Composición de Transformaciones de Modelos en MDD basada en el Álgebra Relacional

Roxana Giandini [(1)] Gabriela Pérez [(1)] Claudia Pons [(1) (2)]

(1) Universidad Nacional de La Plata, Facultad de Informática
LIFIA - Laboratorio de Investigación y Formación en Informática Avanzada
La Plata, Bs As, Argentina, 1900
(2) UAI - Universidad Abierta Interamericana, Argentina
[giandini, gperez, cpons]@lifia.info.unlp.edu.ar

Resumen: En el desarrollo de software dirigido por modelos (MDD), los conceptos claves son los modelos y las transformaciones entre modelos. Esta metodología puede ser vista en forma genérica como un proceso de desarrollo de software implementado mediante una red de transformaciones entre modelos que se combinan o componen en modos diversos.

Este trabajo propone un conjunto de operaciones para composición de transformaciones. La aplicación de estas operaciones fue justificada mediante ejemplos de uso. Se ha observado que la composición de transformaciones se basa fuertemente en analizar y definir operaciones de composición entre las relaciones que componen a las transformaciones, por lo que dichas operaciones están inspiradas en operaciones binarias del Álgebra Relacional. Finalmente se presenta la definición formal de estas operaciones, en base a un *profile* para transformaciones ya desarrollado por nuestro grupo de trabajo.

Palabras Claves: Ingeniería de Software, Desarrollo Dirigido por Modelos, Transformación de Modelos, Composición de Transformaciones

1. Introducción

El desarrollo de software dirigido por modelos - Model Driven Development (MDD) [1] - propone un proceso de desarrollo de software en el que los conceptos claves son los modelos y las transformaciones de modelos. En este proceso, el software se obtiene construyendo uno o más modelos y transformándolos en otros. La visión general de este proceso es que los modelos de entrada son independientes de la plataforma y los de salida son específicos de la plataforma y que, estos últimos pueden ser fácilmente transformados a formato ejecutable. En otras palabras, el proceso dirigido por modelos es comúnmente visto como un proceso de generación de código.
Existe también otra visión más genérica sobre este proceso, en la cual la diferencia entre ser independiente de la plataforma y ser específico no es el punto predominante. La clave para esta vista más genérica es que el proceso de desarrollo de software es implementado mediante una red de transformaciones que se combinan o componen en modos diversos. Esto hace al desarrollo dirigido por modelos mucho más abierto y flexible.
Algunas de las posibles composiciones que se pueden dar entre transformaciones son: - componer dos o más transformaciones existentes en una nueva transformación con sus nuevas relaciones. Esto podría definirse bajo operadores tales como la suma o diferencia de transformaciones.
- encadenar dos o mas transformaciones (encadenamiento secuencial), produciendo una nueva transformación, cuyo dominio será el dominio de la primer transformación y cuyo codominio será el codominio de la segunda.

- combinar dos transformaciones para lograr codominios de transformación más amplios (join acotado).

La composición de transformaciones requiere contar con constructores propios y operadores para composición dentro de un lenguaje simple de transformación entre modelos. Diversos lenguajes para transformación de modelos como ATL [7], Kent [8], TefKat [9] son conocidos actualmente por estar en una etapa avanzada de desarrollo y uso. A modo de síntesis, Czarnecki y Helsen en [15] proponen un framework para la clasificación de propuestas para transformación de modelos existentes. Algunas de estas propuestas incluyen actividades de composición entre modelos; por ejemplo el trabajo de Bézivin [16] introduce tres frameworks para composición de modelos y estudia sus características claves; mientras que Kolovos et al en [11] presentan operaciones para *merging* de modelos basadas en el lenguaje EOL [10]. Sin embargo, pocas de ellas tratan el aspecto de composición de transformaciones. El trabajo de Kleppe [12] presenta un ambiente para transformación de modelos llamado MCC, implementado como un *plug-in* para Eclipse; además describe someramente una clasificación de operaciones para combinar transformaciones como unidades independientes ejecutables.

En este trabajo presentamos un álgebra para composición de transformaciones. El conjunto de operadores definidos permitirá al usuario discernir bajo que condiciones aplicar cada uno y administrar su combinación para obtener los resultados esperados. La definición de estos operadores está inspirada en las operaciones binarias del álgebra relacional. Describimos también una formalización para esta propuesta como extensión de trabajos anteriores [14, 19], donde definimos un metamodelo y construimos un lenguaje minimal para transformación de modelos.

El artículo se estructura de la siguiente manera: la sección 2 presenta un análisis general del concepto de composición de transformaciones basado en el estudio de un ejemplo y en el metamodelo propuesto para transformaciones. La sección 3 justifica el motivo por el que basamos la definición de composición de transformaciones en el Álgebra Relacional y presenta informalmente, a través de ejemplos, las operaciones para composición de transformaciones. La sección 4 presenta la definición formal (en base al *profile* construido) de las operaciones de composición introducidas en la sección anterior. Finalmente la sección 5 presenta conclusiones y líneas de trabajo futuro.

2. Un análisis del Concepto de Composición de Transformaciones

En esta sección analizamos mediante un ejemplo de aplicación real la necesidad de contar con operadores de composición entre transformaciones. Supongamos que queremos definir una transformación que transforme todos los elementos que forman un diagrama de clases UML en elementos Java. Para simplificar esta tarea podría dividirse y asignarse subtareas a distintos grupos de trabajo. Cada uno de estos grupos desarrollará las reglas o relaciones necesarias para transformar un subconjunto de elementos de UML. En este caso, llamemos T1 y T2 a dos de estas transformaciones parciales. La cuestión que se plantea ahora es como componer estas transformaciones para obtener la transformación requerida inicialmente. Surge así la necesidad de definir operadores para combinar transformaciones. En este caso podemos llamar suma a la operación que se aplica a dichas transformaciones parciales para obtener una transformación más completa. Por ejemplo, la Figura 1 muestra dos transformaciones, T1 y T2. T1 transforma algunos elementos del metamodelo UML en elementos Java. Análogamente, T2 transforma otros elementos del metamodelo UML en elementos Java. Llamamos T1+T2 a la transformación resultante de sumar estas dos transformaciones, la cual deberá contener las relaciones necesarias para transformar los mismos elementos fuente, en los mismos elementos destino de las transformaciones originales.

