

RubyTL a ATC: un caso real de transformación de transformaciones

Jesús Sánchez Cuadrado Eduardo Victor Sánchez Rebull

Jesús J. García Molina

Antonio Estévez García

Universidad de Murcia

Open Canarias S.L.

Murcia

Santa Cruz de Tenerife

jesusc, jmolina@um.es

vsanchez,aestevez@opencanarias.com

Resumen

Las transformaciones de modelos constituyen una pieza clave para el éxito de los proyectos desarrollados bajo los principios del DSDM, sin las cuales, la automatización en la evolución de los distintos artefactos manejados puede llegar a resultar muy difícil de precisar. Con el retraso en la llegada del estándar QVT y la aparición de algunos lenguajes alternativos muy prometedores, nos acercamos a un paisaje de opciones variadas pero, lamentablemente, inconexas. En este trabajo presentamos un caso real de traducción (compilación) desde un lenguaje de transformaciones de alto nivel de abstracción (RubyTL) a uno de bajo nivel (ATC), el cual ha sido concebido expresamente para ser objeto de tales traducciones.

1. Introducción

Todo lenguaje de transformaciones de modelos necesita de un soporte de implementación para poder ser ejecutado en un sistema computacional. La pieza de software encargada de ello se denomina comúnmente *motor de transformaciones de modelos*. En general, cada motor se ocupa de resolver el soporte a un único lenguaje, lo cual en sí mismo ya resulta un empeño de considerable envergadura.

Se dan casos esporádicos de especificaciones de arquitecturas de lenguajes que abarcan dos o más lenguajes. Así sucede, por ejemplo, con el estándar QVT [11], que desdobra las partes declarativa e imperativa en lenguajes distintos,

pero con cierto grado de complementación mutua. Otros lenguajes, como ATL [8], son híbridos. En ellos coexisten, de cara a potenciar su expresividad semántica, ambos tipos de sintaxis, con el consiguiente aumento de la complejidad inherente a la implementación de su respectivo motor de soporte, complejidad que en este caso es indivisible. Naturalmente, desarrollar un motor que abarque parte o la totalidad de lenguajes pertenecientes a la misma especificación o a especificaciones independientes será tanto o más costoso cuanto mayor sea el número de lenguajes distintos que deba ser capaz de ejecutar.

El soporte a lenguajes de transformaciones que una herramienta DSDM suele ofertar estará íntimamente ligado al motor o motores que tenga instalados y disponibles. Normalmente un solo motor se corresponde con un único lenguaje. Existen varios mecanismos mediante los cuales es posible aumentar esta oferta. Por ejemplo, si la herramienta permite la integración simultánea de varios motores, cada uno de ellos aportará soporte a su respectivo lenguaje. También se puede modificar un motor para ampliar el número de lenguajes que éste es capaz de soportar, etc. Una alternativa es eludir la responsabilidad de las transformaciones y cederla a sistemas auxiliares especializados, que pueden incluso operar en entornos distribuidos.

Una posibilidad muy atractiva, de la que se ocupa este trabajo, se basa en el uso de un lenguaje provisto de primitivas de bajo nivel, que en nuestro caso se denomina Atomic Transformations

mation Code (ATC), al que pueden ser traducidos múltiples lenguajes para lograr su soporte, indirectamente, en la herramienta mediante la traducción de sus manifestaciones a este lenguaje, en un proceso llamado transformaciones de PIT (Platform-Independent Transformation) a PST (Platform-Specific Transformation), según [3]. ATC es interpretado y ejecutado por un motor llamado Virtual Transformation Engine (VTE), cuya única finalidad es precisamente dar soporte de ejecución a este lenguaje.

En este artículo presentamos un caso real de transformación de transformaciones a modo PIT-PST a partir de un lenguaje de transformaciones de modelos de propósito general, como es RubyTL, presentado en el taller DSDM [5], hacia ATC. En realidad, como se aprecia en [3], la idea no es nueva y ha sido incluso presentada anteriormente en el taller DSDM [10]. Ideas muy parecidas se discuten en [9], donde se pone en juego un lenguaje llamado ATL Virtual Machine, de características similares a ATC. Como ejemplo práctico de este tipo de aproximaciones, ver [7].

