Saying Hello World with QVTR-XSLT
A Solution to the TTC 2011 Instructive Case

Dan Li,* Xiaoshan Li

Faculty of Science and Technology, University of Macau, China

`lidan@iist.unu.edu, xsl@umac.mo`

Volker Stolz

Department of Informatics, University of Oslo, Norway & UNU-IIST, Macau, China

`stolz@ifi.uio.no`

1 Introduction

In this short paper we present our solution for the Hello World case study [3] of the Transformation Tool Contest (TTC) 2011 using the QVTR-XSLT tool [1]. The tool supports editing and execution of the graphical notation of QVT Relations language [6].

The case study consists of a set of simple transformation tasks which covers the basic functions required for a transformation language, such as creating, reading/querying, updating and deleting of model elements. We design a transformation for each of the tasks. The SHARE demo related to the paper can be found at [2].

We begin by giving a brief introduction of the QVTR-XSLT tool in Section 2. Section 3 provides the solutions for the tasks of the case study. We discuss the conclusion in Section 4. Details of the transformation definitions are presented in the appendices.

2 The QVTR-XSLT tool

Model transformation is the core technology for model-driven development, and is used in software model refinement, evolution, refactoring and code generation. To address the need for a common transformation language, the Object Management Group (OMG) proposed the Query/View/Transformation language (QVT) [6] standard. QVT has a hybrid declarative/imperative nature. In its declarative language, called QVT Relations (QVTR), a transformation is defined as a set of *relations* (rules) between source and target models, each conforming to their respective metamodels. Transformations are driven by a single, designated top-level relation.

QVTR combines a textual and a graphical notation. In graphical syntax, a relation specifies how two object diagrams, called *domain patterns*, relate to each other. That is, the *structural* matching of elements in the source- and target model is done diagrammatically. Moreover, QVTR employs a textual language based on essential OCL [5] to define additional (non-structural) constrains in relations. The graphical notation of QVTR provides a concise, intuitive and yet powerful way to specify transformations. However, currently there are very few tools supporting QVTR, and even fewer for its graphical notation.

---

*On leave from Guizhou Academy of Sciences, Guizhou, China

QVTR-XSLT supports the graphical notation of QVT Relations, and an execution engine for a subset of QVTR by means of XSLT programs. It consists of two parts:

- **Graphical Editor**: Building on top of *MagicDraw UML* [4], the editor has a graphical interface for defining metamodels as simple class diagrams, specifying QVTR relations and queries in graphical notation, validating the design, and saving the transformations as an XML file. The toolbar of the graphical editor is showed in Fig. 1.

- **Code generator**: It reads in the XML file, and generates an XSLT stylesheet for a transformation. Fig. 2 illustrates the interface of the code generator.
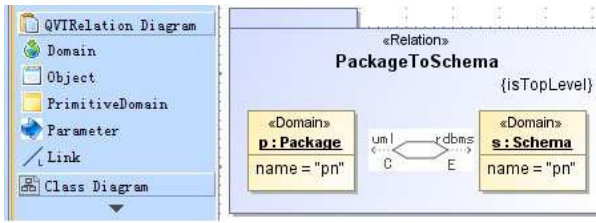
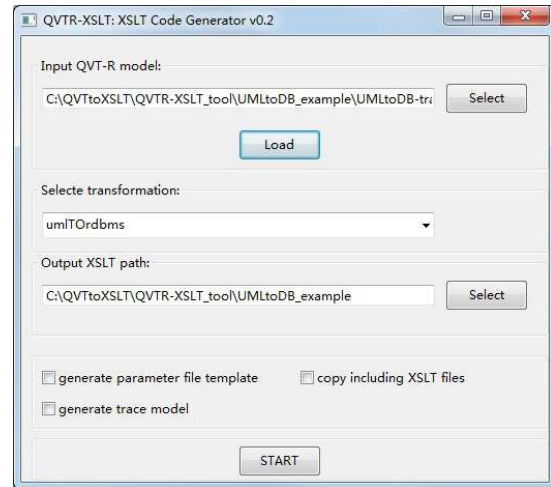Figure 1: Toolbar of QVTR graphical editor   Figure 2: XSLT code generator

The outputs of the code generator are pure XSLT programs which can be directly executed under any XSLT processor on any platform. Additionally, we have also developed a transformation runner, in the form of a Java program invoking the Saxon 9 XSLT processor, to facilitate the execution of generated XSLT stylesheets.

The QVTR-XSLT tool supports transformation parameters, transformation inheritance through rule overriding, and multiple input and output models. Furthermore, *in-place* transformations are defined as modifications (insert, remove, replace) of the existing model elements. QVTR-XSLT-based transformations are used in the rCOS Modeler for use case-driven development of component- and object systems.

3   Solution

As the first step for transformation design, we define all metamodels described in the case specification using the graphical editor of the QVTR-XSLT tool. Simple UML class diagrams are used to specify metamodels in the tool. The source models provided by the case study conform the metamodels. In addition, we design a simple HTML metamodel as the target metamodel for the model-to-text transformation. We also have a *Result* metamodel to store the *results* of querying matched model elements, as well as the descriptions of the queries. Appendix B (Fig. 3–9) shows all the metamodels.

3.1 Hello world

This task consists of three subtasks: two constant transformations and a model-to-text transformation. We complete each subtask using a transformation of a single relation, as shown in Appendix C. There must be a source model for a transformation, even we may not actually need any information of the model. For the convenience we take the *SimpleGraph* metamodel as the source metamodel for the constant transformations, because the case study has already provided a model of the metamode. We use the *HtmlMetaModel* as the target metamodel of the model-to-text transformation, so its output is a web page, which can be easily displayed in a browser.