Supongamos ahora que se requiere definir una transformación UML2Rel que convierta elementos UML en elementos del Modelo Relacional. Contamos con T1+T2 (definida entre UML y Java), y tenemos disponible otra transformación T3, que se aplica entre Java y el Modelo Relacional. Para obtener UML2Rel (llamada T1+T2&T3 en la Figura 1) existen al menos dos posibilidades: la primera es creando una nueva transformación e inicializándola definiendo las relaciones necesarias para transformar cada uno de los elementos; la segunda es realizar el encadenamiento secuencial de las transformaciones ya existentes mediante la aplicación de una operación. Como en el caso de la suma, esto resultará más beneficioso.
Por último, supongamos que no se tiene decidido en que plataforma se implementará el modelo UML debido a que, dependiendo de las propiedades del hardware y del software instalado en los equipos, se puede obtener una mayor performance usando una u otra plataforma. En consecuencia, es deseable mantener las opciones de transformación postergando la selección del modelo destino hasta el momento de la concreción. Las plataformas opcionales para este caso son el Modelo Relacional y un Modelo Orientado a Objetos. Para esto, contamos con una transformación T4 que convierte elementos UML en elementos del Modelo Orientado a Objetos. Se requiere por lo tanto, definir una transformación que permita especificar relaciones que conviertan elementos UML tanto en elementos del Modelo Relacional como en elementos del Modelo Orientado a Objetos. Resulta entonces necesaria la definición de un tercer operador, el cual llamaremos Join Acotado, que produce como resultado la transformación llamada T1+T2&T3#T4 en la Figura 1. Esta operación permitirá ampliar el codominio de la transformación resultante mediante la combinación de los metamodelos de las transformaciones originales. Por ejemplo, para una transformación de clases UML a elementos del Modelo Relacional, compuesta con otra transformación de clases UML a elementos del Modelo Orientado a Objetos, genera una relación que tenga como dominio a las clases UML y como codominio a las tuplas formadas por elementos del Modelo Relacional y elementos del Modelo Orientado a Objetos. Para poder aplicarse, las transformaciones a componer deben tener el mismo metamodelo como dominio.

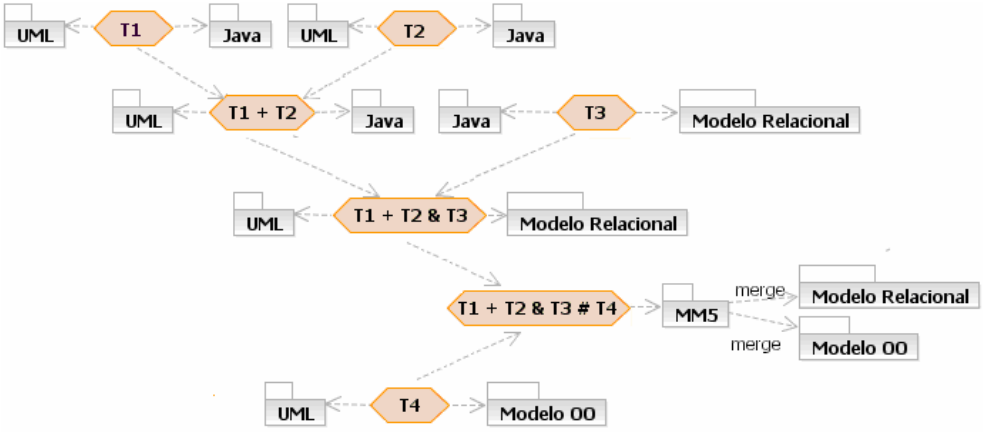

Figura 1: Ejemplo de composición de transformaciones

Mediante el desarrollo de este ejemplo se puede ver que la utilización de operadores de composición entre transformaciones tiene la ventaja de evitar la especificación de una nueva transformación, mediante la reutilización de elementos ya definidos.
En cuanto a la metodología a seguir para componer transformaciones debemos analizar la estructura de una transformación. La figura 2 muestra el metamodelo minimal para transformaciones propuesto en nuestro trabajo previo [19] e inspirado en el lenguaje QVT

(Query/View/Transformations) [4]. Siguiendo este metamodelo, una transformación puede verse como un conjunto de relaciones ya que éstas son sus componentes principales, que aplicadas a un dominio producen elementos del codominio. El dominio de una relación se define como el conjunto de elementos a los cuales efectivamente se aplica la relación. Al tipo de estos elementos lo llamaremos tipo de dominio de la relación. Análogamente se define el codominio como el conjunto de elementos obtenidos al aplicar la relación. Llamaremos tipo del codominio al tipo de los elementos pertenecientes al codominio.
Las relaciones pueden ser de distinto nivel: algunas se ejecutan automáticamente (relaciones *topLevel*), mientras que otras deben ser invocadas explícitamente. Se ha observado que la composición de transformaciones se basa fuertemente en analizar y definir operaciones de composición entre sus relaciones *topLevel*.

