XV Jornadas de Ingeniería del Software y Bases de Datos
JISBD 2006
José Riquelme - Pere Botella (Eds)
© CIMNE, Barcelona, 2006

RUBYTL: UN LENGUAJE DE TRANSFORMACIÓN DE MODELOS EXTENSIBLE

Jesús Sánchez Cuadrado, Jesús J. García Molina y Marcos Menárguez Tortosa

Departamento de Informática y Sistemas
Facultad de Informática
Universidad de Murcia
Campus de Espinardo, 30100
e-mail: {jesusc, jmolina, marcos }@um.es, web: http://gts.inf.um.es

Palabras clave: transformación de modelos, lenguajes de transformación

Resumen. *La transformación de modelos constituye una tecnología esencial en los diversos enfoques de desarrollo de software dirigido por modelos. En los últimos años se han definido un buen número de lenguajes de transformación, pero todavía es necesaria una mejor comprensión de la naturaleza de las transformaciones de modelos y seguir investigando para encontrar las propiedades deseables de un lenguaje de transformación de modelos. En la actualidad el interés, tanto en el ámbito académico como industrial, se centra principalmente en la experimentación con lenguajes de transformación a través de la escritura de transformaciones para problemas reales.*

RubyTL es un lenguaje de transformación híbrido definido como un lenguaje específico del dominio embebido en el lenguaje de programación Ruby, y diseñado como un lenguaje extensible en el que un mecanismo de plugins permite añadir nuevas características al núcleo de características básicas. En este trabajo ofrecemos una presentación de RubyTL: el lenguaje, el algoritmo de transformación y el mecanismo de extensión; también se discutirán, finalmente, sus ventajas e inconvenientes.

1. INTRODUCCIÓN

La transformación de modelos constituye una tecnología clave para el éxito de los diversos enfoques de desarrollo de software dirigido por modelos (DSDM). Desde la propuesta de MDA por parte de OMG, se han definido varios lenguajes y herramientas de transformación de modelos como resultado de un intenso trabajo de investigación y desarrollo tanto desde la industria como desde el mundo académico. Sin embargo, todavía es necesario seguir investigando para conocer mejor la naturaleza de las transformaciones y las características deseables de un lenguaje de transformación. Por ello, el interés actual está centrado

principalmente en la experimentación con los lenguajes existentes a través de la escritura de definiciones de transformaciones para casos reales. En este sentido, marcos teóricos como el modelo de características discutido en [1] y la taxonomía de transformaciones de modelos presentada en [2] son muy útiles para comparar y evaluar las decisiones de diseño.

A principios de 2005, nuestro grupo arrancó un proyecto destinado a la creación de una herramienta para experimentar con las características de lenguajes de transformación híbridos cuyo estilo declarativo estuviese basado en una operación de *binding* similar a la encontrada en ATL [3]. El resultado de este proyecto es RubyTL, un lenguaje de transformación extensible creado como un lenguaje específico del dominio (DSL) embebido en el lenguaje de programación dinámico Ruby. RubyTL proporciona un mecanismo de extensión que posibilita que un conjunto básico de características pueda ser extendido con nuevas características mediante la creación de plugins que implementen algunos de los puntos de extensión predefinidos. En este trabajo, presentamos el lenguaje RubyTL, su algoritmo de transformación y su mecanismo de extensión. En [4][5] se puede encontrar una explicación más detallada de todos estos aspectos. Por ejemplo, en [4] se muestran los diagramas de características, de acuerdo con [1], de las características básicas y de los posibles puntos de extensión de lenguaje y en [5] se describen los puntos de extensión identificados y se presenta un ejemplo de plugin.

El trabajo está organizado en la siguiente forma. En el siguiente apartado se presenta el lenguaje a través de un ejemplo de definición de transformación. En el apartado 3 se describe el algoritmo de transformación y en el apartado 4 el mecanismo de extensión. Una comparación con otros lenguajes se comenta en el apartado 5 y finalmente se presentan las conclusiones.

2. EL LENGUAJE RUBYTL

RubyTL ha sido diseñado para satisfacer tres requisitos principales: i) de acuerdo con las recomendaciones expuestas en [6][7], debería ser un lenguaje híbrido ya que la expresividad declarativa es poco apropiada para transformaciones complejas que pueden requerir un estilo imperativo; el estilo declarativo debería estar basado en una operación *binding* similar a la proporcionada por ATL [3], ii) debería facilitar la experimentación con características del lenguaje, y iii) debería permitir una rápida implementación. Estos requisitos se han satisfecho a través de dos principales decisiones de diseño: la definición del lenguaje como un DSL embebido en Ruby y la incorporación de un mecanismo de extensión basado en plugins que permite añadir nuevas características a un núcleo básico. En esta sección describiremos estas características básicas a través de un ejemplo de definición de transformación. En [4] se puede encontrar una descripción más detallada del lenguaje.

