An XQuery-based Model Transformation Language*

Jesús M. Almendros-Jimenez[1] and Luis Iribarne[1]
and Jesús J. López-Fernández[2] and Ángel Mora-Segura[2]

[1] University of Almeria (SPAIN) {jalmen, luis.iribarne}@ual.es
[2] Autonomous University of Madrid (SPAIN) {jesusj.lopez, angel.moras}@ual.es

Abstract. In this paper we propose a framework for model transformation in XQuery. Our proposal aims to provide the elements for using XQuery as a transformation language. With this aim, our framework provides a mechanism for automatically obtaining an XQuery library for a given meta-model. Meta-models are defined as XML schemas, and the XQuery library serves to query and create elements of an XML Schema. Transformations abstract from XML representation, in the sense of, elements from meta-models are encapsulated by XQuery functions. We have also studied how to use our framework for model validation. Source and target models and transformations are validated by considering constraints. The framework has been tested with a case study of transformation in UML, where the XML-based representation of models is achieved by the standardized language XMI.

1 Introduction

XQuery [6, 3] is a programming language proposed by the W3C as standard for the handing of XML documents. XQuery allows to change the XML-based format of the output of a query, against an XML input document. In spite of the main aim of XQuery is querying XML documents, XQuery can be also used as *XML transformation language*. *XSLT* [14], which was designed for XML transformations, can be encoded in XQuery [4], however XSLT code is very verbose and hard to maintain.

Model Driven Engineering (MDE) is an emerging approach for software development. MDE emphasizes the construction of models from which the implementation is derived by applying model transformations, and provides a framework to developers for transforming their models. Therefore, model transformation [8, 13, 19, 26] is a key technology of MDE. There are many proposals of programming languages specifically designed for transforming models: *QVT* [20, 16], *ATL* [12], *ETL* [15], *RubyTL* [23], *VIATRA2* [2], *GReAT* [1] and AGG [25], among others. Most of languages for describing models (*UML*, *BPMN*, *Petri Nets*, *WebML*, etc) have an XML-based representation (*XMI* [21], *XPDL* [24], *PNML* [11], *IFML* [7], etc). Thus, XML provides a framework for handling many modeling languages, and XQuery can be used for describing transformations in such languages. Usually, the XML-based representation of models is used for exchanging models between applications. One can argue that the XML representation is a machine readable format, and transforming models via XQuery could be a too low-level task. However, XQuery is equipped with high-level mechanisms (i.e.,

* This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2010-15588, and the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

modules, higher-order functions) that could make that transformations abstract from the XML representation.

In this paper we propose a framework for model transformation in XQuery. Our proposal aims to provide the elements for using XQuery as transformation language. In order to use XQuery as transformation language, the developer should be equipped with mechanisms allowing the handling of different meta-models. Models in transformations (source and target models) can conform to different meta-models. With this aim, our framework provides a mechanism for automatically obtaining an XQuery library of functions for each meta-model. This library allows query the given meta-model and to create elements of the meta-model. Meta-models are defined as XML schemas, and the XQuery library is designed for handling an XML schema.

XQuery libraries for meta-models can be used with two ends. Firstly, to define transformations between models. Transformations abstract from XML representation in the sense of elements from meta-models are encapsulated by XQuery functions. Querying and creation of elements from meta-models are achieved by calling suitable XQuery functions. Transformation code uses XQuery constructions, that is, *for* and *let* to traverse source models and *return* to generate target models, and *where* to express applicability conditions of the transformation. In addition, *union* operator is used to decompose a transformation into several transformation cases. In addition, the transformation can make use of *modules* and *auxiliary functions* for particular subtasks.

Secondly, the XQuery library is used for defining transformation properties (i.e., constraints). We have also studied how to use our framework for model validation, in particular, how to validate source and target models, and transformations. Source and target models and transformations are validated by considering constraints. The framework is also used for syntactic validation of source and target models. Syntactic validation is achieved by using XML schema-based validation. Meta-models of source and target models are defined as XML schemas, and therefore syntactic validation of them is equivalent to schema validation.

In order to describe a model transformation in our framework, we have to follow the next steps.

1. **Definition of XML Schemas of Source and Target Models**. We have to define the meta-models of the source and target models. Meta-models have to be described by XML Schemas.
2. **Generation of the XQuery Library**. From XML schemas of source and target meta-models, an XQuery library is automatically generated.
3. **Validation of Source Models**. Source models are syntactically validated from XML Schemas of source meta-models. An XML Schema validator is used for this step. In addition, a set of constraints are described making use of the XQuery library of the source meta-model. Source models are validated w.r.t. constraints. An XQuery interpreter is used for this step.
4. **Definition of the Transformation**. The XQuery library of the source and target meta-models can be used to define the transformations. An XQuery interpreter is used for this step.
5. **Execution of the Transformation**. The transformations are executed obtaining target models. An XQuery interpreter is used for this step.
6. **Validation of Target Models**. Target models are syntactically validated from XML Schemas of target models. An XML Schema validator is used for this step. In addition, a set of constraints are described making use of the XQuery library of the target meta-model. Target models are validated w.r.t. constraints. An XQuery interpreter is used for this step.