El artículo se estructura como sigue: En las secciones 2 y 3 se presentan ATC y RubyTL, respectivamente, enfatizando aquellos aspectos relevantes para la transformación entre ambos; en la Sección 4 se ilustra el lenguaje RubyTL-M; en la Sección 5 se discute el esquema de traducción seguido, mientras que en la Sección 6 se discute la transformación implementada; en la Sección 7 se entra a valorar ambos lenguajes en cuanto a su potencial y sus limitaciones. Cerramos con las conclusiones en la Sección 8.

2. El lenguaje ATC

ATC ha sido concebido como un lenguaje imperativo de bajo nivel de abstracción, para facilitar la traducción de otros lenguajes, generalmente caracterizados por un mayor nivel de abstracción que él mismo. Que sea imperativo hace que se produzca un buen alineamiento con respecto a aquellos lenguajes origen que compartan dicha característica, mientras que para los lenguajes de naturaleza declarativa (o

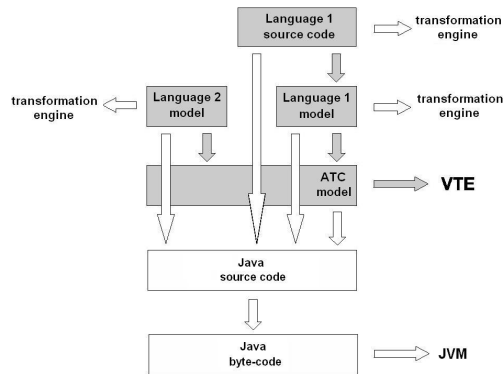


Figura 1: Posibles caminos para la ejecución de lenguajes de transformaciones de modelos de naturaleza diversa. Para los casos de lenguajes con sintaxis concreta gráfica se puede proceder directamente desde su contenido, o reconvertir la información a sintaxis concreta textual o abstracta.

la parte declarativa de los lenguajes híbridos), la traducción debe dejar impronta explícita de sus respectivos algoritmos de resolución de reglas, en los modelos ATC resultantes.

En la Figura 1 se muestran diversas vías de acometida para implementar la ejecución de los lenguajes de transformación, según las representaciones que pueden tomar sus instancias. Se puede, por ejemplo, analizar sintácticamente una entrada textual (o gráfica) y ejecutar su contenido directamente (el motor estará ejerciendo de intérprete), o convertirla en un modelo equivalente y ejecutar éste. O puede que un lenguaje no tenga representación concreta y se ejecute directamente a partir de sus modelos descritos en términos de su sintaxis abstracta. También es posible en cualquier caso traducir las definiciones de transformación origen a un lenguaje intermedio, como puede ser ATC, que será ejecutado por su propio motor. O puede ser incluso traducido a un lenguaje de programación tradicional típico, como Java, quizá para aumentar la eficiencia de la ejecución. En este último caso el motor correspondiente funciona como un compilador.

Un caso que no está contemplado en esta figura atañe a los lenguajes carentes de re-

presentación abstracta. En este grupo se encuentra RubyTL. Para este caso, los motores pueden realizar un análisis sintáctico y ejecutar directamente la información descrita en sus transformaciones originales, sin pasos intermedios. Este camino de ejecución coincide con el explicado para los lenguajes que, aun disponiendo de sintaxis abstracta, prescinden de ella para proceder a la ejecución directa desde el origen.

Cada una de estas aproximaciones tiene sus pros y sus contras, que no entraremos a valorar aquí. Tan sólo destacar que para poder poner en juego un lenguaje como ATC y aprovechar sus ventajas, se requiere la posibilidad de traducir definiciones de transformación descritas en términos del lenguaje origen en forma de sintaxis concreta o abstracta. Y concretamente, para que esta traducción pueda formalizarse como una transformación de modelos se requiere que los elementos de origen, dado que los elementos destino en el caso de ATC ya lo son, tomen la forma de modelos.

Esta transformación es especial porque los modelos implicados son en sí mismos transformaciones, y la transformación puede verse como un caso PIT-PST [3] representativo. Esta transformación, que podemos denominar transformación de traducción, convierte un lenguaje de transformaciones a otro y es única para cada par de lenguajes mapeados, es decir, se formaliza una vez y sirve para todas las definiciones de transformación descritas en el lenguaje origen para obtener sus equivalentes en el lenguaje destino. Y viceversa si la transformación es bidireccional, o en aquellos casos donde sea posible obtener su inversa.

