


3. QVT Relations

Antonio Navarro Martín
Profesor Titular de Universidad
Dpto. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
anavarro@fdi.ucm.es

Referencias

- OMG Meta Object Facility (MOF) 2.0 Query/View/Transformation V1.1
  http://www.omg.org/spec/QVT/1.1/

Índice

- Introducción
- Ejemplo
- Transformaciones y tipos de modelos
- Relaciones y dominios
- Cláusulas `when` y `where`
- Relaciones de nivel superior
- `Checkonly` y `enforced`

Índice

- Emparejamiento de patrones
- Claves y creación de objetos
- Restricciones sobre expresiones
- Propagación del cambio
- Transformaciones internas
- Integración de operaciones de caja negra
- Transformacione `Checkonly`

Índice

- Notación visual
- Ejemplo

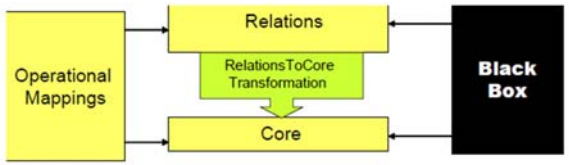
Introducción

- El estándar Query/View/Transformation define tres lenguajes de transformación entre metamodelos descritos en MOF:
  - QVT Core: un lenguaje básico de transformaciones
  - QVT Relations: lenguaje declarativo de transformaciones
  - QVT Operational Mappings: lenguaje de corte imperativo para transformaciones

Introducción

- El estándar define estos lenguajes como metamodelos MOF definidos en tres paquetes:
  - `QVTCore`
  - `QVTRelation`
  - `QVT`
- Nosotros no veremos los metamodelos, sino la racionalidad de uso de `QVTRelation`

Introducción



Relaciones entre metamodelos QVT

Introducción

- La semántica del lenguaje Core (y por tanto del lenguaje Relations) permite los siguientes escenarios de ejecución:
  - Transformaciones de control para verificar que los modelos están relacionados de una determinada forma
  - Transformaciones en una única dirección
  - Transformaciones bidireccionales
  - La capacidad de establecer relaciones entre modelos pre-existentes

Introducción

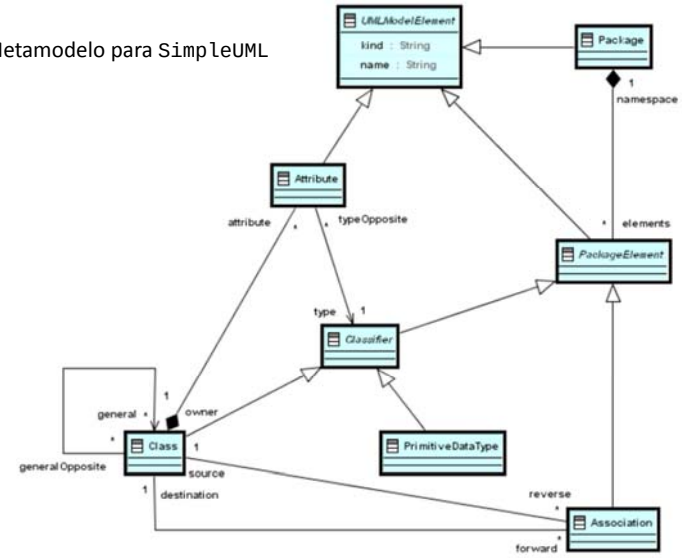
  - Actualizaciones incrementales (en cualquier dirección) cuando se cambia un modelo relacionado tras una ejecución inicial
  - La capacidad de crear y borrar objetos y valores, así como la capacidad de especificar que objetos y valores no deberían ser modificados
- Operational mappings y aproximaciones de caja negra solo permiten transformaciones en una única dirección

Ejemplo

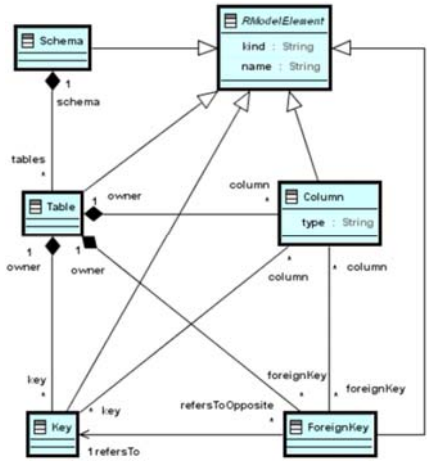
- A lo largo de este tema vamos a trabajar con dos metamodelos de ejemplo:
  - `SimpleUML`
  - `SimpleRDBMS`