7. **Cross Validation of Source and Target Models**. A set of cross constraints are described making use of the XQuery library of the source and target meta-models. Source and target models are validated w.r.t. constraints. An XQuery interpreter is used for this step.

The advantages of the approach are the following. Firstly, we can handle any meta-model having an XML representation. In order to handle a given meta-model is only it is required to have the corresponding XML Schema. Secondly, we use a well-known programming language (XQuery) to write transformations. There are many implementations of XQuery range from academic to commercial ones (see `http://www.w3.org/XML/Query/`). Thirdly, the XQuery language provides a type system, compilation and run-time errors, a module system, higher order programming, among others. XQuery can handle in most implementations large XML documents, and therefore in our proposal large models. Finally, it is worth observing that n-m transformations can be defined making use of natural join operations in database query languages.

The framework has been tested with a case study of transformation in UML, where the XML-based representation of models is achieved by the standardized language XMI. Nevertheless, our framework is not specifically designed for UML. We have to provide the XML schema of XMI in order to transform UML models. The case study of transformation we show is the well-known Entity-relationship model to Relational model, in which source and target models are class diagrams. Thus, we have to provide specifically the XMI Schema of class diagrams.

The implementation of the approach has been achieved by using the *BaseX* interpreter [10] of XQuery. We have developed a library generator from XML Schemas. The library generator has been implemented in XQuery. It takes an XML Schema as input (the XML Schema is an XML document) and generates XQuery code in plain text (as output document). The XQuery code is an XQuery library providing a set of XQuery functions to handle the XML Schema. Basically, each item of the XML Schema is handled by two main functions, one to query the item from an XML document, and one to create the item. Moreover, BaseX is also used in our framework to validate models with respect to XML Schemas (i.e., meta-models) as well as to validate models with respect to constraints. Finally, BaseX is also used to edit transformations and execute transformations. BaseX assists in the development of transformations. BaseX provides error messages from XML Schema validation. BaseX also helps in the development of code of transformations, by providing compilation error messages due to syntactic bugs and run-time error messages when the execution is accomplished. For designing XML Schemas we have used the UML Visual Paradigm tool which is able to translate a stereotyped UML class diagram into an XML Schema. The use of UML VP and BaseX is not actually required in our framework. The implementation can be downloaded from `http://indalog.ual.es/mdd` together with the case study (models, meta-models, validation properties, etc).

The structure of the paper is as follows. Section 2 will describe the use of XQuery as transformation language. Section 3 will show the validation of transformations. Section 4 will summarize related work. Finally, Section 5 will conclude and present future work.

2 Model Transformation with XQuery

Now, we describe the steps to be followed in our framework with a case study. The case study is a well-known transformation from an entity-relationship to relational model.

The model A of Figure 1 represents the modeling of a database. We will call this kind of modeling *"entity-relationship"* modeling of a database in contrast to the model B of Figure 2 which will be called *"relational"* modeling of a database.

Fig. 1. Entity-relationship modeling of the Case Study

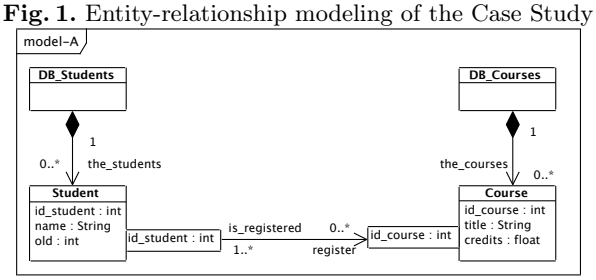
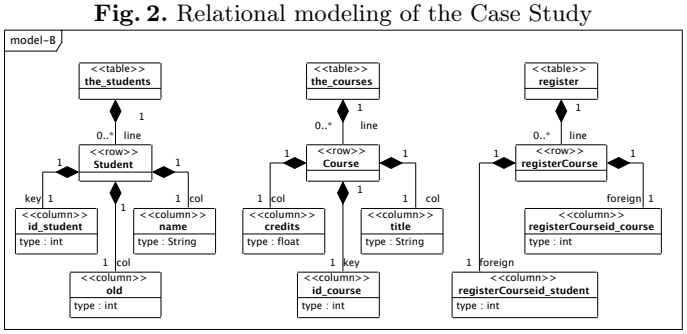