Es interesante ver que la transformación puede ir creciendo gradualmente a medida que se añade soporte de traducción a las distintas partes del lenguaje origen, lo que implica que será compatible con un subconjunto progresivamente mayor de definiciones de transformaciones escritas para el lenguaje original, en un proceso de naturaleza incremental.

2.1. Características de ATC

ATC, por el papel que desempeña, ha de cumplir el requisito de ser exhaustivo en cuan-

to a sus capacidades expresivas, enfocadas en preferencia a las actividades inherentes a las transformaciones de modelos, pero sin descuidar los mecanismos que generalmente se usan durante la programación tradicional. Asimismo, se ha tenido especial cuidado en mantener un alto grado de eficiencia en la ejecución de sus instancias. Estos dos principios, de exhaustividad y eficiencia, determinan la naturaleza y composición del lenguaje, su diseño y su implementación, que se exponen en [6], aunque se repasarán aquí brevemente sus principios generales de funcionamiento.

Las definiciones de transformación de modelos ATC toman la forma de modelos. Como no existe una sintaxis textual “oficial” para el lenguaje, el texto ATC que aparece en las próximas secciones no responde a ninguna sintaxis específica, aunque se ha cuidado para que resulte de fácil comprensión.

Las instrucciones ATC representan elementos constructivos de bajo nivel que, combinados convenientemente proporcionan toda la potencia expresiva del lenguaje. Por este motivo se les llama *átomos*, que se manifiestan como elementos de modelo. Y sus correspondientes metaclasses, las que heredan de `AtcAtomType`, reciben el nombre de *tipos de átomo*.

Muchos tipos de átomo involucran relaciones de composición con otros átomos que les asisten en el cumplimiento de sus respectivas actividades, como por ejemplo, el tipo `AtcGetStructuralFeature`. Los átomos ATC instancias de este tipo contienen un String llamado `elementId` que indica el nombre de la variable donde se almacena el elemento a consultar, una referencia a un átomo llamada `featureNmProvider` que proporciona el nombre de la propiedad perteneciente a este elemento cuyo valor asociado es el objetivo de nuestra consulta, y otro String, `targetVarId`, que representa el nombre de la variable destino donde se guardará el resultado de la consulta.

Un ejemplo típico de un átomo `AtcGetStructuralFeature` con su propio estado se muestra a continuación:

```
attrs <- AtcGetFeature aClass.attrs
```

Donde el nombre de propiedad `attrs` viene proporcionado por un átomo hijo `featureNmProvider`, en este caso de tipo `AtcString`. El valor `s` de `AtcString` está representando una constante. Este átomo toma la siguiente forma:

```
AtcString s = 'attrs'
```

Usando alternativas, como `AtcGetStringFromVar` o cualquier otro tipo de átomo cuyas instancias retornen un valor compatible con `AtcString` al ejecutarse, podemos ejercer control sobre qué propiedad va a ser consultada exactamente dentro de las disponibles en `aClass` (y relegar esta decisión al momento de la ejecución). Esta flexibilidad es crucial, por ejemplo, para dar soporte a variantes dinámicas de ejecución, como sucede en el campo de las líneas de productos de software, con la producción de familias de modelos a partir de modelos de características (ver [2]). Además, el sobreesfuerzo por tener que “programar” los átomos de esta manera es prácticamente inapreciable.

3. El lenguaje RubyTL

RubyTL es un lenguaje de transformación de modelos que posee la característica de que ha sido implementado como un DSL embebido dentro del lenguaje Ruby [13]. Es un lenguaje híbrido, cuya parte declarativa está basada en reglas y *bindings*, mientras que la parte imperativa viene dada por el propio lenguaje Ruby.