Ejemplo

Metamodelo para `SimpleUML`



Ejemplo



Metamodelo para `SimpleRDBMS`

Transformaciones y tipos de modelos

- En el lenguaje relations, una *transformación* entre modelos candidatos se especifica como un conjunto de relaciones que deben cumplirse para que la transformación tenga éxito
- Un *modelo candidato* es un modelo que se ajusta a un tipo de modelo

Transformaciones y tipos de modelos

- Un *tipo de modelo* es una especificación de los tipos de elementos que puede tener un modelo
- Por ejemplo:

```
transformation umlRdbms(uml: SimpleUML,
rdbms: SimpleRDBMS)
```


Transformaciones y tipos de modelos

- La transformación puede invocarse para:
  - Comprobar la consistencia de los dos modelos
  - Modificar un modelo para hacer cumplir la consistencia

Relaciones y dominios

- Las *relaciones* en una transformación declaran restricciones que deben ser satisfechas por los elementos de los modelos candidatos
- Una relación, definida por uno o más dominios y un par de predicados `when` y `where`, especifica una relación que debe cumplirse entre los elementos de los modelos candidatos

Relaciones y dominios

- Un *dominio* es una variable tipada que puede ser emparejada con un modelo de un determinado tipo de modelo
- Un dominio tiene un *patrón*, que es un conjunto de restricciones que los elementos ligados a esas variables deben cumplir para ser una vinculación válida del patrón

Relaciones y dominios

- Así los patrones de dominio son plantillas para objetos y sus propiedades que deben ser localizados, modificados o creados en un modelo candidato para satisfacer la relación
- Por ejemplo:

```
relation PackageToSchema
{ domain uml p:Package { name= pn }
  domain rdbms s:Schema { name = pn }
}
```


Cláusulas `when` y `where`

- Una relación puede ser restringida por dos conjuntos de predicados, las cláusulas `when` y `where`
- Estás cláusulas pueden contener expresiones de QVT Relations o expresiones OCL
- La cláusula `when` especifica la condición bajo la cual debe cumplirse la relación

Cláusulas when y where

- La cláusula `where` especifica la condición que debe ser satisfecha por todos los elementos del modelo que participan en la relación
  - Puede restringir cualquier variable en la relación y en sus dominios

Cláusulas when y where

- Ejemplo:

```
relation ClassToTable {
 domain uml c:Class { namespace = p:Package
     {}, kind='Persistent', name=cn }
 domain rdbms t:Table { schema = s:Schema
     {}, name=cn, column = cl:Column {
     name=cn+'_tid', type='NUMBER'},
     primaryKey = k:PrimaryKey {
     name=cn+'_pk', column=cl} }
when  { PackageToSchema(p, s); }
where { AttributeToColumn(c, t); } }
```


Relaciones de nivel superior

- Una transformación contiene dos tipos de relaciones:
  - De nivel superior
  - Resto
- La ejecución de una transformación requiere que se cumplan todas sus relaciones de nivel superior
  - Se distingue con la palabra clave `top`

Relaciones de nivel superior

- El resto de relaciones sólo deben cumplirse cuando son invocadas directa o transitivamente desde la cláusula `where` de otra relación
- Ejemplo:

```
transformation umlRdbms (uml : SimpleUML,
rdbms : SimpleRDBMS)
{ top relation PackageToSchema {…}
  top relation ClassToTable {…}
  relation AttributeToColumn {…} }
```

`Checkonly` y `enforced`

- El hecho de que una relación se compruebe o se imponga está determinado por el dominio destino, que puede estar marcado como `checkonly` o `enforced`
- Cuando se ejecuta una transformación en la dirección de un dominio `checkonly` se comprueba si hay un emparejamiento válido en el modelo que satisface la relación

`Checkonly` y `enforced`

- Cuando se ejecuta una transformación en la dirección de un modelo de un dominio `enforced`, si la comprobación falla, el modelo destino se modifica para satisfacer la relación
- Ejemplo:

```
relation PackageToSchema
{ checkonly domain uml p:Package {name=pn}
  enforce domain rdbms s:Schema {name=pn} }
```


Emparejamiento de patrones

- Analices las *expresiones de plantillas de objetos*, que emparejan patrones en modelos candidatos
- Por ejemplo, dentro del dominio `uml`:

```
c:Class { namespace = p:Package {},
          kind='Persistent',
          name=cn }
```


