Aplicando los principios del DSDM al desarrollo de transformaciones de modelos en ETL

Álvaro Jiménez, Verónica A. Bollati, Juan M. Vara, Esperanza Marcos

Grupo de Investigación Kybele,
Universidad Rey Juan Carlos, Madrid (España).
{alvaro.jimenez,veronica.bollati
juanmanuel.vara,esperanza.marcos}@urjc.es

Resumen Las transformaciones de modelos son uno de los principales artefactos en el Desarrollo de Software Dirigido por Modelos. Sin embargo, a pesar de ser otro artefacto software más, existen pocas aproximaciones que apliquen los principios del DSDM a su desarrollo. En este trabajo presentamos una aproximación para el desarrollo de transformaciones de modelos dirigido por modelos para el lenguaje *Epsilon Transformation Language* (ETL). Para ello, presentamos un metamodelo para el lenguaje ETL, una transformación que permite obtener un modelo ETL a partir de un modelo de la transformación de alto nivel y la generación del código ETL que implementa la transformación.

Palabras Clave: Desarrollo Dirigido por Modelos, Transformaciones de Modelos, Modelos de Transformación, ETL, ATL.

1. Introducción

En el contexto del Desarrollo de Software Dirigido por Modelos (DSDM, [27], [30]), las transformaciones de modelos actúan principalmente como elementos de enlace entre los diferentes pasos del proceso de desarrollo, refinando los modelos de alto nivel en modelos de menor abstracción, hasta que estos puedan ser ejecutados/interpretados o convertidos en código ejecutable. Además, las transformaciones de modelos pueden emplearse para llevar a cabo otras tareas como la sincronización o la migración de modelos [24],[31]. Por ello, independientemente del objetivo con el que hayan sido construidas, las transformaciones de modelos juegan un papel clave en cualquier propuesta relacionada con la Ingeniería Dirigida por Modelos (*Model-Driven Engineering*, MDE) [26].

Como prueba de ello, durante los últimos años ha surgido un gran número de lenguajes y herramientas que dan soporte al desarrollo de transformaciones [6], [9], [29]. Estos lenguajes y herramientas difieren en múltiples aspectos, como la aproximación que adoptan (declarativa, imperativa, híbrida, basada en grafos, etc.) y esta diversidad trae consigo una complejidad adicional en el desarrollo de transformaciones como la de la selección del lenguaje, el tiempo de aprendizaje, la migración, etc.

En este sentido, dado que el desarrollo de transformaciones es una actividad intrínsecamente compleja [13] y que cada vez existe mayor número de alternativas para la construcción de las mismas, resultaría recomendable aplicar los mismos principios del DSDM a su desarrollo [3], [11], [19]. De esta forma se podrían modelar las transformaciones a alto nivel y refinar dichas especificaciones produciendo modelos de bajo nivel. Además, al ser capaces de expresar las transformaciones en forma de modelos, podremos procesarlas como haríamos con cualquier otro modelo. Por ejemplo, estaremos en condiciones de generar, validar, comparar o refactorizar transformaciones [2]. El modelado de las transformaciones también permitiría establecer puentes tecnológicos entre los diferentes lenguajes de transformación, ya que un modelo de transformación para un lenguaje concreto se podría transformar en un modelo para otro lenguaje diferente.

En este sentido, existen distintas propuestas que se centran en el modelado de las transformaciones de modelos ([3], [11], [19], [20]). Sin embargo, la mayoría de ellas son sólo propuestas teóricas y no ofrecen soporte tecnológico para llevar a la práctica el modelado de las transformaciones, o al menos no soportan el proceso completo.

Por todo ello, este trabajo muestra la aplicación de la propuesta presentada en trabajos anteriores para soportar el desarrollo dirigido por modelos de transformaciones de modelos ([6], [12]) al desarrollo de transformaciones para el lenguaje ETL (*Epsilon Transformation Language*, [17]).

Así, las principales contribuciones de este trabajo son: la implementación de un metamodelo para el lenguaje de transformaciones ETL; el desarrollo de una transformación que permite mapear modelos de transformación de alto nivel a modelos ETL y el desarrollo de una transformación modelo a texto que, a partir del modelo de una transformación ETL, permite obtener el código que implementa la transformación modelada.

El resto del documento se estructura de la siguiente forma: en la Sección 2 se realiza una breve introducción al lenguaje ETL. La Sección 3 presenta la propuesta para el desarrollo dirigido por modelos de transformaciones ETL; la Sección 4 usa un caso de estudio para ilustrar el resultado y finalmente en la Sección 5 se resumen las principales conclusiones, identificando futuras líneas de investigación.

