Model Querying with Graphical Notation of QVT Relations

Dan Li
Faculty of Science and Technology
University of Macau, China
& Guizhou Academy of Sciences
Guiyang, China
e-mail: lidan@iist.unu.edu

Xiaoshan Li
Faculty of Science and Technology
University of Macau, China
e-mail: xsl@umac.mo

Volker Stolz
University of Oslo, Norway
& UNU-IIST, Macau, China
e-mail: stolz@ifi.uio.no

Abstract

As a standard high-level model transformation language, QVT Relations defines a graphical notation, which provides a concise, intuitive way to specify transformations. However, QVT Relations relies only on the textual language OCL for model querying, leading to verbose and complicated OCL expressions. Here, we present a graphical model query facility based on the checking semantics and pattern matching of QVT Relations. The query facility also borrows from QVT Relations the graphical notation. In addition we propose an approach to map the queries into XSLT to facilitate their execution. We have developed a tool for designing the queries and automatically generating the XSLT programs.

Keywords: Graphical model querying, UML, QVT, OCL, XPath

1 Introduction

Over the last decade, model-driven development (MDD) has emerged as a promising paradigm in software engineering. Complete models for large-scale systems are among the most complex information structures, because they capture the relevant aspects of a system from different perspectives and at different levels of abstraction. Thus the MDD paradigm poses a set of challenges to software engineering for model manipulation, including how to precisely and effectively analyze and transform models. For that, a facility to retrieve information from models, such as model querying, is a prerequisite.

To address the need for standardization in model manipulation, the Object Management Group (OMG) has proposed a set of standards. Among them, the Unified Modeling Language (UML) [17] is a visual language for defining models, and the Meta Object Facility (MOF) [14] is another visual language to define the abstract syntax of modeling languages. In addition the MOF 2.0 Query/View/Transformation (QVT) [16] is a standard for model transformations and model querying. For defining metamodels and models, OMG adopts a graphical notation, which is visually attractive since it provides a significant advantage in clarifying understanding and intention among participants of software development [24].

The QVT consists of two user-level languages, QVT-Operational Mappings and QVT Relations (QVT-R), and a low-level language, QVT-Core. QVT-R allows for specifying model transformations declaratively, and thus is more user-friendly than the other QVT languages in our opinion, and enables formal reasoning about transformations. Moreover, QVT-R has another distinguishing feature: it has two concrete syntax styles: textual and graphical. The graphical notation of QVT-R provides a concise, intuitive and yet powerful way to define model transformations.

However, QVT-R only uses the graphical notation to specify transformation rules. The QVT specification does not provide any facility to visually querying models. Instead, QVT-R heavily relies on a textual language, the Object Constraint Language (OCL) [18], for defining model queries. OCL is the *de-facto* standard for expressing complex properties of UML models, and its ability to navigate models and to form collections of objects also make it suitable as a query language. Alas, OCL suffers from several shortcomings for model querying. First, the standard OCL expressions are claimed to be too verbose, even defining simple queries quickly leads to complex query statements [19]. One of the reasons for that is the strong typing of OCL. Type conversions are frequently used in the expressions. Secondly, OCL lacks some features such as using wildcards likes "$*$" or "?" to match textual attributes. It also lacks basic statistical functions, such as *avg()*. In addition, OCL expressions are defined on the visual representations of metamodels and models, textually specifying the connections between model elements and what properties they refer to are tedious, difficult and error-prone. Overall, many users may find a visual programming paradigm more accessible than a text-based one, as the former supports the “correct-by-construction” editing, and only valid programs can be specified.

To overcome these limitations, supporting visual model querying is inevitable for QVT-R to drive for success. Our query facility will help designers in the constructions of a query by graphically selecting model elements from the metamodels, and give them a visual conception of the query. A graphical visualization would facilitate their intuitive comprehension on where a QVT-R rule will actually modify the models. Meanwhile, features needed in model querying will be reinforced in the facility.

In [10], we reported the QVTR-XSLT, a tool for supporting the graphical notation of QVT-R. The tool provides a graphical editor in which a transformation can be specified using the graphical syntax, and a code generator that can automatically generate executable XSLT [26] programs for the transformation. In this paper, we present a graphical model query facility by extending the graphical notation of QVT-R. It has been implemented on the basis of the QVTR-XSLT tool. Thus the queries can be graphically specified, and become executable as XSLT programs.

The remainder of the paper is structured as follows. In Section 2, after briefly reviewing the basic concepts of QVT Relations, we describe the syntax and semantics of the graphical model query facility. We discuss the techniques to implement the query

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

facility in Section 3. Section 4 presents a case study for using the tool to query the model of UML superstructure. Finally, we discuss related work and draw conclusions in Section 5.

2 Model Querying with QVT-R Graphical Notation

In this section we first briefly review QVT Relations, and then introduce the abstract syntax, concrete syntax and semantics of the graphical model query facility.

2.1 QVT Relations

QVT-R is a high-level declarative model transformation language. In QVT-R, a *transformation* is defined as a set of rules called *relations*. A transformation has one or more *typed models* as the source or target models, which are instances of the metamodels. Moreover, a transformation may own *functions* and *queries*, which are side-effect-free operations defined using OCL.

