# Patrones de desarrollo con QVTR

En este documento vamos a analizar diferentes patrones de desarrollo de transformaciones QVTR.

## Trabajo con diferentes metamodelos

Trabajar con diferentes metamodelos puede ser complicado cuando se requiere referenciar operaciones o atributos de éstos. Como ejemplo, veamos cómo podemos referenciar a la operación not() perteneciente a la librería estándar de OCL.

### Ejemplo 1:

Con OCL 2.3 la cosa no es tan complicada, ya que se puede obtener una referencia a la clase de un tipo, y una vez obtenida la clase, encontrar su operación.

### Ejemplo 2:

Con OCL 2.0 esto no es posible, y es necesario acceder al paquete stdlib como modelo de entrada, por ejemplo:

**checkonly** **domain** mm oclstdlib:ecore::EClass {

name='Boolean\_Class'

, eOperations=notOp:ecore::EOperation{

name='not'

}

};

Esto genera el problema de muchas posibles combinaciones de instancias de tipo class para localizar el patrón que encuentre la clase en concreto. Para solucionar este problema, es mejor localizar su paquete, en vez de su clase, ya que habrá menos paquetes que clases normalmente.

**checkonly** **domain** mm oclstdlib:ecore::EPackage{

,name='oclstdlib'

,eClassifiers=notOpClass: ecore::EClass {

,name='Boolean\_Class'

, eOperations=notOp:ecore::EOperation{

name='not'

}

}

};

### Ejemplo 3:

Pongamos como ejemplo una transformación de un modelo a un modelo de instancias de sus metaclases. Para ello es necesario disponer de el metamodelo en UML, ya que un modelo de instancias trabaja con datos de un modelo UML.

Si queremos referenciar el atributo correspondiente a un elemento del modelo, debemos encontrar su correspondiente en el modelo UML de la metaclase.

**checkonly** **domain** mm uml241::Class {

name='Class'

, ownedAttribute = a : um241::Property{

Name=’name’

}

};

Esto otra vez genera una infinidad de combinaciones en la búsqueda de patrones. Para optimizar esta búsqueda, se puede localizar directamente el modelo y así navegar por él con sentencias OCL.

**checkonly** **domain** source \_package:uml241::Model{

name='UML'

};

**where**{

f=\_package.packagedElement-> **select**(c|c.**oclIsTypeOf**(uml241::Class) **and** c.**oclAsType**(uml241::Class).name='Class').at(1).**oclAsType**(uml241::StructuredClassifier).ownedAttribute-> **select**(a|a.name='name').at(1) .**oclAsType**(uml241::Property);

}

### Resumen:

Tal como hemos visto, si no disponemos de OCL 2.3, conviene localizar al elemento raíz en vez del nodo en concreto, para no provocar un número desproporcionado de combinaciones en la búsqueda de patrones de los elementos que necesitamos referenciar.

## Propagación de dependencias

## Ejempo 1 – Elementos accesibles:

En muchos casos, necesitamos crear el elemento raíz antes de poder invocar una relación top, para ello se suele crear una relación de transformación de nodos raíz entre los diferentes metamodelos a transformar.

**top** **relation** Package2Transformation {

n:String;

prefix:String;

**checkonly** **domain** mm ePackage: ecore::EPackage { name = n,nsPrefix=prefix};

**enforce** **domain** qvtr t:qvtrelation::RelationalTransformation {

name = 'Copy' + n,

modelParameter = sourceMM: qvtbase::TypedModel {

name = 'source',

usedPackage = uPackage:ecore::EPackage{ }

,nsPrefix=prefix

},

modelParameter = targetMM: qvtbase::TypedModel {

name = 'target',

usedPackage = uPackage: ecore::EPackage{ }

,nsPrefix=prefix

}

};

**when** {

//indica que es el paquete raíz

ePackage.eSuperPackage.**oclIsUndefined**() **and**

//Al depender de diferentes meta-modelos, necesitamos descartar aquellos meta-modelos que pasamos como entrada, para localizar el meta-modelo de nombre variable que vamos a transformar

ePackage.name<>'ocl' **and** ePackage.name<>'oclstdlib' **and** ePackage.name<>'collections' **and** ePackage.name<>'tuples';

}

**where** {

uPackage = ePackage;

MarkTypedModel(sourceMM, targetMM);

MarkTransformation(t);

}

}

Para poder invocar una relación por búsqueda de patrones sobre todo el modelo, esta relación B (TOP) deberá invocar a la relación A que crea el nodo raíz del nuevo modelo generado, y por tanto las dependencias de A se pueden propagar a la relación B.

