Translation of QVT relations into QVT operational mappings

Raphael Romeikat, Stephan Roser, Pascal Müllender, Bernhard Bauer

Angaben zur Veröffentlichung / Publication details:

Romeikat, Raphael, Stephan Roser, Pascal Müllender, and Bernhard Bauer. 2008. "Translation of QVT relations into QVT operational mappings." *Lecture Notes in Computer Science* 5063: 137–51. https://doi.org/10.1007/978-3-540-69927-9_10.

Translation of QVT Relations into QVT Operational Mappings

Raphael Romeikat, Stephan Roser, Pascal Müllender, and Bernhard Bauer

Programming Distributed Systems Lab,
University of Augsburg, Germany
{romeikat,roser,bauer}@ds-lab.org,
p_muellender@users.sourceforge.net

Abstract. Model transformations play a key role in Model-Driven Engineering solutions. To efficiently develop, specify, and manage model transformations, it is often necessary to use a combination of languages that stand for different transformation approaches. To provide a basis for such hybrid model transformation specification solutions, we developed and implemented a translation of the declarative QVT Relations into the imperative QVT Operational Mappings language.

1 Introduction

Model Driven Engineering (MDE) treats models as primary development artifacts as they are used for model and code generation. MDE uses models to raise the level of abstraction at which developers create and evolve software [8] and reduces complexity of the software artifacts by separating concerns and aspects of a system under development [9]. Largely automated model transformations refine abstract models to more concrete models or simply describe mappings between models of the same level of abstraction.

Model transformations are considered as a kind of metaprogramming since they are specified on the basis of metamodels. People developing model transformations have to respect the rich semantics of the metadata upon which the model transformations operate [4]. Not surprisingly, various authors suggest to use different model transformation approaches for the diverse transformation problems. Declarative transformation approaches are best applied to specify simple transformations and relations between source and target model elements, while imperative approaches lend themselves for implementing complex transformations that involve detailed model analysis [7]. As it is done with other programming languages, it seems beneficial to use several model transformation language to solve complex problems [11]. In the OMG standard for model transformations QVT [12], the imperative *QVT Operational Mappings (OM)* language is defined as an extension of the declarative *QVT Relations (Relations)* language.

Having a closer look at model transformation approaches, one can observe that the various approaches and their implementations support model transformation features like automatic updates, directionality, traceability, etc. to a

different extent [7,11]. In the case of the QVT standard, update is automatically supported by the Relations language, while the user of OM has to implement this transformation feature by hand. The Relations language also allows to specify bidirectional transformations, which reduces effort in model synchronization scenarios. In OM it is in general necessary to specify multiple unidirectional transformations. However, it may not be possible and sensible for people using model transformation languages to construct complex transformations using a fully declarative approach [7].

Though the QVT standard allows to extend Relations with OM (hybrid transformation approach), no engine exists that can execute such a hybrid approach. Some MDE platforms will only provide one optimized execution engine onto which the transformation programs of different model transformation languages are mapped. When implementing such an approach, it is a good heuristic to map declarative and hybrid languages onto imperative languages and provide an execution engine for the imperative language. It is expected that e.g. translating Relations into OM does not expose obstacles [11]. However, the advanced features such as multidirectionality, automatic traceability, special transformation scenarios, etc., that are only supported natively by the Relations language and not by the OM language, have to be translated into imperative OM code and separate transformations.

In this paper we develop a translation of Relations into OM and implement it as a higher-order model transformation. Higher-order transformations take transformations as input and produce other transformations as output [2]. Our translation allows model transformation developers to specify the 'easy' things in a declarative way and profit from the additional features of Relations like support for model synchronization and model updates. By using our translation, one can implement 'hard' model transformation code in OM, execute the Relations code on a possibly optimized OM engine, and use tool support that is available for OM like debuggers, profilers, etc.. We implement our translation approach to show its feasibility and evaluate it with a UML to RDBMS transformation.

This paper is structured as follows: Section 2 gives an introduction to QVT and the Relations and the OM language. Section 3 describes the approach we follow for the translation and Section 4 presents the details of the translation. Section 5 describes the implementation, evaluates our Relations to OM translation and compares it with related work. The paper concludes with a summary in Section 6.