3. La Composición de Transformaciones Basada en el Álgebra Relacional

El lenguaje QVT, especificación estándar de OMG [2] para transformaciones, y otros lenguajes basados en él, definen a las transformaciones en términos de relaciones que transforman elementos particulares de determinados metamodelos en otros elementos posiblemente expresados en otro metamodelo. Como se describió en la sección 2, al componer dos transformaciones, componemos sus relaciones. En cada relación se define dominio y codominio. Dentro de los dominios, se definen variables y expresiones que especifican la transformación entre el tipo del dominio y el tipo del codominio, y que pueden estar embebidas en forma de predicados en las cláusulas *when* y *where* de la relación (ver Figura 2). Combinar estos predicados puede provocar que un mismo elemento del dominio se relacione con más de un elemento en el codominio (como en el caso del join acotado). Esto es correcto ya que hablamos de relaciones en el sentido matemático y no de funciones. Bajo este análisis podemos concluir que para combinar o componer relaciones de transformaciones, resulta válido aplicar las operaciones binarias del Álgebra Relacional.
En el siguiente apartado recordamos la definición de las operaciones del Álgebra Relacional y luego presentamos la definición e introducimos ejemplos de algunas operaciones para composición de transformaciones que se basan en ellas.

3.1 El Álgebra Relacional - Operaciones Binarias

Los matemáticos definen una relación como un subconjunto del producto cartesiano de una lista de dominios. En el área de Bases de Datos Relacionales, esto se corresponde exactamente con la definición de Tabla, por lo que pasa a usarse el término matemático de relación y tupla, en vez de los términos tabla y fila [18].
Un lenguaje de consulta es un lenguaje en el cual un usuario solicita información desde la base de datos. El álgebra relacional es un lenguaje de consulta procedural. Hay cinco operaciones fundamentales en esta álgebra. Dos unarias: selección y proyección, y tres binarias: producto cartesiano, unión o suma y diferencia de conjuntos. Todas estas operaciones producen una nueva relación como resultado. Adicionalmente, otras operaciones que pueden definirse en términos de las fundamentales son: theta -Joint, joint natural, intersección y división.
Introducimos a continuación algunas de estas operaciones, que nos dan base para la definición del álgebra de transformaciones:

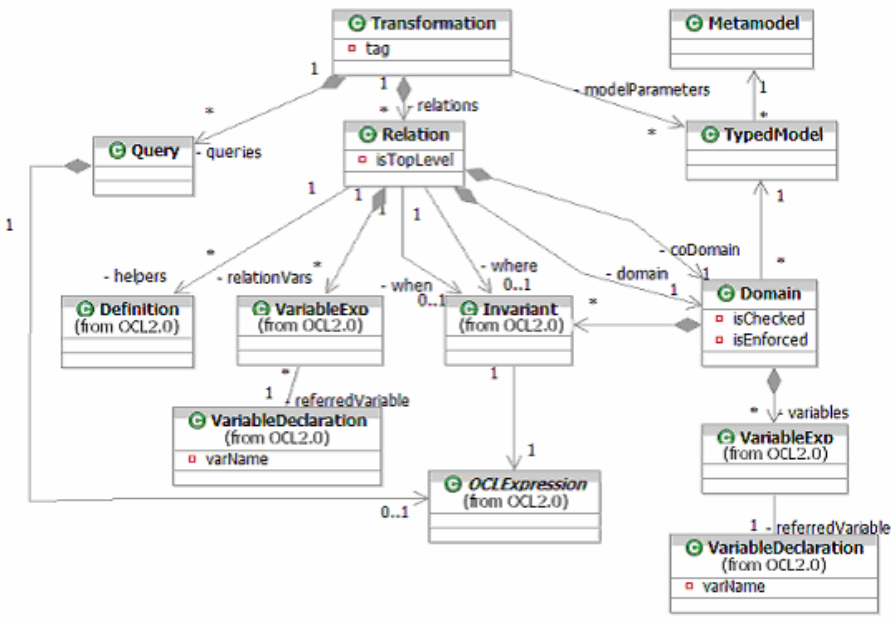

Figura 2. Metamodelo para Transformación de Modelos

Sean las relaciones R y S definidas de la siguiente manera:
R = {(a1, a2) / a1 ∈ D1 y a2 ∈ D2 }
S = {(b1,b2)/ b1 ∈ D3 y b2 ∈ D4 } donde
D1 y D2 son dominio y codomino de R respectivamente y D3 y D4 son dominio y codomino de S respectivamente.
Sobre R y S se definen las operaciones binarias:

- *Unión (U):* R U S es el conjunto de tuplas formado por los elementos que están en R, en S o en ambos. RUS = { (c1,c2) / (c1,c2) ∈ R ó (c1,c2) ∈ S }
- *Producto cartesiano (X):* R X S es el conjunto de todas las tuplas (k1 + k2) cuyos primeros k1 elementos forman una tupla en R y los últimos k2 elementos forman una tupla en S. R X S = { ((a1,a2),(b1, b2)) : (a1,a2) ∈ R y (b1, b2) ∈ S}
  - ❖ *Join Natural (#):* es una operación derivada del Producto cartesiano, donde las tuplas se forman uniendo las tuplas de R y de S que tienen variables del dominio en común.
- *Diferencia de conjuntos (-)* R – S es el conjunto de tuplas que pertenecen a R y que no están en S. R – S = { (c1,c2) / (c1,c2) ∈ R y (c1, c2) ∉ S }