Fig. 2. Relational modeling of the Case Study



The model A of Figure 1 can be summarized as follows. *Data* (i.e. entities) are represented by classes (i.e., *Student* and *Course*), including attributes. *Stores* are defined for each data (i.e., *DB_ Students* and *DB_ Courses*); stores are composed of data, therefore specifying a composition relationship between store and data. The store is responsible for storing the objects of a certain data. Stores are unique for each data. *Relations* are represented by associations and relation names are association names. Besides, *roles* are defined (i.e., *the_ students*, *the_ courses*, *is_ registered* and *register*) for each relation end. *Data attributes* are class attributes. The type of each attribute is the type of the class attribute. Each data has (one or more) key attributes. Relations can be adorned with *role qualifiers* and navigability. Qualifiers are used to specify the key attributes of each data being foreign keys of the corresponding relation, therefore qualifiers have to be selected from the key attributes of the corresponding data.

In model B of Figure 2 *Tables* are composed of *rows*, and rows are composed of *columns*. Stereotypes << *table* >>, << *row* >> and << *column* >> are used for them. Furthermore, *line* is the role of the rows in the table, *key* is the role of the key attributes in rows, *foreign* is the role of the foreign keys in rows, and *col* is the role of non keys and non foreign keys in rows.

Figure 3 represents the meta-model of model A [3] . In the meta-model A, *DB_ Students* and *DB_ Courses* are instances of the class *store*, while *Student* and *Course* are instances of the class *data*, and the attributes of class *Student* and class *Course* are instances of the class *data_ attribute*. *data_ attribute* represents *id_ course*, *title*, etc. having a boolean attribute *key* to represent key attributes. *the_ students* and *the_ courses* are values of attribute *container*, and *register*, *is_ registered* are *roles*, each one with a *min* and *max* values representing the cardinality, and *navigable* which is a boolean value. The attribute *data* of *role* is used as cross reference in the the meta-model (of

[3] Let us observe that our framework works with XML format (models and meta-models). Nevertheless, we use UML class diagrams to graphically describe them.

type IDREF), and represents the association end of the role. Figure 4 represents the meta-model of model B. In the meta-model B, tables and rows of the target model are instances of the corresponding classes, and the same can be said about *key*, *col* and *foreign* classes.

Fig. 3. Meta-model of the Source Model

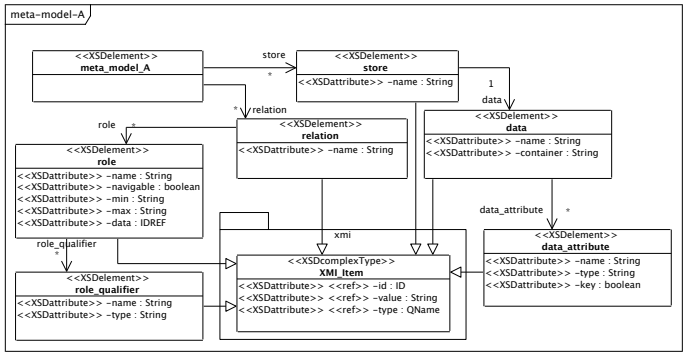
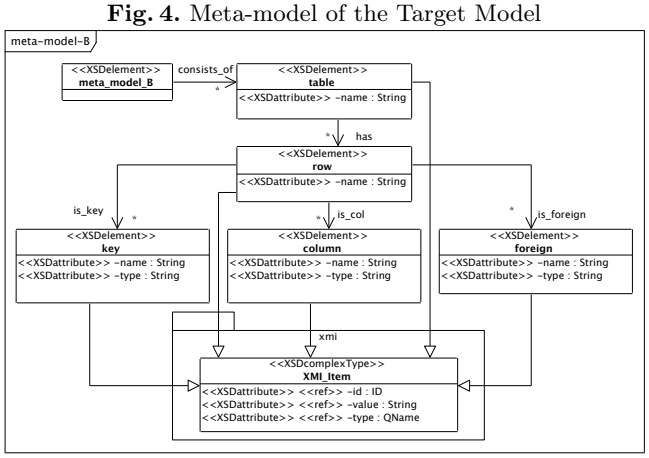


Fig. 4. Meta-model of the Target Model



In both meta-models, we make use of a class called *XMI_Item* from the XMI Schema providing an *id*, *value* and *type* for each XMI Item. It is specific to the case study, which is an XMI based transformation. In addition, in the Visual Paradigm tool, several stereotypes: `XSDelement`, `XSDattribute` and `XSDcomplextype` are used to represent XML Schemas. The stereotype `ref` is used to represent references on XML Schemas.

