Catálogo de refactorizaciones para transformaciones Modelo a Modelo

Salvador Martínez[1], Manuel Wimmer[2], Frédéric Jouault[1], and Jordi Cabot[1]

[1] INRIA & École des Mines de Nantes, France
{salvador.martinez_perez,frederic.jouault,jordi.cabot}@inria.fr
[2] Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

Resumen Las transformaciones de modelos juegan un papel clave en la ingeniería dirigida por modelos. El diseño de estas transformaciones debe hacerse siguiendo principios de ingeniería que garanticen su calidad y su mantenimiento. En programación orientada a objetos, el refactoring es utilizado como uno de los principales mecanismos para mejorar la mantenibilidad del código ya que mejora la estructura del código existente sin modificar su comportamiento. Sin embargo, en el campo de transformaciones de modelos, por el momento, no existe ningún tipo de soporte para el uso de esta técnica. En este artículo se aborda la resolución de esta limitación mediante la adaptación del concepto de refactoring a las transformaciones de modelo a modelo (M2M). En concreto, presentamos un catálogo de refactorizaciones específicas para la mejora de la calidad de transformaciones M2M.

Keywords: Refactoring, Transformación de Modelos, Ingeniería dirigida por modelos

1. Introducción

La manipulación de modelos es una actividad central en muchas de las tareas habituales en la ingeniería dirigida por modelos (MDE). Estas manipulaciones de modelos se implementan normalmente mediante transformaciones de modelo a modelo (M2M). Una transformación M2M transforma un modelo Ma instancia de un metamodelo MMa en un modelo Mb instancia de un metamodelo MMb (donde MMa y MMb pueden ser metamodelos iguales o diferentes)

La investigación actual sobre transformaciones de modelo se centra en el desarrollo de lenguajes para especificar transformaciones (por ejemplo, cf. [5]). Sin embargo, no se han definido técnicas que se centren en el mantenimiento de las transformaciones ya desarrolladas. Estas técnicas de soporte son claramente necesarias, por ejemplo, para mejorar la legibilidad de las transformaciones y para facilitar su evolución.

En el ámbito de la programación orientada a objetos, el refactoring es la técnica elegida para la mejora de la estructura del código existente sin cambiar su comportamiento externo [7,12,14]. Esta técnica ha demostrado ser útil para

mejorar la calidad de los atributos del código fuente, y por lo tanto, para aumentar su capacidad de mantenimiento. Desafortunadamente, ningún catálogo de refactorizaciones ha sido definido para transformaciones de modelo a modelo. Los catálogos disponibles de refactorizaciones en el mundo de los lenguajes orientados a objetos no son directamente reutilizables, porque los enfoques actuales de los lenguajes de transformaciones siguen habitualmente el paradigma de programación basado en reglas y son muy específicos de dominio. Esto obliga a los desarrolladores de transformaciones a realizar las tareas de mejora del código de las mismas sin ningún tipo de soporte dedicado. Dada la posible alta complejidad de las transformaciones del modelo debido a, por ejemplo, las dependencias implícitas entre las reglas (debidas a la trazabilidad interna entre elementos del modelo fuente y destino), una modificación manual puede dar lugar a efectos secundarios no deseados y constituye un proceso de mantenimiento tedioso y propenso a errores.

En este sentido, la principal contribución de este trabajo es ofrecer un catálogo de refactorizaciones para transformaciones M2M escritas en lenguajes basados en reglas. Estas refactorizaciones se han explorado mediante el análisis de ejemplos existentes de transformaciones [3] definidas en ATL [8]. Sin embargo, hay que mencionar que la mayoría de refactorizaciones propuestas no se han centrado específicamente a ATL, siendo aplicables también a otros lenguajes de transformación de modelo a modelo que sigan el paradigma basado en reglas, como por ejemplo, QVT Relations [13]. También cabe señalar que las refactorizaciones presentadas pueden mejorar no sólo la calidad de los atributos relacionados con mantenimiento, como la legibilidad, reusabilidad y extensibilidad de las transformaciones, sino también el rendimiento de las transformaciones. La ejecución de estas refactorizaciones pueden ser semi-automatizada mediante el empleo de 'Higher Order transformations' (HOTs) [17] (transformaciones que tienen como entrada o salida otras transformaciones).

El resto del documento está estructurado de la siguiente manera. En la sección 2, se introducen los conceptos principales de M2M y se presenta un ejemplo ilustrativo. La sección 3 presenta la noción de refactorizaciones para transformaciones M2M, y la sección 4 presenta el catálogo de refactorizaciones y su aplicación sobre el ejemplo propuesto. La sección 5 muestra el impacto de las refactorizaciones en el rendimiento y en la sección 6 se dan algunos detalles sobre su implementación. En la sección 7 se revisan trabajos relacionados y, por último, la sección 8 concluye con propuestas de trabajos futuros siguiendo esta línea.

