MATERIAL ADICIONAL DEL CAPÍTULO 12

EL LENGUAJE QVT: CASO DE ESTUDIO

Antonia Mª Reina Quintero

INTRODUCCIÓN

Para ilustrar el uso de *QVT Operational Mappings*, se ha tomado el caso de estudio denominado "*Model Migration*" propuesto en el taller *Transformation Tool Contest de 2010*[1]. El caso de estudio tiene como objetivo explorar cómo los lenguajes de transformación de modelos se pueden usar para automatizar la evolución de los modelos producida por cambios en el metamodelo. Para ello, se escogen los diagramas de actividad UML, ya que el metamodelo de diagramas de actividad sufrió cambios significativos entre las versiones UML1.4 (OMG, 2001) y UML2.2 (OMG, 2009). La descripción del caso de estudio original se puede encontrar en (Rose, Kolovos, Paige, & Polack, 2010).

DESCRIPCIÓN DEL CASO DE ESTUDIO

El problema que queremos resolver se puede formular así: dado un diagrama de actividad que se ajusta al estándar UML1.4, queremos obtener un diagrama equivalente en UML2.2. Para resolver el problema, lo primero que necesitamos son los metamodelos de los diagramas de actividad UML1.4 y UML2.2. En el caso de estudio propuesto en (Rose, Kolovos, Paige, & Polack, 2010) se usa un metamodelo mínimo de UML 1.4 como metamodelo origen de la

[1] http://planet-research20.org/ttc2010/index.php?Itemid=132

transformación y el metamodelo de UML 2.2 proporcionado por el proyecto de herramientas Eclipse UML 2 (Eclipse MDT / UML2 Project, 2012). La Figura 1 muestra el metamodelo mínimo de los diagramas de actividad UML1.4, mientras que la Figura 2 muestra el metamodelo simplificado de los diagramas de actividad UML 2.2. Los modelos resultados de nuestra transformación tendrán conformidad con este metamodelo.

Antes de acometer la implementación de la propia transformación, merece la pena detenerse en entender cuáles son las diferencias existentes entre los diagramas de actividad en las versiones 1.4 y 2.2 de UML. Para ellos, a continuación se van a detallar las principales relaciones, entre los elementos de los diagramas de actividad de UML 1.4 y los de UML 2.2. Estas relaciones pueden verse como operaciones de migración, y son:

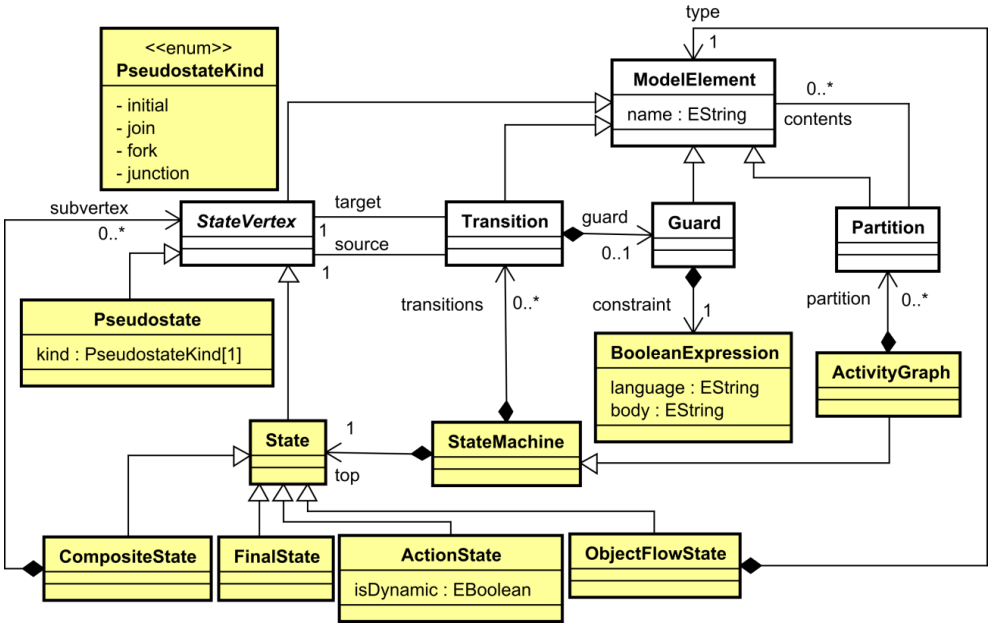

Figura 1. *Metamodelo simplificado de los diagramas de actividad UML 1.4.*

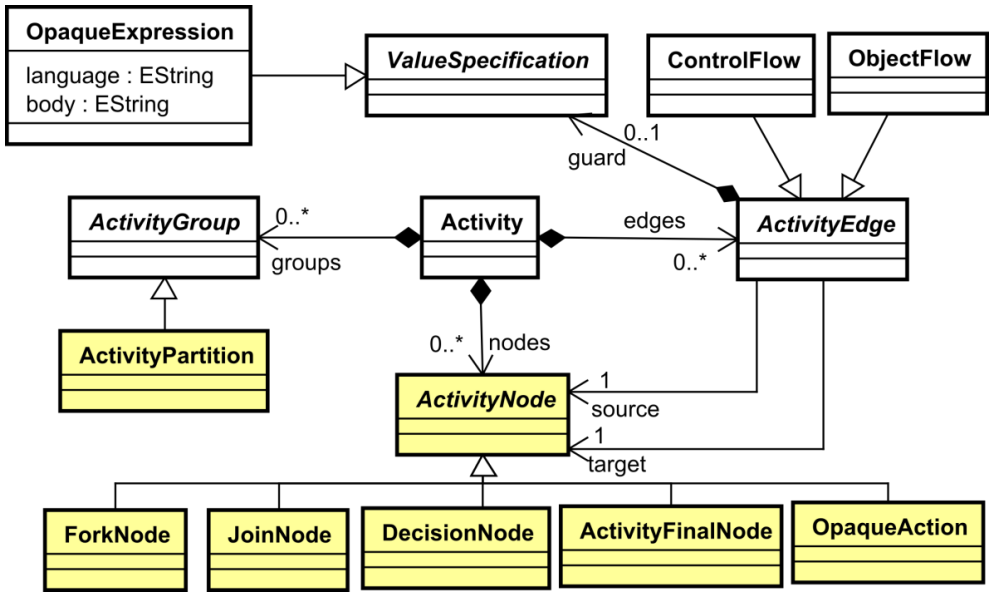

Figura 2. *Metamodelo simplificado de diagrama de actividad UML 2.2*

- (MIG-1) La jerarquía de objetos `ActivityGraph`, `StateMachine` y `CompositeState` de UML 1.4 se convierte en `Activity` en UML 2.2.

- (MIG-2) Los objetos de tipo `Transition` de UML 1.4 pasan a ser objetos de tipo `ActivityEdge` en UML 2.2. Sin embargo, `ActityEdge` es una clase abstracta con dos subtipos: `ControlFlow` y `ObjectFlow`. Si el objeto de tipo `Transition` tiene como origen o como destino un objeto de tipo `ObjectFlowState`, entonces migrará a un objeto de tipo `ControlFlow` en UML 2.2, si no, migrará a un objeto de tipo `ObjectFlow`.

- (MIG-3) Los objetos de tipo `StateVertex` y `State` pasan a ser de tipo `ActivityNode`. Es decir, estas dos clases se mezclan en una.

- (MIG-4) Los objetos de tipo `Pseudostate` pueden migrar a objetos de tipo `InitialNode`, `JoinNode`, `ForkNode` o `DecisionNode`, dependiendo del valor de la propiedad `kind`. Si `kind` tiene el valor `initial`, se convierte en `InitialNode`; si tiene el valor `join`, en `JoinNode`; si tiene el valor `fork`, en `ForkNode`; finalmente, si tiene el valor `junction`, se convierte en `DecisionNode`.

- (MIG-5) Los objetos de tipo `FinalState` migran a objetos de tipo `ActivityFinalNode`.

- (MIG-6) Los objetos de tipo `ActionState` migran a objetos de tipo `OpaqueAction`.

- (MIG-7) Los objetos de tipo `Partition` se convierten en objetos de tipo `ActivityPartition`, subclase de `ActivityGroup`.

- (MIG-8) Los objetos de tipo `Guard` junto con el objeto de tipo `BooleanExpression` que contiene se convierten en un objeto de tipo `OpaqueExpression`.

TRANSFORMACIÓN

En esta sección se explican los detalles de cómo se ha abordado la migración de los modelos de diagramas de actividad UML 1.4 a UML 2.2 presentados en la sección anterior. La transformación completa se puede ver en el Apéndice y se ha implementado usando la implementación de *QVT Operational Mapping* proporcionada por el proyecto de modelado de Eclipse (Eclipse M2M Project). Algunas de las cuestiones en las que merece la pena fijar la atención son las siguientes:

DECLARACIÓN DE LA TRANSFORMACIÓN.