Our framework is based on the automatic generation of an XQuery library for any XML Schema. Basically, the XQuery library consists of a set of XQuery functions that query and create items of an XML Schema (items `xsd:element` and `xsd:complextype`). With this aim, we have defined two abstract XQuery functions as follows:

```
declare function rlib:readsubitem($model as node()*,$nameitem as xsd:QName,$attributes as xsd:string*,$values as xsd:
      string*)
{
for $elements in $model/* where node-name($elements)=$nameitem and
          (every $att in $attributes satisfies rlib:cond($elements,$att,$values[index-of($attributes, $att)]))
return $elements
};

declare function clib:createitem($name as xsd:QName, $atts as xsd:string*, $values as xsd:string*,$body as node()*)
{
  element {$name}{(for $par_n in $atts return (attribute {$par_n}{$values[index-of($atts, $par_n)]})),$body}
};
```

The function `readsubitem`, reads from an XML document (`$model`), the items of a certain name (`$nameitem`). In other words, `readsubitem` queries the nodes of a certain tag. The function has two additional parameters: `$attributes` and `$values`, that restrict the search to tags with attributes of a certain value. Basically, the call to the sentence `rlib:readsubitem (model, nameitem, (`$attribute_1$, ..., $attribute_n$`), (`$value_1, \ldots, value_n$`))` corresponds to the XPath expression given by `model/nameitem [`$attribute_1 = value_1$ *and* ... *and* $attribute_n = value_n$`]`. Let us remark that we have to use an XQuery function for this task given that the number of attributes (and values) varies, and it cannot be simulated with XPath.

The function `createitem` creates an item of name `$name` with `$atts` whose value is given in `$values`. Additionally, the item can include a `$body` as subitem. Basically, a call to `clib:createitem(name, (`$attribute_1, \ldots, attribute_n$`), (`$value_1$, ..., $value_n$`), body)` generates $< name\ attribute_1 = value_1, ..., attribute_n = value_n > body < /name >$

These two functions serve as the basis of the automatically generated XQuery library for any XML Schema. The XQuery library for a particular XML Schema contains functions to query and create XML items following the XML Schema.

For instance, for the XML-Schema of Figure 3, the XQuery library contains the functions `read_store_of_meta_model_A` , `read_data_of_store`, `read_data_attribute_of_data`, etc. to query the elements of the XML document, as well as functions `create_store`, `create_data`, `create_data_attribute`, etc. to create the elements of the XML document. They are defined as follows:

```
declare function mmAl:read_store_of_meta_model_A($model as node()*,$atts as xsd:string*, $values as xsd:string*)
{
        rlib:readsubitem($model,QName('http://xtl.org/mmA','mmA:store'), $atts, $values)
};
declare function mmAl:create_store($xmi_id as xsd:string, $xmi_value as xsd:string,$xmi_type as xsd:string, $name as
      xsd:string, $body as node()*)
{
        clib:createitem(QName('http://xtl.org/mmA','mmA:store'),(QName('http://xtl.org/xmi','xmi:id'),
            QName('http://xtl.org/xmi','xmi:value'),QName('http://xtl.org/xmi','xmi:type'),
            QName('','name')),($xmi_id,$xmi_value,$xmi_type,$name), $body)
};
```

Basically, the functions for querying an element call to the previously defined `readsubitem`, and the functions for creating an element calls `createitem`. The functions for creating elements include as parameters the attributes of the XML Schema.

For instance, `data_attribute` in the XML Schema of Figure 3 has as attributes: *name*, *type* and *key* and, additionally, three attributes inherited from *XMI_Item*: *xmi_id*, *xmi_value* and *xmi_type*, which have been included as parameters of the function `mmAl:create_data_attribute`.

The XQuery library is automatically generated from the XML Schema thanks to an XQuery program we have developed that traverses an XML Schema and generates for each `xsd:element` in an `xsd:choice` two functions (one for querying and another for creation). It generates parameters for each `xsd:attribute` and `xsd:complextype` of an `xsd:extension`.

We have assumed in the case study that models are XMI-based models. This is the reason why the elements of the meta-models inherit from *XMI_Item*. It has as consequence that each element is identified by the attribute *xmi:id*. This identifier is used in meta-models for cross references (see attribute *data* of class *role* in Figure 3). In other words, some meta-model links are defined in XMI-based models, and they are represented by *xmi:id* values.

With this aim, the base library includes a function called `readlink`. The `readlink` function retrieves from `$model` the elements whose *xmi:id* has the value of `$nameatt` of the item `$element`. The `readlink` function is used in the (automatically generated) function `read_data_of_role`. They are defined as follows:

```
declare function rlib:readlink($model as node()*,$element as node(),$nameatt as xsd:string)
{
  for $pointer in (for $atts in $element/@* where name($atts)=$nameatt return data($atts))
  return for $items in $model//* where data($items/@xmi:id)=$pointer
         return $items
};
declare function mmAl:read_data_of_role($model as node()*,$element as node())
{
  rlib:readlink($model, $element, 'data')
};
```

The same could happen in other meta-meta-models, that is, a certain attribute is used for cross references. In such a case, the function `readlink` can be easily modified without affecting our implementation.