2. Ejemplo ilustrativo

En esta sección se introduce un escenario de transformación con el objetivo de ilustrar el funcionamiento de las mismas y para mostrar que el refactoring es necesario para mejorar la calidad del código. Este ejemplo será la base de todos

[3] Por ejemplo, las transformaciones disponibles en www.eclipse.org/m2m/atl/atlTransformations

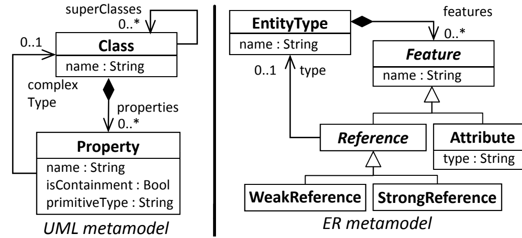

Figura 1. Metamodelos del diagrama de clases de UML y de Entidad-Relación

los ejemplos mostrados a lo largo de este trabajo. El escenario de transformación propuesto consiste en transformar diagramas de clases UML en modelos Entidad-Relación (ER). La figura 1 muestra los metamodelos (simplificados) de los dos lenguajes de modelado utilizados. La mayoría de los conceptos de un lenguaje tienen una contrapartida directa en el otro, excepto el concepto de herencia en UML, que en nuestro metamodelo ER no puede ser representado. Por lo tanto, una tarea importante de la transformación será aplanar los árboles de herencia en el modelo UML, duplicando las propiedades de la superclase en las subclases cuando se creen elementos de tipo Entity en el modelo ER.

Listing 1.1. UML to ER Transformation in ATL

```
1  module UML2ER;
2  create OUT : ER from IN : UML;
3
4  helper context UML!Class def: allClasses() : Sequence(UML!Class) =
5      self.superClasses->iterate(e; acc : Sequence(UML!Class) = Sequence {} |
6      acc->union(Set{e})->union(e.allClasses()) );
7  rule Class {
8   from
9    s: UML!Class
10  to
11   t: ER!EntityType (
12    name <- s.name,
13    features <- Sequence {attributes, weakReferences, strongReferences}
14   ),
15   attributes : distinct ER!Attribute foreach(a in
16         s.allClasses().including(s).flatten()
17         ->collect(e | e.ownedProperty).flatten()
18         ->select (e | not e.primitiveType.oclIsUndefined())) (
19     name <- a.name,
20     type <- a.primitiveType
21   ),
22   weakReferences : distinct ER!WeakReference foreach(a in
23         s.allClasses().including(s).flatten()
24         ->collect(e | e.ownedProperty).flatten()
25         ->select(e | not e.complexType.oclIsUndefined() and not e.isContainment)) (
26     name <- a.name,
27     type <- a.complexType
28   ),
29   strongReferences : distinct ER!StrongReference foreach(a in
30         s.allClasses().including(s).flatten()
31         ->collect(e | e.ownedProperty).flatten()
32         ->select(e | not e.complexType.oclIsUndefined() and e.isContainment)) (
33     name <- a.name,
34     type <- a.complexType
35   )
36 }
```

El listado 1.1 muestra la transformación UML2ER expresada en ATL. Hemos elegido ATL ya que es uno de los lenguajes de transformación más utilizados, tanto en entornos académicos como industriales. Una transformación ATL está compuesta por un conjunto de reglas de transformación y por un conjunto de helpers. Cada regla describe cómo (parte de) el modelo de destino debe ser generado a partir del (parte de) modelo de origen. Hay dos tipos de reglas declarativas, *matched* rules y *lazy* rules. El primer tipo de regla es ejecutado automáticamente por el motor de ATL cuando este encuentra un elemento en el modelo de entrada adecuado, mientras que el segundo ha de ser llamado explícitamente desde otra regla otorgando un mayor control sobre la ejecución de la transformación. Resulta interesante destacar que QVT presenta conceptos similares a estos.

Un helper puede ser visto como una función auxiliar que permite la posibilidad de factorizar código ATL utilizado en diferentes partes de la transformación. En la transformación de ejemplo, la regla *class* se ejecutará para todas las clases en el modelo UML para producir un elemento de tipo Entity en el modelo de salida. El helper *allClasses()* calculará todas las superclases de una clase dada y será llamado para ayudar en el aplanado de la jerarquía de clases duplicando las *features* de las superclases en las subclases.