Tal y como se ha explicado en el capítulo 12 del libro, en la declaración de la transformación se escriben las declaraciones `modeltype` para especificar los tipos de los modelos a los que afecta la transformación (Figura 3). Es necesario resaltar que estos metamodelos tienen que estar registrados en el entorno. La Figura 4 muestra la ventana de Eclipse M2M QVTo que se usa para registrar los metamodelos para que estén disponibles para la transformación. A esta ventana se puede acceder a través de las propiedades del proyecto.

```
001 modeltype UML14 uses minuml1('minuml1');
002 modeltype UML2  uses 'http://www.eclipse.org/uml2/3.0.0/UML';

003 transformation ActDiagUML142UML2(in inModel:UML14, out
outModel:UML2);
```

Figura 3. *Declaración de la transformación ActDiagUML142UML2.*

Finalmente, en la declaración de la transformación también se especifica, mediante la declaración `transformation`, el nombre de la misma, y los tipos y

dirección de los modelos. La transformación `ActDiagUML142UML2` tiene un modelo de entrada de tipo `UML14` y produce un modelo de salida de tipo `UML2`.

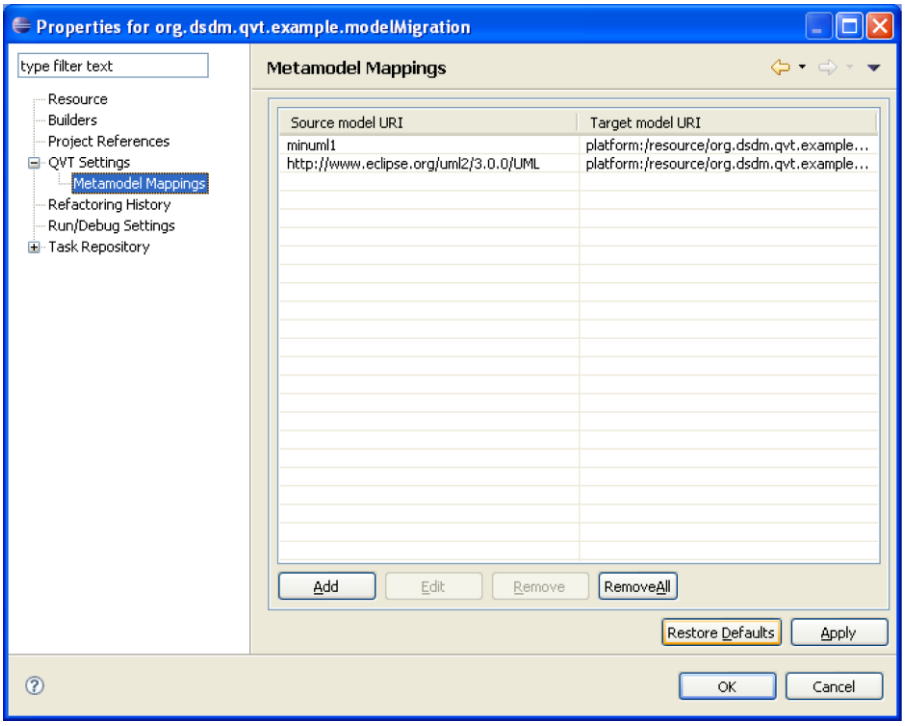

Figura 4. *Registro de los metamodelos usados en la transformación.*

OPERACIÓN MAIN.

La operación `main` es la que se ejecuta primero en la transformación. En este caso, la operación main crea dos objetos en el modelo destino, uno de tipo `Activity`, y otro de tipo `Package`. Los objetos se crean de forma implícita invocando a las operaciones de *mapping* `toActivity` y `toPackage`. Ambas se ejecutan teniendo como contexto a un objeto de tipo `ActiviyGraph`. La operación `rootObjects()` es una operación disponible para todos los modelos, que devuelve una lista de los objetos situados en la raíz del modelo. En el caso de los modelos Ecore, normalmente solo hay un objeto que es la raíz del mismo.

```
004 -- Desencadenante de la transformación
005 main () {

006     inModel.rootObjects ()[minuml1::ActivityGraph].map toActivity();
```

```
007     inModel.rootObjects ()[minuml1::ActivityGraph].map toPackage();
008 }
```

Figura 5. *Operación* `main()`.

OPERACIONES DE MAPPING RELACIONADAS CON LAS MIGRACIONES A REALIZAR.

En esta sección se explican las operaciones de *mapping* implementadas para llevar a cabo las migraciones detalladas en la descripción del caso de estudio.

- (MIG-1) Esta migración se ha implementado con dos operaciones de *mapping*, una para generar un objeto `Activity` a partir de un objeto de tipo `ActivityGraph` (Figura 6), y otra para generar un objeto de tipo `Activity` a partir de un objeto de tipo `CompositeState` (Figura 7). Note que el nombre de estas dos operaciones es el mismo (`toActivity`), y lo que las diferencia es el contexto, que en el primer caso es `UML14::ActivityGraph`, mientras que en el segundo es `UML14::CompositeState`.

```
009 -- Migrar ActivityGraph a Activity
010 mapping UML14::ActivityGraph::toActivity():UML2::Activity{
011      name:= self.name;
012      node+= self.top.oclAsType(
013                  UML14::CompositeState).getSubvertexes().map
toActivityNode();
014      edge+= self.transitions.map toActivityEdge()->asSequence();
015      group+= self.partition.map toActivityPartition();
016 }
```

Figura 6. *Operación de mapping* *`UML14::ActivityGraph::toActivity()`*.