Nuestra propuesta es independiente del lenguaje de programación dinámico elegido. Se ha optado por Ruby por tratarse de un lenguaje muy apropiado para embeber DSL en todo tipo de dominios de aplicación [8][9], y que además gana continuamente aceptación, especialmente desde el éxito del framework *Ruby on Rails*. Ruby es un lenguaje tipado dinámicamente con

una expresividad similar a Smalltalk, ya que también incorpora bloques de código y metaclases. La aplicación de la técnica de embeber un DSL en Ruby nos ha permitido disponer del lenguaje en un corto período de tiempo; además, ha resultado muy simple obtener la naturaleza híbrida del lenguaje puesto que el código Ruby puede ser integrado sin problemas en el DSL: *todo es código Ruby*.

Para mostrar un ejemplo sencillo de definición de transformación que ilustre las características básicas de RubyTL hemos considerado el problema clásico de transformación de un modelo de clases a un esquema relacional, según los metamodelos que muestra la Figura 1.

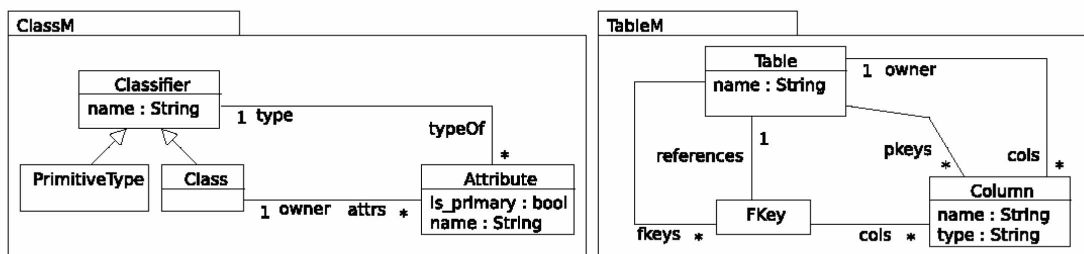

Figura 1. Metamodelos de clases y relacional

```
rule 'klass2table' do
  from  ClassM::Class
  to    TableM::Table

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

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

  mapping do |attr, column|
    column.name = attr.name
    column.type = attr.type.name
    column.owner.pkeys << column if attr.is_primary
  end
end

rule 'reference2column' do
  from    ClassM::Attribute
  to      Set(TableM::Column)
  filter  { |attr| attr.type.kind_of? ClassM::Class }
  end

  mapping do |attr, set|
    table = klass2table(attr.type)
    set.values = table.pkeys.map do |col|
```

```
        TableM::Column.new(:name => table.name + '_' + col.name
                           :type => col.type)
      end
      table.fkeys = TableM::FKey.new(:cols => set)
    end
  end
end
```

Como se puede observar en el ejemplo anterior, una definición de transformación consiste en un conjunto de reglas de transformación. Cada regla tiene un nombre y cuatro partes: la parte *from* que especifica la metaclase del elemento origen, ii) la parte *to* que especifica la metaclase del elemento destino (puede existir más de uno), iii) la parte *filter* que especifica la condición o filtro que deben satisfacer los elementos origen, y iv) la parte *mapping* que especifica las relaciones entre los elementos de los modelos origen y destino, bien de forma declarativa mediante un conjunto de *bindings*, o bien de forma imperativa utilizando instrucciones Ruby.

Un *binding* es una clase especial de asignación que permite declarar "**qué** necesita ser transformado en qué", en vez de "**cómo** debe ser algo transformado en algo". Esta instrucción tiene la siguiente forma `elemento_destino.propiedad = elemento_origen`. La regla `klass2table` incluye dos ejemplos de *bindings*.