**when** {

Package2Transformation(ePackage,transfo);

}

Cuando se puede acceder desde los elementos del dominio de la relación B a los elementos del dominio de la relación A, la dependencia de dominios no se propaga.

**query** rootPackage(ePackage: ecore::EPackage): ecore::EPackage{

**if**(**not** ePackage.eSuperPackage.**oclIsUndefined**()) **then**

rootPackage(ePackage.eSuperPackage.**oclAsType**(ecore::EPackage))

**else**

ePackage

**endif**

}

**top** **relation** Class2MarkerRelation {

sourceMM, targetMM: qvtbase::TypedModel;

**checkonly** **domain** mm eClass: ecore::EClass {

ePackage = ePackage: ecore::EPackage {},

name = n : String{}

};

**enforce** **domain** qvtr rel: Relation {

…

};

**when** {

Package2Transformation(rootPackage(ePackage),transfo);

}

}

## Ejemplo2 – Elementos no accesibles:

Cuando necesitamos invocar una precondición o postcondición, y éstas necesitan de elementos que no pueden localizar desde los elementos del dominio, éstas dependencias se propagan a todas las relaciones clientes de la dependencia.

**top** **relation** Class2CopyNotSubclassRelation {

**checkonly** **domain** mm oclstdlib:ecore::EPackage{

name='oclstdlib'

};

…

**enforce** **domain** qvtr rel:qvtrelation::Relation {…};

…

**when** {

…

Class2CopyRelation(oclstdlib,eSuperClass,rel);

}

**where** {

…

SubClass2MarkerCallInWhen( oclstdlib, subClass, eSuperClass, rel.variable->at(1), rel.variable-> at(2), rel.\_when);

}

}

**relation** SubClass2MarkerCallInWhen {

nott:ecore::EOperation;

**checkonly** **domain** mm oclstdlib:ecore::EPackage{

name='oclstdlib'

,eClassifiers=notOpClass: ecore::EClass {

name='Boolean\_Class'

, eOperations=notOp:ecore::EOperation{

name='not'

}

}

};

…

**enforce** **domain** qvtr whenPattern: qvtbase::Pattern {

predicate = pred: qvtbase::Predicate {

conditionExpression = notCall: ocl::ecore::OperationCallExp {

,referredOperation=notOp.**oclAsType**(ecore::EObject)

,argument=markerCall:qvtrelation::RelationCallExp{

}

}

}

};

}

Como vemos en este ejemplo, tenemos 3 relaciones

A : Top Class2CopyRelation

B : Top Class2CopyNotSubclassRelation

C : SubClass2MarkerCallInWhen

Las relaciones A y C dependen de oclstdlib por algún motivo, y la relación B no lo utiliza. En este caso, las dependencias de A y C se propagan a las relaciones TOP que las invocan, en el ejemplo, B.

### Resumen:

Para realizar transformaciones, primero escribimos la relación en sí, y luego propagamos las dependencias.

## Copia de modelos

QVTR no facilita un mecanismo de copia de elementos, es por ello necesario escribir todas las relaciones de ccopia necesarias que contemplen la posibilidad de que un atributo o referencia aparezcan o no en el modelo. A parte de esto, debemos fijarnos en las características de Ecore que nos interesan: (http://techblog.goelite.org/what-you-need-to-know-about-emf-parameters/)

-Una clase tiene atributos y referencias.

-Los atributos y referencias tienen una cardinalidad mínima y máxima.

-Los atributos y referencias pueden ser ordenados, únicos y derivados

-Una clase puede heredar de otra

-Una clase puede ser abstracta y por lo tanto no se pueden crear instancias de ella.

-Una clase puede ser una interfaz, y por tanto no se pueden crear instancias de ella.