3.2 El Álgebra de Transformaciones - Operaciones Binarias

Abordaremos en detalle en este trabajo las operaciones de suma o unión, encadenamiento secuencial y join acotado entre transformaciones. La definición del resto de las operaciones es parte de nuestro trabajo futuro.
En algunos casos al componer dos transformaciones (por ejemplo en la suma), hay que analizar sus conjuntos de relaciones para determinar cuales predican sobre elementos en común y cuales no. Las relaciones que no tienen elementos en común van a formar parte de la transformación resultante en su forma original, mientras que en el caso de las otras relaciones

habrá que aplicar el operador correspondiente pero esta vez entre relaciones, para obtener una nueva relación que forme parte de la transformación resultante.
En otros casos, como en el encadenamiento secuencial, habrá que analizar otras condiciones que las relaciones deban cumplir para componerse en secuencia.
En conclusión, se puede decir que la composición entre transformaciones puede llevarse al plano de definir operadores de composición en términos de las relaciones que las componen y cumplen con las condiciones necesarias para poder componerse.
En general, las operaciones que presentamos consideran a las relaciones definidas de un solo dominio a un solo codominio, como aparece especificado en el metamodelo de la Figura 2. Para describir en forma clara cada una de las operaciones, a continuación se definen las transformaciones introducidas en el ejemplo de la sección 2, sobre las cuales se aplicaran, paso a paso, las operaciones introducidas también en la misma sección.

Suma de transformaciones

La idea intuitiva de la suma de dos transformaciones T1 y T2, es que la transformación resultado contenga todas las relaciones definidas en las transformaciones originales. En este contexto, se debe analizar cada relación de T1 respecto a todas las relaciones de T2 y viceversa. Para cada relación de T1 si esta definida sobre elementos distintos a los que se aplican las relaciones de T2, dicha relación aparecerá sin modificaciones en la transformación resultante. En cambio, por cada relación existente en T2 que se aplica sobre elementos comunes, se creará una nueva relación resultante de la suma de las dos originales. Las relaciones de T2 que no fueron combinadas con relaciones de T1 aparecerán también sin modificaciones en la resultante.
Para comprender mejor la suma de dos transformaciones, introducimos a continuación la especificación de dos transformaciones T1 y T2 y de la transformación T1+T2, resultante de aplicar la suma entre ellas. La especificación de las transformaciones se muestra (por restricciones de espacio) en formato textual. Este formato se genera automáticamente al imprimir a texto una especificación gráfica construida instanciando nuestro *profile* para transformaciones.
Los ejemplos de transformaciones presentados están inspirados en el Catálogo de Lano[17].

```
Transformation T1 (Uml: UML2.0, Java: JAVA)  {
TopLevel Relation PersistentClass2ClassWithKey {
        checkonly domain Uml c: Class
        checkonly domain Java jc: JavaClass
        when {c.isPersistent}
        where {c.name =jc.name and jc.ownedFields-> exists (jf: JavaField |
jf.name = "primaryKey" and jf.type = "Integer") and jc. ownedMethods -> exists
(jm: JavaMethod | jm.name = "getPrimaryKey" and jm.returnType = "Integer")  } }

Transformation T2 (Uml: UML2.0, Java: JAVA)  {
TopLevel Relation Class2Class {
        checkonly domain Uml c: Class
        checkonly domain Java jc: JavaClass
        when {}
        where { c.name = jc.name and
c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods -> exists (jm1,
jm2 : JavaMethod | Attr2Getter (p, jm1) and Attr2Setter (p, jm2))  }}

Relation Attr2Getter{
checkonly domain Uml a: Property
checkonly domain Java jm: JavaMethod
when { }
```

```
where { 'get' +  p.name.firstUpperCase() = jm.name  } }

Relation Attr2Setter{
 checkonly domain Uml p: Property
 checkonly domain Java jm: JavaMethod
 when { ! p.isReadOnly }
 where {jm.name = 'set' + p.name.firstUpperCase() and jm.ownedParameter-
 >first().name = 'a' + p.name.firstUpperCase()  and jm.ownedParameter-
 >first().type = p.type} }

 Query firstToUpperCase(string: String) : String {
 self.substring(1,1).toUpperCase()+self.substring(2,self.size())}

}
```

Ambas transformaciones T1 y T2 se aplican entre un modelo UML y un modelo JAVA.
T1 define una única relación *topLevel*, llamada PersistentClass2ClassWithKey. El tipo del dominio de esta relación son todas las clases UML. El tipo de codominio de esta relación son las clases Java. La cláusula *when* de esta relación determina su dominio, que esta restringido a todas las clases persistentes. Al aplicar la relación, por cada clase UML, se generará una clase Java con el mismo nombre, a la cual se introduce un nuevo atributo de identidad, llamado primaryKey de tipo Integer para cumplir la función de clave primaria. Además, para permitir la lectura de este atributo se agrega una nueva operación llamada getPrimaryKey.
T2 define una única relación *topLevel*, Class2Class, cuyo dominio son todas las clases dado que la cláusula *when* no tiene condición (se asume true), por lo tanto no restringe los elementos del tipo del dominio. Las clases Java son el tipo de su codominio.
Class2Class invoca a otras dos relaciones, Attr2Getter y Attr2Setter. Al aplicar la relación Class2Class, en el modelo Java resultante se agrega un método getter y uno setter por cada atributo definido en la clase original. Además de estas relaciones, la transformación incluye un *query* que, dado un *string*, retorna el mismo *string* con el primer carácter en mayúscula.
Como dijimos en la sección 2, al sumar T1 con T2 hay que analizar las relaciones que las componen. Attr2Getter y Attr2Setter, al no ser *topLevel*, formarán parte de la suma, tal cual estaban definidas teniendo en cuenta que no se produzcan conflictos de nombre entre las relaciones de una y otra transformación. Lo mismo ocurre con el *query* definido en la transformación. En cambio, entre las relaciones *topLevel* PersistentClass2ClassWithKey y Class2Class, definidas sobre los mismos tipos de elementos, se aplica la operación de suma para relaciones. Esta operación crea una nueva relación con el mismo tipo de dominio y el mismo tipo de codominio que las relaciones originales. El dominio de la relación resultado será la unión de los dominios de las relaciones origen. La cláusula *when* resultante se define como una disyunción entre las cláusulas *when* de las relaciones originales. La cláusula *where* se define como una disyunción entre las dos conjunciones formadas por los *when* y *where* de las relaciones originales:

```
Transformation T1 + T2  (Uml: UML2.0, Java: JAVA)  {
TopLevel Relation PersistentClass2ClassWithKey+Class2Class {
        checkonly domain Uml c: Class
        checkonly domain Java jc: JavaClass
        when { true or c1.isPersistent }
        where {
        (true and c.name = jc.name and
c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods -> exists (jm1,
jm2 : JavaMethod | Attr2Getter (p, jm1) and Attr2Setter (p, jm2))  )
OR
(c.isPersistent and c.name =jc.name and jc.ownedFields-> exists (jf: JavaField
 | jf.name = "primaryKey" and jf.type = "Integer") and jc. ownedMethods ->
```

```
exists (jm: JavaMethod | jm.name = "getPrimaryKey" and jm.returnType =
"Integer")) }  }
```