En el ejemplo, la primera regla (`klass2table`) será ejecutada una vez por cada elemento cuya metaclase (o tipo) es `Class`, creándose un elemento cuya metaclase (o tipo) es `Table`. En la parte *mapping* de esta regla se establecen las relaciones entre las propiedades de una instancia de `Class` y las de una instancia de `Table`. El primer *binding* establece que el nombre de la tabla creada será el de la clase, y el segundo *binding* especifica cómo se transforman los atributos (instancias de `Attribute`) en columnas (instancias de `Column`). Es importante notar que mientras el primer *binding* se resuelve directamente porque el elemento origen es de un tipo primitivo, el segundo disparará la regla `property2column`, que especifica cómo se genera una columna a partir de un atributo de tipo primitivo, o la regla `reference2column`, que especifica cómo se genera un conjunto de columnas a partir de un atributo cuyo tipo es una clase. El filtro se encarga de seleccionar una u otra. En el siguiente apartado se explicará con más detalle este proceso de transformación.

Las reglas `property2column` y `reference2column` ilustran cómo puede escribirse código imperativo Ruby dentro de la parte *mapping*. En la regla `property2column`, dos *bindings* son seguidos de una instrucción Ruby que comprueba si un atributo debe ser convertido en una clave primaria, con lo cual sería añadido al conjunto de claves primarias de la tabla. La regla `reference2column` es un ejemplo de una relación uno a muchos. En esta regla todo el *mapping* está escrito en forma imperativa, un código Ruby que determina cómo es formado el conjunto de columnas que forman parte de la clave ajena.

3. ALGORITMO DE TRANSFORMACIÓN

El modelo de ejecución de RubyTL puede describirse a través de un algoritmo recursivo que es la base para identificar los puntos de extensión para los plugins. Cada definición de transformación debe tener al menos un punto de entrada en el que arranque el proceso de transformación. Por defecto, la primera regla de una definición de transformación es la *regla punto de entrada*, en el ejemplo sería la regla `klass2table`. Cuando arranca una transformación, se ejecuta cada *regla punto de entrada*, aplicándose la regla a todas las instancias de la metaclase especificada en su parte *from* (en el ejemplo a todas las instancias de `ClassM::Class`).

Por tanto, el algoritmo de transformación se puede escribir como un procedimiento principal cuya estructura es un bucle en el que se ejecutan las *reglas punto de entrada*. Para cada una de estas reglas, se crean los elementos del modelo destino que correspondan a cada elemento del modelo origen que satisfaga el filtro de la regla, y entonces se aplica la regla, esto es, se ejecuta su parte *mapping*.

```
reglas-de-entrada = "seleccionar reglas punto de entrada"
para cada regla R en reglas-de-entrada
    instancias-origen = "obtener todas las instancias del tipo origen
                        de la regla R"
    para cada instancia S en instancias-origen
         si S satisface el filtro de R entonces
             T = "crear instancias destino"
             Aplicar_regla(R, S, T)
```

El procedimiento `Aplicar_regla` itera sobre el conjunto de *bindings* en la parte *mapping* de una regla, y se distinguen dos casos: el *binding* se resuelve directamente cuando un valor primitivo (p.e. `table.name = klass.name`) debe ser asignado a la propiedad del elemento destino o el *binding* debe ser resuelto aplicando otras reglas (p.e. `table.cols = klass.attrs`). Como se dijo en el apartado anterior, la forma de un *binding* es `T.property = S`, por tanto `S` y `T` son parte del *binding*, y podemos expresar el procedimiento de la siguiente forma

```
Aplicar_regla (R : regla a aplicar,
                S : elemento origen, T : elementos destino)
     para cada binding B en R.bindings
          si B es primitivo entonces asignar valor a la propiedad
                            si no Resolver_binding(B)
```

El procedimiento `Resolver_binding` establece la resolución de un *binding* a través de un mecanismo de dos pasos. Primero se seleccionan todas las reglas que conforman con el *binding* y el elemento origen satisface el filtro. Una regla conforma con un *binding* cuando el tipo de la parte derecha del *binding*, S, es el mismo tipo (o un subtipo) que el tipo especificado en la parte *from* de la regla, y el tipo de la parte *to* es el mismo tipo (o un subtipo) que el tipo de la parte izquierda del *binding*. Los filtros deben asegurar que sólo una regla conforma con un *binding*. En segundo lugar, para cada regla seleccionada

se crean los correspondientes elementos destino que son conectados a los elementos ya existentes, y finalmente se aplica la parte *mapping* de la regla.

```
Resolver_binding(B : binding)
  S = elemento origen de B
  T = elemento destino de B
  P = propiedad de T especificada en B
  C = lista de reglas que conforman, inicialmente vacía

 para cada regla R en el conjunto de reglas de transformación
       si R conforma con B y R satisface el filtro
        añadir R a C
  para cada  regla R en C
       T' = crear instancias destino
       conectar T' con P de T
       Aplicar_regla (R, S, T')
```

Es importante resaltar la naturaleza recursiva del algoritmo y cómo esta recursión se produce implícitamente a través de los *bindings*: la recursión finaliza cuando en un *mapping* todos los *bindings* son asignaciones de valores de tipo primitivo o cuando se comprueba que un elemento destino ya ha sido generado.