Las siguientes dos reglas formarían parte de una definición de transformación de un modelo de clases (`ClassM`) a un modelo relacional (`TableM`). Las partes `from` y `to` de una regla indican los elementos origen y destino respectivamente, mientras que la parte `mapping` permite especificar las relaciones (*bindings*) entre los elementos origen y destino.

```
top_rule 'class2table' do
  from ClassM::Class
  to   TableM::Table
  mapping do |klass, table|
    table.name = klass.name
    table.cols = klass.attrs
```

```
  end
end

rule 'attr2column' do
  from   ClassM::Attribute
  to     TableM::Column
  mapping do |attr, column|
    column.name = attr.name
  end
end
```

La regla `class2table` será ejecutada una vez por cada elemento origen cuya metaclassa es `Class`, creándose un elemento destino cuya metaclassa es `Table`. En la parte *mapping* de esta regla hay dos *bindings* que relacionan las propiedades de los elementos origen y destino. 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 `attr2column`, que especifica cómo se genera una columna a partir de un atributo.

Al ser RubyTL un DSL embebido dentro de otro lenguaje, no existe un metamodelo subyacente que dé soporte a su sintaxis abstracta. Esto es una limitación a la hora de manipular de manera sencilla definiciones de transformación escritas en RubyTL.

Otra característica novedosa de RubyTL es la posibilidad de organizar una definición de transformación en varias fases[4]. 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.

El mecanismo de fases proporciona un medio para descomponer una transformación en varios pasos, lo que permite manejar la complejidad y aumentar la legibilidad.

4. RubyTL-M

RubyTL-M es una versión simplificada de RubyTL, con la diferencia principal de que en su diseño se ha partido de un metamodelo que describe su sintaxis abstracta. Puesto que RubyTL fue diseñado como un DSL embebido, no es posible manipular una definición de transformación como un modelo, lo cual es necesario para transformar¹ modelos RubyTL a ATC. De ahí la necesidad de diseñar RubyTL-M. En este artículo no se abordará la construcción de un *parser* que convierta la sintaxis textual de RubyTL en un modelo conforme a su sintaxis abstracta.

Al igual que RubyTL, el lenguaje RubyTL-M también está basado en reglas y *bindings* para especificar qué elementos se transforman en qué elementos, aunque existen algunas diferencias significativas entre ambos lenguajes.

1. RubyTL-M es un lenguaje tipado estáticamente, ya que aprovecha la compilación (en este caso a ATC), para realizar comprobación de tipos y resolver reglas de manera eficiente. En cambio, RubyTL es un lenguaje tipado dinámicamente puesto que las comprobaciones de tipo y la resolución de reglas sólo pueden hacerse en tiempo de ejecución.
2. Mientras que en RubyTL es posible escribir cualquier expresión válida del lenguaje Ruby en cualquier lugar de la definición de transformación, el lenguaje de expresiones de RubyTL-M es más simple y sólo es posible escribir expresiones en la parte *mapping* de una regla.

En la Figura 2 se muestra el metamodelo de RubyTL-M. Las dos reglas RubyTL del ejemplo de la Sección anterior conformarían con el metamodelo de RubyTL-M.

4.1. Implicaciones del tipado estático

En un lenguaje de transformación el hecho de que sea “realmente” tipado estáticamente tiene una implicación importante: toda definición de

¹en el sentido de transformaciones de modelos.

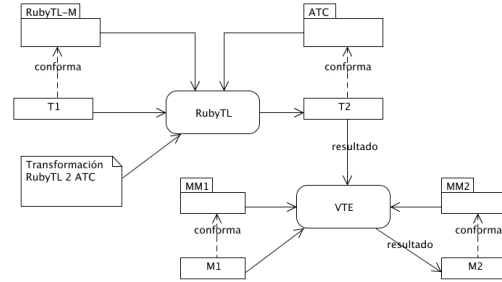


Figura 3: Esquema de compilación de RubyTL-M a ATC.

transformación debe hacer referencia explícita a los metamodelos origen y destino. Esto es así porque el tipo de los objetos que se van a manipular en la definición de transformación está definido por sus metamodelos, y por tanto el compilador de la transformación necesita conocerlos para poder asegurar la corrección de las expresiones en tiempo de compilación.

En el caso de RubyTL-M, esta condición se refleja en el metamodelo porque el tipo de cualquier variable es una metaclass de otro metamodelo (véase el atributo *type* en la Figura 2).