Encadenamiento secuencial de transformaciones

La idea intuitiva de componer en secuencia dos transformaciones es generar una transformación compuesta cuya aplicación sea equivalente a aplicar primero una transformación T1 y a ese resultado, aplicarle una segunda transformación T2. Solo los elementos del modelo de entrada de T1 que tengan como salida elementos del modelo de entrada de T2, pertenecerán a la transformación compuesta. Aquellos que no tengan elementos que los conecten no pertenecerán al resultado.
Esta operación fue definida debido a que al desarrollar ejemplos de composición resultó interesante y útil componer transformaciones en secuencia, donde sus relaciones se encadenen siempre que el tipo de codominio de la primera relación sea igual al tipo de dominio de la segunda relación. Cada tupla se compone entonces de un elemento del dominio de la primera y un elemento del codominio de la segunda.
Sea la transformación T3 definida con una única relación *topLevel* que convierte clases Java en tablas de Modelo Relacional con igual nombre. Además, los atributos simples de las clases JAVA pasan a ser columnas, con igual nombre e igual tipo, en la tabla correspondiente:

```
Transformation T3 (Java: JAVA, Rel: RDBMS)  {
TopLevel Relation Class2Table {
        checkonly domain Java jc: JavaClass
        checkonly domain Rel t: Table
        when { }
        where {jc.name = t.name and jc.ownedFields -> forAll (jf:JavaField |
t.column -> exists (co | Attr2Col (jf, co))
    Relation  Attr2Col {
    checkonly domain Java jf: JavaField
    checkonly domain Rel co: Column
    when {not jf.isMultivalued()}
    where { co.type = jf.type and co.name = jf.name } }
```

Como en otros casos de composición, al encadenar dos transformaciones, hay que analizar las relaciones que las componen. Siguiendo con el ejemplo, si encadenamos en secuencia la transformación T1+T2 con T3, por cada relación de la transformación T1+T2 se estudia su conexión con cada relación de T3. Se aplicará la operación de encadenamiento secuencial pero ahora entre relaciones; en este ejemplo las relaciones que cumplen con la condición para ser encadenadas son PersistentClass2ClassWithKey + Class2Class de T1+T2 y Class2Table de T3. Se genera así una nueva relación *topLevel* que convierte clases UML en tablas del Modelo Relacional. En este caso, el encadenamiento secuencial resultará en la siguiente transformación:

```
Transformation T1+T2 & T3 (Uml: UML2.0, Rel: RDBMS)  {
TopLevel Relation PersistentClass2ClassWithKey+Class2Class & Class2Table {
        checkonly domain Uml c: Class
        checkonly domain Rel t: Table
        when { true or c1.isPersistent }
        where {
LET jc: Class =
        (true and c.name = jc.name and
        c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
        exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods -> exists
        (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and Attr2Setter (p, jm2))
        ) or
```

```
        (c.isPersistent and c.name =jc.name and jc.ownedFields-> exists (jf:
         JavaField | jf.name = "primaryKey" and jf.type = "Integer") and jc.
         ownedMethods -> exists (jm: JavaMethod | jm.name = "getPrimaryKey" and
         jm.returnType = "Integer") )
 AND true
IN
jc.name = t.name and jc.ownedFields -> forAll (jf:JavaField |
t.column -> exists (co | Attr2Col (jf, co)) }
```

Al observar la transformación resultante, podemos enunciar algunas cuestiones sobre las relaciones que fueron encadenadas:

- En la relación compuesta aunque los elementos intermedios no aparecen ni en el dominio ni en el codominio, las condiciones entre ellos deben seguir cumpliéndose, por lo que la cláusula *where* de la primera relación debe cumplirse. Por esta razón debe formar parte de la cláusula *where* de la transformación resultante. Lo mismo ocurre con la cláusula *when* de la segunda relación. Estas condiciones intermedias aparecen especificadas mediante una expresión LET de OCL dentro de la cláusula *where* de la relación resultante.
- Las relaciones invocadas por las *topLevel* (por ej. Attr2Setter) que se aplican a elementos de metamodelos intermedios, deberán también ser parte de la cláusula *where* de la relación compuesta, ya que el metamodelo intermedio esta oculto y en caso contrario quedarían mal definidas. Por lo tanto, deben estar incluidas también como expresiones LET dentro del *where* resultante. Por limitaciones de espacio y por legibilidad, en el ejemplo no se encuentran incluidas en la cláusula *where*.