2 Basics

The OMG adopted the Meta Object Facility (MOF) 2.0 Query/View/Transformation specification (QVT) [12] as standard for model transformations. The QVT specification defines a hybrid transformation language. The three transformation languages *Relations*, *Core*, and *Operational Mappings (OM)* provide declarative and imperative transformation constructs. The languages *Relations* and *Core* can be used to specify declarative transformations. QVT provides

two options to extend declarative specifications with imperative transformation constructs, the *OM* language and *Black Box* operations.

2.1 UML to RDBMS Transformation Example

This paper presents a translation of Relations into OM, which we illustrate via the UML to RDBMS transformation described in the QVT specification [12].

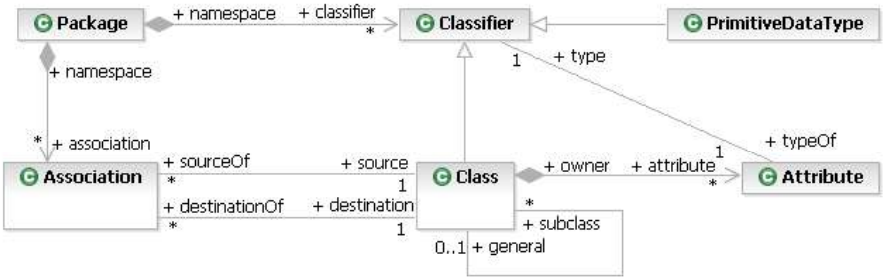

Fig. 1. Simple UML metamodel

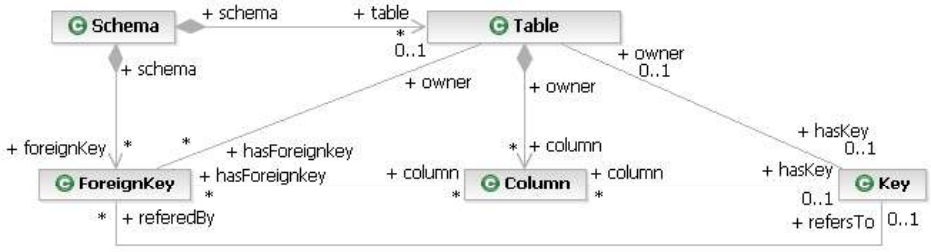

Fig. 2. Simple RDBMS metamodel

The UML to RDBMS transformation maps persistent classes of a UML model to tables of a model of a relational database management system (RDBMS). Figures 1 and 2 show the respective metamodels. The transformation basically works as follows: A persistent class maps to a table. Attributes of the persistent class map to columns of the table. An association between two persistent classes maps to a foreign key relationship between the corresponding tables.

2.2 QVT Relations

Listing 1.1 depicts an excerpt of the UML to RDBMS model transformation implemented in Relations. We will explain the concepts of Relations on the basis of this code.

In the Relations code the transformation *UmlToRdbms* is specified between the candidate models *uml* and *rdbms* as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a model type. For example, the element types of the *uml* model are restricted to those within the *SimpleUML* metamodel. Relations in a transformation like

PackageToSchema or *ClassToTable* declare constraints that must be satisfied by the elements of the *uml* and *rdbms* models. The relation *ClassToTable* is presented in more detail now.

First, a source and a target domain are declared that match elements in the *uml* and *rdbms* models respectively. *ClassToTable* is further constrained by two sets of predicates, a *when* clause and a *where* clause. The *when* clause specifies the conditions under which the relationship must hold, i.e. the relation *ClassToTable* must hold only when the *PackageToSchema* relation holds between the *Package* containing the *Class* and the *Schema* containing the *Table*. The *where* clause specifies the condition that must be satisfied by all model elements participating in the relation. Whenever the *ClassToTable* relation holds, the relation *AttributeToColumn* must also hold.