Emparejamiento de patrones

- Una expresión plantilla empareja resultados en una vinculación de elementos del modelo candidato a variables declaradas por el dominio
- Es posible que algunas variables del dominio ya estén vinculadas a elementos del modelo (p.e. como resultado de la evaluación de una clásusula `where` o de otras expresiones plantilla)

Emparejamiento de patrones

- En este caso, los emparejamientos de las expresiones plantilla vinculan solamente a las variables libres del dominio
- En el ejemplo, el emparejamiento de patrones vinculará las variables c, p y cn
  - Probablemente la variable p ya esté vinculada por la evaluación de la cláusula `when PackageToSchema(p, s)`

Emparejamiento de patrones

  - El proceso de emparejamiento filtrará los objetos de tipo `Class` en el modelo `uml`, eliminando aquellas que no cumplan los valores literales para las propiedades de la expresión plantilla (`kind='Persistent'`)
- Para las propiedades que se comparan con variables (name=cn):
  - Si cn ya está vinculada con un valor, se eliminan la clases que no tengan dicho valor para la propiedad

Emparejamiento de patrones

  - Si cn está libre, se vinculará al valor de la propiedad nombre para todas las clases que no son eliminadas en el filtrado de otras propiedades. El valor cn:
    - Se utilizará en otro dominio
    - Puede tener otras restricciones especificadas en la expresión `where` del dominio o en su propia relación

Emparejamiento de patrones

- Después el emparejamiento continua con las propiedades cuyos valores se comparan con las expresiones anidadas
  - Así, el patrón `namespace=p:Package {}` emparejará solo aquellas clases cuyo propiedad namespace tenga una referencia no nula a un `Package`
  - Al mismo tiempo, la variable p se vinculará para referirse al `Package`
  - Sin embargo en el ejemplo, p ya está vinculada en la cláusula `when`

Emparejamiento de patrones

- Se permite un anidamiento arbitrario de expresiones plantilla
  - El emparejamiento y vinculación de variable procederá recursivamente hasta que haya un conjunto de valores de tuplas correspondientes con las variables del dominio y sus expresiones plantillas
  - En nuestro ejemplo la tupla viene definida por las variables (c, p, cn)

Claves y creación de objetos

- Como ya hemos comentado una *expresión de plantilla de objeto* también sirve como una plantilla para crear un objeto en un modelo destino
- Así, si no hay elementos en el modelo destino que cumplan la plantilla, se crean

Claves y creación de objetos

- Ejemplo:

```
t:Table {
  schema = s:Schema {},
  name = cn,
  column = cl:Column {name=cn+'_tid',
                      type='NUMBER'},
  primaryKey = k:PrimaryKey {name=cn+'_pk',
                             column=cl}
 }
```


Claves y creación de objetos

- Si queremos evitar duplicados a la hora de crear objetos debemos marcar estos objetos con la palabra key, indicando sus propiedades claves:

```
key Table {schema, name}
```

Restricciones sobre expresiones

- Para garantizar la existencia de un algoritmo de vinculación las expresiones en una relación deben cumplir:
  - Debe ser posible organizar las expresiones que aparecen en la cláusula `where`, los dominios origen y la cláusula `where` en un orden squencial que contiene solo alguna de las siguientes expresiones

Restricciones sobre expresiones

- `objeto.propiedad= variable`

donde `objeto` es una variable ligada y `variable` estaba libre, ya que esta expresión la vincula

- `objeto.propiedad= expresión`

donde `objeto` es una variable ligada y `expresión` no contiene variables libres

- No hay otras expresiones con variables libres, ya que se han tenido que vincular en expresiones precedentes

Restricciones sobre expresiones

- Debería ser posible organizar las expresiones en el dominio origen en un orden secuencial que contenga cualquier expresión del tipo:
  - `objeto.propiedad= expresión`

  donde `objeto` es una variable ligada y y `expresión` no contiene variables libres
  - No hay otras expresiones con variables libres, ya que se han tenido que vincular en expresiones precedentes

Propagación del cambio

- En QVT Relations, el efecto de propagar un cambio del modelo origen al modelo destino es semánticamente equivalente a ejecutar toda la transformación de nuevo

Transformaciones internas

- Una transformación se puede considerar interna cuando los modelos candidato y destino se vinculan al mismo modelo en ejecución