Join acotado (#) de transformaciones

Esta operación se basa en el Join Natural del Álgebra Relacional. Tiene como precondición que las transformaciones se apliquen sobre el mismo metamodelo de entrada. La idea intuitiva al componer dos transformaciones T1 y T2 mediante un join es que la transformación compuesta mantenga como entrada los elementos de entrada en común de T1 y T2 y como salida los elementos de salida de ambas transformaciones. Es decir, componer dos transformaciones para lograr codominios de transformación más amplios, para un mismo elemento del dominio. Como en los otros casos, debemos analizar como componer las relaciones. La idea es que se compongan aquellas relaciones de cada transformación que se apliquen sobre el mismo dominio. El codominio se forma con los codominios de ambas relaciones, dando así la posibilidad de que a un mismo elemento del dominio, le correspondan 2 elementos en el codominio.
Consideremos la transformación T4 que convierte mediante una relación *topLevel*, clases UML a clases del Modelo Orientado a Objetos:

```
Transformation T4 (Uml: UML2.0,oodbms: OODBMS)  {
TopLevel Relation Class2 OODBMSClass {
        checkonly domain Uml c: Class
        checkonly domain oodbms c2: Class
        when { }
        where {c.name = c2.name and c. ownedAttributes -> forAll (p: Property |
if (p.type.oclIsKindOf(Datatype)) then c2.attributes -> exists (a: Attribute |
p.name = a.name and p.type = a.type) and c2.attributes -> exists (a:Attribute |
a.name = "OID" and a.type = String) }
```

La aplicación de la operación Join acotado entre las transformaciones T1+T2 & T3 y T4 resulta en la siguiente transformación:

```
Transformation T1+T2&T3 # T4 (Uml:UML2.0,Mm: RDBMSMergeOODBMS){
```

```
TopLevel Relation PersistentClass2ClassWithKey+Class2Class & Class2Table #
Class2 OODBMSClass {
        checkonly domain Uml c: Class
        checkonly domain Mm t: Table,  Mm c2: Class
        when { true or c1.isPersistent and true }
        where {
LET jc: Class =
        (true and c.name = jc.name and
        c. ownedAttribute -> forAll (p : Property |  ( jc.ownedFields->
        exists(jf:JavaField | p.name = jf.name) and  jc.ownedMethods -> exists
        (jm1, jm2 : JavaMethod | Attr2Getter (p, jm1) and Attr2Setter (p,
        jm2))) or
       (c.isPersistent and c.name =jc.name and jc.ownedFields-> exists (jf:
        JavaField | jf.name = "primaryKey" and jf.type = "Integer") and jc.
        ownedMethods -> exists (jm: JavaMethod | jm.name = "getPrimaryKey" and
        jm.returnType = "Integer")  )
IN
        jc.name = t.name and jc.ownedFields -> forAll (jf:JavaField | t.column
-> exists (co | Attr2Col (jf, co))
        AND
        c.name = c2.name and c. ownedAttributes -> forAll (p: Property |  if
(p.type.oclIsKindOf ( Datatype)) then c2.attributes -> exists (a: Attribute |
p.name = a.name and p.type = a.type) and and c2.attributes -> exists
(a:Attribute | a.name = "OID" and a.type = String) } }
```

Puede verse que el metamodelo de la transformación resultante coincide con el metamodelo de las transformaciones originales, mientras que el metamodelo de salida se define como un metamodelo Mm producto del *merge* entre los metamodelos de salida de las transformaciones originales.
Al igual que en los demás casos, al componer transformaciones se componen las relaciones que las forman. En este caso, se componen las dos únicas relaciones *topLevel* definidas en las transformaciones. La relación resultante mantiene el tipo de dominio de las originales. El tipo del codominio esta compuesto por las variables de los tipos de codominio de las relaciones originales. La cláusula *when* de la relación resultado, se define como una conjunción entre las cláusulas *when* de las relaciones originales, formando así el dominio con la intersección de ambos dominios originales. La cláusula *where* se define como una conjunción de las cláusulas *where* de las relaciones originales.

4. Una Formalización para Composición de Transformaciones

Presentamos en esta sección una especificación formal para las operaciones introducidas intuitivamente en la sección anterior. Esta formalización se basa en el *profile* que implementa el metamodelo para Transformaciones inspirado en QVT. Dicho *profile* utiliza el lenguaje para restricciones OCL [5] y extiende a la Infrastructure 2.0 [6], instancia de MOF [3]. La implementación completa del *profile* puede encontrarse en nuestro trabajo anterior [19].
En este trabajo, las operaciones están especificadas en términos de las metaclases pertenecientes al metamodelo de la Figura 2, abstrayéndonos de las particularidades de implementación del *profile*.

4.1 Definición de las operaciones de Composición

Las operaciones están definidas en OCL especificando sus pre y postcondiciones, en el contexto de un elemento Relation. Una Relation cumple con ciertas restricciones y se

compone de dominio, codominio, cláusulas *when* y *where*, helpers y eventualmente un conjunto de variables globales (relationVars) de tipo de dato primitivo que describen generalmente nombres de variables comunes a los dominios de la relación.
En general, las operaciones tienen otra Relation como parámetro, con la cual se compone la receptora, y dan como resultado una nueva Relation. Los prerrequisitos que piden las operaciones, se expresan en forma de precondiciones, mientras que las definiciones de cada elemento que conforma a la Relation resultante (dada por la pseudovariable result de OCL), se expresa en las postcondiciones de la operación.