Un problema potencial de esta aproximación está relacionado con la compilación incremental utilizada en los editores de código modernos. Los metamodelos origen y destino deben existir y ser completos al mismo tiempo que se edita la definición textual de la transformación para poder compilarla.

5. Esquema de compilación

En la Figura 3 se muestran los elementos implicados en el proceso de traducción y la posterior ejecución de la transformación.

El punto de partida de la traducción es un modelo conforme al metamodelo del lenguaje RubyTL-M. Este modelo corresponde a una definición de transformación. El resultado de la traducción es un modelo conforme al metamodelo del lenguaje de bajo nivel ATC. Es decir, el código RubyTL es compilado a un código ejecutable, que está expresado conforme

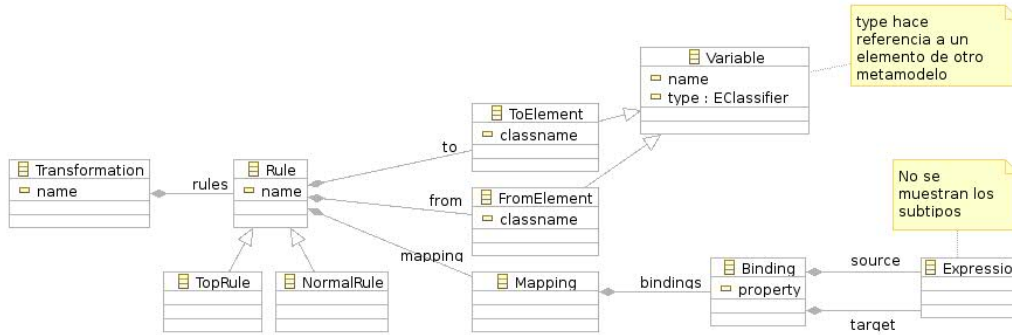


Figura 2: Metamodelo de RubyTL-M.

al metamodelo de ATC.

La definición de transformación, que al ser ejecutada efectuará el proceso de traducción, ha sido implementada en RubyTL [13].

Una vez compilado el modelo origen es posible alimentar el motor de transformación VTE con el modelo ATC resultante, de manera que la definición de transformación especificada en RubyTL-M es finalmente ejecutada por el motor de transformación de ATC, esto es VTE.

Por último, destacar que el motor de transformación VTE debe recibir como entrada un modelo conforme al metamodelo origen *MM1* que se pretendía transformar en la definición de transformación RubyTL-M (*T1*), y el resultado es un modelo conforme al metamodelo destino *MM2*, tal como vendrá especificado en *T1*.

6. Compilación de RubyTL-M a ATC

La compilación de una definición de transformación en RubyTL-M a ATC se ha implementado como una transformación modelo-modelo, escrita en RubyTL, que transforma un modelo conforme al metamodelo RubyTL-M en un modelo conforme al metamodelo de ATC. En esta sección se explicarán los aspectos más importantes de la transformación realizada.

En [12] se describe el algoritmo de transfor-

mación de RubyTL, que está dividido en tres funciones mutuamente recursivas. El algoritmo se ha adaptado para RubyTL-M, creando una versión “estática” del mismo, esto es, en vez de existir tres funciones genéricas se han generado funciones ATC específicas para cada regla. En concreto, se genera una función `aplicarRegla` por cada regla, y una función adicional `recorrerRegla` para cada regla *top*:

- **aplicarRegla.** Ejecuta el mapping de la regla, esto es, su conjunto de bindings. El código generado para esta función consistirá en la evaluación de las expresiones de la parte derecha e izquierda de los bindings y la llamada a otras funciones `aplicarRegla` para transformar la parte derecha de cada binding en la izquierda.
- **recorrerRegla.** Para las reglas *top* esta regla se encarga de recorrer todos los elementos del tipo origen, crear los elementos destino correspondientes y llamar a `aplicarRegla` para ejecutar su conjunto de *bindings*.