Now, the problem of model transformation is how to transform a class diagram of the type A (like Figure 1) into a class diagram of type B (like Figure 2). The transformation is as follows.

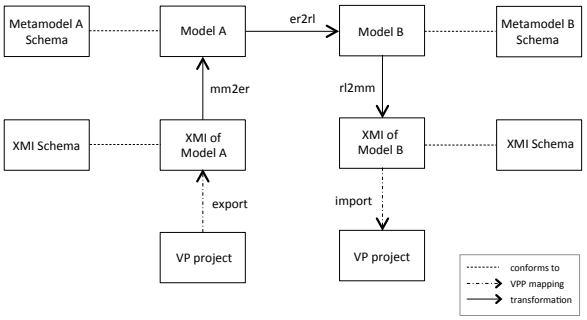
The transformation generates two tables called *the_ students* and *the_ courses* each including three columns that are grouped into rows. The table *the_ students* includes as columns for each student the attributes of *Student* of Figure 1. Key columns are key attributes. The same can be said for the table *the_ courses*. Given that the association between *Student* and *Course* is navigable from *Student* to *Course*, a table of pairs, called *register*, is generated to represent the assignments of students to courses, using *registerCourse* as the name of the row. The columns *registerCourseid_ student* and *registerCourseid_ course* taken from role qualifiers, play the role of *foreign* keys.

The transformation can be considered as a transformation of elements of meta-model A (see Figure 3) into elements of meta-model B (see Figure 4). Such a transformation has to query elements of meta-model A and create elements of meta-model B. Therefore we can use the XQuery libraries for meta-models A and B, in particular those functions for querying A and creating B. The code of the transformation is as follows.

```
<meta_model_B xmlns="http://xtl.org/mmB" xmlns:xmi="http://xtl.org/xmi" xmlns:mmB="http://xtl.org/mmB">
{
let $model :=  doc($file)/mmA:meta_model_A
let $body := ((
for $s in mmAl:read_store_of_meta_model_A($model,(),()),
  $p in mmAl:read_data_of_store($s,(),())
  return (
    let $attrs := mmAl:read_data_attribute_of_data($p,(),())
    let $columns := er:columns($attrs)
    let $id_has := clib:create_id(($s/@xmi:id,$p/@xmi:id, "has"))
    let $id_table := clib:create_id(($s/@xmi:id,$p/@xmi:id, "table"))
    let $r:= mmBl:create_row($id_has,"","mmB:has",$p/@name,$columns)
    let $t:= mmBl:create_table($id_table,"","mmB:table",$p/@container, $r)
    return $t
  )
)
union
(
for $s in mmAl:read_relation_of_meta_model_A($model,(),()),
  $p in mmAl:read_role_of_relation($s,(),()) where (($p/@navigable = "true") and ($p/@max = "*"))
    return (
    let $data := mmAl:read_data_of_role($model,$p)
    let $columns := er:foreign_keys($model,$s,$p)
    let $id_has := clib:create_id(($s/@xmi:id,$p/@xmi:id, "has"))
    let $id_table := clib:create_id(($s/@xmi:id,$p/@xmi:id, "table"))
    let $r:= mmBl:create_row($id_has,"","mmB:has",concat($p/@name,$data/@name), $columns)
    let $t:= mmBl:create_table($id_table,"","mmB:table",$p/@name,$r)
    return $t
)))
return $body
}
</meta_model_B>
```

Basically, the XQuery code of the transformation generates an XML document conforming to the XML Schema of meta-model B. Therefore the result is an XML document whose root label is `meta_model_B` and contains a `$body` including two kinds of tables: those representing `the_students` and `the_courses`, and those representing `register`. In other words, the XQuery code represents two cases and they are joined by the *union* operator of XQuery.

Fig. 5. Schema of the Whole Transformation



In both cases a *for* expression is used to traverse the meta-model A, the *let* expression retrieves some elements of the model, the *where* expression describes applicability conditions, and the *return* expression builds the elements of the meta-model B.

In the case study, the code creates tables, rows and columns thanks to the use of creation functions of the library for meta-model B. A central element of the transformation code is the use of an auxiliary function, called `create_id` for creating identifiers of XML nodes.