-Un atributo o referencia de una clase abstracta o interfaz se propaga a las subclases no interfaces ni abstractas.

-Un atributo o referencia derivad@ no se copia porque no se serializan.

Ahora veamos qué cómo hacer una relación de copia para modelos.

Respecto del meta-modelo:

\*Para cada clase o interfaz, creamos una relación de marcado.

\*Para cada clase no abstracta o interfaz, y sin clases que hereden de ella, creamos una relación de copia, que llame como post-condición a la relación de marcado.

\*Para cada superclase no abstracta o superinterfaz, creamos una relación de copia, que llame como post-condición a la relación de marcado, y como precondición, que no se cumplan las relaciones de copia de las subclases.

\*Para cada atributo no derivado de una clase no abstracta o interfaz creamos una relación de copia que se ejecutará una vez se haya marcado la clase a la que pertenece.

\*Para cada referencia no derivada de una clase no abstracta creamos una relación de copia que se ejecutará una vez se haya marcado la clase a la que pertenece.

\*Para cada atributo no derivado de una superclase abstracta o interfaz (siguiendo la generalización recursivamente hacia arriba, las que sean directamente abstractas o interfaces) creamos una relación de copia que se ejecutará una vez se haya marcado la clase a la que pertenece.

\*Para cada referencia no derivada de una superclase abstracta o interfaz (siguiendo la generalización recursivamente hacia arriba, las que sean directamente abstractas o interfaces) creamos una relación de copia que se ejecutará una vez se haya marcado la clase a la que pertenece.

## Refinamiento de modelos

Para poder realizar el refinamiento de un modelo M, alterando sólo las partes que nos interesan, debemos mezclar relaciones de copia RC con relaciones de refinamiento RF. Veamos cómo:

Generamos las relaciones A de copia según lo explicado en el apartado correspondiente.

Generamos las relaciones B de modificación según lo deseado.

\*Por cada relación top de B, y por cada meta-clase de los dominios patrones de B, añadimos a su relación de copia A la precondición de que no se ejecute cuando se cumpla B. Las meta-clases de los dominios patrones de B pueden ser tanto clases, atributos, relaciones, etc.

No llamamos a las relaciones de marcado, ya que podrían disparar otras las relaciones que no nos interesan.

\*Para cada elemento del punto anterior, también modificamos las relaciones de copia de las superclases no abstractas, para que no se ejecuten cuando se cumple la relación B.

## Tejido automático de refinamientos

Debido a la complejidad de ciertos meta-modelos, es necesario realizar transformaciones meta-modelo a transformación de copia.

Debido a la complejidad de ciertas transformaciones, es necesario que las transformaciones de refinamiento se simplifiquen, y sea otra transformación la que haga el tejido entre la transformación de refinamiento y la transformación de copia.

Para ello es necesario disponer del modelo (AST) de la transformación de refinamiento, lo cual hace necesario disponer de un analizador de QVT.

## Obtención del AST

Cada lenguaje tiene un modelo AST concreto, obtenido a partir de un modelo de sintaxis concreta del lenguaje. Es necesario realizar una relación de transformación del CST al AST.

## Obtención del CST

Cada lenguaje suele ser descrito mediante una sintaxis BNF o alguna variación de ésta. Los parsers son herramientas que a partir de una descripción del BNF del lenguaje, y determinadas condiciones, obtienen su modelo CST correspondiente.

El BNF suele ser un meta-modelo que incluye además de los elementos del meta-modelo del lenguaje, cadenas terminales y otros tokens.

## Manda huevos

Si ejecutamos esto en una top relation, se invoca con cst\_models1=cst\_models2 también, así que es necesario distinuirlos en una cláusula when

**checkonly** **domain** cst tAS:ast::qvt::TransformationAS{

,name=name

//,\_extends=\_extends

//,relations=cst\_relations:ast::qvt::RelationAS{},

,modelDeclarations = cst\_models1:ast::qvt::ModelDeclarationAS{

modelId=modelId1

}

,modelDeclarations = cst\_models2:ast::qvt::ModelDeclarationAS{

modelId=modelId2

}

};

**when**{

cst\_models1<>cst\_models2;

}