De esta forma, para la transformación mostrada en la Sección 3, el código generado para la regla `class2table` tendrá la siguiente estructura:

```
AtcFunction 'recorrerRegla_class2table'
model <- GetModel('ClassM')
AtcForEach obj <- model
```

```

AtcIf obj instanceof Class
  table <- AtcCreateElement 'Table'
  aplicarRegla_class2table(obj, table)

AtcFunction 'aplicarRegla_class2table'
  AtcParameter 'class'
  AtcParameter 'table'

# table.name = class.name
name <- AtcGetFeature class.name
AtcSetFeature table.name <- name

# table.columns = class.attrs
attrs <- AtcGetFeature class.attrs
AtcForEach attr <- attrs
  col <- AtcCreateElement 'Column'
  AtcSetFeature table.columns <- col
  aplicarRegla_attr2column(attr, col)

```

La función `recorrerRegla_class2table` obtiene el modelo origen y recorre todos los elementos. Cada vez que encuentra un elemento del tipo indicado en la parte `from` de la regla, `Class` en este caso, crea un elemento destino y llama a la función `aplicarRegla_class2table`.

La función `aplicarRegla_class2table` ejecuta el *mapping* de la regla. Para esto, primero evalúa la asignación entre tipos primitivos `table.name = class.name`, estableciendo el valor del nombre de la tabla. Después ejecuta el *binding* `table.columns = class.attrs`. Para ello evalúa la expresión en la parte derecha del *binding* (`class.attrs`) y obtiene un resultado. En este caso, el resultado es una lista de elementos, que es recorrida, y para cada elemento origen se crea un elemento destino, de acuerdo al tipo de la parte `to` de la regla `attr2column`. Esta regla se evaluará a continuación mediante la llamada a `aplicarRegla_attr2column(attr, col)`

A continuación se describe cómo es posible crear código estático para la resolución de *bindings*.

6.1. Resolución de bindings en tiempo de compilación

Un aspecto importante que se ha logrado gracias a la compilación a un lenguaje de bajo ni-

vel como es ATC, es que es posible determinar el conjunto de reglas que resuelven un *binding* en tiempo de compilación. Esta mejora implica un incremento del rendimiento con respecto a RubyTL, donde deben recorrerse todas las reglas antes de resolver cada *binding*.

Así, dado un *binding* en la forma `expizq.propiedad = expder` el algoritmo para generar código se basa en los siguientes puntos:

- Generar átomos ATC para la expresión `expder`. Almacenar el tipo de la expresión en la variable `tipoder`
- Generar átomos ATC para la expresión `expizq`. Almacenar el tipo de `expizq.propiedad` en `tipoizq`.
- Para cada regla r_i de la transformación, seleccionarla si:
 1. `tipoder` coincide o es un subtipo del tipo de r_i .`from`, y
 2. `tipoizq` coincide o es un supertipo del tipo de r_i .`to`

En el ejemplo de transformación a ATC mostrado anteriormente, la regla seleccionada para resolver el *binding* `table.cols = klass.attrs` es `attr2column`.

- Para cada regla seleccionada, crear un elemento del tipo destino y ejecutar el código de la parte *mapping*. Esto es, ejecutar la función `aplicarRegla` correspondiente, en el caso anterior `aplicarRegla_attr2column`.

Este algoritmo resulta sencillo de implementar de manera imperativa. Puesto que en RubyTL es posible escribir código imperativo dentro de la parte *mapping* de una regla, la implementación del algoritmo ha sido sencilla escribiendo una pequeña cantidad de código Ruby imperativo.

6.2. Comprobación de tipos

Puesto que RubyTL-M es un lenguaje tipado estáticamente es necesario determinar el tipo de cada expresión antes de transformarla a su correspondiente representación en ATC. De

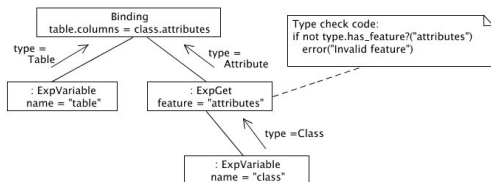


Figura 4: Ejemplo de cálculo de expresiones. La información de tipo se propaga desde los nodos hijos a los padres.

esta forma se evita generar código de comprobación de tipos en tiempo de ejecución. Este aspecto también es importante para poder implementar el algoritmo anterior de resolución de *bindings*.