A relation specifies a set of *domains*, one for each participating typed model, relates to each others. A domain is defined by a *domain pattern* that can be viewed as an object diagram of object nodes, their properties and links originating from the metamodels. In addition, free *variables* can be declared and bound to the objects or the values of properties. In fact, it is the free variables that make a pattern different from a normal object diagram. A domain can be marked as *checkonly* or *enforced*.

Optionally, a relation may have *primitive domains*, a *when* and a *where* clause. The *primitive domains* serve as parameters of the relation. The *when* clause indicates the conditions under which the relation holds, and the *where* clause provides additional conditions, apart from the ones expressed by the domain patterns themselves, that must be satisfied by all pattern elements in the relation [16]. New variables can be defined in the *where* clause. Furthermore *functions*, *queries* and *relations* can be called from the *clauses*.

2.2 Syntax of the query facility

A QVT-R transformation may contain a set of *queries*, each has an OCL expression as the body. By evaluating the OCL expression, a query retrieves information from a model. Very much like the role the source domain pattern plays in a relation, a query uses the pattern matching mechanism to search for model elements that conform to the structural pattern and attribute constraints given by the OCL expression. This inspires us that a query can be expressed like a relation with only one domain pattern using the graphical notation. Naturally, our approach still depends on OCL, as the graphical notation mostly covers *structural* aspects of a query. Operations related to iteration and aggregation will be specified textually either in OCL or XSLT-functions.

We define the abstract syntax of the graphical query facility using the metamodel shown in Fig. 1. The metamodel is adapted from the QVT Relations package [16] by adding two metaclasses, *Query* and *Parameter*. The metaclasses in light yellow (or light grey in black-and-white print) boxes are imported from EMOF specification [14]. In the metamodel, a *query* consists of:

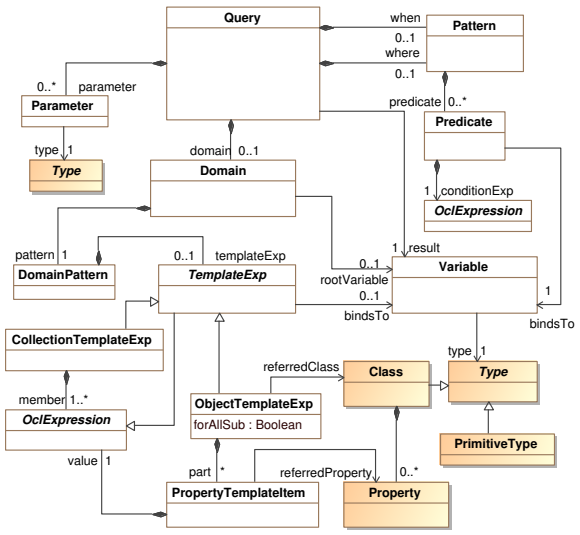

Figure 1: Metamodel of the model query

- a *domain* defined by a *domain pattern* that consists of a set of linked *template expressions (templateExp)*, each of them can be an *object template expression* (*objectTemp*) or a *collection template expression*. The former specifies a pattern that refers to a class of the corresponding metamodel. An *objectTemp* may contain a collection of *property template items* (*propertyTemp*) corresponding to the properties of the referred class. A collection template expression specifies a pattern that matches a collection of elements. Note the *templateExp* itself is also an OCL expression, it may be hierarchically contained in other *templateExps*. The *forAllSub* attribute of the *objectTemp* indicates whether we want to search all contained child at any level of depth in the hierarchy of a mode. A *templateExp* or a *propertyTemp* may declare a variable and *bind to* it. We denote all variables declared in the domain as a set V_d. In addition, a *propertyTemp* can be bound to an OCL expression which contains no variables.

- a *when* clause is a *pattern* that consists of set of *predicates* separated by the semicolon ";". A predicate here is a *boolean* expression in the form of $\langle \mathsf{var} \rangle \langle \mathsf{op} \rangle \langle \mathsf{exp} \rangle$, where $\mathsf{var} \in V_d$, exp is an OCL expression which involves only variables defined in the *where* clause or parameters, and $op \in \{=, <>, >, >=, <, <=\}$. These predicates provide additional constraints on the *templateExps* bound to the variables. The relationship between the predicates is conjunction.

- a *where* clause is also a set of *predicates*. Each of them defines a variable in the form: $\langle \mathsf{var} \rangle = \langle \mathsf{exp} \rangle$, where var is a free variable, and exp is an OCL expression that only involves parameters, variables bound in the domain pattern or variables that have been defined in the preceding predicates of the *where* clause. We denote all variables defined in the *where* clausen as a set V_w.

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

- a set of *parameters* used to pass values to the query, denoted as V_p. A parameter can be of primitive type or type of a class defined in a metamodel.

