QVT-Based Model Transformation Using XSLT

Dan Li * Xiaoshan Li
Faculty of Science and Technology
University of Macau, China
e-mail: lidan@iist.unu.edu xsl@umac.mo

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

Abstract

Model transformations are one of the key technologies in model-based development. The graphical notation of relational QVT provides a concise, intuitive way to specify transformations. But this notation is not directly applicable for practitioners because of the lack of tool support. On the other hand, XSLT is a common and powerful language for XML transformations, but not suitable for directly programming transformations of semantically complex models due to its low level syntax. We combine the best of both techniques by using QVT graphical notation to specify a transformation as a set of QVT relations, and implementing each relation as an XSLT rule template. A prototype tool with a QVT graphical editor and an automatic XSLT program generator has been developed to support the approach.

Keywords:
Model transformations, QVT Relations, Graphical notation, XSLT

1 Introduction

Model transformation is the core of model-driven development [11]. Based on a problem-specific model description (in machine-readable format), further development tasks are carried out. This model serves as specification, documentation, and for communication between the different stakeholders in the software development process. This usually includes database and graphical user interface design, and application programming. Instead of manually doing those tasks, it would be beneficial to automate them as far as possible. Especially in the case of changes to the domain model, these changes need to be propagated appropriately.

This can only be successful if the artifacts in the different stages or phases of the development can be processed automatically, effectively treating them as models themselves. From the domain model as input, we would like to calculate as output its corresponding representation suitable to be used in the further development process. This could be a specification in a data definition language, source code in a particular target programming language for an object-relational mapping, or documentation. Such a calculation, or transformation, can only be executed successfully if the input model is well-defined, that is, if it conforms to a metamodel. Likewise, a transformation can only make sense if its output domain, the target metamodel, is known.

Just as a transformation translates a particular input to an output model, we can generalize this to a transformation language that allows us to express arbitrary transformations between instances of two given source and target metamodels. This transformation language would obviate the need for a plethora of transformation languages for specific problem domains. It would certainly improve maintainability of transformations implemented in conventional programming languages such as Java, C/C++, scripting languages, or XML processors.

The Query/View/Transformation language (QVT) [15] standardized by the Object Management Group (OMG), together with the supporting MetaObject Facility (MOF) [12] standard defining the notation for metamodels (MOF thus being a meta-metamodel), addresses this need for a common transformation language.

QVT comes in two flavours: the QVT Operations language, for writing imperative model transformations, and the QVT Relations language, for specifying them declaratively. From the perspective of theoretical computer science, declarative transformations are generally preferred, since they are usually more concise, and make reasoning about them easier.

The QVT Relations language has another distinguishing feature: it comes with a corresponding graphical notation. Structural matching is specified similar to object-diagrams in the Unified Modeling Language (UML) [16]. Semantic constraints and local definitions can be introduced by the Object Constraint Language (OCL) [13]. As the "graphical vocabulary" of the transformation language is restricted by the metamodels, this should make the language easier to use for novice users, as a wide variety of syntax errors are immediately ruled out [19].

Alas, tool support for QVT Relations is still immature [8], presumably because of the complex requirements of the runtime environment that will execute the transformations: many of the features that are implicit in a declarative language, require elaborate routines for iteration, pattern matching, and back-tracking in an implementation. A full implementation of the QVT Relations language is certainly an ambitious undertaking, with its features apart from "plain" model transformation, such as checking conformance of two given models with regard to a relation, or reverse execution from target back to source model.

In our approach, we use the expressiveness of another declarative language for manipulating structured data to implement a suitable subset of the QVT Relations language, mainly supporting transformations from source to target model.

As the Eclipse Modeling Framework (EMF) [5] has evolved to one of the major standards for manipulating models, and EMF models are normally stored in the form of XML Metadata Interchange (XMI) [14], a well-studied and standardized technique for mapping graph-based models to and from tree-based XML documents, using an XML processor to manipulate the concrete representation of a model comes as a natural choice. Of course, we need to ascertain that the necessary semantic features of the metamodel and the transformation language can be adequately sup-

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

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

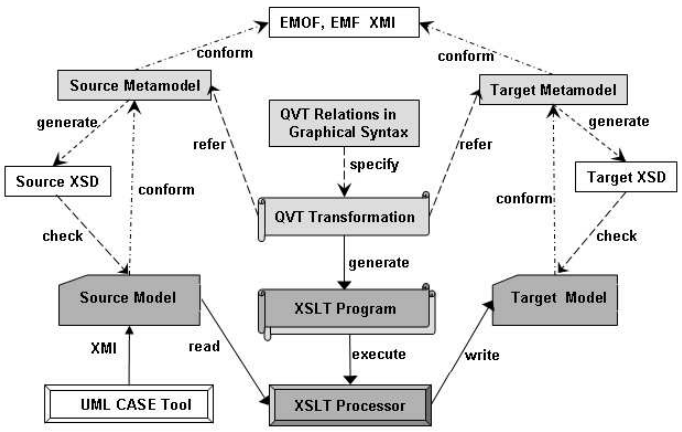

Figure 1: Approach overview

ported. Having the characteristic of both declarative and imperative languages, XSL Transformations (XSLT) [22], a standard language recommended by the W3C for transforming XML documents, is powerful in defining rules, pattern matching and processing (tree) structured data [7]. Thus, it becomes our transformation implementation language.