2. Conceptos previos: Lenguaje de Transformación ETL

ETL (*Epsilon Transformation Language*, [17]) es un lenguaje para el desarrollo de transformaciones de modelos que sigue una aproximación híbrida (estilos declarativo e imperativo). Forma parte de la familia de lenguajes de Epsilon y por lo tanto del proyecto Eclipse GMT (*Generative Modeling Technologies*). Actualmente es usado ampliamente por la comunidad de desarrolladores, proporcionando amplia documentación, ejemplos de uso y un foro de discusión.

El lenguaje ETL potencia el uso de las construcciones declarativas mediante la definición y ejecución de reglas. Sin embargo, para soportar construcciones imperativas, como operaciones o iniciación de variables, se apoya en el lengua-

je EOL (*Epsilon Object Language*, [16]). Para definir la transformación, ETL proporciona distintos tipos de reglas:

- *Matched Rule*: son las reglas principales de la transformación. Permiten definir qué elemento o elementos se deben generar en el modelo destino a partir de la existencia de uno o varios elementos en el modelo origen.
- *Lazy Rule*: son reglas que deben ser invocadas explícitamente por otras reglas.
- *Abstract Rule*: reglas abstractas que pueden ser extendidas por otras reglas.
- *Greedy Rule*: a diferencia de otras reglas, los elementos que las invocan no tienen porqué ser exactamente del mismo tipo definido, sino que también pueden ser de un subtipo de dicho tipo (*Type-of vs. Kind-of*). Por ejemplo, si se tiene un elemento X del que heredan los elementos Y y Z, las reglas *greedy* definidas para X también se ejecutarán sobre los elementos Y y Z.
- *Primary Rule*: para devolver los elementos generados a partir de un elemento concreto, una de las operaciones que proporciona ETL es `equivalents()`. Por defecto, la operación `equivalents()` devuelve una lista de elementos, ordenada según el orden en el que se han definido las reglas que los han creado. Sin embargo, cuando una regla es definida como *primary* sus resultados preceden, en dichas listas, a los del resto de tipos de reglas.

3. Desarrollo Dirigido por Modelos de Transformaciones en ETL

Este trabajo ha sido abordado en el contexto de MeTAGeM-Trace, un entorno para el desarrollo dirigido por modelos de transformaciones de modelos que incluyen generadores de trazas [13]. En concreto, el trabajo aquí presentado constituye parte de la prueba de concepto de MeTAGeM-Trace. En la Fig. 1 se muestra una visión global del proceso propuesto para el desarrollo dirigido por modelos de transformaciones ETL a diferentes niveles de abstracción.

Con el objetivo de aumentar el nivel de abstracción al que se especifican las transformaciones ETL, definimos las transformaciones de acuerdo a los siguientes niveles: PSM, PDM y Código. A nivel específico de plataforma (PSM, *Platform Specific Model*), se modelarán las transformaciones de modelos siguiendo una aproximación de transformaciones de modelos concreta (declarativa, imperativa, híbrida, etc.). En este trabajo, se define un modelo de transformación conforme a un metamodelo que incluye las abstracciones propias de la mayoría de los lenguajes de transformación que siguen una aproximación híbrida como ETL, ATL [14] o RubyTL [25]. Es importante mencionar, que de la misma manera, a este nivel podrían definirse metamodelos para otras aproximaciones, como por ejemplo: declarativo, imperativo, orientado a grafos, etc. A nivel dependiente de plataforma (PDM, *Platform Dependent Model*), se modelaran las transformaciones de acuerdo a un lenguaje de transformación de modelos en concreto, dicho lenguaje, deberá seguir la aproximación seleccionada previamente. En este trabajo, se incluyen los modelos de transformación conformes al metamodelo definido para ETL (Sección 3.1). Por último, a nivel código se obtiene el código que

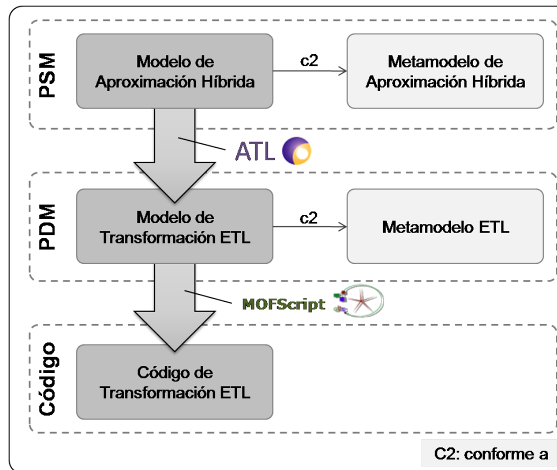

Figura 1. Proceso de Desarrollo Dirigido por Modelos de Transformaciones ETL

implementa la transformación en el lenguaje seleccionado en el nivel anterior, por tanto se obtiene el código ETL que implementa la transformación.