En la migración de `ActivityGraph` a `ActivityGraph` (Figura 6), en primer lugar se copia el nombre de objeto origen de la operación de *mapping*, al que nos referimos como `self`, en el nombre del objeto creado implícitamente en el *mapping* (línea 011). Note que el operador de asignación es `:=`.

En segundo lugar (línea 012), se accede a los subvértices del objeto de tipo `CompositeState` que hace de contenedor de los objetos del diagrama de actividad, y cada subvértice se transforma mediante la invocación de una operación de *mapping* a su correspondiente objeto de tipo `ActivityNode`. En este caso, la propiedad `node` del objeto resultado de la transformación es una colección, por lo que el operador usado, es `+=` en lugar de `:=`. Note también que la operación `getSubvertexes()` es una operación de *query*

que se ha implementado en la transformación para mejorar la legibilidad. Para ver el código de esta operación de *query* acuda al Apéndice (líneas 037-041).

Finalmente, los objetos de tipo `Transition` se convierten en `ActivityEdge` invocando a la operación de *mapping* `toActivityEdge` (línea 014) y los de tipo `Partition` en `ActivityPartition`, mediante la operación `toActivityPartition` (línea 015). En estas líneas solo cabe destacar la invocación a la operación OCL `asSequence`, que convierte la colección resultado de las invocaciones de la operación de *mapping* sobre los distintos objetos de tipo `Transition` a una secuencia. Esto es necesario porque la propiedad `edge` es de tipo `Sequence`.

```
132 -- Migrar CompositeState (cuando no es el top)
133 mapping UML14::CompositeState::toActivity():UML2::Activity
134     when{not self.isTopComposite()}
135 {
136     name:= self.name;
137     node:= self.subvertex->map toActivityNode()->asSet();
138 }
```

Figura 7. Operación de *mapping* *UML14::CompositeState::toActivity()*.

La operación de mapping de `CompositeState` a `Activity` (Figura 7) solamente tendrá lugar cuando el objeto de tipo `CompositeState` no sea el *top*, es decir, no sea el contenedor de los objetos del diagrama de actividad. Esto se consigue mediante la claúsula `when` (línea 133), que se apoya en la operación de query `isTopComposite ()` para comprobar si el objeto de tipo `CompositeState` es el contenedor principal. Esta operación se puede ver en el Apéndice (líneas 139-145). Del resto de la operación solamente merece la pena destacar la necesidad de convertir a un conjunto el resultado de la invocación de `toActivityNode` sobre cada uno de los subvértices del estado compuesto (línea 137). Esto se hace mediante la operación OCL `asSet`.

- (MIG-2) Esta migración se ha resuelto mediante tres operaciones de *mapping*: una operación con un claúsula `disjuncts`, que permita invocar a una operación de *mapping* u otra, y sendas operaciones de *mapping* con una claúsula `when` que permita indicar las condiciones en las que se debe invocar dicha transformación. En la Figura 8 la operación de *mapping* `toActivityEdge` es la que contiene la claúsula `disjuncts`, e invocará bien a la operación de *mapping* `toControlFlow`, bien a la operación `toObjectFlow`.

```
082 -- Migración de Transition a ActivityEdge
083 mapping UML14::Transition::toActivityEdge():UML2::ActivityEdge
084 disjuncts  UML14::Transition::toControlFlow,
UML14::Transition::toObjectFlow {
085 }

086 -- Migración de Transition a ControlFlow
087 mapping UML14::Transition::toControlFlow():UML2::ControlFlow
088     when{not self.target.oclIsTypeOf (UML14::ObjectFlowState) and
089          not self.source.oclIsTypeOf (UML14::ObjectFlowState)}
090 {
091     name := self.name;
092     source:= self.source.resolveone (UML2::ActivityNode);
093     target:= self.target.resolveone (UML2::ActivityNode);
094     guard := self.guard.map toOpaqueExpression();
095 }

096 -- Migración de Transition a ObjectFlow
097 mapping UML14::Transition::toObjectFlow():UML2::ObjectFlow
098     when{self.isSurroundingObjectFlow()}
098 {
100     source:= self.source.resolveone (UML2::ActivityNode);
101     target:= self.getTargetSurroundingObjectFlow().resolveone
(UML2::ActivityNode);
102     name := result.source.name + '2'+ result.target.name;
103    guard := self.guard.map toOpaqueExpression();
104 }
```

Figura 8. *Operaciones de mapping para migrar un objeto* `Transition`

En estas la Figura 8 también merece la pena destacar el uso de la operación `resolveone` (líneas 092, 093, 100, 101), que permite acceder a los objetos de tipo `ActivityNode` que se han creado en operaciones de *mapping* anteriores. Es decir, los nodos `source` y `target` son objetos que se han creado anteriormente en invocaciones implícitas a operaciones de *mapping*. Para acceder a los mismos QVTo usa la información que tiene almacenada en la traza. Finalmente, hay que indicar que la palabra reservada `result` (línea 102) hace referencia al objeto que se crea como resultado de la transformación, de esta forma se puede acceder a las operaciones `source` y `target`, que se han inicializado en las dos líneas anteriores.