We take metamodels prepared with a CASE tool such as MagicDraw [10] or IBM's Rational, specify a particular transformation using QVT graphical notation also in a CASE tool, and save the transformation as an XML file. Then, we process the XML file with an XSLT processor, which will generate the *actual* transformation as an XSLT program. The source models are prepared with CASE tools and exported as EMF XMI files. The generated XSLT program is then executed on the source models, and will in turn generate the target models. Additionally, we generate XML Schemas (XSD) [20] from the metamodels to validate the models. This process, the various artifacts and their relation to each other are shown in Fig. 1. We have highlighted both phases (generation of XSLT from QVT in light colour, and running the generated XSLT to execute the transformation in dark colour) separately.

We have developed a prototype tool to support our approach. In this tool, we specify QVT transformation rules using the graphical notation, and automatically generate XSLT programs for the transformations. Since currently there exists no practical tool support for QVT Relations in graphical notation, our approach and the tool attempt to bring a large subset of the language to practice.

The paper is organized as follows. We start by briefly explaining in Section 2 the basics of QVT Relations and XSLT, and give a simple comparison between them. In Section 3 we illustrate the major steps and features of our approach. We describe our tool and apply it to a case study in Section 4. After that, we discuss related work and present our conclusions and future work in Section 5.

2 Basic Concepts

In this section we first discuss QVT Relations on an abstract level with a short example, and then introduce the necessary concepts of XSLT which we need in the translation process.

QVT Relations

QVT Relations is a declarative model transformation language proposed by the OMG as part of the MOF Query/View/Transformations (QVT) standard [15]. The QVT Relations language specifies a *transformation* as a set of *relations* between candidate model elements. QVT Relations has both textual and graphical notation. In graphical syntax, the basic idea is to specify how two types of object diagrams, called *domain patterns*, relate to each other. In addition, a transformation may own *keys*, that uniquely identify an instance of model element, and *functions*, which are side-effect-free operations. Fig. 2 depicts an example QVT relation between instances of *Class* and *Table* (*ClassToTable*). And its textual counterpart is shown for comparison in Listing 1. This example is adapted from Annex A of [15].

A *relation* is defined by two or more *checkonly* or *enforce* relation *domains*: an enforce domain specifies the target domain, while the checkonly domain act as the source. The target model may be empty at the beginning of the transformation. In case of a relation that only has one *checkonly source domain* and one *enforced target domain*, we assume the left part of the diagram is the source domain, and the right part the target domain. Each domain is defined by a *domain pattern* which is composed of a set of linked *template expressions*, that may be *object template expressions* or *collection template expressions*. The former refers to the classes of the corresponding metamodel. The *object template expression* with tag ≪*domain*≫ is called *root variable*. It is the root element of the pattern and serves as the relation's parameter. The links between *template expressions* must conform to the associations between their typed classes in the metamodel. An *object template expression* (e.g. the box labelled with "c:Class" in Fig. 2) may own a set of *property template items* (e.g. "name=cn" in that box). Each of them binds a *property* of the typed class to a *variable* or an *OCL expression*.

A relation may define a pair of optional *when*- and *where*-clauses which consist of arbitrary *predicates*, that are specified by OCL expressions which may contain references to variables of the domain patterns. 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 pattern itself, that must be satisfied by all pattern elements in the relation. In addition, new variables can be declared in the *where*-clause and referenced by *predicates* in both *when*- and *where*-clauses. A *relation call expression* in a clause specifies the invocation of a subsequent relation. A relation may also have *primitive domains* in order to pass parameters of primitive types (numeric values, strings) between the relations. Furthermore, a relation is either designated as a *top-level* relation, or a *non-top-level* relation. A *top-level* relation is invoked from the transformation framework. Non-top-level relations are invoked through *when*- and *where*-clauses from other relations.