El paso de los modelos entre los niveles de abstracción se realiza mediante la ejecución de transformaciones, en el mapeo entre los modelos PSM y PDM se emplea una transformación de modelo a modelo (M2M), implementada con el lenguaje ATL [14] Para la generación del código ETL (PDM-código) se emplea una transformación de modelo a texto (M2T), implementada con el lenguaje MOFScript [22].

3.1. Metamodelo para ETL

Aunque en la literatura se han encontrado algunas especificaciones de la sintaxis abstracta del lenguaje ETL ([17], [18]), no se ha encontrado ninguna implementación de su metamodelo. Por tanto, para dar soporte al modelado de transformaciones ETL ha sido necesario especificar e implementar un metamodelo para este lenguaje. Para llevar a cabo esta tarea, se ha analizado en profundidad la sintaxis del lenguaje así como ciertos aspectos del lenguaje EOL y los ejemplos disponibles en el sitio web de la herramienta Epsilon. En la Fig. 2 se muestra el metamodelo para el lenguaje de transformaciones ETL implementado en términos de un modelo Ecore [7].

En un modelo Ecore, al utilizar la representación XML, se debe definir un elemento raíz a partir del cual se definan el resto de elementos. En este caso, el elemento raíz del metamodelo de ETL es la metaclase `EtlModule` que representa a la transformación en sí misma. A partir de dicho elemento se pueden definir: reglas de transformación (`TransformationRule`); bloques de código EOL (`EolBlock`) para definir pre-condiciones y post-condiciones de la transformación; y operaciones EOL (`Operation`) que se pueden invocar a lo largo de la transformación.

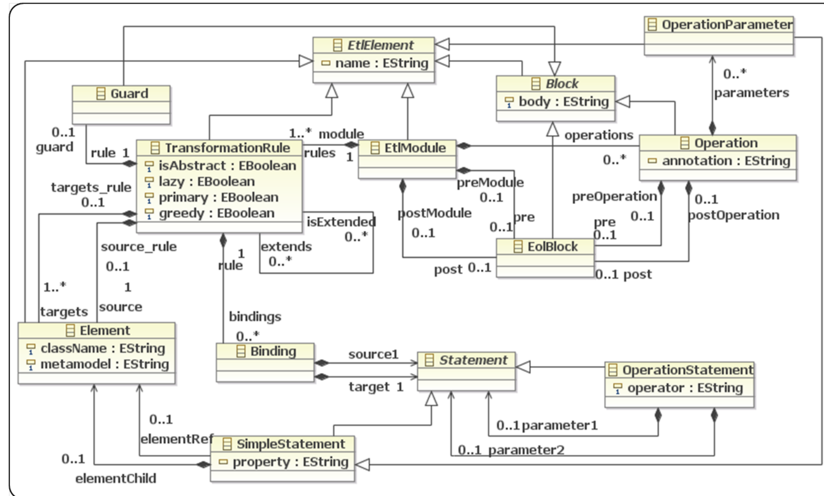

Figura 2. Metamodelo para el lenguaje ETL

Respecto a la definición de las reglas de transformación conviene mencionar que, como se ha descrito en la sección 2, ETL permite la definición de varios tipos de reglas, sin embargo, en el metamodelo definido todas ellas se representan mediante la metaclase `TransformationRule`, en la que se definen atributos booleanos (`isAbstract`, `lazy`, `primary` y `greedy`), que distinguen cada uno de los tipos de reglas. Además, en cada regla se podrán definir los elementos que intervienen en las mismas (`Element`) y asignaciones (`Binding`).

3.2. Transformación de Modelos de Aproximación Híbrida a Modelos ETL

En esta sección nos centramos en la generación de modelos de transformación ETL (conformes al metamodelo presentado en la sección anterior) a partir de modelos de la transformación definidos de acuerdo a las características de la aproximación híbrida. Para ello, se desarrolla una transformación de modelos, denominada `Hybrid2ETL`, que establece las reglas de mapeo entre los elementos del metamodelo Híbrido, definido en [13], y del metamodelo ETL.