The set of all variables of the query denoted as $V = V_p \cup V_d \cup V_w$. Particularly, a query may have a *root variable* var_{rt} which binds to the *root objectTemp* that adheres to tag «`domain`», and $\text{var}_{rt} \in V_d \wedge \text{var}_{rt} \in V_p$. Moreover, a query must define a variable var_{rst} with name "`result`", which returns the result of the query, where $\text{var}_{rst} \in V_d \cup V_w$.

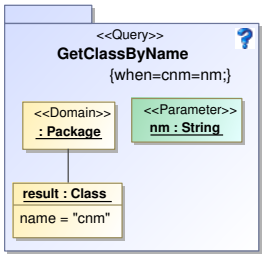

Figure 2: Example of the model query

Fig. 2 depicts a query example which acquires all classes of a UML model that have the name given by the parameter. Following the style of QVT-R graphical notation, a query is displayed inside a box, where its name shows in the middle top of the box with tag «`Query`», and the *when* and *where* clauses display also in top part of the box, prefixed with "`when=`" and "`where=`", respectively. If there is no *when* clause, the "`where=`" prefix can be omitted. The domain pattern is shown in the middle of the box as a special kind of object diagram, where an object or a property may decorate with a variable name. Note that in the diagrams a variable bound to a property may contain additional quote-characters that are an artefact of the visualization, not string delimiters. The small box with tag «`Parameter`» represents a formal parameter and its type.

2.3 Semantics of the query facility

A query can be called from QVT-R relations, queries and functions. It is used to find parts of models that fulfill given constraints specified in the query. We adapt the QVT-R checking semantics and define the semantics of a query as: for each valid binding of variables of the domain and the parameters, that satisfy the *when* condition, there exists a valid binding of the result variable var_{rst} that satisfies the domain pattern and the *where* condition.

The execution of a query starts from pattern matching. In a similar way as for a QVT-R relation, the matching begins with the root *objectTemp*. If it has been bound to an input parameter, that is to say $\exists\, var_{rt} \cdot var_{rt} \in V_p$, then the search will start from the model element specified by the parameter, otherwise the search will start from the root of whole model. In the matching process, predicates specified in the *when* clause act as additional constraints. If matching model elements are found, variables bound in the domain pattern are instantiated, and then variables defined in the *where* clause calculate their values. Finally, the value of var_{rst} returns as the result of the query.

3 Mapping Queries to XPath

As we have discussed previously, the graphical query facility supports complex object patterns, which may involve sets of objects, non-existence of objects and ordered links. A pattern may be recursively defined. In addition, a pattern might need to navigate and gather information from different places of a model. In order to handle such complex object patterns, pattern matching becomes a core technique to implement the query facility.

Models are normally stored as tree-structured XML files. Another OMG standard related to MOF is the XML Metadata Interchange (XMI) [15], which provides a set of rules that map the graph-based models to/from the tree-based XML documents. Many modeling frameworks, such as the Eclipse Modeling Framework (EMF) [6], provide practical tools to output models as XMI files.

The great advantage of XMI is that the whole range of generic XML processors is available in every popular programming language. In XML technical space, XPath is a suitable tool for the task of pattern matching. As one of the W3C's standard, XPath (XML Path Language) [25] is a declarative language for locating nodes in XML documents at a higher level of abstraction. Complex searching conditions, which may require tree-oriented navigation, and may involve logic, arithmetic, string, and set operators on attributes and nodes, can be adequately expressed using XPath [7]. Thus XPath provides a powerful and convenient way to implement pattern matching.

XPath acts as a sub-language within XSLT. Recommended by W3C, the Extensible Stylesheet Language for Transformations (XSLT) [26] is a declarative rule-based language for transforming XML documents. While XPath expressions can locates elements in the source models, handling these elements and creating new elements in the target model are the job of XSLT. For the detailed syntax and semantics of XPath and XSLT, we refer to [25, 26]. In the following, we discuss only the subset of them that are used in our approach.

3.1 XPath path expression

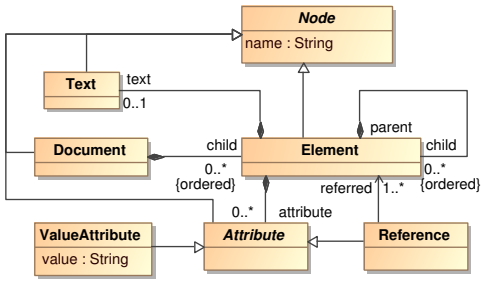

Figure 3: An excerpt of XML data model

XPath is a textual language for representing search expressions in hierarchical XML documents. It works on an abstract, logical tree-structure data model. Fig. 3 shows an excerpt of the data model we used. In this model, every item is a *Node*. The *Document* is the distinguished root node of the tree, denoted by

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

a global variable `xmiXMI`. An *element* owns a set of attributes, each of them either belongs to the set of *ValueAttribues* denoted as $\{\alpha_v\}$, or is a *Reference* in the set $\{\alpha_r\}$. Moreover, we denote the set of parameters mapped from the parameters of the query as $\{\nu_p\}$.