In order to simplify the code some auxiliary functions have been implemented, for instance, functions `columns` and `foreign_keys` which, respectively, compute the columns of a table, in the first case, and the foreign keys in the second case. The code of the first auxiliary function is as follows.

```
declare function er:columns($attrs as node()*)
{
for $attr in $attrs return
  let $idkey := clib:create_id(($attr/@xmi:id,"is_key"))
  let $idcol := clib:create_id(($attr/@xmi:id,"is_col"))
  return
  if ($attr/@key="true")
  then mmBl:create_key($idkey,"","mmB:key",$attr/@name,$attr/@type,())
  else mmBl:create_column($idcol,"","mmB:is_col",$attr/@name,$attr/@type,())
};
```

We have presented how to encode the transformation from meta-model A to meta-model B. However, the case study of transformation involves two additional transformations, that is, from the UML meta-meta-model to meta-model A, and from meta-model B to the UML meta-meta-model. In other words, the whole transformation takes an XMI file with the model A and transforms model A into an XMI file with model B. Thus, the XMI representation of model A has to be transformed into the XML representation of meta-model A, and also the XML representation of meta-model B has to be transformed into the XMI representation of model B (see Figure 5, where mm2er and rl2mm are the additional transformations, and er2rl in the main transformation). Direct transformations from XMI to XMI are harder to encode.

3 Validation of the Transformation

Now, we would like to show how we can describe validation properties on transformations. Source and target models and transformations are validated by considering *constraints*. Moreover, we have considered two kinds of properties to be validated: *syntactic (SynR)* and *semantic requirements (SemR)*. In addition, we have considered that some requirements are required to have *well-formed models* (WR), while some of them are required by the transformation (TR). We can see these requirements in Table 1.

For instance, (v2) requires that each data has a unique key attribute. This is a semantic requirement. Key attributes are attributes having *key* set to *true*, and the

Table 1. Model validation: properties

		SemR	SynR	WF	TR
	Source Model Constraints				
(v1)	All attribute names of a data are distinct	✓	-	✓	-
(v2)	Each data has a unique key attribute	✓	-	-	✓
(v3)	Each attribute is associated to exactly one data	-	✓	✓	-
(v4)	Each data is contained in exactly one store	-	✓	✓	-
(v5)	All data have distinct names	✓	-	-	✓
(v6)	All data have distinct containers	✓	-	-	✓
(v7)	Each qualifier is associated to exactly one role	-	✓	-	✓
(v8)	All qualifier names of a relation are distinct	✓	-	-	✓
(v9)	All qualifiers are key attributes	✓	-	✓	-
(v10)	Each relation has two roles	-	✓	✓	-
(v11)	Each role is associated to exactly one relation	-	✓	-	✓
(v12)	Each role is associated to exactly one data	-	✓	-	✓
(v13)	All role names of a data are distinct	✓	-	-	✓
(v14)	Each store is associated to exactly one data	-	✓	✓	-
	Target Model Constraints				
(v15)	All column names of a row are distinct	✓	-	✓	-
(v16)	All foreign keys of a row are keys of another row	✓	-	✓	-
(v17)	Each table is associated to exactly one row	-	✓	✓	-
(v18)	Each row is associated to exactly one table	-	✓	✓	-
(v19)	Each key is associated to exactly one row	-	✓	-	✓
(v20)	Each column is associated to exactly one row	-	✓	-	✓
(v21)	Each foreign key is associated to exactly one row	-	✓	-	✓
(v22)	All table names are distinct	✓	-	✓	-
(v23)	All row names are distinct	✓	-	✓	-
(v24)	All rows have exactly one key	-	✓	-	✓
(v25)	All rows have either all keys and 'cols' or all foreign keys	✓	-	-	✓
	Cross Constraints				
(v26)	Key and col names and types are names and types of data 'atts'	✓	-	-	✓
(v27)	Table names are either container names or role names	✓	-	-	✓
(v28)	Row names are data names or 'concats' of role and data names	✓	-	-	✓
(v29)	Foreign key names are concatenations of role, data and key names	✓	-	-	✓

existence of a unique key attribute cannot be expressed by the XML Schema. Moreover, this requirement is a constraint on the source model because key attributes are used as foreign keys in the target model. Case (v4) is a syntactic requirement on well-formed models: each data is contained in exactly one store. It is not needed in the transformation and can be expressed by the XML Schema with a cardinality constraint. Cases (v5), (v6), (v8) and (v13) are related to naming of elements of source models, and therefore they are semantic requirements. They are required by the transformation: data and container names are used for naming tables and rows in the target model, while role and qualifier names (concatenated with data names) are also used for naming rows and foreign keys.

In the target model tables, rows, columns, keys and foreign keys are not shared (cases (v15), (v22) and (v23)). Case (v16) is a semantic requirement that describes the relationship between foreign keys and keys in a well-formed target model. Case (v25) is required by the transformation which assigns either keys and columns or foreign keys to rows. Finally, cases (v26)-(v29) describe the relationship between names of the target model and names of the source model.

It is worth observing that the requirements about source and target models in isolation are not enough for the soundness of the transformation. For instance, source and target models can both have keys, but a transformation requirement is needed: the keys of the target models are the keys of the source model.