When a relation is executed, the source domain pattern is searched in the source model by way of *pattern matching*: in the example, the pattern matching starts from the root variable "p" (source domain) of type *Package*. There are two cases. If the variable "p" is a free variable, such as in a top-level relation, and it is also not bound by a relation call expression in the *when*-clause, the

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

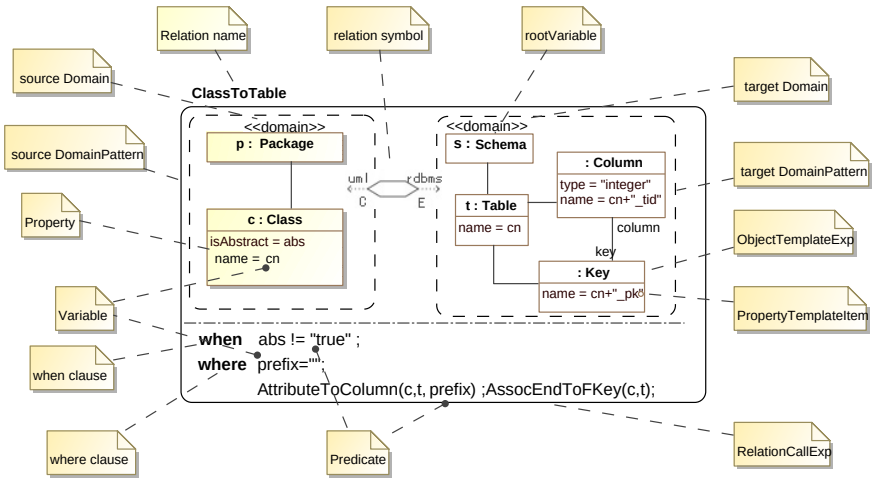

Figure 2: Example QVT relation in graphical notation

```
relation ClassToTable {
    cn, prefix: String;

    checkonly domain uml p:Package {
          class = c:Class {isAbstract!=true, name=cn}
    };
    enforce domain rdbms s:Schema {
          table = t:Table{name=cn,
                column=cl:Column {name=cn+'_tid',
                                  type='Integer'},
                key=k:Key {name=cn+'_pk', column=cl}}
    };
    where {
      prefix = '';
      AttributeToColumn(c, t, prefix); AssocEndToKey(c,t);
    }
}
```

Listing 1: QVT relation in textual notation

pattern matching searches the whole source model for a *Package* "p". Such an instance may be found nested inside other elements in the source model. Otherwise, if the variable has already been bound to a particular "p" by the invoker of the current relation, all child elements of "p" are searched to locate elements of type *Class* which satisfy the *when* condition, that the *isAbstract* property is not *true* (*isAbstract=abs*, *abs!="true"*). If a matched model element is found, the variable "c" is bound to the element, and the value of the element's *name* property is bound to the variable "cn" (*name=cn*).

The target domain pattern acts as a template to create corresponding objects and links in the target model. In the example of Fig. 2, under the context of root variable "s" (target domain) of type *Schema*, an element "t" of type *Table* is created and its *name* property gets assigned the value of the bound variable "cn". In the same way, the model elements of type *Column* and *Key* are created, their properties are set, and the links *column* and *key* among them are established as specified by the relationship in the domain pattern. Finally, in the *where*-clause, the relations *AttributeToColumn* and *AssocEndToFKey* are invoked using variables "c" and "t" as the arguments (*AttributeToColumn* needs one more argument, "prefix", an empty string as the *primitive domain*).

XSLT

The Extensible Stylesheet Language for Transformations (XSLT, recommended by W3C) [22] is a flexible declarative rule-based programming language for transforming XML. XSLT provides a powerful capability that enables rule declaration, transformation, navigation, and presentation of XML content, which makes it a potential tool for QVT implementation.

An XSLT program is called a *stylesheet* that consists of a set of *rule templates*. Each template matches one or more elements in the source model, and defines the actions to take and how to produce the output when a match is found. The result of processing all of the elements in the source model generates the target model. An example of an XSLT rule template that implements the QVT relation *ClassToTable* is depicted in Listing 2.

```
1  <!-- pattern matching -->
2  <xsl:template match="packagedElement[@xmi:type='uml:Class'
3        and @isAbstract!='true']" mode="ClassToTable">
4    <!-- variable binding -->
5    <xsl:variable name="c" select="current()"/>
6    <xsl:variable name="cn" select="current()/@name"/>
7    <xsl:variable name="prefix" select="''"/>
8    <!-- create target model -->
9    <xsl:element name="Table">
10     <xsl:attribute name="name" select="$cn"/>
11     <xsl:element name="Key">
12       <xsl:attribute name="name" select="concat($cn,'_pk')"/>
13       <xsl:attribute name="column" select="concat($cn,'_tid')"/>
14     </xsl:element>
15     <xsl:element name="Column">
16       <xsl:attribute name="name" select="concat($cn,'_tid')"/>
17       <xsl:attribute name="type" select="'Integer'"/>
18       <xsl:attribute name="key" select="concat($cn,'_pk')"/>
19     </xsl:element>
20     <!-- invoke rule templates -->
21     <xsl:apply-templates select="$c" mode="AttributeToColumn">
22           <xsl:with-param name="prefix" select="$prefix"/>
23     </xsl:apply-templates>
24     <xsl:apply-templates mode="AssocEndToFKey" select="$c"/>
25   </xsl:element>
26 </xsl:template>
```