Las reglas se componen principalmente de un *patrón de entrada* y un *patrón de salida*. El patrón de entrada filtra el subconjunto de los elementos del modelo de entrada que se transformarán con la regla. El patrón de salida detalla como elementos del modelo de salida son creados a partir de elementos del modelo de entrada. Cada elemento del patrón de salida puede tener varios *bindings* que se pueden utilizar para inicializar los valores de los elementos en el modelo de salida. En el ejemplo, hemos definido un elemento patrón de entrada, que selecciona elementos de tipo *Class*, y un patrón de salida que crea cuatro tipos de elementos: *types*, *attributes* y los dos tipos de *references*. Los *bindings* se utilizan, por ejemplo, para inicializar el nombre de los elementos Entity con el nombre de las clases correspondientes. La cláusula *distinct-foreach* que aparece en el patrón de salida indica que se puede producir mas de un elemento de salida del tipo correspondiente a la vez (en el ejemplo se recorren todas las propiedades pare crear las correspondientes features en el modelo de salida).

Aunque el ejemplo de transformación proporcionado funciona, es decir, produce correctamente modelos ER a partir de modelos UML, contiene varios defectos que comprometen su calidad en términos de mantenibilidad y rendimiento:

1. La transformación consiste en una única y compleja regla haciendo todo el trabajo en lugar de descomponer la transformación en varias reglas basándose en los diferentes tipos de elementos en el metamodelo de entrada.
2. La existencia de código duplicado dificulta la mantenibilidad de la transformación. Por ejemplo, si la referencia *ownedProperty* es renombrada en el metamodelo UML, tendremos que reescribir tres complejas expresiones OCL
3. Se hacen llamadas innecesarias (por haberse hecho ya antes) comprometiendo el rendimiento de la transformación. El helper *allClasses()* es llamado varias veces para el mismo elemento obligando a recalcular el valor devuelto cada vez.

4. Se hace uso de constructos del lenguaje marcados como obsoletos y no recomendados. La clausula *distinct-foreach* es de este tipo (no se recomienda su uso porque rompe la trazabilidad interna y porque existen otros mecanismos para lograr el mismo objetivo)

Los desarrolladores de transformaciones pueden no darse cuenta de estos problemas o simplemente no saber con certeza como evitarlos sin modificar el funcionamiento de la transformación. El catálogo de refactorizaciones que presentamos en la seccion 4 mejora substancialmente esta situación.

3. Refactorización en transformaciones M2M

En esta sección explicaremos como el concepto de refactorización es adaptado al campo de las transformaciones M2M.

Como para cualquier otro tipo de refactorizaciones, el comportamiento del objeto refactorizado, la transformación, en nuestro caso, debe ser conservado. Desafortunadamente, la comunidad científica no ha llegado a un consenso sobre el significado de preservación de comportamiento [12]. No existe, por tanto, una definición universal y en consecuencia, las definiciones adaptadas a lenguajes y dominios específicos proliferan.

Adaptando la definición más amplia de conservación de comportamiento [14] a las transformaciones M2M llegamos a la siguiente definición. La preservación de comportamiento estará asegurada si para cualquier modelo de entrada el modelo de salida producido será el mismo antes y después de la refactorización de la transformación. Esta definición de preservación de comportamiento puede ser probada disponiendo de un conjunto exhaustivo de pruebas o en caso de que dispusiéramos de una semántica formal demostrando la equivalencia semántica de la transformación refactorizada y la original. Existen algunos esfuerzos para dar una semántica formal a ATL, cf. por ejemplo, [18], sin embargo, el lenguaje aún no cuenta con una semántica formal completa (al igual que muchos lenguajes de programación utilizados en la práctica). En este trabajo se ha decidido por tanto utilizar la primera técnica. Ésta proporciona feedback de manera rápida y ha sido probada cómo útil y eficaz (aunque no formal) para ingenieros de software en el campo de las refactorizaciones en lenguajes orientados a objetos.

4. Catálogo

En esta sección se ofrece una visión general sobre el catálogo de refactorizaciones propuesto. Para exponer las refactorizaciones en el presente trabajo, estamos usando un formato inspirado en el usado por Fowler [7]. En particular, presentamos algunas de las refactorizaciones con más detalle de la siguiente manera:

1. Dándole un nombre a la refactorización
2. Describiendo la situación típica en la que la refactorización debiera ser utilizada, por ejemplo, identificando el problema que resuelve.
3. Describiendo la solución para mejorar la situación problemática.
4. Estableciendo las precondiciones necesarias para poder aplicar la refactorización.