Listing 1.1. UML to RDBMS transformation in QVT Relations

```
transformation UmlToRdbms(uml: SimpleUML; rdbms: SimpleRDBMS) {
    key Table {schema, name};
    ...
    top relation PackageToSchema {...}
    top relation ClassToTable {
        cn, prefix: String;
        checkonly domain uml c:Class {
            namespace = p:Package {},
            kind = 'Persistent',
            name = cn
        };
        enforce domain rdbms t:Table {
            schema = s:Schema {},
            name = cn,
            column = cl:Column {
                name = cn+'_tid',
                type = 'NUMBER'
            },
            hasKey=k:Key {
                name = cn+'_pk',
                column = cl
            }
        };
        when {
            PackageToSchema(p, s);
        }
        where {
            prefix = '';
            AttributeToColumn(c, t, prefix);
        }
    }
    top relation AssocToFKey {...}
    ...
}
```

Each of the domains is also associated with several object template expressions used to match patterns in the candidate models. A template expression match for the *uml* domain results in a binding of the matching classes to the root variable *c* of the *uml* domain. Such template expression matches are only performed with regard to to the free variables of the domain. For the *uml* domain this applies to the variables *c*, *p*, and *cn*. The variable *p* is not free as it already has a binding resulting from the evaluation of the *when* clause expression. Pattern matching

proceeds by filtering any class with its *kind* property not set to *'Persistent'*. As *cn* is free, it gets a binding to the value of the name property for all remaining classes. Matching proceeds to the property pattern *namespace = p:Package*. As *p* is already bound in the *when* clause, the pattern only matches those classes whose *namespace* property has a reference to the same package that is bound to *p*. The three variables *c*, *p*, and *cn* make a three tuple and each valid match results in a unique tuple representing the binding.

The *uml* domain is marked *checkonly* and the *rdbms* domain is marked *enforce*. Thus, when executing the transformation in the direction of the *uml* domain, no elements are created in the *uml* model. If for example a table in *rdbms* exists with no corresponding class in *uml*, this is simply reported as an inconsistency. If the transformation is executed in the direction of the enforced domain *rdbms*, elements are created or modified in the target model *rdbms* so the relations between the candidate models hold. For example, for each valid class there must exist at least one valid table that satisfies the *where* clause. Otherwise, tables are created and properties are set as specified in the template expression associated with the *rdbms* domain. Also, for each valid table there must exist at least one valid class that satisfies the *where* clause. Otherwise, tables are deleted from the *rdbms* model so there it is no longer a valid match.

To create objects in the target model, object template expressions of the target domain are used. The template associated with *Table* specifies that a table object is created with the properties *schema*, *name*, *column*, and *hasKey* set to values as specified in the template expression. When creating objects, Relations ensures that duplicate objects are not created if the required objects already exist. The existing objects are updated. For this purpose, the concept of *key* is used defining a set of properties that uniquely identify an object instance. A *Table* is uniquely identified by its name and the *schema* it *belongs* to.

2.3 QVT Operational Mappings

OM is the target language of our translation. In the following, we shortly introduce the basic language concepts of OM. The concepts and the presented language constructs are used to specify the Relations into OM translation in Sections 3 and 4. Listing 1.2 illustrates a short example of OM code.

Listing 1.2. QVT Operational Mappings example

```
1  modeltype UML "strict" SimpleUml;
2  modeltype RDBMS "strict" uses SimpleRDBMS;
   transformation UmlToRdbms(in uml:UML,out rdbms:RDBMS) {
4    main() {
       uml.objectsOfType(Package)->map packageToSchema();
6    }
     mapping Package::packageToSchema() : Schema {
8      init { ... }
       population { ... }
10     end { ... }
     }
12 }
```

An operational transformation represents the definition of a unidirectional transformation that is expressed imperatively. It defines a signature indicating the models involved in the transformation. The signature of the *UmlToRdbms* transformation declares that an *rdbms* model of type *RDBMS* is produced from an *uml* model of type *UML*. The *UML* and *RDBMS* symbols represent model types. The model types are defined by the metamodels *SimpleUML* and *SimpleRDBMS*.

A operational transformation defines an entry operation for its execution named *main*. The main operation first retrieves the list of objects of type *Package* and then applies a mapping operation called *packageToSchema* on each *Package* of the list. A mapping operation like *packageToSchema* is an operation that implements a mapping between one or more source model elements into one or more target model elements. The *init* section contains some code to be executed before the instantiation of the declared outputs. The *population* section contains code to populate the result parameters and the *end* section contains additional code to be executed before exiting the operation. Between the *init* and the *population* sections, there is an implicit instantiation section which creates all output parameters that have a null value at the end of the *init* section.