Dado que que las expresiones se organizan en forma de árbol, el problema que aparece es cómo obtener el tipo de una sub-expresión para poder calcular el tipo de la expresión padre. Por ejemplo, dada la expresión padre `class.attrs`, su tipo se obtiene consultando el tipo de la característica `attrs` en el tipo de la sub-expresión `class`. El problema es que el tipo de esta sub-expresión (`class` en este caso), debe ser propagado hacia la expresión padre. La Figura 4 muestra esta situación.

La solución que se ha adoptado ha sido anotar los nodos del árbol de expresiones (elementos del modelo origen) con información de tipo, utilizando un esquema similar a los *atributos sintetizados* de las definiciones dirigidas por la sintaxis [1].

6.3. Organización de la transformación en fases

La definición de transformación para la compilación de RubyTL-M a ATC, ha sido implementada utilizando el mecanismo de fases de RubyTL, comentado brevemente en la Sección 3.

En la primera fase se ha creado el código de inicialización de ATC, como por ejemplo definir los parámetros de la transformación (modelo origen/destino), obtener los elementos del modelo origen, y crear la función *main* de ATC. También se crean las funciones men-

cionadas anteriormente para cada regla, pero sin completar el cuerpo de las mismas. Así, en esta fase se crea la estructura del modelo destino (ATC), pero sin completar.

La segunda fase crea código ATC para las expresiones de RubyTL-M. Al mismo tiempo calcula el tipo de cada expresión y realiza la comprobación de tipos en “tiempo de compilación”.

La tercera fase es una continuación de la primera fase, en la cual se retoma la definición de las funciones, pero ahora se dispone de toda la información necesaria para generar su contenido. Es en esta fase donde se implementa el algoritmo de resolución de *bindings*, utilizando para ello las expresiones cuyo tipo se ha calculado en la fase anterior.

Con esta organización se ha conseguido un mayor grado de modularidad que si se hubiera implementado como un único conjunto de reglas.

Por ejemplo, si se decidiera implementar el mecanismo de fases de RubyTL en RubyTL-M, sería posible encapsular la traducción a ATC en una fase. Esta fase crearía los elementos ATC necesarios para invocar las reglas (implementadas como funciones ATC) cuando fuese necesario. De esta forma, es posible desacoplar la traducción de las estructuras de control (las fases, en este caso), y la aplicación de reglas.

7. Valoración

En esta experiencia se ha podido comprobar que ATC, a pesar de ser un lenguaje de bajo nivel de abstracción, permite manipular con sencillez los elementos de los modelos origen y destino. También es posible implementar un lenguaje de expresiones utilizando los mecanismos explicados en la Sección 2. Una decisión de diseño que ha demostrado ser acertada es el hecho de que ATC permita organizar el código en funciones, ya que resulta muy útil a la hora de estructurar el código ATC generado, evitando repeticiones y facilitando la depuración.

Por otra parte, la experiencia también ha puesto de manifiesto la necesidad de que ATC

ofrezca mecanismos de más alto nivel, comunes a cualquier lenguaje de transformación, como pueden ser facilitar aún más la obtención del modelo origen, potenciar la creación de estructuras de datos auxiliares (por ejemplo, para ayudar a implementar estructuras de traza), o la posibilidad de definir variables globales que mantengan el estado de la transformación o para propósitos de configuración.

También es interesante poder evaluar la capacidad que ATC actualmente ostenta para tratar con tipos dinámicos, como hace RubyTL, y con elementos de modelo en contextos donde el tipo específico no sea un aspecto crucial, como al acceder a la propiedad `name` de un elemento cuando, sin importar el tipo concreto del elemento, se puede asegurar que dicha propiedad está presente en él. Aunque el lenguaje ya consta de componentes que habilitan consultas reflectivas sobre los elementos, un aumento de la experiencia práctica en este campo podría contribuir a mejorar este aspecto del lenguaje.

En cuanto a RubyTL, la experiencia ha mostrado que para poder compilar expresiones de manera práctica, es necesario un mecanismo para anotar los nodos del árbol de sintaxis abstracta, bien sea con atributos sintetizados o con otro mecanismo. En el caso de RubyTL, ha sido posible implementar de manera sencilla atributos sintetizados porque está basado en Ruby, un lenguaje dinámico que permite modificar en tiempo de ejecución la estructura de las clases (metaclases en este caso).