4. MECANISMO DE EXTENSIÓN

RubyTL es un lenguaje extensible, es decir, es un lenguaje que ha sido diseñado como un núcleo de características básicas con un mecanismo de extensión que permite extender ese núcleo. En este apartado se comentan los aspectos básicos del mecanismo de extensión de RubyTL basado en el uso de plugins, y una explicación detallada se encuentra en [5]. En primer lugar, explicaremos las ideas que subyacen tras el diseño del lenguaje y a continuación comentaremos algunos ejemplos de extensiones del lenguaje.

El lenguaje puede ser extendido en varios aspectos dando lugar a tres categorías de extensiones: (1) relacionadas con el algoritmo de transformación, (2) relativas a la creación de nuevas reglas y a la gestión del ciclo de ejecución de las reglas, y (3) extensiones de la sintaxis del lenguaje. El algoritmo de transformación que se ha presentado en el apartado anterior permite identificar algunas operaciones que podrían modificar el comportamiento del algoritmo dependiendo de cómo sean implementadas. Estas operaciones pueden considerarse como *puntos de extensión* del algoritmo. Asimismo, dicho algoritmo de transformación establece un ciclo de ejecución para las reglas: instanciación, chequeo del filtro sobre un elemento origen, etc. La transición entre los estados por los que pasa cada regla está dirigida por dicho algoritmo. Cada estado está implementado por un método que define el comportamiento por defecto del lenguaje. Estos métodos son también puntos de extensión que permiten gestionar el ciclo de ejecución de las reglas y crear nuevos tipos de reglas. Además, la sintaxis del lenguaje puede ser extendida con nuevas palabras clave que permitan definir nuevas estructuras, y, por tanto, nuevas construcciones en el lenguaje. Cada extensión sintáctica tiene asociada una operación que es llamada cada vez que la palabra clave aparece

en la transformación y que implementa su semántica. Por todo ello, RubyTL puede considerarse como un framework que proporciona un conjunto de *puntos de extensión* que pueden ser implementados para añadir cierta funcionalidad al lenguaje a través de plugins desarrollados en Ruby.

Los puntos de extensión del lenguaje pueden implementarse utilizando dos estrategias diferentes. La primera de ellas utiliza el concepto de *hook*, un método asociado a un punto de extensión que puede ser redefinido por un plugin con el fin de modificar la funcionalidad por defecto del lenguaje. La segunda estrategia de implementación está basada en el uso de filtros que interceptan la invocación de un *hook* con el fin de añadir funcionalidad sin alterar la implementación del *hook*, pudiendo ejecutarse antes o después de su invocación. Esta estrategia está basada en el patrón *Intercepting Filter* empleado en el desarrollo de aplicaciones web [10]. Por tanto, los filtros ofrecen un mecanismo de colaboración entre plugins para la implementación de un punto de extensión. Es más, un filtro permite implementar determinadas extensiones al lenguaje sin interferir con otras existentes.

A continuación se describen algunas características del lenguaje implementadas como plugins:

- **Invocación explícita de reglas**. Por defecto, el algoritmo de transformación está basado en la ejecución implícita de reglas a través de *bindings*. Se ha definido un plugin que permite invocar a las reglas por su nombre en un fragmento de código imperativo. El plugin implementa un filtro asociado a la instanciación de las reglas donde se crea un método con el mismo nombre que la regla para que pueda ser llamado explícitamente.
- **Regla creadora**. En RubyTL una regla nunca transforma un elemento origen dos veces, ya que éste es el comportamiento esperado en la mayoría de ocasiones. Sin embargo, en ciertas transformaciones es necesario que una regla sea ejecutada más de una vez para un mismo elemento origen. Por este motivo, se ha añadido un plugin al lenguaje que añade la palabra clave "copy_rule" que define un nuevo tipo de regla e implementa el *hook* asociado a la aplicación de la regla para que siempre evalúe el *mapping*.
- **Transformación en fases**. Algunas transformaciones podrían ser simplificadas si pudieran ser expresadas en fases. Este concepto permite pensar en una transformación como un conjunto de pasos de refinamiento, donde cada fase confía en el trabajo realizado por las fases anteriores para cumplir su tarea. El plugin que implementa este tipo de transformaciones añade palabras clave al lenguaje e implementa varios puntos de extensión. En [5] se describe detalladamente la implementación de este plugin.