Now, we can see how to validate source models in our approach. Firstly, syntactic requirements (SynR) can be validated by XML schema validation. Fortunately, BaseX is equipped with `validate:xsd-info` which is a built-in function that validates an XML document against an XML Schema and provides error messages. Secondly, semantic requirements (SemR) can be validated by XQuery boolean expressions that use the

XQuery library. For instance, (v2) "Each data has a unique key attribute" can be checked as follows:

```
let $model :=  doc($file)/mmA:meta_model_A
return every $stores in mmAl:read_store_of_meta_model_A($model,(),())
  satisfies (every $data in mmAl:read_data_of_store($stores,(),())
    satisfies (let $attr := mmAl:read_data_attribute_of_data($data,("key"),("true"))
        return (count($keys) = 1))
```

The same can be said for the validation of target models. They can be syntactically validated by the XML Schema, and semantically validated by XQuery boolean expressions. For instance, (v16) "All foreign keys of a row are keys of another row" can be checked by the following expression:

```
let $model :=  doc($file2)/mmB:meta_model_B
return
  every $table in mmBl:read_table_of_meta_model_B($model,(),())
  satisfies
  (every $row in mmBl:read_row_of_table($table,(),())
  satisfies
    (every $foreign in mmBl:read_foreign_of_row($row,(),())
    satisfies
    (some $table2 in mmBl:read_table_of_meta_model_B($model,(),())
    satisfies
    (some $row2 in mmBl:read_row_of_table($table2,(),())
    satisfies
    (some $key in mmBl:read_key_of_row($row2,(),())
    satisfies $foreign/@name=concat($row/@name,$key/@name)
  )))))
```

Finally, cross constraints can be expressed in our framework. For instance, (v26) "Key and column names and types are names and types of data attributes" can be expressed as follows:

```
let $modelB :=  doc($file2)/mmB:meta_model_B
let $modelA :=  doc($file)/mmA:meta_model_A
return
  every $table in mmBl:read_table_of_meta_model_B($modelB,(),())
  satisfies
  (every $row in mmBl:read_row_of_table($table,(),())
  satisfies
  (every $keycol in mmBl:read_key_of_row($row,(),()) union mmBl:read_column_of_row($row,(),())
  satisfies
  (some $store in mmAl:read_store_of_meta_model_A($modelA,(),())
  satisfies
  (some $data in mmAl:read_data_of_store($store,(),())
  satisfies
  (some $data_attribute in mmAl:read_data_attribute_of_data($data,(),())
  satisfies $data_attribute/@name=$keycol/@name and $data_attribute/@type=$keycol/@type)
))))
```

4 Related Work

The use of XML and transformation languages for XML in the context of model transformation have been proposed in some works. All of them, as far as we know, fall on the use of XSLT.

In an early work [22], the authors propose a language that uses XSLT for transforming models. The language can be seen as a first attempt to represent models and transformations in XSLT.

In [17] the authors propose the use of the transformation language QVT, but they map QVT transformations into XSLT. XSLT programs are automatically obtained from QVT transformations. Therefore QVT is encoded by an XSLT transformation. They also create XML schemas for source and target meta-models using syntactic validation of source and target models. This work is the closest to our proposal. Although they use an standardized language (i.e., QVT) for describing transformations, they adopt as transformation engine an XML-based transformation language. Since XSLT can be encoded in XQuery, the proposal of these authors could be handled by XQuery instead of XSLT.

The UMT (UML Model Transformation) Tool [9] also is based on XSLT (and Java). In this tool a simplified version of XMI (called XMI light) is used as representation of

UML models. They directly use XSLT to write transformations. In [5], M2T transformations are also proposed in XSLT using a simplified version of XMI. We believe that to simplify XMI makes easier the definition of transformations. The drawback is that tools (especially UML VP) usually generate and require fully fledged XMI files, and it limits import and export of models.

UMLX [27] uses a graphical notation that is again translated into XSLT transformations. User interface model transformation has been studied in [18], where a graphical notation is translated into XSLT (and QVT). The main aim of the transformations is to obtain a prototype of the user interface (using the XML based user interface language OpenLaszlo). The use of graphical notation is an interesting research line in model transformation: developers would like to use a graphical notation to depict transformations instead of using code. Thus, transformations of graphical notations to code aims to ease the adoption of transformation technologies.

In conclusion, XSLT has been the basis of works dealing with model transformations. We can see our work similar to them in the sense of the proposed XQuery library of functions provides a high-level language for describing transformations, and at a low-level XQuery handles models in XML format. Obviously, our proposal could be adapted to XSLT.

