras empresas para abordar la integración de sistemas software existentes utilizando DSDM y otros problemas de generación de código.

Por otra parte, se continúa el estudio de qué construcciones son necesarias en los lenguajes de transformación para mejorar la legibilidad y modularidad de las transformaciones.

Agradecimientos

Este trabajo ha sido parcialmente financiado por los proyectos 2I05SU0018 y TIC-INF 06/01-0001 de la Consejería de Educación y Cultura (CARM).

Referencias

[1] Object Management Group. *MDA Guide* version 1.0.1. omg/2003-06-01, 2003.

[2] Sitio web de DSM Forum, http://www.dsmforum.org/

[3] T. Stahl, M. Völter. *Model-Driven Software Development*. John Wiley, 2006.

[4] AndroMDA. http://www.andromda.org

[5] ArcStyler. http://www.arcstyler.org

[6] OpenArchitectureWare. http://www.openarchitectureware.org/

[7] MOFScript. http://www.eclipse.org/gmt/mofscript

[8] J. Sánchez, J. García, M. Menárguez. *RubyTL: A practical extensible transformation language*. LCNS, Proceedings of the 2nd European Conference on Model Driven Architecture (ECMDA, 2006). Vol. 4066, pag.158-172, julio, 2006.

[9] J. Sánchez, J. García. *A plugin-based language to experiment with model transformation*. LCNS, Proceedings of 9th International Conference on Models Driven Engineering Languages and Systems (MoDELS 2006). Vol. 4199, pag. 336-351.

[10] J. Sánchez, J. García. *A phasing mechanism for model transformation languages*. ACM Symposium on Applied Computing, MT Track. Pag 1020-1024. Marzo, 2007.

[11] I. Kurtev, J. Bézivin, M. Aksit. *Technological Spaces: An Initial Appraisal*. CoopIS, DOA'2002 Federated Conferences, Industrial track, 2002.