- (MIG-3) Esta migración se ha implementado con dos operaciones de *mapping*, una para migrar objetos de tipo `StateVertex`, y otra para migrar objetos de tipo `State`. La Figura 9 muestra estas dos operaciones. Ambas se construyen como una disyunción de otras operaciones de *mapping*.

```
029 -- Migrar vertices a ActivityNode
030 mapping UML14::StateVertex::toActivityNode():UML2::ActivityNode
031 disjuncts  UML14::Pseudostate::toActivityNode,
032           UML14::State::toActivityNode {}
```

```
-- Migrar State a ActivityNode
mapping UML14::State::toActivityNode():UML2::ActivityNode
disjuncts  UML14::ActionState::toOpaqueAction,
           UML14::FinalState::toActivityFinalNode {}
```

Figura 9. *Operaciones de mapping* `toActivityNode`

- (MIG-4) El hecho de que un objeto de tipo `Pseudostate` pueda convertirse en objetos de distinto tipo en el modelo destino, se ha implementado mediante una operación de *mapping* con una cláusula `disjuncts`, y una operación de *mapping* con una cláusula `when` por cada tipo de objeto destino al que se tiene que convertir (Figura 10).

```
-- Migración de pseudoestados
mapping UML14::Pseudostate::toActivityNode():UML2::ActivityNode
disjuncts  UML14::Pseudostate::toInitialNode,
           UML14::Pseudostate::toJoinNode,
           UML14::Pseudostate::toForkNode,
           UML14::Pseudostate::toDecisionNode{}

-- Migración de pseudoestado a InitialNode
mapping UML14::Pseudostate::toInitialNode():UML2::InitialNode
    when{ self.kind = PseudostateKind::initial}
{
    name:= self.name;
}

-- Migración de pseudoestado a InitialNode
mapping UML14::Pseudostate::toJoinNode():UML2::JoinNode
    when{ self.kind = PseudostateKind::join}
{
    name:= self.name;
}

-- Migración de pseudoestado a ForkNode
mapping UML14::Pseudostate::toForkNode():UML2::ForkNode
    when{ self.kind = PseudostateKind::fork}
{
    name:= self.name;
}

-- Migración de pseudoestado a DecisionNode
mapping UML14::Pseudostate::toDecisionNode():UML2::DecisionNode
    when{ self.kind = PseudostateKind::junction}
{
    name:= self.name;
}
```

Figura 10. *Operaciones de mapping para migrar objetos de tipo* `Pseudostate`

- (MIG-5) La Figura 11 muestra la operación de *mapping* para realizar esta migración. Es una operación sencilla en la que se inicializa la propiedad `name` del objeto de tipo `ActivityFinalNode` resultado de la operación con el valor de la propiedad `name` del objeto origen de la operación.

```
047 -- Migrar FinalState a ActivityFinalNode
048 mapping UML14::FinalState::toActivityFinalNode():UML2::ActivityFinalNode
049 {
050     name:= self.name;
051 }
```

Figura 11. *Operación de mapping para migrar objetos de tipo* `FinalState`

- (MIG-6) La Figura 12 muestra la operación de *mapping* para realizar esta migración.

```
042 -- Migrar ActionState a OpaqueAction
043 mapping UML14::ActionState::toOpaqueAction():UML2::OpaqueAction
044 {
045     name:= self.name;
046 }
```

Figura 12. *Operación de mapping para migrar objetos de tipo* `ActionState`

- (MIG-7) Lo destacable de la operación de *mapping* que implementa esta migración y que se muestra en la Figura 13 es el uso de la operación `resolve`, que permite obtener referencias a aquellos objetos de tipo ActivityNode que se crearon como resultado de la ejecución de operaciones de *mapping* que se invocaron anteriormente.

```
023 -- Migrar Partition a ActivityPartition
024 mapping UML14::Partition::toActivityPartition():UML2::ActivityPartition
025 {
026     name:= self.name;
027     node += self.contents.resolve (UML2::ActivityNode);
028 }
```