5. Describiendo los pasos implicados en la misma.
6. Exponiendo un ejemplo de aplicación concreto.

Las refactorizaciones se centran principalmente en construcciones del lenguaje ATL que también forman parte de otros lenguajes de transformación M2M, por ejemplo, QVT relations, así como en elementos de OCL ya que estos forman parte integral de muchos lenguajes de transformaciones. Gracias a esto, la mayoría de las refactorizaciones son útiles independientemente del lenguaje de transformaciones M2M utilizado, con la condición de que el mismo tenga elementos similares. Es importante resaltar que en este trabajo no se ha tenido en cuenta la parte imperativa del lenguaje ATL (y otros lenguajes M2M). Consideramos que los problemas de refactorización en código imperativo han sido ya estudiados en el ámbito de los lenguajes de programación orientados a objetos [7,14].

Las refactorizaciones propuestas se dividen en cuatro categorías: Renombrado, reestructuración, herencia y optimización de expresiones OCL.
Mientras que la primera categoría puede ser vista como un conjunto de refactorizaciones básicas para mejorar la legibilidad de la transformación, el resto de categorías presentan refactorizaciones que se utilizan para, en gran medida, cambiar la estructura de la misma. En particular, también se han incluido refactorizaciones para la eliminación, de manera explícita, de construcciones obsoletas del lenguaje así como de malas prácticas de codificación. La tabla 1 resume el catálogo de refactorizaciónes. En los apartados siguientes, se expone cada categoría con mayor detalle.

4.1. Renombrado

Tal como ocurre en cualquier lenguaje de programación [7], una de las cosas mas sencillas y a la vez mas útiles que pueden hacerse para mejorar la calidad del código es modificar nombres. Tener reglas y helpers nombrados adecuadamente ayuda a obtener una idea precisa de cual es la funcionalidad que implementan. Otros elementos del lenguaje, como las variables, también pueden ser renombradas (De nuevo, para mejorar el entendimiento del código y también porque el lenguaje impone ciertas restricciones con respecto al nombrado de variables cuando se hace uso de herencia). En la tabla 1, las refactorizaciones 1 a 3 se encargan de realizar estas acciones de renombrado.

4.2. Restructuración

Además de refactorizaciones de renombrado, otras, encargadas de mejorar la estructura de las transformaciones son también necesarias. Esto supone, reestructurar reglas y helpers. Las refactorizaciones propuestas en esta categoría atacan el problema del crecimiento de las reglas (asumiendo que el aumento de tamaño de una regla tiene un gran impacto en la complejidad de entendimiento de la misma). No es extraño encontrar transformaciones en las que una sola regla realice todo el trabajo generando demasiados elementos mediante el uso de un patrón de salida que contiene una gran cantidad de elementos de salida.

Nombre de la refactorización	Objetivo
Renombrado	
1. *Renombrar patrones de entrada/salida*	Cambia el nombre de los patrones de entrada/salida por otros que expliquen mejor la intención del elemento.
2. *Renombrado de los modelos de entrada/salida*	Cambia el nombre de los modelos de entrada/salida por otros que identifiquen mejor el modelo.
3. *Renombrado de reglas y helpers*	Cambia el nombre de las reglas/helpers por otros que expliquen mejor la intención de los mismos.
Restructuración	
4. *Extraer Helper/Regla**	Extraer un helper adicional a partir de una regla o un helper ya existente o extraer una regla a partir de otra.
5. *Inline Helper/Regla**	Inline a helper en otro helper o regla existente o inline una regla en otra ya existente.
6. *Mezclar Helper/Regla**	Mezclar dos reglas o dos helper en una sola regla o helper respectivamente.
7. *Dividir Helper/Regla**	Dividir una regla o helper en dos reglas o helpers respectivamente.
8. *Mezclar Binding*	Mezclar dos *bindings* en uno solo si ambos inicializan la misma propiedad.
9. *Dividir Binding*	Dividir un *binding* en dos si varios elementos son asignados a la misma propiedad.
10. *Modificar tipo de regla**	Cambiar el tipo de una regla de regla *Matched* a *LazyRule* y viceversa.
Herencia	Aplicable en reglas *Matched* y reglas *Lazy*
11. *Extraer super-regla*	Introduce una super-regla común para un conjunto de subreglas que compartan supertipos comunes para los elementos de entrada/salida.
12. *Subir binding*	Mueve *bindings* comunes de subreglas a una super-regla común.
13. *Subir filtro*	Mueve filtros comunes de subreglas a una super-regla común.
14. *Eliminar super-regla*	Elimina una super-regla y las relaciones de herencia de las subreglas con ésta.
15. *Bajar Binding*	Mueve los *bindings* de una super-regla a las subreglas.
16. *Bajar Filtro*	Mueve los filtros de una super-regla a las subreglas.
Relacionados con OCL	
17. *Modificar tipo de helper**	Cambia el tipo de un helper de dinámico a estático.
18. *Eliminar navegaciones inseguras del modelo de salida*	Introduce la operación *resolveTemp* para evitar navegaciones inseguras del modelo de salida.
19. *Mejora del cálculo de referencias opuestas*	Si el metamodelo carece de referencias opuestas, estas son calculadas mediante iteraciones. Sustituirlas por la operación *refIntermediateComposite* .
20. *Acortamiento de navegaciones mediante cambio de contexto*	Acorta expresiones OCL mediante la optimización de la longitud de las navegaciones tras seleccionar un contexto mas apropiado.
21. *Reemplazar patrón Select/First por Any*	Sustituye el patrón Select/First por la operación Any para encontrar un elemento específico que cumpla una determinada condición.
22. *Reemplazar el uso de allInstances con navegación*	Sustituir el uso de la operación allIntances por navegación.
23. *Introducir cortocircuito*	Simular evaluación en cortocircuito mediante el uso adecuado de IF ELSE