En cuanto al desarrollo de esta transformación, aunque en ocasiones se recomienda seguir un proceso de desarrollo completo (elicitación de requisitos, análisis, diseño, implementación y pruebas) [11], en [23] se indica que el mapeo que describen las transformaciones puede realizarse en lenguaje natural, algoritmos o en modelos de mapeo. A partir de la experiencia en trabajos anteriores ([6],[29]) y dado que la definición de las reglas de transformación en lenguaje natural puede considerarse, en cierto modo, como una aproximación a las fases de análisis y diseño, se ha optado por seguir los siguientes pasos para el desarrollo de las transformaciones de modelos:

1. Definir las transformaciones entre los modelos en lenguaje natural.

2. Estructurarlas en un conjunto de reglas de mapeo, expresadas nuevamente en lenguaje natural.
3. Implementar dichas reglas usando un lenguaje de transformación existente. En particular, se ha optado por emplear ATL [15], que es considerado el estándar *de-facto* para el desarrollo de transformaciones de modelos([4], [28]). Además, comparado con otros lenguajes ([6], [29]), en nuestra opinión es el más adecuado debido a la cantidad de documentación disponible, al soporte tecnológico que ofrece y al número de casos de éxito.

De acuerdo a este proceso, en la Tabla 1 se describe, de forma resumida, las reglas de mapeo para transformar modelos de transformación de alto nivel en modelos de transformación ETL.

Metamodelo de Aproximación Híbrida	**Metamodelo ETL**
`Module`	`EtlModule`
`Rule` (`sources`==1 and `targets` > 0)	`Transformation Rule` - `source`: `rule.sources` - `target`: `rule.targets`
`Binding`	`Binding`
`LeftPattern`	`SimpleStatement`
`TransformationElement` (incluido en un `OpDefinition` o en un `OpArgument`)	`Element`
`Source` (incluido en una `Rule`)	`Element` `parent` = `Rule`
`Source` (incluido en un `RightPattern`)	`Element` `parent` = `Element`
`Target` (incluido en una `Rule`)	`Element` `parent` = `Rule`
`Target` (incluido en un `LeftPattern`)	`Element` `parent` = `Element`
`Guard`	`Guard`
`Operation`	`Operation`
`OpDefinition` (`return`==`Boolean` or `return`==`Integer` or `return`==`String`)	`SimpleStatement`
`OpArgument` (`return`==`Boolean` or `return`==`Integer` or `return`==`String`)	`OperationParameter`

Tabla 1. Reglas de mapeo: de modelos de transformación de aproximación híbrida a modelos ETL

Conviene mencionar que debido a su complejidad, esta tabla no incluye el mapeado del elemento `RightPattern`, que puede ser convertido en elementos `SimpleStatement` o `OperationStatement`, dependiendo de sus referencias y del

tipo de relación a la que pertenezca (para más información acerca de estas relaciones, consulte [13]).

El siguiente paso consiste en implementar dichas reglas usando ATL. A modo de ejemplo, a continuación se muestra el código de la regla ATL que implementa el mapeo entre el elemento `Module` del metamodelo de aproximación híbrida y el elemento `EtlModule` del metamodelo de ETL, que se corresponden con los elementos raíz de cada uno de estos metamodelos. Al mismo tiempo que se genera el elemento destino, se generan sus atributos y referencias que representan el nombre, las reglas y las operaciones de la transformación que se está generando (`name`, `rules` y `operations`). En esta implementación, estos elementos se construyen directamente a partir de los elementos de mismo nombre registrados en el modelo de la transformación a alto nivel.

```
rule Module{
from
   h_module: Hybrid!Module
to
   etl_module: ETL!EtlModule(
     name <- h_module.name,
     rules <- h_module.rules,
     operations <- h_module.operations
   ) }
```

Una vez implementada la regla anterior, se procede a implementar el resto de reglas de transformación mostradas en la Tabla 1. Así, por ejemplo, para pasar de elementos de tipo `rule` (alto nivel) a elementos de tipo `TransformationRule` en ETL se ha implementado la regla `createRule`:

```
rule createRule{
from
   r: Hybrid!Rule(r.sources.size()=1 and r.targets.size()>0)
to
   etl_r: ETL!TransformationRule(
     name <- r.name,
     isAbstract <- r.isAbstract,
     "extends" <- r."extends",
     "lazy" <- not r.isMain,
     guard <- r.guard,
     source <- r.sources.first(),
     targets <- r.targets,
     bindings <- r.targets->collect(t|t.bindings).asSequence()
   ) }
```

Dado que ETL no permite definir reglas con más de un elemento de entrada ni reglas sin elementos destino, en la regla de transformación `createRule` se ha definido una condición (o guarda) que impide transformar reglas que no cumplan estas características. Asimismo, conviene destacar la creación de los atributos

`isAbstract`, `extends` y `lazy` que establecen las características de la regla ETL generada.

Como muestra la Fig. 1, ejecutando estas reglas de transformación en el motor de ATL se consume un modelo de la transformación siguiendo la aproximación híbrida y se produce un modelo de la transformación conforme al metamodelo de ETL.