3.2 Count matches with certain properties

The task focuses on querying a simple graph model to find elements with certain properties. We design four *Queries* (Fig. 15–18) using the graphical notation for counting the number of nodes, isolated nodes, looping edges and dangling edges. In all these queries, OCL function *size()* is used to count the number of elements satisfying the conditions, and a predefined variable *result* returns the results of the queries. Moreover, we define a function *GetAllCircleNodes* (Fig. 20) for counting the number of matches of a circle consisting of three nodes. The function is directly written in XSLT, the back-end language of our tool, and calls within it another XSLT function *GetCircleNodes* and a query *GetLinkedNodes* (Fig. 19), which returns all target nodes of a given source node. In actually, the two functions offer the capacity to calculate general *k*-circles, and the *k* is given as the last parameter of function *GetCircleNodes*. It shows how XSLT code can be seamlessly integrated with QVTR to provide more powerful features.

The transformation starts from the relation *GraphToResult* (Fig. 13), in where queries and functions are invoked one by one, and the count results and their explanations are sent to relation *ShowIntResult* for adding to the result model.

3.3 Reverse edges

The transformation takes the *SimpleGraph* as both the source and target metamodels. We design three top-level relations for copying graphs, copying nodes and copying edges while exchanging the source and target nodes, as shown in Appendix E. This transformation can properly handle dangling edges.

3.4 Simple migration

Using the *EvolveGraph* as the target metamodel, the transformation is also designed as three top-level relations for migrating graphs, nodes and edges respectively (see Appendix F).

3.5 Topology-changing migration

With the *MoreEvolveGraph* as the target metamodel, the transformation only has two relations to migrate graphs and nodes, and a query is used to get the destination nodes of a node (see Appendix G).

3.6 Delete node with specific name and its incident edges

Different from all above transformations, the task is completed by an *in-place* transformation, where two relations are used to mark the node with name "n1", along with the incident edges of the node, as *remove* in the *xmiDiffOp* property (see Appendix H). For an in-place transformation, the source and

target models are the same during the execution, and the model elements can be added, deleted and updated.

When an in-place transformation is executed, the modifications to the model will be collected into a *difference model* as a set of adding, deleting, or altering operations. Then the source model is modified according to the records of the *difference model* to get the result model. For example, the node and the edges will be deleted in this task. This process is user transparent and runs automatically.

3.7   Insert transitive edges

Similar to the other transformations, we use three relations to copy graphs, nodes, and edges from the source model to the target model. Furthermore, with the help of two queries that obtain a node's destination nodes (except the node itself) and the edges from one node to another node, the relation *LookTransitive* calculates the transitive closure, and invokes relation *InsertEdge* to insert an edge between the two indirectly related nodes. Appendix I depicts the definitions of the transformation.

4   Experiments and Conclusion

Table 1: Transformations and experiment results

Task	Number of relations /queries/functions	Lines of XSLT	Input model (.xmi)	Exec time (ms)
Hello world (constant)	1	47	Graph1	< 1
Hello world (extended constant)	1	52	Graph1	< 1
Hello world (model-to-text)	1	60	Greetingext	< 1
Count matches	2/5/2	172	Graph1	13
Reverse edges	3	81	Graph1	< 1
Simple migration	3	82	Graph1	< 1
Topology-changing migration	2/1	73	Graph1	< 1
Delete node	2	81	Graph1	< 1
Insert transitive edges	5/2	139	Graph1	15

We have solved all mandatory and optional tasks of the case study. We generate XSLT stylesheets for all the transformations through our code generator. Table 1 shows the number of relations/queries/-functions and lines of XSLT code for each transformation. There are only about 30 lines of XSLT code that are directly hand-written and embed into the transformation counting matches, and all others are generated from the graphical notation of QVTR.

Using our transformation runner, the transformations are executed in a laptop of Intel 2.13 GHz M330 CPU, 3 GB memory, and running Windows 7 home version. The results are also shown in Table 1. The execution time includes the time for loading and saving model files from/to disk.

Conclusion

We presented transformations for the Hello World case study of TCC 2011 to show how basic model transformation problems can be solved with the QVTR-XSLT tool. These transformations are designed

using the standard graphical notation and OCL expressions of QVT Relations in a straightforward, concise and intuitive way. We hope the case study will help to demonstrate that the language and the tool can be efficiently applied to model transformations in practice.

References

[1] Dan Li, Xiaoshan Li & Volker Stolz (2011): *QVT-based model transformation using XSLT. SIGSOFT Softw. Eng. Notes* 36, pp. 1–8, doi:`10.1145/1921532.1921563`.

[2] lidan@iist.unu.edu (2011): *Online demo: QVTR-XSLT solutions to the TTC11 Hello World and Compiler Optimization case studies*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC11_TTC11_QVTR-XSLT.vdi`.

[3] Steffen Mazanek (2011): *Hello World! An Instructive Case for the Transformation Tool Contest*. In Pieter Van Gorp, Steffen Mazanek & Louis Rose, editors: *TTC 2011: Fifth Transformation Tool Contest, Zürich,Switzerland, June 29-30 2011*, EPTCS.

[4] NoMagic, Inc.: *MagicDraw*. `http://www.magicdraw.com`.

[5] Object Management Group (2006): *Object Constraint Language, version 2.0*. Available at `http://www.omg.org/spec/OCL/2.0/`.

[6] Object Management Group (2009): *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*.