Listing 2: An example of XSLT rule template

An XSLT template is endowed with a *match* property which has a *selection pattern* as its value, along with a *mode* property which can be regarded as the template's name. Inside its body, a set of variables are declared and obtain their values using <xsl:variable> instructions. A template contains also a collection of hierarchical <xsl:element> and <xsl:attributes> instructions to construct new elements and their attributes. A template can invoke another template through <xsl:apply-templates> instructions, and pass parameters to it.

The *selection pattern* is an XPath [21] *path expression* that locates the elements in a source model to which the template applies. A path expression matches a sequence of elements in the tree of a source model by following a path in a series of steps from a given

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

starting point. The starting point may be either the context element, or the root element of the model. Steps are defined in terms of *node expressions* or *attribute expressions*. Each step takes as its starting point an element, and from this starting point, matches other elements. E.g., the path expression in lines 2–3 of Listing 2 selects all children of type *Class* (as a *packagedElement* with an *xmi:type* attribute of value "uml:Class" in EMF XMI) of the current context element "p", that do not have an *isAbstract* attribute with value "true". All matched elements will be processed by the template one by one.

Besides the *source context* which denotes an element of the source model under which the *current processing elements* are selected, there is also a *target context*, that is, an existing element of the target model. The newly created elements will be children of that *target context* element.

In addition, XSLT provides an index mechanism <xsl:key> for rapid searching of a particular model element through its attributes. This is particularly useful for EMF models, since they use the internal XMI ID as the identifier of a model element, and most elements in an EMF model have interdependencies and cross-references. XSLT also supports user-defined functions through <xsl:function> that can be invoked from any XPath expression.

Execution of an XSLT stylesheet is by recursive application of an individual template to whole source model or a selected part of the model. A template is selected by its mode property and matching condition (selection pattern). Once a template has been matched, the variables defined in the template get their values, and the <xsl:element> and <xsl:attributes> instructions are executed to build a fragment of the target model using these variables. Finally, the successive rule templates are invoked.

As we have seen in the previous part of this section, there are apparent similarities and analogies between QVT Relations and XSLT. Therefore, a mapping can be specified from QVT Relations to XSLT. Details of implementation will be discussed in next section. Table 1 summarizes the correspondences between QVT Relations and XSLT.

Table 1: Comparison of QVT Relations and XSLT

QVT Relations	XSLT
MOF metamodel	XML schema
source model	input XML file
target model	result XML file
transformation	stylesheet
relation	rule template
source domain pattern	selection pattern
target domain pattern	construct instruction
primitive domain	template parameter
source domain	source context element
target domain	target context element
objectTemplateExp in source domain pattern	node expression in selection pattern & <xsl:variable>
propertyTemplateItem in source domain pattern	attribute expression in selection pattern & <xsl:variable>
objectTemplateExp in target domain pattern	<xsl:element>
propertyTemplateItem in target domain pattern	<xsl:attribute>
OCL expression	XPath expression
predicate in when clause	path expression
predicate in where clause	<xsl:variable>
function	<xsl:function>
key	<xsl:key>
not template	not() function

3 XSLT-based Approach

Our approach addresses the model transformation task in the following principal steps.

1. Given the transformation as a model with its set of QVT relations in graphical notation, translate the transformation into XSLT by
   - creating the XML Schemas for source and target metamodel;
   - creating an XSLT stylesheet corresponding to the QVT transformation;
   - defining keys and functions for the transformation;
   - implementing each QVT relation as an XSLT rule template.
2. Given the source model as an EMF XMI file, execute and validate the transformation by
   - validating the source model against the source XML Schema;
   - performing the transformation by executing the XSLT stylesheet in an XSLT processor, which reads in the source XMI file and outputs the target model in XML format, and
   - validating the target model against the target XML Schema.

Naturally, once a QVT transformation has been translated to an XSLT stylesheet in step 1, it can directly be run repeatedly as in step 2 on different input models (conforming to the same source metamodel). In the following, we shall focus on implementing a QVT relation through an XSLT rule template, as it is the central phase of our approach.