*Representan subcategorías que incluyen otras refactorizaciones similares

Cuadro 1. Catálogo de refactorizaciones para transformaciones M2M

Con el fin de garantizar la legibilidad y facilidad de mantenimiento del conjunto de reglas de una transformación, se necesitan refactorizaciones que sean capaces de transformar reglas complejas en varias mas simples. Este objetivo se puede lograr mediante la división de la regla original en varias reglas *matched* o delegando cierta funcionalidad de la regla en reglas *lazy*.

En la tabla 1, las refactorizaciones 4 y 5 se encargan de extraer e insertar reglas y *helpers*. Por su parte, las refactorizaciones 6 y 7 permiten mezclar o dividir dos reglas o dos *helpers*. Las refactorizaciones 8 y 9 hacen lo mismo para *bindings* y finalmente la refactorización 10 permite cambiar el tipo de una regla. Para mostrar el funcionamiento de estas refactorizaciones de reestructuración se presenta un ejemplo en el que se eliminan los elementos *distinct-foreach* del listado 1.1 mediante la extracción de reglas *matched*.

Extraer una regla *matched* a partir de un elemento *distinct-foreach*
Problema: Una regla *matched* utiliza un elemento del lenguaje marcado como obsoleto, *distinct-foreach*, para producir una colección de elementos de salida a partir de una colección de elementos de entrada.
Solución: Extraer una regla *matched* a partir del elemento de patrón de salida y modificar el elemento de patrón de salida sustituyendo el uso de *distinct-foreach* por la simple navegación a elementos del modelo de entrada (delegando de esta manera en la resolución automática de ATL).
Precondiciones:

1. Los *bindings* del elemento del patrón de salida usan sólo la variable del iterador, i.e., el elemento del patrón de salida a refactorizar es auto-contenido.
2. No existe ninguna regla de tipo *matched* para el mismo conjunto de elementos de entrada.

Pasos:

1. Determinar los tipos de los elementos de los patrones de entrada y de salida de la nueva regla *matched*. El primer elemento del patrón de entrada será del mismo tipo que el del iterador del elemento *distinct-foreach* mientras que el tipo del elemento de patrón de salida de la nueva regla será igual al de la original. Elementos de entrada adicionales pueden ser necesarios para asegurar que la nueva regla es ejecutada exáctamente tantas veces como lo era el elemento *distinct-foreach*
2. Crear la regla *matched* dándole un nombre adecuado.
3. Mover los *bindings* del elemento *distinct-foreach* a la nueva regla.
4. Sustituir los *bindings* que utilicen el elemento de patrón de salida sujeto a refactoring con la navegación hacia el elemento de entrada referido en el mismo. La recolección de los elementos de salida producidos es hecha de manera automática por el sistema de traza implícito de ATL.

Ejemplo: Veamos de nuevo el elemento de patrón de salida *distinct-foreach* en el listado 1.1. Éste es refactorizado en el ejemplo que se muestra a continuación. Es necesario resaltar que el elemento de patrón de entrada de la regla Attributes tiene que ser ejecutado tantas veces (y para los mismos elementos) como lo era el elemento *distinct-foreach* en la regla original. Por consiguiente, utilizar *property*

como elemento del patrón de entrada no es suficiente sino que tendremos que utilizar el producto cartesiano de *property* y *class* y seleccionar las combinaciones apropiadas mediante el uso de un filtro encargado de comprobar si una clase posee directa o indirectamente una determinada propiedad.

```
rule Class {
 from
  s: UML!Class
 to
  t: ER!EntityType (
   name <- s.name,
   features <- s.allAttributes() -> collect(p | Tuple {s = p, c = s},
   ...
  )
}
rule Attributes {
 from
  s : UML!Property,
  c : UML!Class (
   c.allAttributes()->includes(s)
  )
 to
  t: ER!Attribute (
   name <- s.name,
   type <- s.primitiveType
  )
}
```

4.3. Herencia

Tal y como ocurre en los catálogos de refactorizaciones de lenguajes orientados a objetos, el concepto de herencia, en nuestro caso entre reglas, constituye una categoría en si misma. En la tabla 1, por consiguiente, se presentan refactorizaciones para trabajar sobre este concepto.