A   A Brief Introduction to QVT Relations

QVT Relations (QVTR) is a declarative model transformation language proposed by the OMG as part of the MOF Query/View/Transformations (QVT) standard [6]. QVTR specifies a *transformation* as a set of *relations* between source and target metamodels. A metamodel is defined in our tool as a simple class diagram. In addition, a transformation may own some *functions*, which are side-effect-free operations, and *queries* used to retrieve information from the models.

In the graphical notation, a *relation* defines how two object diagrams, called *domain patterns*, relate to each other. The object with tag ≪*domain*≫ is the *root* of a domain pattern, and it also serves as a parameter of the relation. In general, we assume the left domain pattern is the source domain, and the right the target domain. An *object* or a property of an object could be given a name that is taken as a *variable*. If the object is in the source domain pattern, then the object or the value of the property is bound to the variable. Otherwise the object in target domain pattern means assigning the value of the variable to the object or property. Note that a property variable in the diagrams may contain additional quote-characters that are an artefact of the visualization, and not string delimiters.

When a relation is executed, the source domain pattern is searched in the source model by way of *pattern matching* which starts from the domain root. When a match is found, all variables defined in source domain pattern are bound to values. The target domain pattern acts as a template to create corresponding objects and links in the target model using the values of the variables in the pattern.

A relation may define a pair of optional *when-* and *where*-clauses which consist of a set of OCL expressions. The *when*-clause indicates additional matching conditions for the relation. And new variables can be defined in the *where*-clause. Other relations could be invoked in the *where*-clause and variables can be passed as arguments. A relation may also have *primitive domains* in order to pass additional parameters between the relations. Furthermore, a relation is either designed as a *top-level* relation, or a *non-top-level* relation. A *top-level* relation is invoked from the transformation framework, and *non-top-level* relations are invoked by other relations.
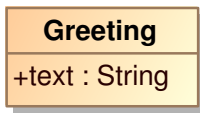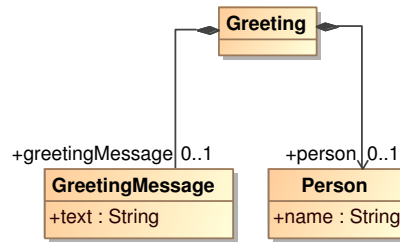
B   Metamodel definitions

Greeting

+text : String

Greeting

+greetingMessage 0..1

GreetingMessage

+text : String

+person 0..1

Person

+name : String

html

1

body

1

head

+#title : String

1..*

table

1..*

tr

+#td : String

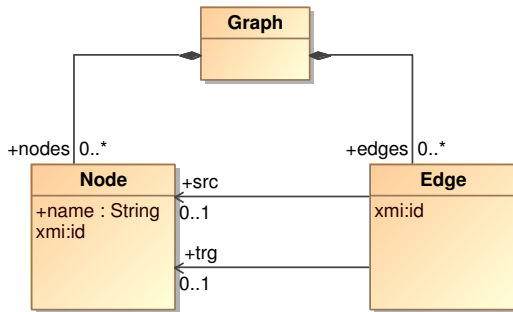Figure 3: HelloWorld   Figure 4: HelloWorldExt   Figure 5: HtmlMetaModel

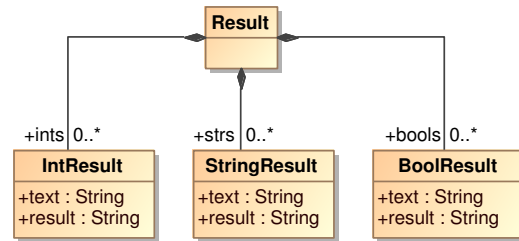**Graph**

+nodes 0..*

Node

+name : String
xmi:id

+src 0..1

+trg 0..1

+edges 0..*

Edge

xmi:id

Result

+ints 0..*

IntResult

+text : String
+result : String

+strs 0..*

StringResult

+text : String
+result : String

+bools 0..*

BoolResult

+text : String
+result : String
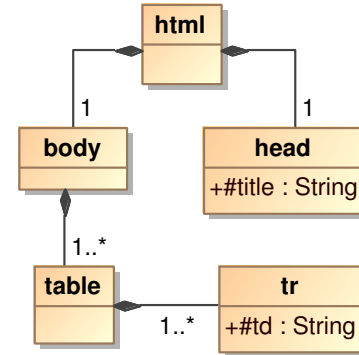
Figure 6: SimpleGraph   Figure 7: Result

Graph

+gcs 0..*

GraphComponent

+text : String
xmi:id

Node

+src 0..1

+trg 0..1

Edge

Graph

+nodes 0..*

Node

+text : String
xmi:id

+linksTo 0..*
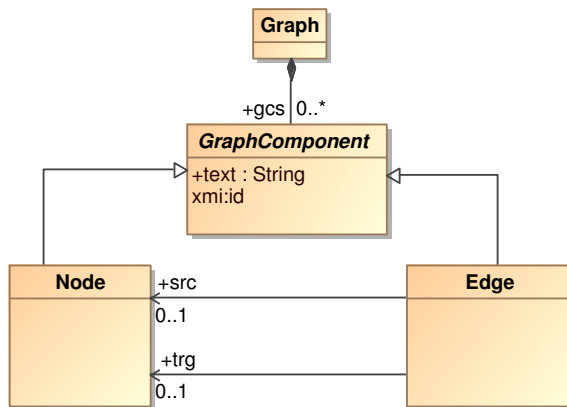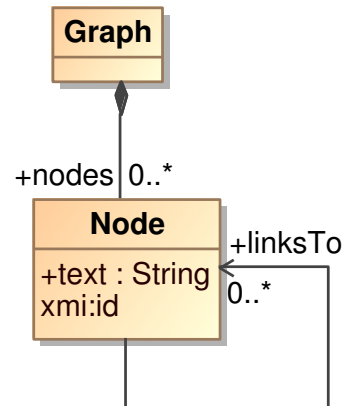
Figure 8: EvolvedGraph   Figure 9: MoreEvolvedGraph