Implementation of QVT relations

The whole procedure of generating an XSLT rule template from a QVT relation can be broken down into the following four steps:

- **Map the source domain pattern and the *when*-clause into the selection pattern**

  We map the following three kinds of QVT expressions into *path expressions* of the *selection pattern* of an XSLT rule template:

  – object template expressions;

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

– property template items that bind a property to a literal expression;

– predicates in the *when*-clause, that impose restrictions on the object templates and property templates.

We use an algorithm similar to depth-first search to generate the path expression. The mapping starts from the root variable of a source domain pattern, and goes down along the links. When an *object template expression* is met, we add a *node expression* to the path expression, along with constraints retrieved from predicates in the *when*-clause concerning the object and its attributes. For each *property template item* of the object, we add an *attribute expression* to the path expression.

- **Create variable declarations from the *source domain pattern* and the *where*-clause**

  Variables may be declared in the source domain pattern and the *where*-clause. We start from the *domain* element, along the links down to build a path expression, and instantiate every variable (as the name of an object template or a property template item) we meet with the path expression. In this way, a variable declared as the name of an object template expression (e.g. "c:Class") is bound to an element of the source model, and a variable declared in a property template item is bound to the value of the property (e.g. "name = cn").

  For each predicate in a *where*-clause that takes the form *var = Oexp*, where *var* is a variable name and *Oexp* is an OCL expression, we parse the OCL expression and map it into an XPath expression *Xexp*. Then this variable is defined through the XSLT variable instruction <xsl:variable name="*var*" select="*Xexp*"/>.

  Since XSLT requires that variables have to be defined before use, all variables from the source domain pattern and *where*-clause are sorted to account for dependencies between them.

- **Generate construction instructions from the *target domain pattern***

  As mentioned previously, a target domain pattern acts as a template to create corresponding objects and links in the target model. If the relation is a top-level relation, the creation process starts building a new target model using the root variable as its root element. Otherwise, the creation process starts from the pattern element directly linked to the root variable, and a new fragment of the target model is created as the child of the root variable. In this process, an object template expression *o:C* turns into an XSLT *element* instruction <xsl:element name="*C*"/>; a property template item of the form *"p=Oexp"*, where *"Oexp"* is an OCL expression, becomes an attribute instruction <xsl:attribute name="p" select="Xexp"/>, where *Xexp* is an XPath expression mapped from *Oexp*. Other linked object template expressions, depending on the associations in the metamodel, become contained *elements* (for composition relationships) or key referenced *attributes* (for normal associations).

- **Translate *relation call expressions* in the *where*-clause into template invoking instructions**

  In our approach, only two non-primitive domains are allowed, one source and one target domain. For a relation call expression *Rcall(svar,tvar,pvar$_1$,...,pvar$_n$)*, where *Rcall* is the relation name, and *svar*, *tvar*, *pvar*$_1$ to *pvar*$_n$ represent the source, target and primitive domains of the relation *Rcall*, respectively, we translate the relation call expression into an XSLT instruction:

  <**xsl:apply−templates** select="$svar" mode="Rcall">
  <**xsl:with−param** name="pvar$_1$" select="$pvar$_1$"/>
  ...
  <**xsl:with−param** name="pvar$_n$" select="$pvar$_n$"/>
  </<**xsl:apply−templates**>

  where *svar* is a variable already defined in the rule template that bound to an object template expression, and *pvar*$_1$ to *pvar*$_n$ are variables of primitive types or literal expressions. This *apply-templates* instruction will be put as a child of the current target domain pattern element *tvar*. This location serves as the target domain of the relation *Rcall*, from here the relation *Rcall* will subsequently create the substructures of the target model.

Features of the approach

Our approach has the following characteristics:

- XML has rapidly emerged as a dominant standard for data representation and exchange. As one of the most successful XML technologies, XSLT (along with XPath) has many industrial-strength implementations, such as *Saxon*, *Xalan* and Microsoft *MSXML*, and had been widely used in data-intensive applications, e.g., querying XML databases. Stored as XMI files, even large-scale models can be efficiently processed by XSLT and translated to target XML files.

- Both QVT Relations and XSLT are declarative languages. In the procedure of implementing QVT Relations as XSLT, we can focus on how to express the semantics of QVT relations with XSLT constructs, rather than on realization details, such as arranging the iteration and search-order, that have to be considered in an imperative or operational language.

- Even XSLT is widely employed by many people in their daily work, coding in XSLT is still quite a challenge because its "low level" syntax. As the final product of our tool is a piece of XSLT which can be executed without any other dependencies except for an XSLT processor, it can be integrated into other development environments that can profit from model transformations, without the overhead of a complex runtime system.

Next we will briefly discuss the restrictions of our approach.