3.3. Generación de Código

El siguiente paso consiste en generar el código que implementa la transformación ETL. Para ello, se construye una transformación M2T, siguiendo los mismos pasos que para la construcción de la transformación M2M anterior: definición en lenguaje natural, estructuración en reglas de mapeo e implementación con un lenguaje de transformaciones.

Metamodelo ETL	**Código ETL**

`EtlModule`	’pre’ pre.name ’{’
	    pre.body
	’}’
	module.transformationRules
	module.operations
	’post’ post.name ’ {’
	    post.body
	’}’

`TransformationRule`	[(’@greedy ’, ’@abstract ’, ’@lazy ’, ’@primary ’)]
	’rule’ name
	’transform’ source
	’to’ targets
	[’extends’ extends]
	’{’
	    [’guard:’ guard.body]
	    bindings
	’}’

`Binding`	target ’:=’ source ’;’
`SimpleStatement(Binding)`	[ref ’.’] property
`SimpleStatement(Operation)`	[child.model ’!’ className] [ref ’.’] property
`OperationStatement`	[parameter1] operator [parameter2]
`Element`	name ’:’ model ’!’ class
`Operation`	[’@’ annotation] [’@’ pre] [’@’ post] ’operation’ context [’(’ parameters ’)’] ’: return’ return ’{’ body ’}’
`OperationParameter`	name ’:’ [ref.model ’!’ ref.className] [’.’ property]

Tabla 2. Reglas de generación de código: de modelos ETL a código ETL

Así, la Tabla 2 muestra las relaciones existentes en la transformación de los modelos ETL a código ETL. En este caso, la descripción en lenguaje natural incluye algunos términos propios de la descripción de la gramática, como por ejemplo el uso de corchetes para describir elementos opcionales.

Para la implementación del generador de código ETL se ha utilizado el lenguaje MOFScript [22]. Se ha escogido este lenguaje porque ofrece soporte completo en Eclipse, existe gran cantidad de documentación y su curva de aprendizaje, desde nuestro punto de vista, es menor que para otras alternativas como xText [10] o Acceleo [21]. Además, ya fue utilizado satisfactoriamente en trabajos anteriores ([13], [29]).

Para la implementación del generador de código con MOFScript, se debe crear un fichero (extensión `.m2t`) donde se especifica el código que se generará a partir de cada elemento del metamodelo. Las transformaciones MOFScript tienen un comportamiento imperativo por lo que se debe definir qué regla se ejecutará en primer lugar (palabra reservada `main`). En este caso, la generación del código ETL comienza por la regla que genera el código correspondiente al elemento raíz del modelo ETL (`EtlModule`). El código que implementa esta regla es el siguiente:

```
eco.EtlModule::main () {
   var name:String
   if (self.name.size()=0)
    name="generated.etl"
   else
    name=self.name + ".etl"
   file (name)

   println("pre "+self.pre.name+" {")
    if (self.name.size()!=0)
    println("'Running ETL: "+self.name+
    " Transformation'.println();")
    self.pre.body println("}")

   println("post "+self.post.name+" {")
    self.post.body println("}")

   self.rules->forEach(r:eco.TransformationRule){
      r.generateRule()}

   self.operations->forEach(o:eco.Operation){
      o.generateOperation()}
}
```

Dado que se trata de la regla principal, esta regla debe definir dónde se almacenará el código resultante. Para ello, se emplea la función `file`. En este caso, se ha decidido que el fichero de código se guarde con el nombre del elemento raíz y la extensión propia de los ficheros de transformación ETL (`.etl`). En caso

de que no se haya definido el nombre del módulo de la transformación, el nombre por defecto del fichero de código obtenido será `generated.etl`.

Como se puede observar en el código anterior, para generar el código relativo a los bloques *pre* y *post*, dado que los atributos de la metaclase `EolBlock` (`name` y `body`) son de tipo `String`, tan solo es necesario incluir su nombre para añadirlos al código resultante. En el caso de las reglas y las operaciones, dado que son estructuras más complejas, se invocan las reglas `generateRule()` y `generateOperation()`, que generan el código correspondiente a estos elementos. A modo de ejemplo, a continuación se muestra el código que implementa la regla `generateOperation()`:

```
  eco.Operation::generateOperation(){
    if(self.annotation!=null and self.annotation.size()>0)
     println("@"+self.annotation)
    if(self.pre!=null and self.pre.body.size()>0)
     println("@pre "+self.pre.body)
    if(self.post!=null and self.post.body.size()>0)
     println("@post "+self.post.body)
    print("operation ") self.context.generateOPstatement()
    print(" "+self.name+"(")
    self.parameters->forEach(p:eco.OperationParameter){
         p.generateOpParameter()
         if(p!=self.parameters.last())print(",")
    }
    print(") : ")
    self._getFeature("return").generateOPstatement() println(" {")
    println("\t"+self.body)
    println("}")
 }
```