Otros ejemplos de plugins permiten: definir varios puntos de entrada en una transformación mediante la definición de reglas "top"; establecer *mappings* muchos-a-uno o especificar reglas transitorias, es decir, reglas que crean elementos que sólo existen durante la ejecución de la transformación. Por limitaciones de espacio no se comenta la implementación de estos plugins, pero en [4] y [5] pueden encontrarse los detalles de algunos de ellos.

5. TRABAJO RELACIONADO

Varios esquemas de clasificación de lenguajes de transformación han sido propuestos como [1][6]. De acuerdo con estas clasificaciones, los lenguajes de transformación pueden clasificarse en tres grandes categorías según su expresividad: imperativos, declarativos e híbridos. Como ejemplos de lenguajes podemos destacar QVT, ATL, Tefkat, MTF, MTL y Kermeta. OMG ha definido QVT [15] como lenguaje para expresar transformaciones. La propuesta de estándar, que aún no ha sido aceptada, establece un lenguaje declarativo y otro imperativo, que pueden ser implementados sobre un lenguaje base, que es un lenguaje de transformación con primitivas de bajo nivel. Actualmente no existe ninguna implementación completa de la propuesta actual.

MTL y Kermeta [11] son metalenguajes ejecutables imperativos que no han sido diseñados específicamente para transformaciones modelo-a-modelo, pero que son usados debido a que la versatilidad de sus construcciones proporciona una expresividad muy alta. Sin embargo, la verbosidad y el estilo imperativo propio de estos lenguajes dificulta la escritura de las transformaciones complejas debido a que su nivel de abstracción sobre los detalles de una transformación es muy bajo, lo que provoca definiciones de transformaciones muy largas y poco legibles.

ATL [3][12] es un lenguaje híbrido con una sintaxis muy clara. Incluye varios tipos de reglas que facilitan escribir reglas en un estilo declarativo. Sin embargo, la implementación completa del lenguaje no está disponible todavía, y sólo un tipo de regla puede ser utilizado, por lo que resulta complicado escribir algunas transformaciones de forma declarativa. ATL y RubyTL comparten las mismas abstracciones básicas, como regla y *binding*, pero ATL es tipado estáticamente, mientras que RubyTL tiene tipado dinámico.

Tefkat [13] es un lenguaje declarativo relacional con alta expresividad que está totalmente implementado. Como se indica en [2], escribir transformaciones complejas en un estilo completamente declarativo no es sencillo y el estilo imperativo puede resultar más apropiado. Por esta razón se recomiendan lenguajes de transformaciones híbridos para que tengan utilidad práctica.

MTF [14] es un conjunto de herramientas que incluye un lenguaje declarativo y bidireccional que incorpora la semántica de QVT para la transformación de modelos: chequeo y mantenimiento de la consistencia. Asimismo, MTF proporciona un mecanismo de extensibilidad que permite extender el lenguaje de expresiones. Por lo que conocemos, RubyTL es el primer lenguaje de transformación de modelos extensible en el sentido que proporciona un mecanismo que permite no sólo extensiones relacionadas con la sintaxis sino con el algoritmo de transformación y la creación de nuevos tipos de reglas. En cualquier caso, la idea de lenguaje extensible no es novedosa, sino que ha sido aplicada desde hace más de dos décadas, y que es ampliamente usada en el contexto del lenguaje Lisp que proporciona un potente sistema de macros para extender su sintaxis.

6. CONCLUSIONES

El éxito del DSDM requiere de lenguajes de transformación de modelos apropiados. Estos lenguajes deberían tener buenas propiedades, como ser usables y tener una expresividad apropiada para escribir transformaciones complejas. Recientemente, se han definido un buen número de lenguajes y han sido identificados requisitos de calidad en propuestas como [2]. Actualmente, la experimentación con estos lenguajes es una actividad clave dentro del ámbito del DSDM.

RubyTL es un lenguaje híbrido extensible que proporciona expresividad declarativa a través de una construcción de *binding*. Es un lenguaje usable con una sintaxis clara y un buen equilibrio entre concisión y verbosidad. Además, su aprendizaje no resulta complicado y requiere conocimientos mínimos de Ruby.