- **Relation domains**: QVT Relations supports multiple domains in a relation, that means, it can have multiple source and target models. Our approach is restricted to one source model and one target model. Through applying model merge

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

and split techniques, it is possible for our tool to support multiple source and target models, and this process is transparent to the users.

- **Transformation directions**: As one of its key features, QVT Relations allows to specify bidirectional transformations. Transformations can be executed in reverse direction, and supplied with a given target model instead. In our approach, the generated XSLT stylesheet is unidirectional. But given a bidirectional transformation, we could generate a corresponding XSLT stylesheet that performs the transformation in the opposite direction on demand.

- **Check-before-enforce semantics**: The semantics of QVT Relations ensures that before creating a new object, it is checked whether an existing one satisfying the constraints of the relation can be reused. In this way, QVT Relations supports creating new, and modifying existing, target model elements. In our approach, we have to create all target model elements in a transformation.

- **Rule scheduling**: A QVT transformation may have a set of top-level relations, and if the conditions specified in the source domain pattern and the *when*-clause are satisfied, they will be automatically executed. In our approach, we allow one top-level relation, which serves as the entry-point of the transformation, and constructs the root element of the target model. All the other relations will be explicitly invoked from the *where*-clauses of relations.

- **OCL support**: In QVT Relations, the *when*- and *where*-clauses can contain arbitrary expressions of extended essential OCL. Our approach permits OCL expressions evaluating in the context of QVT relations, and allows source domain pattern elements, properties, links and variables to be referenced.

4 Tool and Case Study

Tools are crucial for the promotion of model transformation techniques in practice. In order to support our approach, we have developed a prototype tool to implement a QVT transformation as an XSLT stylesheet. This tool consists of two part: a QVT Relations graphical editor and a code generator.

The graphical editor has been built on top of MagicDraw [10], a popular UML CASE tool. First, we define a UML profile which allows us to represent the concepts of QVT Relations in UML models. Through customizing MagicDraw's diagram interface, we design a particular "QVT Relation Diagram", and its toolbar. Applying the UML profile and the diagram, we can specify QVT rules by selecting objects and links from the metamodels in a pop-up window.

We feel that it is quite easy and user-friendly to design a QVT rule in the editor, because all objects and links are restricted by the source and target metamodels, effectively providing a graphical domain-specific language. Such syntax-directed editing should reduce the number of errors, as it only permits mostly well-typed models. An exception are of course the *when*- and *where*-clauses, which are given textually, and are only checked when running the transformation generation (although they could be checked in a dedicated editor).

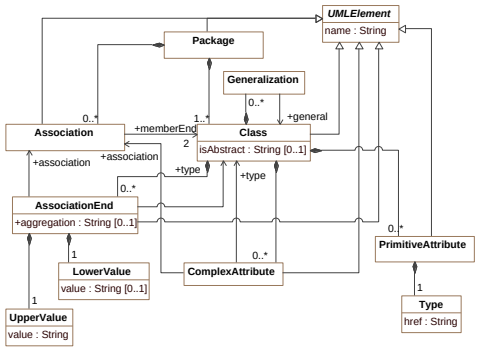

Figure 3: UML metamodel

With the QVT editor, we can construct a QVT Relations model which includes metamodels and transformations. A transformation consists of a set of *relations* (as QVT relation diagrams), along with *Key* and *Function* definitions. The QVT model is saved as an XML file. The code generator, that itself is also an XSLT stylesheet, reads in the XML file, analyzes the model's structure, parses the OCL expressions, and generates an XSLT file that represents the transformation specified by the QVT Relations. The generated XSLT stylesheet can be directly executed under an XSLT processor.

Currently, the tool supports complex pattern matching of QVT *object templates*, *property templates*, *collection templates* (member selection), and *not templates*. These cover all kinds of QVT expressions except some collection template expressions which are difficult to represent graphically. The tool is able to output execution traces. And the source model and target model can share the same metamodel, or have different metamodels.

We support a large subset of types, expressions and operations of essential OCL, such as *sequence*, *ordered set*, *relation call*, *property call*, *assignment*, *literal*, *if*, *select*, *forAll*, etc. The OCL constructs the tool does not support include *let*, *oclIsKindOf*, *iterate*, and *tuple* type.

The case study we have chosen here is a slight variation of the well known simple UML model to simple RDBMS transformation taken from [15]. This transformation is usually used to demonstrate the concepts and capabilities of transformation languages. Using our tool, we configure the transformation with the appropriate metamodels: we use the UML metamodel depicted in Fig. 3 as the source metamodel, and a RDBMS metamodel same as the one in Annex A of [15] as the target metamodel.