Figura 13. *Operación de mapping para migrar objetos de tipo* `Partition`

- (MIG-8) La operación de *mapping* para implementar esta migración se muestra en la Figura 14. Como el nombre de la propiedad `body` coincide con una palabra reservada de QVTo, se le antepone el carácter subrayado para indicar que no se refiere a la palabra reservada, sino a la propiedad (línea 130).

```
126 -- Migración de Guard a OpaqueExpression
127 mapping UML14::Guard::toOpaqueExpression():UML2::OpaqueExpression{
128     name:= self.name;
129     language += self.expression.language;
130     _body+= self.expression._body;
131 }
```

Figura 14. *Operación de mapping para migrar objetos de tipo* `Guard`

OTRAS OPERACIONES DE MAPPING

Además de las operaciones de *mapping* que se han visto en el apartado anterior, y que están directamente relacionadas con las migraciones a realizar para

convertir un modelo de diagrama de actividad UML 1.4 en un modelo de diagrama de actividad UML 2.2. Se ha definido una transformación extra para generar, a partir de un objeto `ActivityGraph`, un paquete que sea el contenedor del modelo (Figura 15). En la implementación de esta operación de *mapping* se ha usado la operación `resolveIn` que obtiene los objetos de tipo `UML2::Activity` que se han creado al invocar a la transformación `UML14::ActivityGraph::toActivity`.

```
017 -- Migrar ActivityGraph a Package
018 mapping UML14::ActivityGraph::toPackage():UML2::Package{
019     name:='graph';
020     packagedElement+=self.resolveIn (UML14::ActivityGraph::toActivity,
021                                      UML2::Activity);
022 }
```

Figura 15. *Operación de mapping para la creación de objetos de tipo* `Package`

CONFIGURANDO LA EJECUCIÓN DE LA TRANSFORMACIÓN

Para poder ejecutar la transformación presentada en el Anexo, es necesario configurar las propiedades de ejecución de la misma. Esto, en el entorno del proyecto Eclipse M2M QVTo se realiza mediante el asistente que se muestra en la Figura 16, en el que hay que rellenar, como mínimo, las siguientes propiedades:

- La ruta del archivo que contiene el código de la transformación (*Transformation Module*). El archivo ha de tener extensión `.qvto`.
- La ruta del archivo que almacenará la traza de la transformación, si es que se quiere generar este archivo. Para indicar la intención de generarlo se ha de marcar el botón de opción *Generate trace file.* El archivo de traza resultado tendrá extensión `.qvtotrace`.
- La URI del modelo de entrada a la transformación (*inModel*).
- La URI del modelo de salida de la transformación.
- Una referencia a una característica de metamodelo destino en la que se guardará el resultado de la transformación (*Feature*).

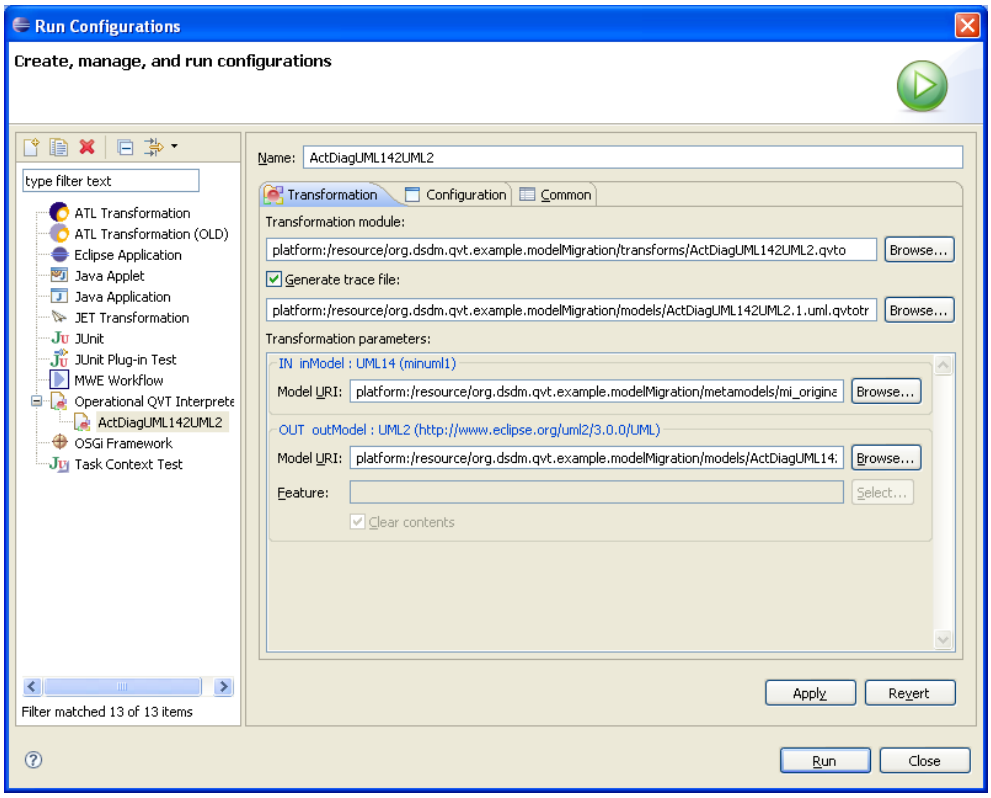

Figura 16. *Ventana de configuración de propiedades de ejecución de la transformación*