We require an element has an identity attribute $ID \in \{\alpha_v\}$. Thus an attribute α_r contains the *IDs* of other elements. We define a function xmiRefs : Reference $\rightarrow$ *Sequence*(Element) to obtain the referred elements. Moreover, an element has an *xmi label* that consists of the element's name and some particular attributes. For example, a *class* in a UML model is represented in XMI as *<packagedElement xmi:type='uml:Class' .../>*, so the *xmi label* for it is "`packagedElement[@xmi:type='uml:Class']`". We introduce a function xmiLabel() to get the xmi label for each model element defined in the metamodel of the query's typed model. The set of xmi labels for all elements in the XML tree is denoted as $\{\ell\}$.

XPath uses *path expressions* to locate particular elements in XML documents. The syntax of the path expressions is defined by:

⟨Expr⟩	=	⟨NodeExpr⟩ | ⟨AttributeExpr⟩;
⟨NodeExpr⟩	=	'$' (⟨$\nu_p$⟩ | '**xmiXMI**') (('/' | '//') ⟨Step⟩)∗;
⟨Step⟩	=	⟨Axis⟩ ('**[**' ⟨Predicate⟩ '**]**')∗;
⟨Axis⟩	=	⟨ℓ⟩ | '∗' | '**parent()**' | '**xmiRefs(**' ⟨RefExp⟩ '**)**';
⟨AttributeExpr⟩	=	⟨NodeExpr⟩ '/' '**@**' ⟨α_v⟩;
⟨RefExp⟩	=	⟨NodeExpr⟩ '/' '**@**' ⟨α_r⟩;

A path expression Expr is either a NodeExpr which is used to select a sequence of nodes, or an AttributeExpr that gets a set of *value attributes*. The selection starts from the variable representing the context node, which can either be a parameter ν_p, or the global variable `xmiXMI`, and is followed by location steps (Step) separated by a path operator "/" or "//", where the "/" selects only the direct children of the context node, and the "//" finds all its descendants at arbitrary depth. A step is used to navigate from one node to a sequence of related nodes along an Axis, and the resulting sequence can be filtered by a set of Predicates that are enclosed by the brackets "[" and "]".

An axis gives the direction of navigation. Among the possible axes are the *child axis*, which is represented by an xmi label in set $\{\ell\}$, the *parent axis* as function parent(), and the *reference axis* where the function xmiRefs() is used to get the related nodes. The wildcard "*" can be used to match all nodes. A predicate defines arbitrary constraints that the resulted items must satisfy. It can either be a boolean expression selected the items for which the predicate is *true*, or a numeric expression that selects the item at the position given by the value of the expression, for example "`[2]`" selects the second item. To get values of attributes, "@" is prefixed to the name of the attributes.

3.2 Mapping domain pattern to path expressions

We map the domain pattern into XPath path expressions using an algorithm similar to the *depth-first search*. The mapping starts from the *root objectTemp*, which may bind to a *root variable* var_{rt}. We support two kinds of queries:

- A **global model query** matches the entire model to retrieve all results fulfilling the query. If $\mathsf{var}_{rt} \notin V_p \vee \nexists\, \mathsf{var}_{rt}$, that means the *root objectTemp* is not bound to an input parameter, the pattern matching will search over all the model. So the path expression expr will begins as expr="$"+"xmiXMI"+"//"+$\ell_e$, where ℓ_e is the xmi label of the *objectTemp*'s referred class type, `xmiXMI` is the global variable representing the whole model.

- A **local model query** retrieves information specific for a given input model element. In the case, $\mathsf{var}_{rt} \in V_p$, the pattern matching will search under the context of the root variable. Thus we build the path expression expr starting from the variable as expr="$"+$\mathsf{var}_{rt}$.

Then from the root *objectTemp*, say, *o:C*, where *C* is the referenced type of *o*, we go down along the links to obtain other *objectTemps*, for example, *b:B*. Consider the association between class *C* and *B* in the metamodel, two cases arise:

- **Querying down the containment hierarchy**: If the association is composition, that's to say *B* is a child of *C*, we get the label of *B* by $\ell_b = xmiLabel(B)$, then we add to the path expression a *child* step: expr = expr + "/" + ℓ_b. In case of the *forAllSub* feature of *b* is *true*, that means we want to query all descendant nodes, then the path expression becomes : expr = expr + "//" + ℓ_b.

- **Querying up the containment hierarchy**: On the other hands, if *B* is the parent of *C*, we add to the path expression a *parent* step: expr = expr + "/" + "parent()".

- **Querying the normal references**: If the relationship is a normal association with name *asso*, there must be a reference attribute α_r with the same name *asso* in the corresponding XML element of *o*, and the referred elements can be gotten by function xmiRefs(), then the path expression becomes: expr = xmiRefs(expr + "/@" + *asso*).

If the *objectTemp b* is bound to a variable var_b, we construct an XSLT variable declaration as <xsl:variable name=var_b select=expr/>.