We model the *UMLtoRDBMS* transformation as seven QVT relations with their respective diagrams, namely *PackageToSchema*, *ClassToTable*, *AttributeToColumn SuperAttributeToColumn*, *ComplexAttributeToColumn*, *PrimitiveAttributeToColumn* and *AssocEndToFKey*. The toolbar and QVT relation diagram *PackageToSchema* is shown in Fig. 4. This is the only top-level relation that maps the UML root element *Package* to the RDBMS root element *Schema*. This relation provides the context necessary to apply the subsequent relations.

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

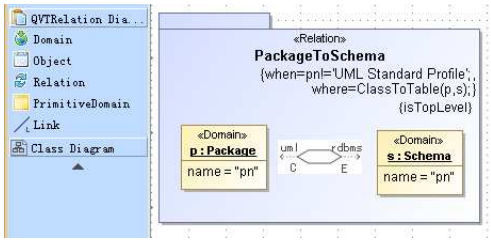

Figure 4: A QVT relation diagram and its toolbar

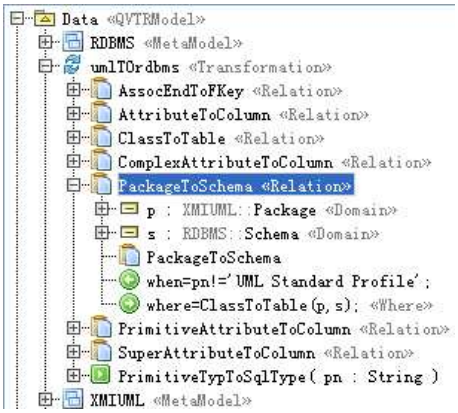

Figure 5: QVT model for the UMLtoRDBMS

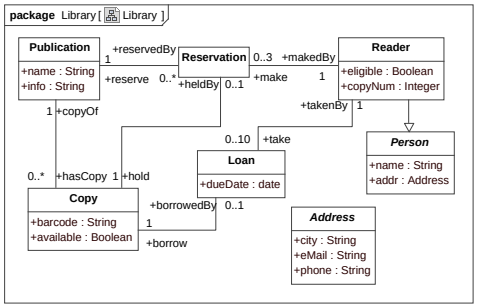

Figure 6: UML model of a library system

Fig. 5 shows the abstract representation of the QVT model designed in the above steps. The code generator of our tool reads in the model, and produces an output XSLT stylesheet. The seven QVT relations of our case study, along with a user defined function that transfers the UML primitive types to the data types of RDBMS, the *key* function and the *xmiXMIrefs* function used to retrieve elements in the UML XMI model, translate to an XSLT stylesheet about 130 lines of code.

As a comparison, the same transformation specified in QVT Relations textual syntax is about 120 lines, in QVT Operational language about 100 lines, in QVT Core language about 400 lines [15], and in MT [18] about 140 lines. Considering our generated XSLT code sufficiently structured as not to skew the line count, we judge that it is not more complex than the other specifications.

To test the generated XSLT stylesheet, we use a simple UML model of a library system, as shown in Fig. 6. In the execution of the transformation on the example, the relation *PackageToSchema* runs first. The relation finds the *Package* in the library model, binds the name ”Library” to the variable ”pn”, builds a *Schema* with the same name as the root of the target model. Next, the relation call defined in the *where*-clause, *ClassToTable*, is invoked to search *Classes* owned by the *Package*. If a class, for example, *Reader*, is located, then the class name, ”Reader”, is bound to variable ”cn” which is used to create a corresponding *Table* under the *Schema*. After that, relations *AttributeToColumn* and *AssocEndToFKey* are called to deal with properties of the class *Reader*.

5 Related Work and Conclusion

Different model transformation techniques have been discussed and compared in details in [4, 9]. As stated in [8], tool supports for QVT Relations has made some progress, but is still in its infancy. The most often mentioned QVT Relations tools, such as medini QVT, ModelMorf or MOMENT-QVT, provide support for specification and execution of QVT Relations in textual language.

Several other research projects, such as UMLX QVTr, Visual QVT/R, and VMTS QVT, propose to develop graphical editors for QVT Relations, but they are all not deployable yet.

There are a number of works also using XSLT for model transformation. MTrans [17] proposes a language that is placed on top of XSLT to describe model transformations. The UMT tool [6] supports UML-to-UML model transformation based on XMI Light and uses XSLT as transformation language. Willink [19] proposed a graphical transformation language UMLX to do model-to-model transformations. UMLX is transformed into XSLT that executes the transformation between XMI instances. Bichler [2] discussed model-to-text transformations using XMI as the input. An intermediate XML format is used to simplify the XMI, and the XSLT operates on this simplified XML representation of the XMI content. In [1], processing UML models in XMI format through XSLT has already been suggested, but not described in detail.