C   Transformation for Hello world

C.1   The constant transformation

• **Configuration:** name : *TTC_HelloWorld*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *src*, target: *HelloWorld*, targetKey:*text*, targetName : *trg*.
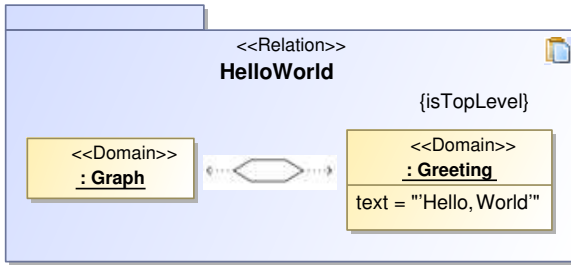
Figure 10: Constant transformation
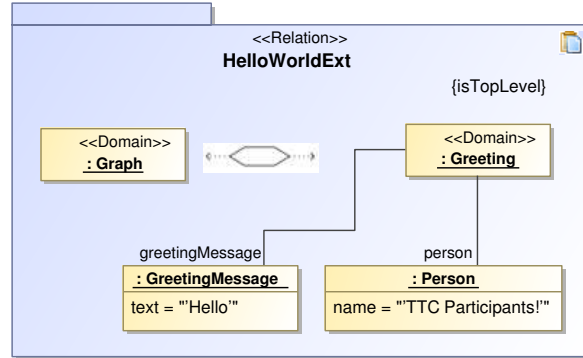
Figure 11: Constant transformation with extended model

C.2 The constant transformation with extended model

● **Configuration:** name : *TTC_HelloWorldExt*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *src*, target: *HelloWorldExt*, targetKey:*text*, targetName : *trg*.

C.3 The model-to-text transformation

● **Configuration:** name : *TTC_HelloWorldText*, output : *html*, source : *HelloWorldExt*, sourceKey : *text*, sourceName : *src*, target: *HtmlMetaModel*, targetKey:*name*, targetName : *html*.
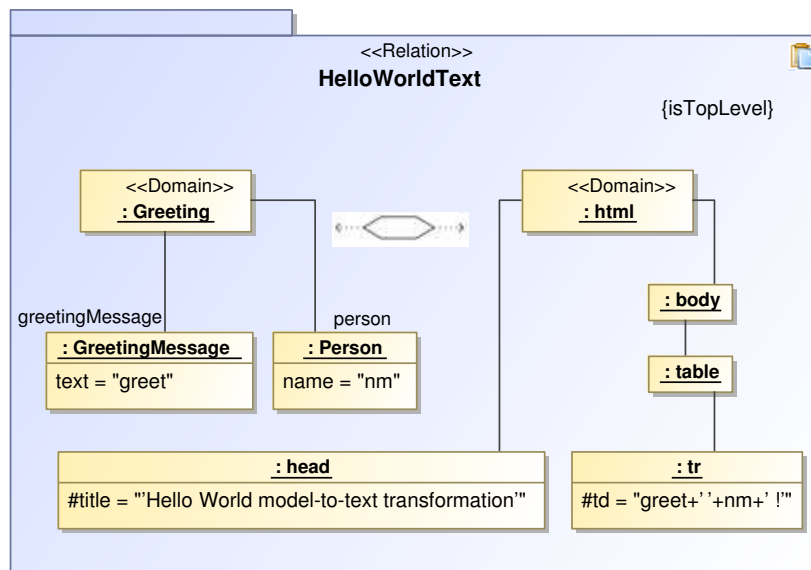
Figure 12: Model-to-text transformation

D Transformation for Count Matches with Certain Properties

• **Configuration:** name : *TTC_CountElement*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *graph*, target: *Result*, targetKey:*text*, targetName : *result*.
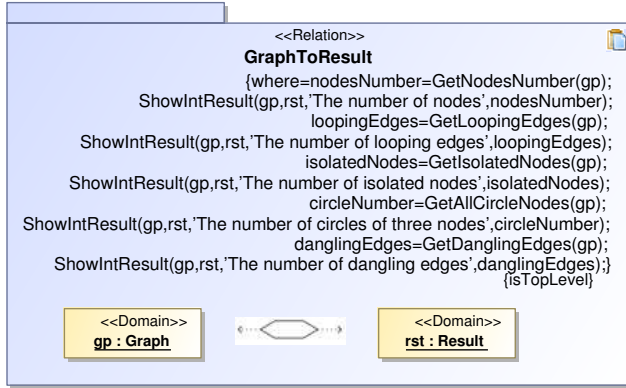
Figure 13: Starting top level relation
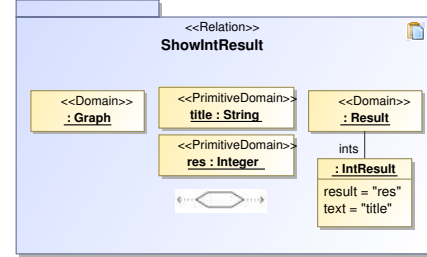
Figure 14: Add count result and explanation to result model
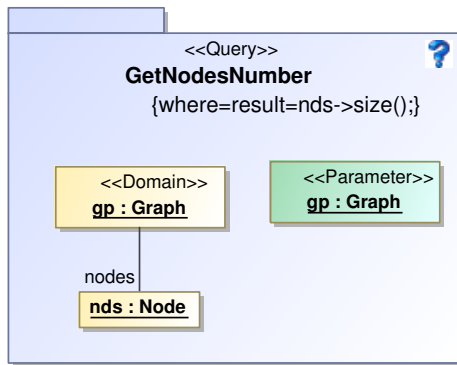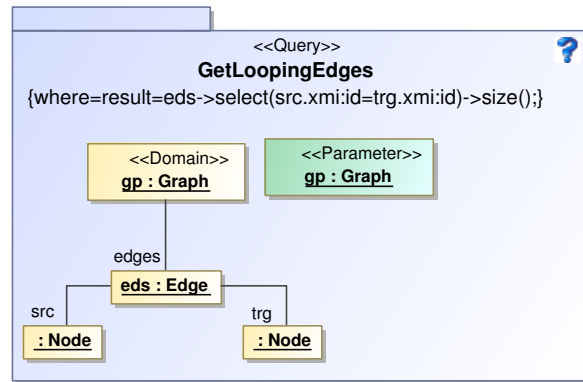