Now we consider the *propertyTemps* of the *objectTemp b:B* and the predicates defined in the *when* clause. Let ρ be a *propertyTemp* referred to a property of the class B, there are two situations:

- if ρ is bound to a variable var_ρ, and if there is a predicate in the *when* clause that is related to the variable, we translate the OCL expression of the predicate into an XPath boolean expression xexp by the function OCLtoXPath(), which is specifically designed to map an OCL expression to an XPath expression. Then we have expr = expr + "[" + xexp + "]". Otherwise if there is no such a predicate, the expr will remain unchanged. Consequently an XSLT variable is defined as <xsl:variable name=var_ρ select=expr+"/@"+ρ/>.

- if ρ is bound to an OCL expression oexp, we get xexp= OCLtoXPath(oexp). Thus, we have expr = expr + "[" + xexp + "]". In particular, oexp could be a regular expression in our approach.

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

In this way, we traverse all linked *templateExps* and their *propertyTemps* to construct path expressions, and instantiate every variable we meet on the way.

3.3 Generating XSLT functions for queries

We create an XSLT user-defined function for a query using the <xsl:function> instruction. For each parameter of the query, we generate a corresponding XSLT parameter using the <xsl:param> construct. Inside the function body, we collect all variable definitions instantiated from the domain pattern. Then, for each predicate in the *where* clause that takes the form of ⟨var⟩ = ⟨oexp⟩, where var is a free variable, and oexp is an OCL expression, we map the oexp into an XPath expression xexp using the function OCLtoXPath(), and generate an XSLT variable declaration for the predicate as <xsl:variable name=var select=xexp/>. After that, a <xsl:sequence> instruction will be used to return the value of the *result* variable. Furthermore, an OCL *tuple* expression can be used to return multiple values.

As an example, the following XSLT function is generated from the example query shown in Fig. 2. Here "my:" is the name space we give to XSLT user-defined functions:

```
<xsl:function name="my:GetClassByName">
  <xsl:param name="nm"/>
  <xsl:variable name="result" select="$xmiXMI//packagedElement
                                [@xmi:type='uml:Class'][@name=$nm]"/>
  <xsl:variable name="cnm" select="$result/@name"/>
  <xsl:sequence select="$result"/>
</xsl:function>
```

3.4 Tool implementation

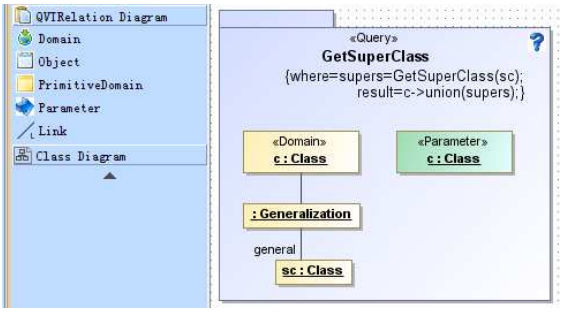

Figure 4: Toolbar for QVT-R graphical query

We extend the QVTR-XSLT tool to support the graphical query facility. Based on the metamodel discussed in Section 2.2, we propose a UML profile that defines stereotypes and tagged values to implement the abstract syntax. The profile has been added to the profiles of the QVTR-XSLT tool, thus the model query facility is supported by the graphical editor of the tool. The current version of the editor is built on top of *MagicDraw UML*, a popular UML modeling and CASE tool.

As shown in Fig. 4, the toolbar of the editor contains the necessary buttons to graphically design queries, such as *Domain*, *Object*, *Parameter*, and *Link*. We can select an element from user-defined metamodels or UML standard metamodel in a pop-up window, and drop it into the diagram. The *when* and *where* clauses can be input in a text editing window. In addition, we have defined a set of well-formedness constraints for queries using OCL. The design of a query can be validated by just pressing the "*check*" button of the editor. The elements that violate the OCL constraints are reported in a result window, and are also shown in the diagram surrounded by red boxes.

We also add support for the query facility into the QVTR-XSLT code generator. The code generator reads in the XML file exported from the graphical editor, traverses and analyzes the XML representations of the queries, parses the OCL expressions, and generates an XSLT function for each query. The generated XSLT program contains only pure XSLT code without extensions, so it can be directly executed with any XSLT processor and from most major programming languages.

4 Case Study

We test the graphical query tool with a suite of popular queries from [4, 22] on the model of UML Superstructure version 2.4.1, a typical EMOF model which includes 11 packages, 341 classes, and in total about 9500 model elements stored in an XMI file of 1.7 MB in size. We design a QVT-R transformation to collect and execute these queries. This transformation takes the UML superstructure model as the source model, and outputs a HTML file to show the querying results.

Using the QVTR-XSLT tool, we configure the transformation with appropriate metamodels: a slight variant of EMOF model as the source metamodel, and a simple HTML model as the target metamodel. The transformation includes two relations: a top-level relation *QueryUML* that starts the transformation and invokes the queries, and a relation *Display* puts the result of a query into the HTML model. Fig. 5 shows the outline of the transformation model.