3 Translation Approach

In this section we describe requirements, restrictions, and challenges for the translation of Relations into OM and introduce the overall translation algorithm.

3.1 Transformation Execution Direction

A crucial difference between the two languages is the execution direction of the transformation. OM transformations are unidirectional. Their execution direction is explicitly defined by their imperative statements, specifying which models are read and which ones are written. Relations transformations can be executed in any direction by selecting one of the candidate models as target. One may change the execution direction of a Relations transformation by selecting another target model, which is not possible in OM. The QVT specification does not clarify whether multiple target models are supported in the enforce mode. Examples are only provided for transformations with one target model. Hence, the described translation algorithm is restricted to one target model, which is is assumed to be the last parameter of the transformation and to appear as enforce domain in at least one relation. The translation algorithm generates the operational transformation in one direction towards the target model.

3.2 Model Transformation Execution Semantics

In the translation of programming languages into other programming languages, it is not sufficient to only map statements of the source language onto statements of the target language. The crucial and normally more challenging part is to

develop an accurate mapping of the execution semantics. This is also the main challenge when translating Relations into OM.

Relations performs a model transformation in a declarative way based on a powerful pattern matching mechanism and OCL constraints on the candidate models. This facilitates developing a consistent transformation for the user, but at the same time involves complex execution semantics with nested loops of object tuples for the execution engine [12]. In contrast, a transformation in OM is defined as sequence of statements executed by the engine step by step in the defined order. When performing a translation from Relations into OM, the transformation semantics must remain the same in spite of the different programming paradigms. The following aspects of the execution semantics have to be considered when translating Relations into OM:

Rule Scheduling. Relations uses implicit rule scheduling which is based on the dependencies among the relations. OM uses explicit internal scheduling where the sequence of applying the transformation rules is specified within the transformation rules. Our Relations into OM translation has to make the implicit rule scheduling of the Relations execution semantics explicit in the OM transformations. This has to be done in a way that has no (bad) side effects on the pattern matching and binding of the variables in the transformation occurs.

Pattern Matching. Relations uses pattern matching to find bindings of source and target model elements to the variables declared by the transformation. Pattern matching is based on the internal rule scheduling of Relations. When translating this mechanism to OM, the expressions in the relation domains must be organized into a sequential order and one has to take care that in the final OM code only variables are accessed that have been bound or at least defined before (cp. [12, p.17f]). Hence, we deal with pattern matching at various points in our translation; the most important issues are described in the Sections 3.3 and 4.4.

Check-Before-Enforce Semantics. The Relations semantics first performs a step where it checks whether a valid match exists in the target model that satisfies the relationship with the source model. Based on the checking results, the enforcement semantics modifies the target model so it satisfies the relationship with the source model. Through this check-before-enforce semantics Relations provides support for both generating new and updating existing target models. OM does not support updates of existing models automatically by its execution semantics. In OM this has to be implemented in the model transformation. The *generation* scenario can be realized by translating the checking semantics into rules that generate new model transformation elements. For *update* scenarios this has to be enhanced with functionality to modify and delete model elements.

3.3 Overall Translation Algorithm

Algorithm 1 gives an overview about the different steps performed during the translation. The algorithm first translates the transformation declaration. Before

the relations are translated one after another, they are sorted topologically to account for dependencies between them.

The main building blocks of a relation are domains, when clause, and where clause. In order to address the challenges and to keep the relational execution semantics in the imperative environment, it is essential to translate the building blocks in the designated sequence. The main issue is to ensure that assigned expressions only contain variables that have been bound before. For this purpose, the algorithm stores all variable values for each relation at all times in order to determine which variables have already been bound and which ones are still free.

Algorithm 1. Translation algorithm overview

```
 1: procedure RelationsToOperationalMapping(RelTrans) : OperationMapping
 2:     OmTrans ← ∅
 3:     OmTrans ← translateTransformationDeclaration(RelTrans)
 4:     Relations ← sortRelationsTopologically(RelTrans)
 5:     for all relation ∈ Relations do
 6:         OmTrans ← OmTrans ∪ translateRelationDeclaration(relation)
 7:         OmTrans ← OmTrans ∪ translateDomainDeclaration(relation)
 8:
 9:         OmTrans ← OmTrans ∪ translateWhenClause(relation)
10:         OmTrans ← OmTrans ∪ translateSourceDomains(relation)
11:
12:         OmTrans ← OmTrans ∪ translateWhereClause(relation)
13:         OmTrans ← OmTrans ∪ translateTargetDomain(relation)
14:     end for
15:     return OmTrans
16: end procedure
```

4 Realizing the Translation

In this section, the rules of the translation algorithm that implement the Relations into OM translation are presented. The structure of this section is aligned with the steps of the overall translation algorithm (cp. Algorithm 1).