Figure 15: Count the number of nodes
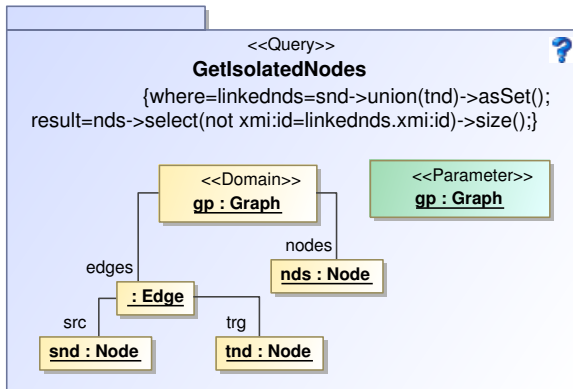

Figure 16: Count the number of looping edges

Figure 17: Count the number of isolated nodes
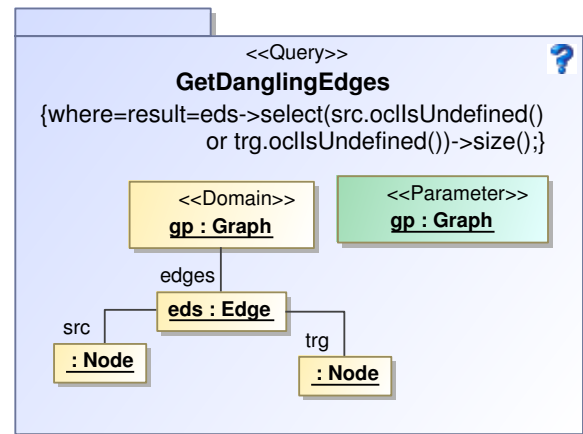
Figure 18: Count the number of dangling edges
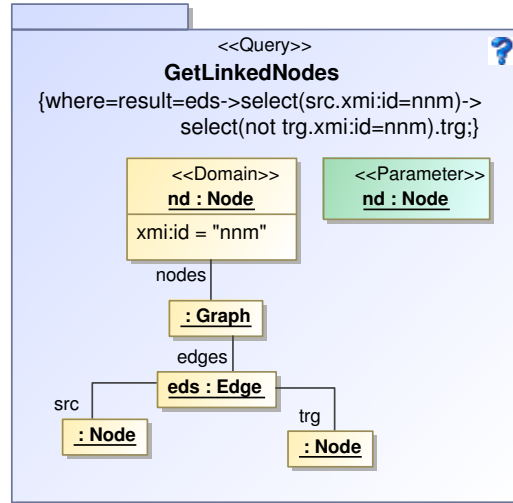
Figure 19: Get target nodes of a node

```
<xsl:variable name="nds" select="$gp/nodes"/>
<xsl:variable name="allcnodes" as="item()∗">
  <xsl:for−each select="$nds">
      <xsl:sequence select="my:GetCircleNodes(.,.,2)"/>
  </xsl:for−each>
</xsl:variable>
<xsl:sequence select="count(distinct−values($allcnodes))"/>
```

Figure 20: Function **GetAllCircleNodes**(gp : Graph)

```
<xsl:variable name="lnds" select="my:GetLinkedNodes($nd)"/>
<xsl:for−each select="$lnds">
  <xsl:variable name="cnd" select="."/>
  <xsl:choose>
    <xsl:when test="$counter=0 and $list[1][@xmi:id=$cnd/@xmi:id]">
      <xsl:variable name="rrr" as="item()∗">
        <xsl:for−each select="$list">
          <xsl:sort select="@xmi:id" data−type="text" order="ascending"/>
          <val><xsl:value−of select="./@xmi:id"/></val>
        </xsl:for−each>
      </xsl:variable>
      <val><xsl:value−of select="string−join($rrr,'')"/></val>
    </xsl:when>
    <xsl:when test="$counter>0 and not($list[@xmi:id=$cnd/@xmi:id])">
      <xsl:variable name="newlist" select="insert−before($list,count($list)+1,$cnd)"/>
      <xsl:sequence select="my:GetCircleNodes($cnd,$newlist,$counter − 1)"/>
    </xsl:when>
  </xsl:choose>
</xsl:for−each>
```

Figure 21: Function **GetCircleNodes**(nd: Node, list: Set, counter : Integer): Node