Las ventajas principales que proporciona RubyTL con respecto a otros lenguajes no extensibles son: i) el lenguaje puede ser adaptado a una familia particular de de problemas de transformación, ii) nuevas construcciones pueden ser añadidas sin modificar el núcleo básico, y iii) proporciona un entorno para experimentar con las características de los lenguajes de transformación. Varios plugins están ya disponibles como hemos comentado en el apartado 4.

Una limitación de RubyTL es que la extensibilidad está limitada a una determinada familia de lenguajes de transformación de modelos, aquellos basados en el concepto de *binding*. No obstante, la idea puede ser aplicada para otros tipos de lenguajes. Asimismo, actualmente estamos investigando cómo controlar que no se puedan conectar plugins incompatibles.

En nuestros experimentos con RubyTL hemos identificado e implementado un conjunto de características deseables para este tipo de lenguajes: reglas que se puedan ejecutar en varias fases, la necesidad de cuatro tipos diferentes de reglas: *normales* como las del ejemplo de este trabajo, *top* para permitir tener más de un punto de entrada, *copy* que posibilitan que un mismo elemento origen pueda ser transformado más de una vez, y *transient* que generan elementos que sólo existen mientras se ejecuta la transformación. En la actualidad se ha desarrollado un plugin que soporta transformaciones muchos-a-uno, aunque estamos estudiando la mejora de la eficiencia de este tipo de transformaciones. Otra importante característica sobre la que estamos trabajando, no considerada en muchos lenguajes, es la consistencia incremental. Asimismo, cabe señalar la creación de un plugin que permite la invocación explícita de reglas dentro del código imperativo del lenguaje.

Hemos creado un entorno de desarrollo dirigido por modelos denominado AGE (*Agile Generative Enviroment*) que integra un editor con resaltado de sintaxis y asistencia de contenido para el lenguaje RubyTL, una herramienta de configuración de plugins, un motor de plantillas para las transformaciones modelo-código y un editor de metamodelos. Este entorno es interoperable con otros entornos de metamodelado basado en MOF a través de XMI. Finalmente, destacar que estamos escribiendo transformaciones para problemas reales, por ejemplo estamos aplicando RubyTL en un proyecto relacionado con el desarrollo de portlets en colaboración con el grupo de Oscar Díaz de la Universidad del País Vasco.

REFERENCIAS

[1] Krzysztof Czarnecki and Simon Helsen. *Classification of model transformation approaches*. In *Proceedings of the 2nd OOPSLA Workshop on Generative Technique in the Context of the Model Driven Architecture*, Anaheim, October 2003.

[2] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. *A taxonomy of model transformations*, 2005.

[3] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. In *OOPSLA 2003 Workshop*, Anaheim, California, 2003.

[4] Jesús Sánchez, Jesús García, and Marcos Menárguez. RubyTL: *A Practical, Extensible Transformation Language*. In *2nd European Conference on Model Driven Architecture*, volume 4066, pages 158–172. *Lecture Notes in Computer Science*, June 2006.

[5] Jesús Sánchez, and Jesús García: *A plugin-based language to experiment with model transformation*. In proceedings of 9th International Conference MoDELS 2006, Genova, Italy. October 2006.

[6] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. IEEE Software, 20(5):42–45, Sept./October 2003.

[7] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. Review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards final Standard, 2003.

[8] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, June 2005.
http://www.martinfowler.com/articles/ languageWorkbench.html.

[9] Jim Freeze. *Creating DSLs with Ruby*, March 2006.
http://www.artima.com/rubycs/articles/ruby as dsl.html.

[10] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns*: *Best Practices and Design Strategies*. 2001.

[11] Pierre Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. *On executable meta-languages applied to model transformations*. In *Model Transformations in Practice*, Jamaica, 2005.

[12] 14. Frédric Jouault and Ivan Kurtev. *Transforming models with ATL*. In *Model Transformations in Practice Workshop*, Montego Bay, Jamaica, 2005.

[13] Michael Lawley and Jim Steel. *Practical Declarative Model Transformation* with *Tefkat*. In *Model Transformations in Practice Workshop*, Montego Bay, Jamaica, 2005.

[14] Sebastien Demathieu, Catherine Griffin, and Shane Sendall. *Model Transformation with the IBM Model Transformation Framework*, 2005. http://www-128.ibm.com/developerworks/rational/library/05/503 sebas/index.html.

[15] OMG, *MOF 2.0 Query/View/Transformation Final Adopted Specification,* Noviembre, 2005, http://www.omg.org/docs/ptc/05-11-01.pdf.