BIBLIOGRAFÍA

Eclipse MDT / UML2 Project. (2012). *Eclipse MDT / UML2 Wiki*. Recuperado el Junio de 2012, de Eclipse MDT / UML2 Wiki: http://wiki.eclipse.org/MDT-UML2

OMG. (Septiembre de 2001). *Unified Modelling Language 1.4 Specification*. Recuperado el 4 de Mayo de 2012, de Unified Modelling Language 1.4 Specification: http://www.omg.org/spec/UML/1.4/

OMG. (Febrero de 2009). *Unified Modelling Language 2.2 Specification*. Recuperado el 4 de Mayo de 2012, de Unified Modelling Language 2.2 Specification: http://www.omg.org/spec/UML/2.2/

Rose, L. M., Kolovos, D. S., Paige, R. F., & Polack, F. A. (2010). Model Migration Case for TTC 2010. *Transformation Tool Contest 2010.*

APÉNDICE. CÓDIGO FUENTE COMPLETO

La Figura 17 muestra una transformación QVTo que toma como entrada un modelo conforme a un metamodelo reducido de los diagramas de actividad UML14, y devuelve un modelo UML2, conforme al metamodelo de UML2, desarrollado dentro del marco del proyecto de modelado en Eclipse. Para entender mejor la transformación, es necesario conocer y entender antes estos metamodelos.

```
001 modeltype UML14 uses minuml1('minuml1');
002 modeltype UML2  uses 'http://www.eclipse.org/uml2/3.0.0/UML';

003 transformation ActDiagUML142UML2(in inModel:UML14, out outModel:UML2);

004 -- Desencadenante de la transformación
005 main () {

006     inModel.rootObjects ()[minuml1::ActivityGraph].map toActivity();
007     inModel.rootObjects ()[minuml1::ActivityGraph].map toPackage();
008 }

009 -- Migrar ActivityGraph a Activity
010 mapping UML14::ActivityGraph::toActivity():UML2::Activity{
011      name:= self.name;
012      node+= self.top.oclAsType(
013                    UML14::CompositeState).getSubvertexes().map
toActivityNode();
014      edge+= self.transitions.map toActivityEdge()->asSequence();
015      group+= self.partition.map toActivityPartition();
016 }
017 -- Migrar ActivityGraph a Package
018 mapping UML14::ActivityGraph::toPackage():UML2::Package{
019     name:='graph';
020     packagedElement+=self.resolveIn (UML14::ActivityGraph::toActivity,
021                                      UML2::Activity);
022 }
023 -- Migrar Partition a ActivityPartition
024 mapping UML14::Partition::toActivityPartition():UML2::ActivityPartition
025 {
026     name:= self.name;
027     node += self.contents.resolve (UML2::ActivityNode);
028 }

029 -- Migrar vertices a ActivityNode
030 mapping UML14::StateVertex::toActivityNode():UML2::ActivityNode
031 disjuncts  UML14::Pseudostate::toActivityNode,
032           UML14::State::toActivityNode {}


033 -- Migrar State a ActivityNode
034 mapping UML14::State::toActivityNode():UML2::ActivityNode
035 disjuncts  UML14::ActionState::toOpaqueAction,
036            UML14::FinalState::toActivityFinalNode {}

037 -- Obtener el conjunto de vertices de un estado compuesto
038 query
UML14::CompositeState::getSubvertexes():OrderedSet(UML14::StateVertex)
039 {
```

```
040     return self.subvertex;
041 }

042 -- Migrar ActionState a OpaqueAction
043 mapping UML14::ActionState::toOpaqueAction():UML2::OpaqueAction
044 {
045     name:= self.name;
046 }

047 -- Migrar FinalState a ActivityFinalNode
048 mapping UML14::FinalState::toActivityFinalNode():UML2::ActivityFinalNode
049 {
050     name:= self.name;
051 }

052 -- Migración de pseudoestados
053 mapping UML14::Pseudostate::toActivityNode():UML2::ActivityNode
054 disjuncts  UML14::Pseudostate::toInitialNode,
055            UML14::Pseudostate::toJoinNode,
056            UML14::Pseudostate::toForkNode,
057            UML14::Pseudostate::toDecisionNode{}

058 -- Migración de pseudoestado a InitialNode
059 mapping UML14::Pseudostate::toInitialNode():UML2::InitialNode
060     when{ self.kind = PseudostateKind::initial}
061 {
062     name:= self.name;
063 }

064 -- Migración de pseudoestado a InitialNode
065 mapping UML14::Pseudostate::toJoinNode():UML2::JoinNode
066     when{ self.kind = PseudostateKind::join}
067 {
068     name:= self.name;
069 }

070 -- Migración de pseudoestado a ForkNode
071 mapping UML14::Pseudostate::toForkNode():UML2::ForkNode