Suma de Relaciones (+)

```
Context Relation:: suma (r2: Relation) : Relation
```
Precondiciones*:*
Los metamodelos del dominio y del codominio de ambas relaciones coinciden, y los tipos del dominio y codominio también.
```
Pre: self.domain.typedModel.metamodel = r2.domain.typedModel.
metamodel and self.coDomain.typedModel.metamodel= r2.coDomain.
typedModel.metamodel
```

```
Pre: self.domain.variables-> collect(v|v.type) = r2.domain.variables -
> collect(v| v.type) and self.codomain.variables->collect(v|v.type) =
r2.codomain.variables->collect(v| v.type)
```

Suponemos que los nombres de variables de dominio y codominio de ambas relaciones coinciden. En caso contrario, se deben renombrar las variables de r2 para que haya coincidencia, modificando también las ocurrencias de tales variables en el body del *when* / *where* donde sea necesario.

Postcondiciones*:*
El nombre de la relación resultado es la unión de los nombres de ambas relaciones con '+' intercalado.
```
Post: result.name = self.name.concat ('+').concat (r2.name)
```

La cláusula *when* se forman como la disyunción entre los *when* de las relaciones originales. Si alguna relación no tiene definida su cláusula *when*, se entiende que aplica a todos los elementos del tipo de dominio, por lo cual se puede interpretar que la cláusula tiene valor *true*.
```
Post: result.when.bodyExpression = self.when. getBodyExpression or
r2.when. getBodyExpression
```

La operación getBodyExpression retorna la expresión de la cláusula *when* en caso que exista, o una expresión true en caso contrario.

La cláusula *where* se forma con la disyunción entre las conjunciones formadas por los *when* / *where* de las relaciones originales.
```
Post: result.where.bodyExpression = (self.when.getBodyExpression and
self.where.bodyExpression) or (r2.when.getBodyExpression and
r2.where.bodyExpression)
```

Los metamodelos de dominio y codominio son los de la relación receptora.
```
Post:result.domain.typedModel.metamodel =
self.domain.typedModel.metamodel

Post:result.coDomain.typedModel.metamodel=
self.coDomain.typedModel.metamodel
```

Las variables del dominio y codominio coinciden con las variables de la relación receptora.
```
Post:result.domain.variables = self.domain.variables
Post:result.coDomain.variables = self.coDomain.variables
```

Los helpers se forman con la unión de los helpers de las relaciones originales.
Post: `result.helpers = self.helpers ->union (r2.helpers)`

Las variables globales se forman con la unión de las variables globales de las relaciones originales.
Post: `result. relationVars = self.relationVars -> union (r2. relationVars)`

Encadenamiento secuencial de relaciones (&)

`Context Relation:: secuencia (r2: Relation) : Relation`
Precondiciones:
Los elementos del codominio de la relación receptora coinciden con los elementos del dominio de la relación parámetro.
Pre: `self.coDomain.variables-> collect(v|v.type)= r2.domain.variables->collect(v| v.type)`
Suponemos que los nombres de variables de dominio de la relación r2 coinciden con los de las variables de codominio de la relación receptora. En caso contrario, se deben renombrar como se especificó en la suma.

Postcondiciones:
El nombre de la relación resultado es la unión de los nombres de ambas relaciones con '&' intercalado.
Post: `result.name = self.name.concat ('&').concat (r2.name)`

El metamodelo del dominio de la relación resultado coincide con el metamodelo del dominio de la relación receptora.
Post:`result.domain.typedModel.metamodel = self.domain.typedModel.metamodel`

El metamodelo del codominio de la relación resultado coincide con el metamodelo del codominio de la relación parámetro.
Post:`result.coDomain.typedModel.metamodel = r2.coDomain.typedModel.metamodel`

Las variables de dominio coinciden con las variables de la relación receptora. Las variables del codominio coinciden con las variables de la relación parámetro.
Post:`result.domain.variables = self.domain.variables`
Post:`result.coDomain.variables = r2.coDomain.variables`

La cláusula *when* es la cláusula *when* de la relación receptora.
Post: `result.when.bodyExpression = self.when. bodyExpression`

La cláusula *where* de la relación resultante se forma con el *where* de la relación parámetro. Como parte de esta cláusula se define, mediante una expresión LET de OCL, la conjunción de la cláusula *where* de la relación receptora con la cláusula *when* de la relación parámetro. De la misma manera, las relaciones restantes, no *topLevel*, deben ser rescritas también en forma de LETs anidados, dentro de la cláusula *where*.

```
Post: result.where. bodyExpression =
LET self.codomain.variables. first.eferredVariable =
self.where.bodyExpression and r2.when. bodyExpression
IN r2.where.bodyExpression
```

Los helpers y las variables globales se forman de la misma manera que en el caso de la suma.

Join acotado de Relaciones (#)

Context Relation:: join# (r2: Relation):Relation
Precondiciones*:*
Los metamodelos del dominio de ambas relaciones coinciden.
Pre: `self.domain.typedModel.metamodel= r2.domain.typedModel.metamodel`

Los elementos del dominio de ambas relaciones coinciden.
Pre: `self.domain.variables->collect(v|v.type)= r2.domain.variables->collect(v| v.type)`
Suponemos que los nombres de variables del dominio de ambas relaciones coinciden. En caso contrario, se deben renombrar como se especificó en la suma.