La refactorización número 11 se encarga de extraer una super-regla (las subreglas, a parte de presentar funcionalidad común, deben tener como super-clase de los elementos de patrón de entrada y de salida super-clases comunes), mientras que la refactorización 14 realiza el trabajo contrario. Los *Bindings* comunes en subreglas que inicialicen atributos de las super-clases pueden ser extraídos hacía una super-regla común. Lo mismo ocurre con los filtros utilizados para restringir el lanzamiento de las reglas. Las refactorizaciones 12 y 13 ejecutan estas extracciones mientras que las refactorizaciones 14 y 15 realizan la operación contraria. Debe resaltarse que para la implementación de estas refactorizaciones se necesita conocimiento de los metamodelos de entrada y de salida. Por ejemplo, para determinar si un filtro puede ser extraido a una super-regla o para emplazar una super-regla extraída en una jerarquía de reglas ya existente.

En el siguiente ejemplo se presenta en detalle una de estas refactorizaciones de herencia.

Extraer una regla abstracta a partir de una regla *matched*
Problema: Dos o mas reglas implementan una funcionalidad similar (código duplicado) y comparten super-reglas comunes para sus elementos de patrón de entrada y salida.

Solución: Extraer una super-regla para albergar elementos comunes de subreglas, como *bindings* y filtros similares.
Precondiciones: Los elementos del patrón de salida y entrada de cada subregla tienen mismo tipo o tienen supertipos comunes.
Pasos:

1. Encontrar las super clases mas específicas en los metamodelos de entrada y salida para los elementos de patrón de entrada salida.
2. Si las subreglas ya tienen alguna super-regla, encontrar el lugar adecuado en la jerarquía de reglas para insertar la nueva.
3. Añadir una regla abstracta actuando como super-regla.
4. Añadir relaciones de herencia entre las subreglas y la nueva super-regla.
5. Extraer a la super-regla *bindings* y filtros comunes en las subreglas.

Ejemplo: Teniendo en cuenta nuestro ejemplo, después de haber extraído varías reglas *matched* para eliminar los elementos *distinc-foreach*, estas presentan *bindings* comunes. Para eliminar estos elementos de código duplicado se introducirá una nueva regla abstracta, *Property*, que actuará como super-regla para las anteriormente mencionadas subreglas. Aplicando luego la refactorización para extraer los *bindings* a la super-regla, el código duplicado quedará ahora encapsulado en la super-regla mejorando la mantenibilidad de la transformación.

```
abstract rule Property{ from
  s : UML!Property,
  c : UML!Class
 to
  t: ER!Feature (
   name <- s.name,
              ...
  )
}
rule Attribute extends Property{...}
rule WeakReference extends Property{...}
rule StrongReference extends Property{...}
```

4.4. Optimización y mejora de expresiones OCL

En muchos lenguajes de transformación basados en reglas, como ATL y QVT, se utiliza OCL para realizar consultas y cálculos sobre los modelos de entrada y salida. Parece por tanto necesario proporcionar refactorizaciones que ataquen los posibles problemas que estas expresiones puedan presentar. Para proporcionar refactorizaciones para las expresiones OCL que podemos encontrar en las transformaciones podemos reutilizar reglas de diseño ya existentes como las introducidas por [2] para mejorar la calidad de expresiones OCL en aspectos de legibilidad y mantenibilidad. En la tabla 1, las refactorizaciones 19 a 23 proporcionan dichas mejoras.

A parte de estas, en este trabajo se proponen nuevas refactorizaciones especialmente adaptadas a expresiones que aparecen en las transformaciones. La refactorización 17 permite optimizar *helpers* y la 18 eliminar la navegación insegura del modelo de salida.

Para mostrar el uso de estas refactorizaciones presentamos un ejemplo de refactorización que se puede aplicar de manera bastante frecuente. Se trata de la

conversión de un *helper* de operación en un *helper* (de nuevo, cabe destacar que aunque los términos son específicos de ATL, otros lenguajes tienen elementos similares) estático. Esta refactorización ayuda a mejorar el tiempo de ejecución debido a las técnicas de caching usadas comúnmente por los motores de transformación para evitar recalcular expresiones cuando usan el mismo contexto o parámetros.