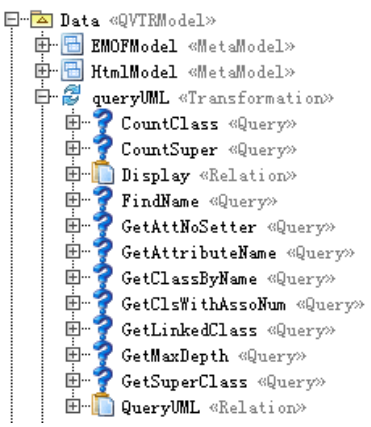

Figure 5: The transformation model of case study

4.1 The example queries

The queries we discuss here are frequently used for model analysis and are of different complexity and structure. The first three

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

search the entire model to retrieve the results, while the following two are local queries.

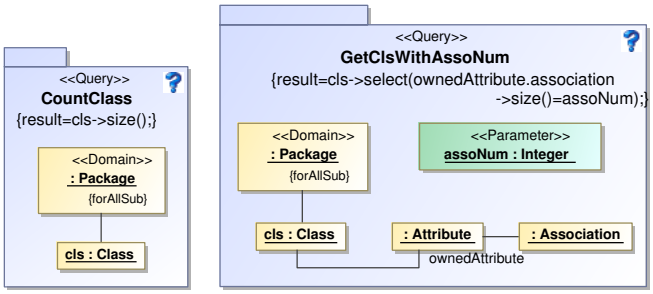

Figure 6: Count number of classes; classes with given number of associations

Count elements of a given type: The first query is to retrieve the number of overall elements of a given type, say, classes, in a model. The query *CountClass* shown in Fig. 6 (left) counts the number of classes contained in all packages. We set the `forAllsub` feature of the `Package` to *true*. The OCL function `size()` is used to count the number, then the result is returned by variable `result`. The code generator produces the following XSLT function for the query:

```
<xsl:function name="my:CountClass">
  <xsl:variable name="cls" select="$xmiXMI//packagedElement
                                   [@xmi:type='uml:Class']"/>
  <xsl:variable name="result" select="count($cls)"/>
  <xsl:sequence select="$result"/>
</xsl:function>
```

All classes with a given number of associations: The query *GetClsWithAssoNum* shown in Fig. 6 (right) obtains a list of classes with a given number of associations. The parameter `assoNum` passes the number to the query. The OCL expression in the *where* clause uses the *select* operation to choose the classes that have the number of outgoing associations. With such a query one could find the classes which have too many or too few associations. A reference definition for the query using OCL is given in [4]:

```
context Class
  def:getAssociationsFromClass():Set(Association)=
    self.ownedElement→asSet()→select(a|a.oclIsTypeOf(Property)).
        oclAsType(Property).association→select(a|not(a=null))→asSet()
context Package
  def:getAllClasses():Set(Class)=
      self.packagedElement→asSet()→select(t|t.oclIsKindOf(Class)).
          oclAsType(Class)→asSet()
  def:getClassesWithNumberOfAssociations(NoA:Integer):Set(Class)=
     self.getAllClasses()→asSet()→select(c|c.getAssociationsFromClass()→
         size()=NoA)
```

It is obvious the OCL specification is more complicated, and relatively difficult to understand, comparing to its graphical representation in our query facility.

Maximal depth of class inheritance: The depth of the class inheritance is a typical metric in object oriented modeling or programming. The OCL specification for this query given in [4] consists of three user-defined operations, in total 26 lines of complex OCL code.

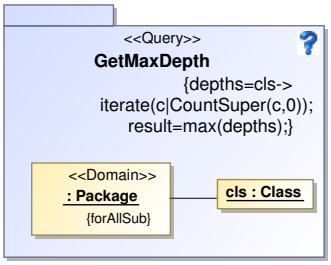

Figure 7: Maximal depth of inheritance

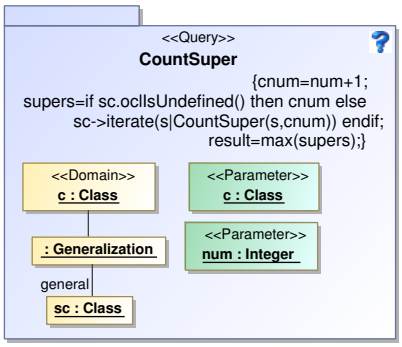

Figure 8: Number of super classes

Fig. 7 shows the query *GetMaxDepth* specified in the graphical notation. For each class `c` in the set `cls`, the query *CountSuper* (see Fig. 8) is iteratively called to get a collect of depth numbers. The query *CountSuper* again recursively calls itself to count the number of super classes. A notable feature of our query facility is that a set of standard functions defined in XPath can be mixed use with OCL operations. As an illustration, we directly use XPath function `max` here to get the maximal one in a collection instead of its OCL counterpart.