E   Transformation for Reverse Edges

• **Configuration:** name : *TTC_ReverseEdges*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *msrc*, target: *SimpleGraph*, targetKey:*xmi:id*, targetName : *mtrg*.
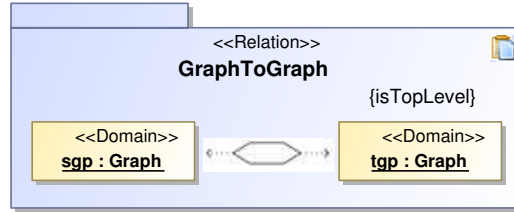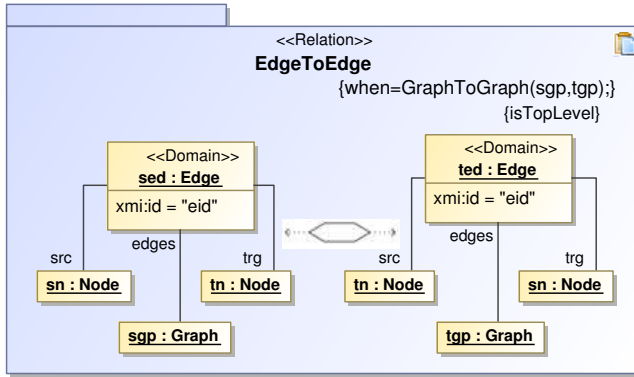

Figure 22: Starting relation—copy a graph

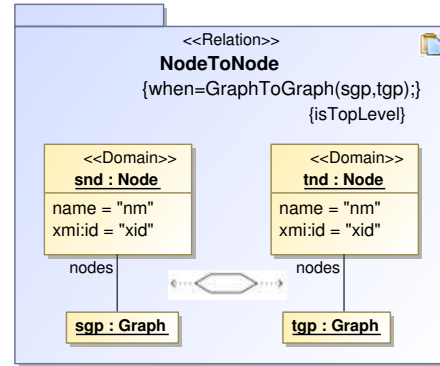Figure 23: Copy an edge with exchanged source and target nodes

Figure 24: Copy a node

F   Transformation for Simple Migration

• **Configuration:** name : *TTC_SimpleMigration*, source : *SimpleGraph*, sourceKey : *xmi:id*, source-Name : *msrc*, target: *EvolveGraph*, targetKey:*xmi:id*, targetName : *mtrg*.
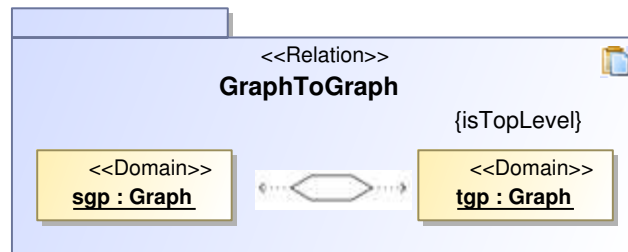

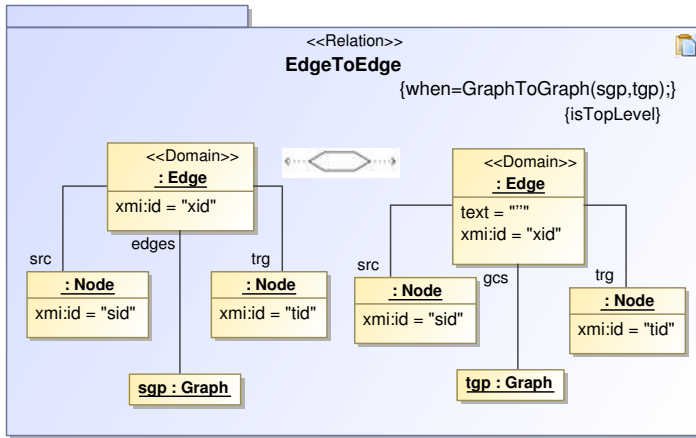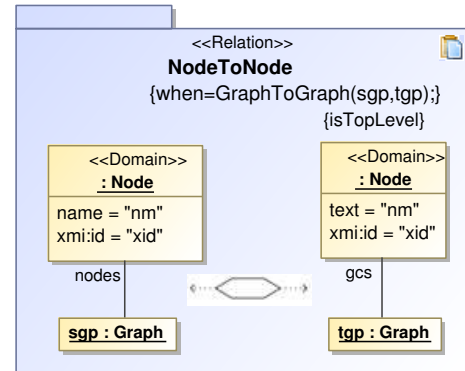Figure 25: Starting relation—migrate a graph

Figure 26: Migrate an edge   Figure 27: Migrate a node

G   Transformation for topology-changing migration

• **Configuration:** name : *TTC_TopologyMigration*, source : *SimpleGraph*, sourceKey : *xmi:id*, source-Name : *msrc*, target: *MoreEvolveGraph*, targetKey:*xmi:id*, targetName : *mtrg*.
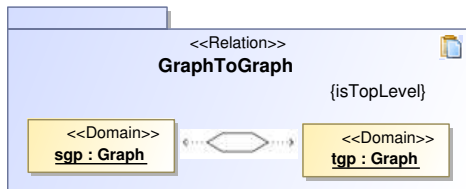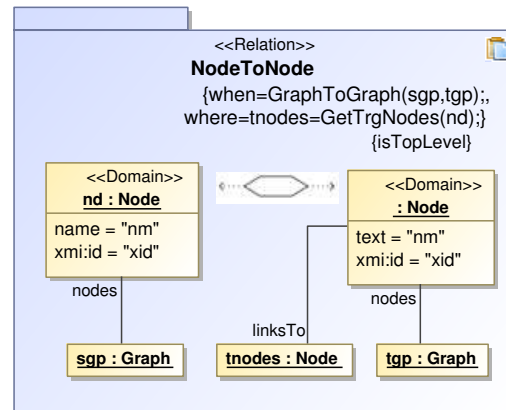