En esta regla, se genera el código de la operación que incluye: pre-condiciones y post-condiciones, cabecera y cuerpo de la operación. La cabecera, a su vez, se compone del contexto de la operación, el nombre, los parámetros y el tipo de retorno. Para facilitar la comprensión de esta regla, la Fig. 3 muestra el código generado para una operación ETL que puede invocarse sobre un elemento `Families!Father` (perteneciente al caso de estudio que se muestra en la siguiente sección) y devuelve un `String` que se forma a partir del valor de dos atributos del elemento de entrada.

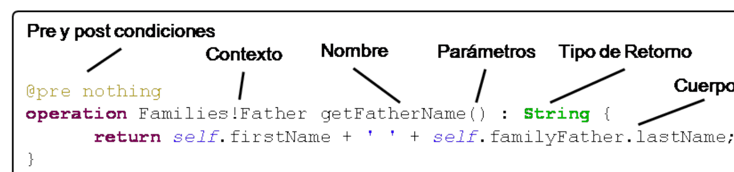

Figura 3. Código generado para una operación ETL

El resto de relaciones mostradas en la Tabla 2 se han implementado de forma similar, mediante reglas en MOFScript.

4. Caso de Estudio

Para ilustrar la propuesta presentada, en esta sección, se presenta un caso de estudio sencillo pero completo que es comúnmente empleado en el ámbito de las transformaciones de modelos: `Families2Persons` [1]. consiste en el desarrollo de una transformación para convertir modelos conformes al metamodelo `Families` (Fig. 4.a) en modelos conformes al metamodelo `Persons` (Fig. 4.b). Conviene mencionar que esta propuesta también ha sido utilizada en escenarios más complejos que pueden ser consultados en [13].

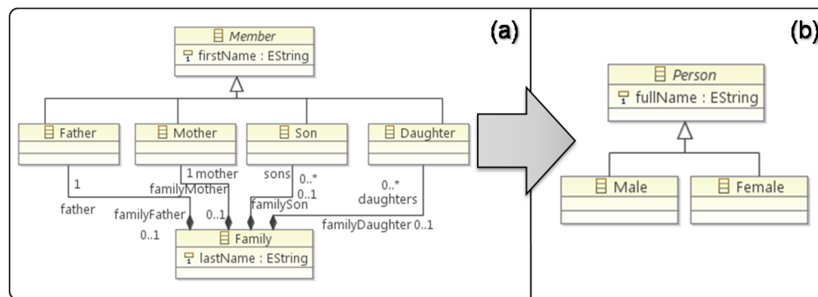

Figura 4. Metamodelos `Families` y `Persons`

Para desarrollar una transformación entre estos dos metamodelos usando la propuesta presentada, en primer lugar se debe modelar la transformación a alto nivel mediante un modelo de aproximación híbrida (`families2persons.hybrid`). Posteriormente, se debe ejecutar la transformación M2M presentada en la sección anterior que consumirá dicho modelo y producirá un modelo de transformación ETL (`families2persons.etl_model`). A partir del modelo ETL obtenido, se genera el código que implementa la transformación (`families2persons.etl`), mediante la ejecución del generador de código implementado con MOFScript.

La Fig. 5 muestra el desarrollo de la transformación `Families2Persons` usando la propuesta dirigida por modelos que se presenta en este trabajo. En concreto, la figura se centra en mostrar la regla `Mother_2_Female` a lo largo de los distintos niveles de abstracción en los que se ha definido la transformación.

5. Conclusiones y Trabajos Futuros

El desarrollo de transformaciones de modelos es una actividad inherentemente compleja. Dado que las transformaciones son otro artefacto software, sería conveniente también aplicar los principios de MDE [26] a su desarrollo. Así, el desarrollo de transformaciones de modelos podría beneficiarse de las ventajas