Transformar un *helper* de operación en un *helper* estático
Problema: Un *helper* de operación que realiza un cálculo intensivo necesita ser llamado varias veces para los mismos elementos.
Solución: Convertir *helper* de operación en un *helper* estático.
Precondiciones: El helper no tiene parámetros.
Pasos:
1. Convertir el *helper* de operación en un *helper* estático
2. Sustituir las llamadas a *helper* de operación por llamadas al nuevo *helper* estático

Ejemplo: Trabajando de nuevo sobre el ejemplo propuesto, podemos aplicar esta refactorización al *helper* de operación que calcula todas las super clases de una clase dada (allClasses). Con respecto a la sintaxis, solamente hay que eliminar los paréntesis para transformar un *helper* de operación en uno estático. Aunque esta refactorización implica simplemente una pequeña modificación en la sintaxis, puede llevar a una gran mejora en cuanto al rendimiento de la transformación. Esto lo podremos ver en la siguiente sección dedicada a evaluar el impacto en el rendimiento de las refactorizaciones propuestas.

5. Evaluación de Rendimiento

Después de haber visto las capacidades de las refactorizaciones para mejorar la estructura interna de una transformación, ahora evaluaremos su impacto con respecto al rendimiento en ejecución.

Organización de la evaluación. Se han evaluado las siguientes versiones de la transformación propuesta como ejemplo. (T1) Transformación inicial (List. 1.1), (T2) transformación refactorizada para cambiar *helpers* de operación por estáticos, (T3) para extraer reglas *matched*, (T4) para extraer reglas *matched* con herencia, (T5) para extraer reglas *lazy*, (T6) para extraer reglas *lazy* con herencia. En el experimento, el modelo de entrada para las transformaciones contenía 30 clases donde cada clase contenía a su vez 30 propiedades. Además, cada clase heredaba al menos de otra clase sin contar, obviamente, con la raíz del modelo.
Resultados. Los resultados de la evaluación de rendimiento pueden verse en la tabla 2 Para evaluar el rendimiento en ejecución se han usado las siguientes métricas:

- **Tiempo de CPU**: Tiempo de ejecución en segundos. Sin contar el tiempo de carga del modelo de entrada ni el tiempo de serialización del modelo de salida.

- **Instrucciones**: Instrucciones ejecutadas. Cuántos *bytecode* (sentencias) son ejecutadas para producir el modelo de salida.
- **Aceleración**: Relación entre el tiempo de ejecución de la transformación original y la transformación refactorizada.

Cuadro 2. Resultados de la evaluación

Transformación	Instrucciones	Tiempo de CPU	Aceleración
T1	243.913	0,45 s	1,00
T2	238.308	0,42 s	1,07
T3	879.361	1,79 s	0,24
T4	507.073	0,97 s	0,44
T5	190.672	0,34 s	1,26
T6	179.684	0,31 s	1,38

Discusión. Como se puede ver en la tabla 2, la cadena de refactorizaciones que hemos aplicado a la transformación original no empeora el rendimiento y en algunos casos, incluso lo mejora de manera sustancial. Otros experimentos realizados (y que no podemos detallar aquí por falta de espacio) muestran resultados similares.

6. Implementación

Siguiendo un enfoque MDE, en este trabajo proponemos implementar las refactorizaciones propuestas como transformaciones de modelos. En ATL (y en otros lenguajes de transformaciones) las transformaciones están también expresadas como modelos. Por consiguiente, estas transformaciones pueden actuar como entrada a otras transformaciones (HOTs) encargadas de realizar la refactorización (transformación de modelo).

Además de la transformación a ser refactorizada, en algunos casos, como se ha ido indicando en la descripción del catálogo, necesitaremos los metamodelos de entrada y salida sobre los que esta actúa. Esto es así porque para algunas de las refactorizaciones propuestas será necesario razonar sobre los tipos de los elementos, que son instancia de elementos de estos metamodelos.

7. Trabajos relacionados

Numerosos trabajos han estudiando el uso de transformaciones para la implementación de refactorizaciones en modelos [16] [11] [10] [1] [15], [19], [9]. Sin embargo, ninguna de estas aproximaciones se ha centrado en las refactorizaciones de las propias transformaciones de modelos.

Existen algunos trabajos enfocados a la refactorización de expresiones OCL, lo que es, por supuesto, relevante para todos aquellos lenguajes de transformaciones que las utilizan para realizar consultas y cálculos. En nuestro catálogo

de refactorizaciones hemos reutilizado algunas refactorizaciones propuestas en otros trabajos [2,3] que son aplicables a transformaciones como por ejemplo *Acortamiento de navegaciones mediante cambio de contexto*. Nuestro catálogo complementa estas refactorizaciones proporcionando algunas refactorizaciones específicas para transformaciones como por ejemplo *Modificar tipo de helper* o *Eliminar navegaciones inseguras del modelo de salida*. Además, se han desarrollado refactorings para introducir, de manera retrospectiva, los patrones de optimización de [4] en transformaciones de modelos existentes como *Mejora del cálculo de referencias opuestas*.