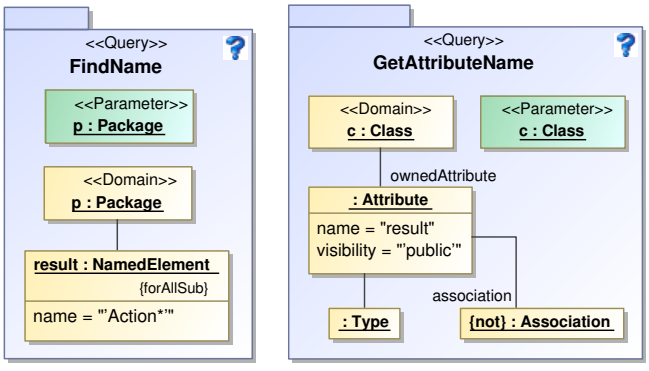

Figure 9: Find elements by name; getting attribute names

Textual pattern matching: One of the most frequent queries is to do a text search for a given string over a model. However, regular expressions are still not supported in the OCL specification. An example query might be "find all model elements whose names begins with 'Action' in a package". Fig. 9 (left) depicts this query. In the EMOF model, the NamedElement is an abstract

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

Table 1: Results of the first three queries

Name	Explanation	Result	Time
CountClass	count all classes of the model	331	78ms
GetClsWithAssoNum(3)	classes with a given number of asso.	30	65ms
GetMaxDepth	maximal depth of class inheritance	10	15s

class. Since there is a wildcard "*" in the string bound to *propertyTemp* name, we take the string as a regular expression. In the implementation, XPath function matches() is used to test whether the names matches the regular expression:

```
<xsl:function name="my:FindName">
  <xsl:param name="p"/>
  <xsl:variable name="result" select="$p//*[matches(@name,'Action*')]"/>
  <xsl:sequence select="$result"/>
</xsl:function>
```

Structural matching for existing and non-existing elements: The query *GetAttributeName* shown in Fig. 9 (right) gets a set of names of primitive attributes in a given class. The query selects a primitive attribute by (1) it owns an element of *Type*; (2) it does not have an *association* link; (3) its visibility is *public*.

4.2 Experiments

Using our *transformation runner*, we execute the queries on the model of the UML superstructure in a laptop of Intel M330 2.13 GHz CPU, 3 GB memory, and running Windows 7 home version. The results of the first three queries are shown in Table 1. For the query *GetClsWithAssoNum*, we use 3 as the value for the parameter, and the table only shows the number of classes found. As depicted in the table, there are in total 331 classes in the model, 30 of them have 3 associations to other classes. The maximal depth of class inheritance is 10.

The execution time includes loading and saving model files from/to the hard disk. We can see from the results the first two queries works well, and the straightforward algorithm of the third query needs to be optimized.

5 Related Work

A model query facility is an important part of model transformations. Medini QVT [12] and ModelMorf [13] are two of the most widely used tools for supporting QVT Relations. But even the standard graphical notation are not supported by these two tools, not to mention supporting queries in graphical formats. However, almost all QVT-like languages supports only the textual notation, such as ATL (Atlas Transformation Language), SmartQVT and Borland Together.

Today, OCL is the *de-facto* standard language for querying models. The work likes [1] tries to extend OCL with the help of relational algebra. *Model Manipulation Toolkit* (MoMaT) [21] and *Logical Query Facility* (LQF) [22] use Prolog to query UML models. Moreover, the aforementioned model transformation languages all provide facilities for model querying. In general, a textual query language does not blend well with a visual modeling language, and formalisms like first order logic are not acceptable to domain modelers [23].

Visual model querying attracts the attentions of many researchers. Visual OCL [3] is an attempt to visualize OCL expressions by providing graphical symbols for all OCL keywords. In consequence, users are still confronted with the full load of OCL complexity. On the other hands, graph matching theory can be used as a foundation to graph query languages, which are normally used in conjunction with graph transformation languages. Notable examples are AGG (Attributed Graph Grammars), Fujaba, MOFLON and ATOM^3. Moreover, XML-GL [5] is a graphical query language for XML data with semantics based on graph matching. In this way, querying can be made visual as the process of locating a subgraph in a graph. However, all these languages define a query on the bases of model's abstract syntax.

Model querying can be based on the concrete syntax of the model language. In [19], *Join Point Designation Diagrams* (JPDD) are used to represent queries graphically in terms of user model entities and user model properties, rather than meta model entities. The *Query Models* (QM) [20] is another similar work, where a variant of the host language is used to express additional constraints, and then translated into OCL. The BP-QL language [2] allows visually querying business processes based on the intuitive model of BPs. The more recent VMQL [23] also used the respective modeling language of the source model as the query language. The selection criteria are expressed as tags attached to the model elements.

Different from the above approaches, our purpose is to extend the graphical notation of QVT-R to support model querying, on the basis of the check-only semantics and pattern matching of QVT Relation. The queries are specified on the level of abstract syntax. We provide an editor for designing the graphical queries, and we also produce corresponding XSLT programs to execute them.

6 Conclusion

In this paper, we report on our work on model querying using the graphical notation of QVT Relations. By adapting the graphical syntax of QVT-R, we present the concrete and abstract syntax of the graphical query facility, and its semantics is built upon the checking semantics and pattern matching of QVT-R. We propose an approach to map the selection criteria of the queries into XPath expressions, thus a query can be implemented as an XSLT function to facilitate its execution. As a standard model transformation language, QVT-R relies only on OCL to query models, even in its graphical syntax. Our work complements QVT Relations by providing it with a concise, easy to understand, and yet expres-

DOI: 10.1145/2237796.2237808 http://doi.acm.org/10.1145/2237796.2237808

sive graphical query facility. Since we allow queries consisting of only a textual part, our query facility is at least as expressive as pure textual OCL.