5 Conclusions and Future Work

In this paper we have presented a framework to use XQuery as transformation language. We have studied how to adapt MDE to XQuery in the following sense. Meta-models are represented by XML Schemas. From XML Schemas we are able to automatically generate an XQuery library to handle elements of a certain meta-model. It allows to write transformations in XQuery abstracting from the XML representation. The XQuery code becomes elegant and concise thanks to the use of the XQuery library. We have also shown how to use the library to validate transformations by constraints as semantic requirements. Syntactic requirements checking comes for free from the use of XML Schema validation. As future work we would like to extend our work as follows. Firstly, the XML type system is not still used in our proposal. In other words, types in the XML Schema are not used in the generation of the XQuery library. We believe that introducing types we could improve compilation (error detection) of XQuery programs for transformation. Secondly, we would like to implement an oracle to guide the user in transformation validation. The XML Schema validator of BaseX already provides information about syntactic (compilation) errors in source and target models, whereas the semantic validation we have proposed only gives true or false when checking a constraint. It could be useful to provide more accurate information about the type of error the validator has found. We are also interested to apply our framework to other modeling languages: BPM, Ontologies, etc. Finally, we are interested in test case generation for transformations in order to detect programming errors and to help transformation debugging.

References

1. Agrawal, A.: Graph rewriting and transformation (GReAT): a solution for the model integrated computing (MIC) bottleneck. Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on pp. 364–368 (6-10 Oct 2003)
2. Balogh, A., Varró, D.: The Model Transformation Language of the VIATRA2 Framework. Science of Programming 68(3), 187–207 (October 2007)
3. Bamford, R., Borkar, V., Brantner, M., Fischer, P.M., Florescu, D., Graf, D., Kossmann, D., Kraska, T., Muresan, D., Nasoi, S., et al.: XQuery reloaded. Proceedings of the VLDB Endowment 2(2), 1342–1353 (2009)

4. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2003)
5. Bichler, L.: A flexible code generator for MOF-based modeling languages. In: 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture (2003)
6. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J., Stefanescu, M.: XQuery 1.0: An XML query language. W3C working draft 12 (2003)
7. Brambilla, M., Fraternali, P.: Large-scale Model-Driven Engineering of Web User Interaction: The WebML and WebRatio experience. Science of Computer Programming (2013)
8. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: 2nd OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
9. Grønmo, R., Oldevik, J.: An empirical study of the UML model transformation tool (UMT). Proc. First Interoperability of Enterprise Software and Applications, Geneva, Switzerland (2005)
10. Grün, C.: BaseX. The XML Database (2013), `http://basex.org`
11. Hillah, L.M., Kordon, F., Petrucci, L., Trèves, N.: PNML framework: An extendable reference implementation of the Petri Net Markup Language. In: Applications and Theory of Petri Nets, pp. 318–327. Springer (2010)
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
13. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. Sci. Comput. Program. 68(3), 114–137 (2007)
14. Kay, M., et al.: XSL transformations (XSLT) version 2.0. W3C Recommendation 23 (2007)
15. Kolovos, D.S., Paige, R.F., Polack, F.A.: The epsilon transformation language. In: Theory and Practice of Model Transformations, pp. 46–60. Springer (2008)
16. Kurtev, I.: State of the Art of QVT: A Model Transformation Language Standard. In: 3rd Int. Symposium on Applications of Graph Transformation with Industrial Relevance. pp. 377–393. Springer (2008)
17. Li, D., Li, X., Stolz, V.: QVT-based model transformation using XSLT. ACM SIGSOFT Software Engineering Notes 36(1), 1–8 (2011)
18. López-Jaquero, V., Montero, F., González, P.: T: XML: A Tool Supporting User Interface Model Transformation. In: Model-Driven Development of Advanced User Interfaces, pp. 241–256. Springer (2011)
19. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electr. Notes Theor. Comput. Sci. 152, 125–142 (2006)
20. OMG: MOF 2.0 Query/Views/Transformations RFP. Tech. rep., `http://www.omg.org/spec/QVT/` (2008)
21. OMG: XML Metadata Interchange (XMI). Tech. rep., `http://www.omg.org/spec/XMI/` (2011)
22. Peltier, M., Bézivin, J., Guillaume, G.: MTRANS: A general framework, based on XSLT, for model transformations. In: Workshop on Transformations in UML (WTUML), Genova, Italy (2001)
23. Sánchez-Cuadrado, J., García-Molina, J., Menárguez-Tortosa, M.: RubyTL: A Practical, Extensible Transformation Language. In: Procs of Model Driven Architecture - Foundations and Applications. pp. 158–172. LNCS 4066, Springer (2006)
24. Shapiro, R.M.: XPDL 2.0: Integrating process interchange and BPMN. Workflow Handbook pp. 183–194 (2006)
25. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance, pp. 446–453. Springer (2004)
26. Tratt, L.: Model transformations and tool integration. Software and System Modeling 4(2), 112–122 (2005)
27. Willink, E.: UMLX: A graphical transformation language for MDA. In: Model Driven Architecture: Foundations and Applications. pp. 03–27 (2003)