Integración de op. de caja negra

- Una relación podría tener asociada una operación de caja negra para hacer cumplir una relación en un dominio
- Dicha operación se invoca cuando la relación se ejecuta y la relación se evalúa a falsa
- La operación invocada es responsable de hacer los cambios necesarios en el modelo para que se cumpla la relación

Integración de op. de caja negra

- Si la relación se evalúa a falsa después de la ejecución de la operación, se produce una excepción de ejecución
- La signatura de la operación se puede derivar de la especificación de dominio de la relación:
  - Un parámetro de salida para el dominio sobre el que se impone la relación
  - Un parámetro de entrada para el resto de dominnios

Integración de op. de caja negra

- Las relaciones que pueden ser implementadas por operaciones de operational mappings y operaciones de caja negra están restringidas de la siguiente forma:
  - Sus dominios deberían ser primitivos o contener una plantilla de objetos simple (sin sub-elementos)
  - Las cláusulas `when` y `where` no deberían definir variables

Integración de op. de caja negra

- Estas restricciones permiten una semántica simple de llamada, que no necesita evaluar ninguna restricción y restringe la comprobación de restricciones después de la invocación de la operación

Transformación `Checkonly`

- Una transformación se puede ejecutar en modo `Checkonly`
- En este modo la transformación sólo comprueba si las relaciones se cumplen en todas las direcciones e informa en caso contrario
- No se fuerza nada, con independencia de que haya dominios marcados como `checkonly` o `enforced`

Notación visual

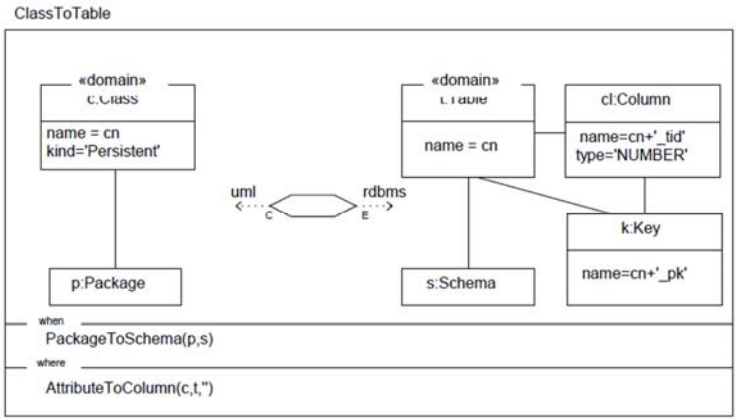
- El estándar define una notación visual para representar las relaciones
- Ejemplo:

Notación visual

```
top relation ClassToTable   // map each persistent class to a table
{
   cn, prefix: String;

   checkonly domain uml c:Class {namespace=p:Package {},
                                 kind='Persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
          column=cl:Column {name=cn+'_tid', type='NUMBER'},
          key=k:Key {name=cn+'_pk', column=cl}};
   when {
      PackageToSchema(p, s);
   }
   where {
            prefix = '';
            AttributeToColumn(c, t, prefix);
         }
      }
```

Notación visual



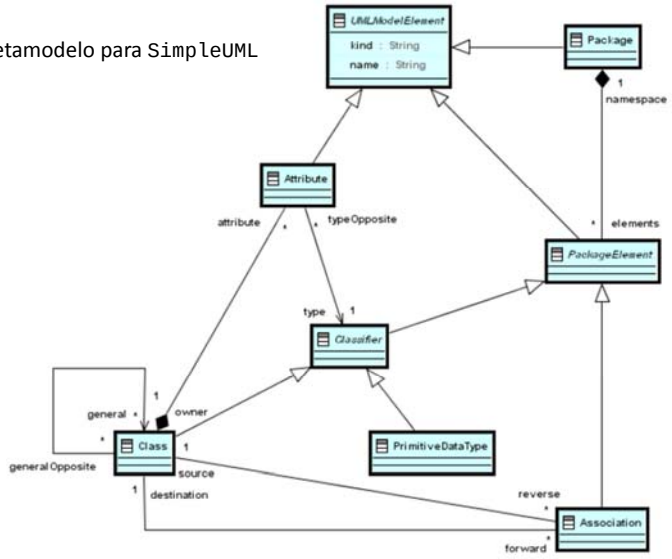
Representación visual de la relación anterior