8. Conclusiones y trabajo futuro

En este artículo se ha presentado una experiencia de compilación de un lenguaje de transformación de modelos, a modo PIT-PST, utilizando para ello otro lenguaje de transformación². El lenguaje que se ha compilado es RubyTL-M, una versión simplificada de RubyTL, el lenguaje que se ha utilizado como medio para realizar la compilación. El resultado de la transformación es un modelo conforme al lenguaje ATC, que es ejecutable uti-

²La implementación realizada puede descargarse en <http://gts.inf.um.es/age>.

lizando el motor VTE. Como consecuencia se logra ampliar el catálogo de transformaciones de modelos para el segundo lenguaje con todas las transformaciones disponibles en el primero.

Como trabajo futuro se pretende ampliar RubyTL-M con un lenguaje de expresiones más rico, incluyendo un lenguaje de consultas como pueda ser OCL. Implementar un lenguaje de expresiones más rico requerirá añadir comprobación de tipos en tiempo ejecución en el algoritmo de resolución de *bindings*.

Por su parte, ATC aún debe completar algunos aspectos para aumentar su adecuación a las traducciones desde otros lenguajes. No cabe duda que la experiencia con RubyTL-M ha sido muy provechosa para sacar conclusiones sobre su idoneidad como lenguaje intermedio.

9. Agradecimientos

Trabajo parcialmente subvencionado por el *Ministerio de Educación y Ciencia* (PTQ2004-1495, PTR1995-0928-OP), el *Fondo Social Europeo* y la DGUI, Consejería de Educación, Cultura y Deportes, Gobierno de Canarias (PI042005/007), y por la Consejería de Educación y Cultura de la CARM (TIC-INF 06/01-0001).

Gracias también a la Red de Desarrollo de Software Dirigido por Modelos (DSDM), ref: TIN2005-25866-E.

Referencias

- [1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.
- [2] O. Avila-García, A. E. García, V. S. Rebull, and J. L. R. García. Using software product lines to manage model families in model-driven engineering. In *SAC 2007: Proceedings of the 2007 ACM Symposium on Applied Computing, track on Model Transformation*, pages 1006–1011. ACM Press, Mar 2007.

- [3] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective Model Driven Engineering, 2003. Available at <http://www.lina.sciences.univ-nantes.fr/Publications/2003/BFJLP03>.
- [4] J. S. Cuadrado and J. G. Molina. A phasing mechanism for model transformation languages. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1020–1024, New York, NY, USA, 2007. ACM Press.
- [5] J. S. Cuadrado, J. J. G. Molina, and M. M. Tortosa. RubyTL: un Lenguaje de Transformación de Modelos Extensible. In *DSDM'06: Proceedings of the III Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones*, Oct 2006.
- [6] A. Estévez, J. Padrón, V. Sánchez, and J. L. Roda. ATC: A low-level model transformation language. In *MDEIS 2006: Proceedings of the 2nd International Workshop on Model Driven Enterprise Information Systems*, May 2006.
- [7] F. Jouault. ATL Use Case - QVT to ATL Virtual Machine Compiler, Jun 2007. <http://www.eclipse.org/m2m/atl/usecases/QVT2ATLVM/>.
- [8] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005: Proceedings of the Model Transformations in Practice Workshop*, Oct 2005.
- [9] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In *SAC 2006: Proceedings of the Symposium on Applied Computing*. ACM Press, Apr 2006.
- [10] J. P. Lorenzo, J. D. G. Luna, E. V. S. Rebull, and A. E. García. Implementación de un motor de transformaciones con soporte mof 2.0 qvt. In *DSDM'05: Proceedings of the II Taller sobre Desarrollo de Software Dirigido por Modelos. MDA y Aplicaciones*, Sep 2005.
- [11] OMG. MOF 2.0 Query/Views/Transformations. Technical Report ptc/05-11-01, Nov 2005. Available at <http://www.omg.org/docs/ptc/05-11-01.pdf>.
- [12] J. Sánchez and J. García. A plugin-based language to experiment with model transformations. In *9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199, pages 336–350. Lecture Notes in Computer Science, October 2006.
- [13] J. Sánchez, J. García, and M. Menarguez. 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.