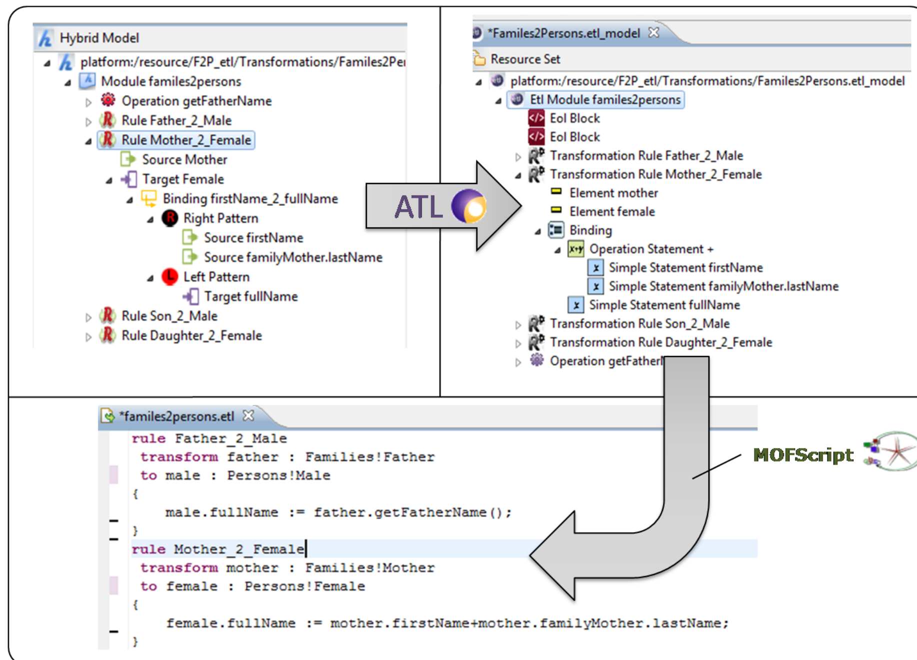

Figura 5. Caso de Estudio 'Families2Persons': de modelo de alto nivel a código ETL

que ofrece la aplicación de MDE: disminuir el nivel de complejidad, aumentar la productividad, facilitar la construcción de puentes tecnológicos entre diferentes tecnologías, etc.

Para poner en práctica esta idea, en este trabajo se ha presentado un proceso para el desarrollo dirigido por modelos de transformaciones de modelos en ETL [16], un lenguaje de transformaciones que adopta la aproximación híbrida (combinación de los estilos declarativo e imperativo). En concreto, se ha presentado la implementación de un metamodelo para dicho lenguaje, una transformación de modelo a modelo que permite generar modelos ETL a partir de especificaciones de alto nivel propias de la aproximación híbrida y una transformación de modelo a texto para la generación del código ETL que implementa la transformación modelada. Finalmente, para ilustrar el funcionamiento de la propuesta se ha presentado un caso de estudio en el que se ha seguido el proceso propuesto para el desarrollo de transformaciones de modelos.

El resultado de este trabajo proporciona diferentes líneas de trabajo futuras. Dado que en trabajos anteriores ([6], [12]) ya se mostró la validez de esta propuesta para el desarrollo de transformaciones para los lenguajes ATL y RubyTL, la línea más inmediata pasa por aplicar las mismas ideas para el desarrollo de transformaciones en otros lenguajes, como por ejemplo VIATRA [8]. Además, una vez que se disponga de los metamodelos de los diferentes lenguajes de transformación será posible construir puentes tecnológicos entre estos lenguajes, mejorando el nivel de interoperabilidad entre ellos. Por otra parte, al igual

que en este trabajo se han definido las transformaciones a nivel PSM siguiendo la aproximación híbrida, podrían definirse otros metamodelos a este nivel que sigan otras aproximaciones (puramente imperativa o declarativa, basada en grafos, etc.). Así, el desarrollador podría seleccionar la aproximación que quiere seguir y luego escoger entre los lenguajes que adopten dicha aproximación.

Agradecimientos

Este trabajo de investigación se ha llevado a cabo en el marco de trabajo del proyecto MASAI (TIN-2011-22617), financiado por el Ministerio de Ciencia e Innovación del Gobierno de España.