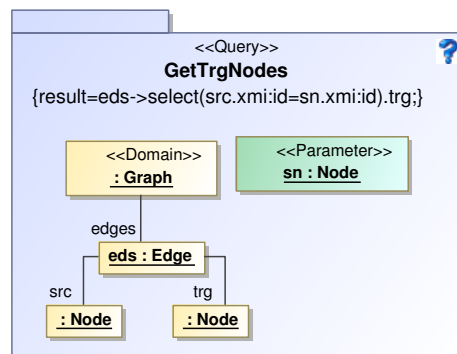Figure 28: Starting relation—migrate a graph   Figure 29: Migrate a node

Figure 30: Get target nodes of a node

H   Transformation for Delete Node

• **Configuration:** name : *TTC_DeleteNode*, isInPlace : *true*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *msrc*, target: *SimpleGraph*, targetKey:*xmi:id*, targetName : *mtrg*.
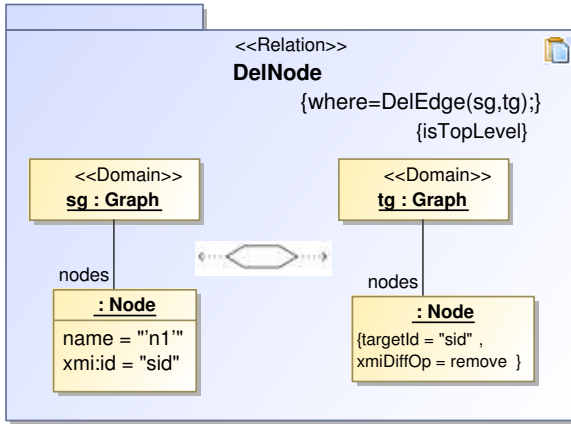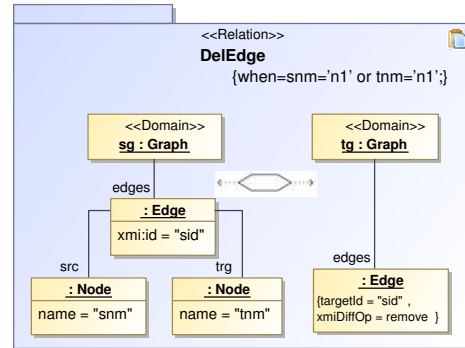

Figure 31: Delete a node with name "n1"

Figure 32: Delete the incident edges

I   Transformation for Insert Transitive Edges

• **Configuration:** name : *TTC_InsertTransitiveEdges*, source : *SimpleGraph*, sourceKey : *xmi:id*, sourceName : *msrc*, target: *SimpleGraph*, targetKey:*xmi:id*, targetName : *mtrg*.
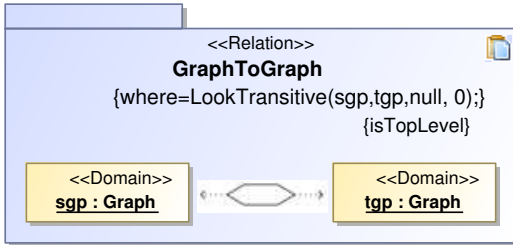

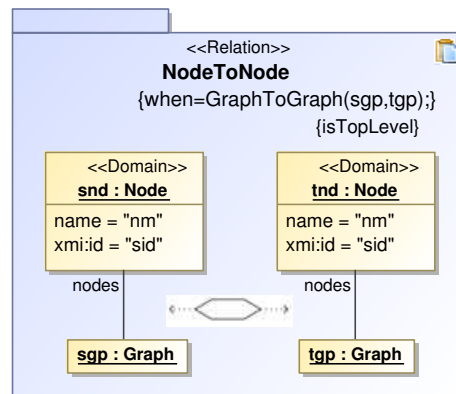Figure 33: Starting relation—copy a graph
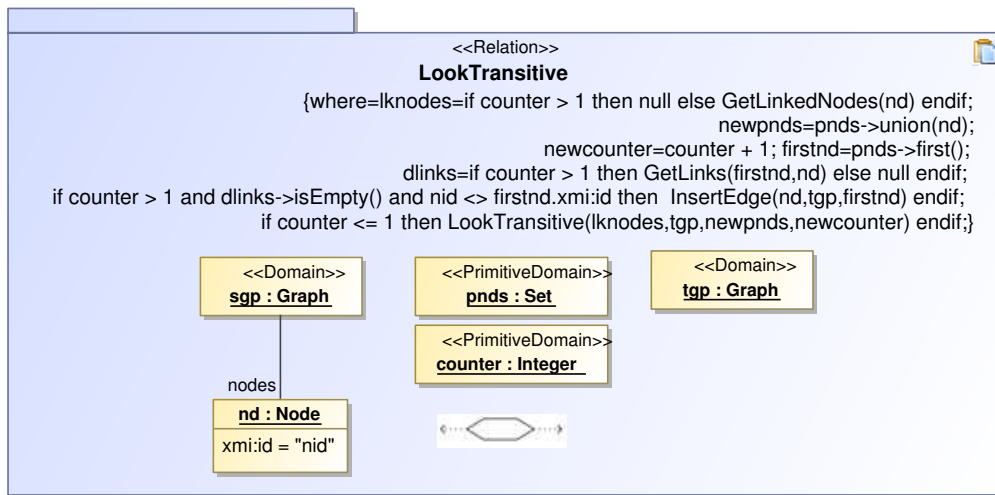