Ejemplo

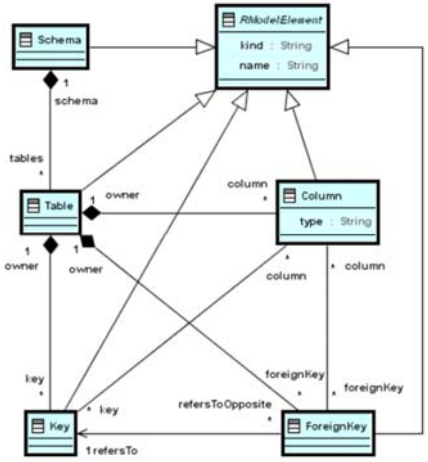
- Veamos el ejemplo que transforma modelos de `simpleUML` a `simpleRDBMS`

Ejemplo

Metamodelo para `SimpleUML`



Ejemplo



Metamodelo para `SimpleRDBMS`

Ejemplo

```
transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)
{
key Table (name, schema);
key Column (name, owner);   // owner:Table opposite column:Column
key Key (name, owner); // key of class ëKeyí;
                       // owner:Table opposite key:Key

top relation PackageToSchema   // map each package to a schema
{
   pn: String;

   checkonly domain uml p:Package {name=pn};
   enforce domain rdbms s:Schema {name=pn};
}
```


Ejemplo

```
top relation ClassToTable   // map each persistent class to a table
{
   cn, prefix: String;

   checkonly domain uml c:Class {namespace=p:Package {},
                                 kind='Persistent', name=cn};
   enforce domain rdbms t:Table {schema=s:Schema {}, name=cn,
          column=cl:Column {name=cn+'_tid', type='NUMBER'},
          key=k:Key {name=cn+'_pk', column=cl}};
   when {
      PackageToSchema(p, s);
   }
   where {
            prefix = '';
            AttributeToColumn(c, t, prefix);
         }
      }
```


Ejemplo

```
relation AttributeToColumn
{
   checkonly domain uml c:Class {};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      PrimitiveAttributeToColumn(c, t, prefix);
      ComplexAttributeToColumn(c, t, prefix);
      SuperAttributeToColumn(c, t, prefix);
   }
}
```


Ejemplo

```
relation PrimitiveAttributeToColumn
{
   an, pn, cn, sqltype: String;

   checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                 type=p:PrimitiveDataType {name=pn}}};
   enforce domain rdbms t:Table {column=cl:Column {name=cn,
                                 type=sqltype}};
   primitive domain prefix:String;
   where {
      cn = if (prefix = '') then an else prefix+'_'+an endif;
      sqltype = PrimitiveTypeToSqlType(pn);
   }
}
```

Ejemplo

```
relation ComplexAttributeToColumn
{
   an, newPrefix: String;

   checkonly domain uml c:Class {attribute=a:Attribute {name=an,
                                 type=tc:Class {}}};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      newPrefix = prefix+'_'+an;
      AttributeToColumn(tc, t, newPrefix);
   }
}
```


Ejemplo

```
relation SuperAttributeToColumn
{
   checkonly domain uml c:Class {general=sc:Class {}};
   enforce domain rdbms t:Table {};
   primitive domain prefix:String;
   where {
      AttributeToColumn(sc, t, prefix);
   }
}
```



```
// map each association between persistent classes to a foreign key
top relation AssocToFKey
{
   srcTbl, destTbl: Table;
   pKey: Key;
   an, scn, dcn, fkn, fcn: String;

   checkonly domain uml a:Association {namespace=p:Package {},
            name=an,
            source=sc:Class {kind='Persistent',name=scn},
            destination=dc:Class {kind='Persistent',name=dcn}
         };
   enforce domain rdbms fk:ForeignKey {schema=s:Schema {},
            name=fkn,
            owner=srcTbl,
            column=fc:Column {name=fcn,type='NUMBER',owner=srcTbl},
            refersTo=pKey
         };
   when {   /* when refers to pre-condition */
      PackageToSchema(p, s);
      ClassToTable(sc, srcTbl);
      ClassToTable(dc, destTbl);
      pKey = destTbl.key;
   }
   where {
      fkn=scn+'_'+an+'_'+dcn;
      fcn=fkn+'_tid';
   }
}
```


Ejemplo

```
function PrimitiveTypeToSqlType(primitiveTpe:String):String
{
   if (primitiveType='INTEGER')
   then 'NUMBER'
   else if (primitiveType='BOOLEAN')
      then 'BOOLEAN'
      else 'VARCHAR'
      endif
   endif;
}

}
```

3. QVT Relations

Antonio Navarro Martín
Profesor Titular de Universidad
Dpto. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
anavarro@fdi.ucm.es