Referencias

1. Allilaire, F., Jouault, F. (2007): ATL Use Case - Families to Persons. ATLAS, INRIA and University of Nantes. http://www.eclipse.org/m2m/atl/doc/ATLUseCase_Families2Persons.pdf
2. Bernstein, P. (2003): Applying model management to classical meta data problems. First Biennial Conference on Innovative Data Systems Research, Asilomar, USA.
3. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A. (2006): Model Transformations? Transformation Models!. 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2006.
4. Bohlen, M.: QVT und Multi-Metamodell-Transformationen in MDA. In: OBJEKTspektrum, vol. 2. Translated in: http://goo.gl/boRs4. (2006)
5. Bollati, V. A., Sánchez, E. V., Vela, B., Marcos, E. (2009): Análisis de QVT Operational Mappings: un caso de estudio. VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM). Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos (JISBD2009). vol. 3 (2).
6. Bollati, V. A. (2011): MeTAGeM: un Entorno de Desarrollo de Transformaciones de Modelos Dirigido por Modelos. Tesis Doctoral, Universidad Rey Juan Carlos.
7. Budinsky, F., Merks, E., Steinberg, D. (2008). Eclipse Modeling Framework 2.0 (2nd Edition): Addison-Wesley Professional.
8. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D. (2002): VIATRA-visual automated transformations for formal verification and validation of UML models. 17th IEEE International Conference on Automated Software Engineering (ASE 2002), pp. 267-270.
9. Czarnecki, K., Helsen, S. (2003): Classification of model transformation approaches. OOPSLA'03, workshop on Generative Techniques in the Context of MDA.
10. Efftinge, S., Völter, M. (2006): oAW xText: a framework for textual DSLs. Workshop on Modeling Symposium at Eclipse Summit.
11. Guerra, E., de Lara, J., Kolovos, D., Paige, R., dos Santos, O. (2010): transML: A Family of Languages to Model Model Transformations. Model Driven Engineering Languages and Systems. vol. 6394, pp. 106-120, Springer.
12. Jiménez, Á., Granada, D., Bollati, V. A., Vara, J. M. (2011): Using ATL to support Model-Driven Development of RubyTL Model Transformations. 3rd International Workshop on Model Transformation with ATL (MtATL2011), Zürich, Switzerland.

13. Jiménez, Á. (2012): Incorporando la Gestión de la Trazabilidad en un entorno de Desarrollo de Transformaciones de Modelos Dirigido por Modelos. Tesis Doctoral, Universidad Rey Juan Carlos, Abril 2012.
14. Jouault, F., Kurtev, I. (2006): Transforming Models with ATL. Satellite Events at the MoDELS 2005 Conference. vol. 3844, Springer, pp. 128-138.
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I. (2008): ATL: A model transformation tool. Science of Computer Programming, vol. 72 (1-2), pp. 31-39.
16. Kolovos, D. S., Paige, R. F., Polack, F. A. C. (2006): The Epsilon Object Language (EOL). Model Driven Architecture - Foundations and Applications. vol. 4066, Springer Berlin / Heidelberg, pp. 128-142.
17. Kolovos, D. S., Paige, R. F., Polack, F. A. C. (2008): The Epsilon Transformation Language. Theory and Practice of Model Transformations. vol. 5063, pp. 46-60, Springer Berlin / Heidelberg.
18. Kolovos, D. S., Rose, L. M., Paige, R. F., Polack, F. A. C. (2010): The Epsilon Book. http://www.eclipse.org/epsilon/doc/book/
19. Kusel, A. (2009): TROPIC - a framework for building reusable transformation components. Proceedings of the Doctoral Symposium at MODELS 2009, School of Computing, Queen's University, Denver, 2009.
20. Lano, K. Kolahdouz-Rahimi, S. (2011): Model-Driven Development of Model Transformations. Theory and Practice of Model Transformations. vol. 6707, pp. 47-61, Springer.
21. Musset, J., Juliot, E., Lacrampe, S. (2006): Acceleo User Guide. http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf.
22. Oldevik, J. (2006): MOFScript user guide, Version 0.6 (MOFScript v1.1. 11).http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf
23. OMG (2003): MDA Guide Version 1.0.1 Document number omg/03-06-01. Ed.: Miller, J. and Mukerji, J. http://omg.org/cgi-bin/doc?omg/03-06-01.
24. Rose, L. M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D. S., Garcés, K., Paige, R. F., and Polack, F. (2010): A Comparison of Model Migration Tools. Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10), Springer, Berlin, pp. 61-75.
25. Sánchez, J., García, J., Menarguez, M. (2006): RubyTL: A Practical, Extensible Transformation Language. European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA.
26. Schmidt, D. C. (2006): Model-driven engineering. IEEE Computer, vol. 39 (2), pp. 25-31.
27. Stahl, T., Völter, M., and Czarnecki, K. (2006): Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.
28. van Amstel, M., van den Brand, M.: Using Metrics for Assessing the Quality of ATL Model Transformations. In: 3rd International Workshop on Model Transformation with ATL (MtATL2011), Zürich, Switzerland 2011, pp. 20-34. CEUR-WS
29. Vara, J. M. (2009): M2DAT: A Technical Solution for Model-Driven Development of Web Information Systems. Tesis Doctoral, Universidad Rey Juan Carlos, 2009.
30. Völter, M. (2009): Best Practices for DSLs and Model-Driven Development. Journal of Object Technology, vol. 8 (6), pp. 79 - 102, September-October 2009.
31. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H. (2007): Towards automatic model synchronization from model transformations. Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Atlanta, USA, 2007, pp. 164-173.