Sólo se tiene conocimiento de un trabajo que mencione explícitamente refactorizaciones de transformaciones de modelos. En [6] los autores muestran como co-evolucionar transformaciones de grafos en el caso de que los metamodelos de los modelos implicados en la transformación cambien. En cualquier caso, esta es una noción totalmente diferente de refactoring. Mientras que nosotros estamos utilizando el termino refactorización denotando la mejora de la transformación sin modificar su semántica, en [6] la semántica de las transformaciones es modificada debido a los cambios en los metamodelos. Además, la noción de refactorizado utilizada en [6] no se preocupa en absoluto por la mejora de la calidad de la transformación sino por la adaptación de esta a los cambios entre versiones entre metamodelos.

8. Conclusiones y trabajos futuros

En este trabajo se ha mostrado como mejorar la mantenibilidad de las transformaciones de modelo a modelo mediante la adopción del concepto de refactorización, ya existente en otros dominios. Concretamente, hemos presentado un extenso catálogo de refactorizaciones para transformaciones M2M así como su aplicación a ejemplos concretos.

Como trabajos futuros consideramos que una serie de retos deben ser abordados. En primer lugar, la automatización del proceso de refactorización. Actualmente, las refactorizaciones son realizadas de una manera semi-automática. Los usuarios deben identificar manualmente que transformaciones (y dónde, dentro de éstas) deben ser refactorizadas y determinar igualmente cuales son las refactorizaciones más adecuadas a aplicar. Una vez hecho esto, la refactorización es aplicada de manera automática mediante la ejecución de la correspondiente transformación HOT. Planeamos desarrollar un conjunto de patrones para la identificación de *bad smells* en transformaciones que ayuden a los desarrolladores a identificar candidatos potenciales para ser refactorizados y a sugerir que refactorización del catálogo aplicar en cada caso.

La preservación del diseño del código es también otro reto. Las refactorizaciones son mas sencillas de implementar cuando se hace directamente en el nivel de sintaxis abstracta del lenguaje. Sin embargo, realizando la refactorización a este nivel se pierde el diseño hecho por los desarrolladores en el código fuente. Deben ser desarrollados por tanto, mecanismos para preservar este diseño.

Finalmente, uno de nuestros objetivos es que las refactorizaciones sean tan genéricas como sea posible, tanto en cuanto a su definición como a su implementación. Para lograr este objetivo sería necesario disponer de una definición mas

abstracta (independiente del lenguaje concreto) de transformaciones de modelo a modelo.

Referencias

1. P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Specifying integrated refactoring with distributed graph transformations. In *2nd Int. Workshop on Applications of Graph Transformations*, volume 3062 of *LNCS*, pages 220–235. Springer, 2003.
2. J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Sci. Comput. Program.*, 68(3):179–195, 2007.
3. A. L. Correa and C. Werner. Refactoring object constraint language specifications. *Software and System Modeling*, 6(2):113–138, 2007.
4. J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin. Optimization patterns for ocl-based model transformations. In *Models in Software Engineering*, volume 5421 of *LNCS*, pages 273–284. Springer, 2008.
5. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. volume 45, pages 621–646, 2006.
6. H. Ehrig, K. Ehrig, and C. Ermel. Refactoring of model transformations. *ECEASST*, 18, 2009.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code.* 1999.
8. F. Jouault and I. Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.
9. D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.
10. T. Mens. On the Use of Graph Transformations for Model Refactoring. In *Int. School on Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 219–257. Springer, 2006.
11. T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *1st Int. Conf. on Graph Transformation*, pages 286–301. Springer, 2002.
12. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.
13. OMG. *MOF QVT Final Adopted Specification.* Object Modeling Group, 2005.
14. W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, 1992.
15. I. Porres. Rule-based Update Transformations and their Application to Model Refactorings. *Software and System Modeling*, 4(4):368–385, 2005.
16. G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML Models. In *4th Int. Conf. on the Unified Modeling Language*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.
17. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *5th European Conf. on Model Driven Architecture*, pages 18–33. Springer, 2009.
18. J. Troya and A. Vallecillo. Towards a rewriting logic semantics for atl. In *3rd Int. Conf. on Theory and Practice of Model Transformations*, volume 6142 of *LNCS*, pages 230–244. Springer, 2010.
19. J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Model-driven Software Development—Research and Practice in Software Engineering*, pages 199–217. Springer, 2005.