Postcondiciones*:*
El nombre de la relación resultado es la unión de los nombres de ambas relaciones con '#' intercalado.
Post: `result.name = self.name.concat ('#').concat(r2.name)`

El metamodelo del dominio es el metamodelo del dominio la relación receptora.
Post:`result.domain.typedModel.metamodel = self.domain.typedModel.metamodel`

El nombre del metamodelo del codominio de la relación resultado es la concatenación de los nombres con la palabra 'Merge' intercalada.
Post:`result.codomain.typedModel.metamodel.name= self.codomain.typedModel.metamodel.name.concat('Merge'). concat(r2.codomain.typedModel.metamodel.name)`

El metamodelo del codominio de la relación resultado se forma aplicando un *merge* con el metamodelo del codominio de la relación receptora y un *merge* con el codominio de la relación parámetro.
Post: `result.coDomain.typedModel.metamodel.packageMerge-> exists (pm | pm.mergedPackage = self.coDomain.typedModel.metamodel) and result.coDomain.typedModel.metamodel.packageMerge-> exists (pm | pm.mergedPackage = r2.coDomain.typedModel.metamodel)`

Las variables del dominio coinciden con las variables de la relación receptora.
Post:`result.domain.variables = self.domain.variables`

Las variables del codominio se forman con la composición de las variables de los codominios de las relaciones originales.
Post:`result.coDomain.variables = self.coDdomain.variables -> union(r2.coDomain.variables)`

Post: `result.when.bodyExpression = self.when.bodyExpression and r2.when.bodyExpression`
Post: `result.where. bodyExpression = self.where.bodyExpression and r2.where.bodyExpression`

Los helpers y las variables globales se forman de la misma manera que en el caso de la suma.

5. Conclusiones

El desarrollo de software dirigido por modelos puede ser visto en forma genérica como un proceso de desarrollo de software implementado mediante una red de transformaciones entre

modelos que se combinan o componen en modos diversos. Esto hace al desarrollo dirigido por modelos mucho más abierto y flexible.
La utilización de operadores de composición entre transformaciones tiene la ventaja de evitar definir nuevas transformaciones, mediante la reutilización de elementos ya definidos.
En este trabajo hemos propuesto un conjunto de operaciones para composición de transformaciones. La aplicación de estas operaciones fue justificada mediante ejemplos de uso. Además presentamos la definición formal en OCL de ellas, en base al *profile* para transformaciones ya desarrollado por nuestro grupo de trabajo. Se ha observado que la composición de transformaciones se basa fuertemente en analizar y definir operaciones de composición entre las relaciones que componen a las transformaciones, por lo que dichas operaciones están inspiradas en las operaciones binarias del Álgebra Relacional.
En lo inmediato, implementaremos estas operaciones integrándolas a la herramienta *Case* [13] que nuestro grupo de investigación está desarrollando. Resta abordar el estudio y formalización de otras operaciones de composición entre transformaciones que surgen del Álgebra Relacional, como diferencia, producto Cartesiano, intersección, etc. El análisis incluirá en que casos el desarrollador puede encontrarle una utilización conveniente a la aplicación de estas operaciones.
Resulta interesante también extender este *profile* para soportar mecanismos de *Traceability* que permitan recuperar información acerca del origen de los elementos que forman las diferentes composiciones.

Referencias

1. MDA Guide, v1. 0. 1, omg/03-06-01, June 2003. http://www. omg. org.
2. OMG (Object Management Group) http://www. omg. org
3. Meta Object Facility (MOF) 2. 0. OMG Adopted Specification. 2003. http://www. omg. org.
4. MOF 2. 0 Query/View/Transformations (QVT) - OMG Adopted Specification. March 2005. http://www. omg. org.
5. OMG. The Object Constraint Language Specification – Version 2. 0, for UML 2. 0, April 2004.
6. The Unified Modeling Language Infrastructure version 2. 0, OMG Final Adopted Specification. March 2005. 7. Jouault F. , Kurtev I. Transforming Models with ATL Workshop in Model Transformation in Practice at the MoDELS 2005 Conference. Montego Bay, Jamaica, Oct 3, 2005
8. Akehurst D , Howells W. , McDonald-Maier K. Model Transformation Language. Workshop in Model Transformation in Practice - MoDELS 2005 Conference, Jamaica, Oct 3, 2005
9. Lawley M. , Steel J. Practical Declarative Model Transformation with TefKat. Workshop in Model Transformation in Practice - MoDELS 2005 Conference. Jamaica, Oct 3, 2005
10. Dimitrios Kolovos, Richard Paige and Fiona Polack. The Epsilon Object Language (EOL). In Proc. Model Driven Architecture Foundations and Applications: 2nd European Conference, ECMDA-FA, vol 4066 of LNCS, pages 128–142, Spain, June 2006.
11. Dimitrios Kolovos, Richard Paige and Fiona Polack. Merging Models with the Epsilon Merging Language (EML). In Proceedings of MoDELS 2006 Conference, Session Model Integration. Genova, Italy, October 2006.
12. Anneke Kleppe. MCC: A Model Transformation Environment. A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 173 – 187, Bilbao, Spain, June 2006.
13. Pons C., Giandini R., Pérez G., et al. Precise Assistant for the Modeling Process in an Environment with Refinement Orientation. "UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers". LNCS #3297. Springer, 2004.
14. Giandini, R. , Pons, C. Un lenguaje para Transformación de Modelos basado en MOF y OCL. Actas de XXXII CLEI 2006. Santiago, Chile. Agosto, 2006.
15. Czarnecki, Helsen. Feature-based survey of model transformation approaches.IBM System Journal,V45,N3, 2006
16. Bézivin J. et al. A canonical Scheme for Model Composition. A. Rensink and J. Warmer (Eds.) ECMDA-FA 2006, LNCS 4066, pp. 346 – 360, Bilbao, Spain, June 2006.
17. Lano K., Catalogue of Model Transformation. 2006, http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf
18. Ullman J. Principles of Database System. Computer Science Press. 1980
19. Giandini, R, Pérez, G, Pons, C. A minimal OCL-based Profile for Model Transformation. VI Jornada Iberoamericana de Ing. de Soft. e Ing. del Conocimiento JIISIC'07.Lima, Perú. Feb, 2007.