072     when{ self.kind = PseudostateKind::fork}
073 {
074     name:= self.name;
075 }

076 -- Migración de pseudoestado a DecisionNode
077 mapping UML14::Pseudostate::toDecisionNode():UML2::DecisionNode
078     when{ self.kind = PseudostateKind::junction}
079 {
080     name:= self.name;
081 }

082 -- Migración de Transition a ActivityEdge
083 mapping UML14::Transition::toActivityEdge():UML2::ActivityEdge
084 disjuncts  UML14::Transition::toControlFlow,
UML14::Transition::toObjectFlow {
085 }

086 -- Migración de Transition a ControlFlow
087 mapping UML14::Transition::toControlFlow():UML2::ControlFlow
088     when{not self.target.oclIsTypeOf (UML14::ObjectFlowState) and
```

```
089         not self.source.oclIsTypeOf (UML14::ObjectFlowState)}
090 {
091     name := self.name;
092     source:= self.source.resolveone (UML2::ActivityNode);
093     target:= self.target.resolveone (UML2::ActivityNode);
094     guard := self.guard.map toOpaqueExpression();
095 }

096 -- Migración de Transition a ObjectFlow
097 mapping UML14::Transition::toObjectFlow():UML2::ObjectFlow
098     when{self.isSurroundingObjectFlow()}
098 {
100     source:= self.source.resolveone (UML2::ActivityNode);
101     target:= self.getTargetSurroundingObjectFlow().resolveone
(UML2::ActivityNode);
102     name := result.source.name + '2'+ result.target.name;
103     guard := self.guard.map toOpaqueExpression();
104 }

105 -- Devuelve cierto si la transición rodea a un objeto de tipo
106 -- ObjectFlowState
107 query UML14::Transition::isSurroundingObjectFlow():Boolean{
108 var res :Boolean:=null;
109     res:= self.target.oclIsTypeOf(UML14::ObjectFlowState) and
110           inModel.objectsOfType(UML14::Transition)->exists(t |
111
      t.source.oclIsTypeOf(UML14::ObjectFlowState) and
112                                 t.source = self.target);
113 return res;
114 }

115 -- Devuelve el vértice destino de una transición que rodea a un objeto
de -
116 -- tipo ObjectFlowState
117 query
UML14::Transition::getTargetSurroundingObjectFlow():UML14::StateVertex{
118 var res :UML14::StateVertex:=null;
119 var transition: UML14::Transition:=null;

120     transition := inModel.objectsOfType(UML14::Transition)->selectOne
(t |
121                       t.source.oclIsTypeOf(UML14::ObjectFlowState) and
122                       t.source = self.target);
123     res := transition.target;
124 return res;
125 }

126 -- Migración de Guard a OpaqueExpression
127 mapping UML14::Guard::toOpaqueExpression():UML2::OpaqueExpression{
128     name:= self.name;
129     language += self.expression.language;
130     _body+= self.expression._body;
131 }

132 -- Migrar CompositeState (cuando no es el top)
133 mapping UML14::CompositeState::toActivity():UML2::Activity
134     when{not self.isTopComposite()}
135 {
136     name:= self.name;
```

```
137     node:= self.subvertex->map toActivityNode()->asSet();
138 }

139 query UML14::CompositeState::isTopComposite():Boolean{
140 var ags:
Set(UML14::ActivityGraph):=inModel.rootObjects()[minuml1::ActivityGraph];
141 var agsq : Sequence(UML14::ActivityGraph):= ags->asSequence();
142 var ag: UML14::ActivityGraph:= agsq->first();
143 var res: Boolean:= ag.top = self;
144     return  res;
145 }
```

Figura 17.*Transformación QVTo para migrar diagramas de actividad UML1.4 a diagramas de actividad UML2.2.*

Autores

ANTONIA Mª REINA QUINTERO

Doctora e Ingeniera en Informática por la Universidad de Sevilla. Actualmente es Profesora del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Sevilla, aunque también ha trabajado como Ingeniera en Informática en Telvent Tráfico y Transporte participando en el desarrollo de Sistemas de Control y Gestión de Tráfico. Actualmente es miembro del grupo de investigación TDG, y sus trabajos se centran en la Ingeniería Dirigida por Modelos y el Desarrollo de Software Orientado a Aspectos en entornos web. Es autora y co-autora de varios artículos en revistas y conferencias nacionales e internacionales. Igualmente, ha participado en diferentes proyectos de I+D a nivel regional y nacional.