We have developed a tool to support the graphical notation of QVT-R and the query facility. With the graphical editor of the tool, we can specify queries using the graphical syntax. The code generator of the tool can automatically translates the queries into executable XSLT programs. We have applied the tool for designing model transformations in the developing of rCOS Modeler, a CASE tool for component-based model-driven development [8]. One of the cases is the transformation from object-oriented to component-based models [9], in where more than 30 queries are defined. We also participated the fifth transformation tool contest (TTC2011) using the tool [11].

The tool with the case study is available from `http://rcos.iist.unu.edu/qvttoxslt/qvtquery-example.html`. For the graphical editor, MagicDraw UML v16.5 or later is required.

Acknowledgements Partially supported by the ARV, GAVES and SAFEHR grants of the Macau Science and Technology Development Fund, and the Guizhou International Scientific Cooperation Project G[2011]7023 and GY[2010]3033.

References

[1] D. Akehurst and B. Bordbar. On Querying UML Data Models with OCL. *«UML» 2001-The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of Lecture Notes in Computer Science:91–103, Springer, 2001.

[2] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes with BP-QL. *Information Systems*, 33(6):477–507, 2008.

[3] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL using Collaborations. *«UML» 2001-The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of Lecture Notes in Computer Science:257–271, Springer, 2001.

[4] J. Chimia-Opoka, M. Felderer, C. Lenz, and C. Lange. Querying UML models using OCL and Prolog: A performance study. In *Software Testing Verification and Validation Workshop, ICSTW'08*, pages 81–88. IEEE, 2008.

[5] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over XML data. *ACM Transactions on Information Systems (TOIS)*, 19(4):371–430, 2001.

[6] Eclipse Foundation. Eclipse Modelling Framework. http://www.eclipse.org/modeling/emf/.

[7] W. Janssen, A. Korlyukov, and J. Van den Bussche. On the tree-transformation power of XSLT. *Acta Informatica*, 43(6):371–393, 2007.

[8] W. Ke, X. Li, Z. Liu, and V. Stolz. rCOS: a formal model-driven engineering method for component-based software. *Front. Comput. Sci. in China*, 6(1), 2012.

[9] D. Li, X. Li, Z. Liu, and V. Stolz. Interactive Transformations from Object-Oriented Models to Component-Based Models. In *Proc. of Formal Aspects of Component Software (FACS'11)*. University of Oslo, Norway, July 2011.

[10] D. Li, X. Li, and V. Stolz. QVT-based model transformation using XSLT. *SIGSOFT Softw. Eng. Notes*, 36:1–8, Jan. 2011.

[11] D. Li, X. Li, and V. Stolz. Solving the TTC 2011 Compiler Optimization Case with QVTR-XSLT. In Proceedings Fifth *Transformation Tool Contest*, Zürich, Switzerland, volume 74 of *Electronic Proceedings in Theoretical Computer Science*, pages 54–69, 2011.

[12] medini QVT, ikv++ Technologies, `http://projects.ikv.de/qvt`.

[13] ModelMorf, A Model Transformer, `http://www.tcs-trddc.com/`.

[14] Object Management Group. Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.0/PDF, Jan. 2006.

[15] Object Management Group. MOF 2.0/XMI Mapping, v2.1.1. http://www.omg.org/spec/XMI/2.1.1/PDF, 2007.

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

[17] Object Management Group. Unified Modeling Language: Superstructure, version 2.4.1, August 2011.

[18] Object Management Group. OCL Specification Version 2.3.1, January 2012.

[19] D. Stein, S. Hanenberg, and R. Unland. A graphical notation to specify model queries for MDA transformations on UML models. In *MDAFA*, volume 3599 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2004.

[20] D. Stein, S. Hanenberg, and R. Unland. Query models. In *«UML» 2004-The Unified Modeling Language. Modelling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.

[21] H. Störrle. A PROLOG-based Approach to Representing and Querying Software Engineering Models. In *Proceedings of the VLL 2007 workshop on Visual Languages and Logic, Idaho, USA, September*, pages 71–83, 2007.

[22] H. Störrle. A Logical Model Query Interface. *In Proceedings of the VLL 2009 workshop on Visual Languages and Logic, Dalhousie, Canada, September*, pages 18–36, 2009.

[23] H. Störrle. VMQL: A visual language for ad-hoc model querying. *Journal of Visual Languages & Computing*, 22:3–29, 2010.

[24] E. Willink. On Challenges for a Graphical Transformation Notation and the UMLX Approach. *Electronic Notes in Theoretical Computer Science*, 211:171–179, 2008.

[25] WWW Consortium. XML Path Language (XPath) 2.0, W3C Recommendation. http://www.w3.org/TR/2007/REC-xpath20-20070123/, January 2007.

[26] WWW Consortium. XSL Transformations (XSLT) Version 2.0, W3C Recommendation, January 2007.