As a distinguishing feature from the above approaches, our transformation approach is based on QVT Relations, supported by the graphical notation, and we can automatically generate a corresponding executable XSLT stylesheets. Textual QVT Relations could be parsed into our QVT models and immediately benefit from the graphical modeling mechanism.

Conclusion

In this paper we report our work on a practical model transformation approach. In the approach, we specify a transformation as a set of QVT relations between the elements of the metamodels using the graphical notation of QVT Relations. Then the transformation is implemented as an XSLT stylesheet, where every relation becomes an XSLT rule template. We have discussed how a QVT relation can be translated to an XSLT rule template.

Our approach is founded on the standards introduced by OMG and W3C, and makes use of well-known and commonly adopted CASE tools and languages. The benefit to the user lies in having access to a *graphical* general purpose model transformation language, instead of having to learn, for example, XSLT, and use it

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563

to implement model transformations.

We have developed a prototype tool to apply our approach. The QVT graphical editor of the tool supports specification of QVT transformation in graphical notation. The code generator of the tool (implemented in XSLT) automatically translates the QVT specification into an executable XSLT program. Interestingly, after boot-strapping our approach, we could now also use it to rewrite the code generator in QVT, as the graphical model is saved as an XMI model, instead of directly implementing it in XSLT (ignoring OCL expressions).

While currently this tool does not support the full feature-set of QVT Relations, like conformance checking or bidirectional relations, it is still complete enough for the more traditional forward-engineering tasks of computing a target model from an input model.

We are now working on support for in-place transformations through generating only the changed part of a model, and then merging the changed part into the original model. We also plan to extend our tool to support multiple input and output models, as well as parameterized transformations.

In the scope of our rCOS Modeler [3] for component-based modeling in UML, we are now re-implementing a set of model transformations currently given in Java as graphical QVT relations in our framework. Apart from hopefully reducing the complexity of the transformations by giving them in a declarative framework, this will further also extend our set of sample transformations to validate the viability of our approach in practice.

The prototype tool with examples is available from `http://rcos.iist.unu.edu/qvttoxslt/`. For QVT graphical editing, MagicDraw v16.5 or later is required, and the XSLT transformations have been tested under the Microsoft XSLT processor.

Acknowledgments: Partially supported by the national natural science foundation of China (NNSFC) 90718009 and 90718014, STCSM 08510700300, and the ARV grant of the Macao Science and Technology Development Fund. The first author would like to thank Dr. Zhiming Liu of UNU-IIST, Prof. Anders P. Ravn of Aalborg University, and Prof. Wuwei Shen of Western Michigan University for their valuable discussion.

References

[1] M. Angelaccio and A. DAmbrogio. A model transformation framework to boost productivity and creativity in collaborative working environments. In *Collaborative Computing: Networking, Applications and Worksharing*, pages 464–472, 2007.

[2] L. Bichler. A flexible code generator for MOF-based modeling languages. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[3] Z. Chen, Z. Liu, and V. Stolz. The rCOS tool. In J. Fitzgerald, P. G. Larsen, and S. Sahara, editors, *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, number CS-TR-1099 in Technical Report Series. Newcastle University, May 2008.

[4] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

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

[6] R. Grønmo and J. Oldevik. An empirical study of the UML model transformation tool (UMT). *Proc. First Interoperability of Enterprise Software and Applications, Geneva, Switzerland*, 2005.

[7] M. Kay. *XSLT 2.0 Programmer's Reference, Third Edition*. Wrox Press, 2004.

[8] I. Kurtev. State of the art of QVT: A model transformation language standard. *Applications of Graph Transformations with Industrial Relevance*, pages 377–393, 2008.

[9] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[10] No Magic Inc. MagicDraw UML User's Manual. http://www.magicdraw.com/, 2010.

[11] Object Management Group. MDA Guide, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

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

[13] Object Management Group. Object constraint language, version 2.0, May 2006.

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

[15] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification v1.1. http://www.omg.org/spec/QVT/1.1/Beta2/PDF/, Dec. 2009.

[16] Object Management Group. Unified Modeling Language: Superstructure, version 2.3, May 2010. http://www.omg.org/spec/UML/2.3/Superstructure.

[17] M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *Workshop on Transformations in UML (WTUML), Genova, Italy, April*, 2001.

[18] L. Tratt. Model transformations in MT. *Science of Computer Programming*, 68(3):169–186, 2007.

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

[20] WWW Consortium. XML Schema, W3C Recommendation. http://www.w3.org/XML/Schema.

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

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

DOI: 10.1145/1921532.1921563 http://doi.acm.org/10.1145/1921532.1921563