Figure 34: Copy a node

<<Relation>>
LookTransitive
{where=lknodes=if counter > 1 then null else GetLinkedNodes(nd) endif;
newpnds=pnds->union(nd);
newcounter=counter + 1; firstnd=pnds->first();
dlinks=if counter > 1 then GetLinks(firstnd,nd) else null endif;
if counter > 1 and dlinks->isEmpty() and nid <> firstnd.xmi:id then InsertEdge(nd,tgp,firstnd) endif;
if counter <= 1 then LookTransitive(lknodes,tgp,newpnds,newcounter) endif;}

<<Domain>>
sgp : Graph

<<PrimitiveDomain>>
pnds : Set

<<Domain>>
tgp : Graph

<<PrimitiveDomain>>
counter : Integer

nodes

nd : Node
xmi:id = "nid"

Figure 35: Select two nodes for inserting an additional edge

<<Relation>>
InsertEdge
{where=eid='_0.@edges.'+fnd.name+'.'+nd.name;}

<<Domain>>
sgp : Graph

<<PrimitiveDomain>>
fnd : Node

<<Domain>>
tgp : Graph

edges

: Edge
xmi:id = "eid"

nodes

nd : Node

src

fnd : Node

nd : Node
trg
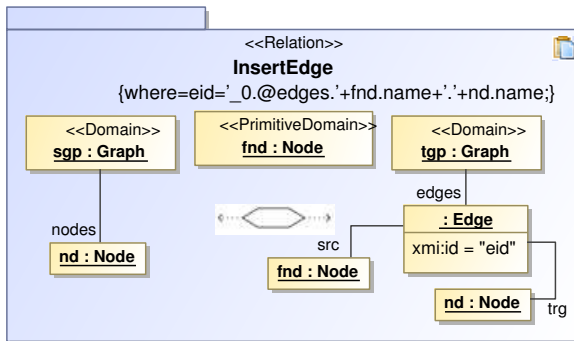
Figure 36: Insert a new edge

<<Relation>>
EdgeToEdge
{when=GraphToGraph(sgp,tgp);}
{isTopLevel}

<<Domain>>
sed : Edge
xmi:id = "eid"

<<Domain>>
ted : Edge
xmi:id = "eid"

src

sn : Node

edges

tn : Node
trg

src

sn : Node

edges

tn : Node
trg

sgp : Graph

tgp : Graph

Figure 37: Copy an edge

<<Query>>
GetLinkedNodes
{where=result=eds->select(src.xmi:id=nid)->
select(not trg.xmi:id=nid).trg;}

<<Domain>>
nd : Node
xmi:id = "nid"

<<Parameter>>
nd : Node

nodes

: Graph

edges

eds : Edge

src

: Node

trg
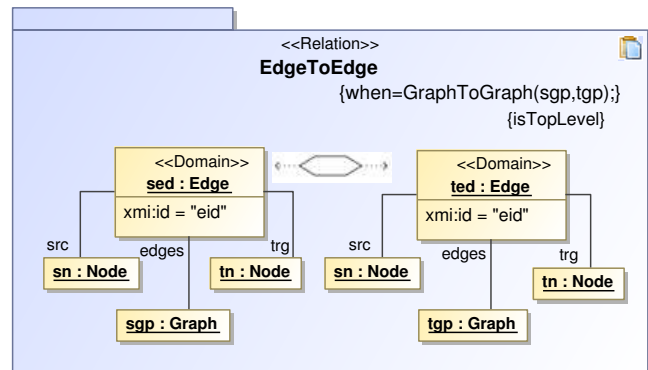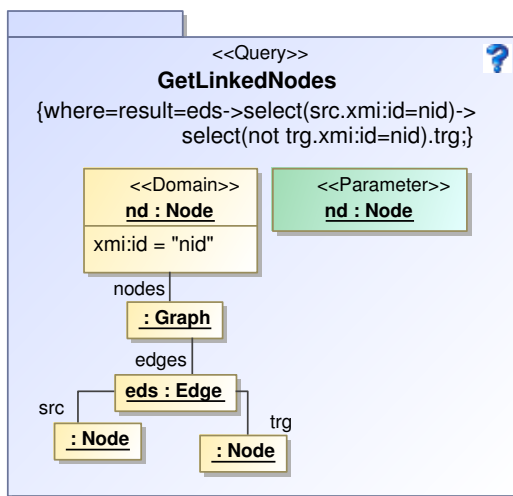
: Node

Figure 38: Get target nodes of a node

<<Query>>
GetLinks
{where=result=eds->select(src.xmi:id=snd.xmi:id '
and trg.xmi:id=tnd.xmi:id);}

<<Domain>>
snd : Node

<<Parameter>>
snd : Node

<<Parameter>>
tnd : Node
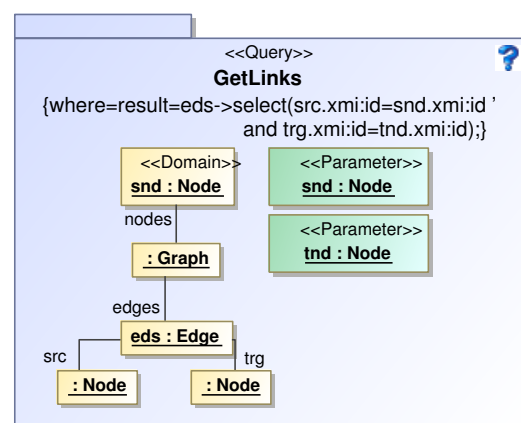
nodes

: Graph

edges

eds : Edge

src

: Node

trg

: Node

Figure 39: Get edges between nodes