The translation rules we describe in this paper cover all Relations language concepts that are relevant for the UML to RDBMS transformation. These are transformation and modeltypes, relations and domains, when and where clauses, pattern matching and restriction expressions, as well as keys and object creation.

4.1 Transformation Declaration

First, the transformation declaration is translated from Relations into OM. Lines 1 to 3 of Listing 1.3 depict the OM transformation declaration that is generated in the UML to RDBMS Relations example (cp. Listing 1.1). As a relational transformation is bidirectional, the direction of the parameters must be determined for OM. Source models are translated into *in* parameters. If the target model

is only used as *enforce* domain, it is translated into an *out* parameter; if it is used as *checkonly* domain in one or more relations, it is translated into an *inout* parameter. Each parameter of the relational declaration is also translated into a *modeltype* reference to import the respective metamodel packages. According to the Relations specification, type checking for the modeltypes is *strict*. This implies that all objects passed as parameter of the translation must be instances of the respective *modeltype*; subclasses of that type are not allowed.

4.2 Calculate Relations Topology Tree

In Relations it is not necessary to specify an explicit sequence of execution as rule scheduling automatically considers dependencies between relations. This is e.g. the case if a relation occurs as precondition in the when clause of another relation. In OM rule scheduling is explicit. OM requires a *main* operation as an entrance point as shown in lines 4 to 8 of Listing 1.3. For each toplevel relation in Relations, invocations are generated in that *main* operation that specify in which sequence the OM mappings are executed. If there are no dependencies between the toplevel relations, the respective OM mappings can be executed in arbitrary sequence. Otherwise, the correct sequence of execution is determined by a topological sorting algorithm in an iterative process.

That sorting algorithm regards the dependencies between the toplevel relations as a directed acyclic graph (DAG) whereas a node represents a relation and an edge represents a dependency between two relations. The initial structure of the DAG is built as follows. For each relation, a node is added. If a relation R1 is referenced in the when clause of another relation R2, an edge from R1 to R2 is added. If the where clause of R2 contains a reference to R3, an edge from R2 to R3 is added. For each toplevel relation, the algorithm now determines the number of incoming edges. In the first iteration, all toplevel relations are determined that have no incoming edges, which means they are not dependent from any other relation. The respective nodes and outgoing edges are removed from the DAG. This may result in some more toplevel relations that have no incoming edges, which are then processed in the same way in the next iteration. The algorithm terminates as soon as there are no toplevel relations with zero incoming edges left. Finally, calls to the respective OM mappings are generated in the main operation according to the determined sequence.

4.3 Relation and Domain Declarations

Relation declarations are translated into OM mapping declarations (cp. lines 9, 10, and 31 of Listing 1.3). For each domain in a relation, the algorithm generates a parameter with same type and name in the respective OM mapping. In doing so, the translation differentiates between the three kinds of domains.

- **Primitive domains** represent simple datatypes and are translated into *in-out* parameters in OM.
- The **enforce domain** is translated into the *result* variable in OM. If the relation is not top level, the *result* variable has already been bound before

the OM mapping is executed. For this reason, the generated mapping requires a parameter to which the previously bound result is passed and which initializes the *result* variable in the init block.
– All other domains are **checkonly domains**. The first one is translated into the context variable, which is then accessible using the *self* keyword in OM. Any further checkonly domains are translated into *in* parameters.

Listing 1.3. UML to RDBMS transformation in QVT Operational Mappings

```
modeltype SimpleUML "strict" uses UmlMM;
modeltype SimpleRDBMS "strict" uses RdbmsMM;
transformation UmlToRdbms(in uml: SimpleUML, out rdbms: SimpleRDBMS) {
  main() {
    uml.objects[Package]->map PackageToSchema();
    uml.objects[Class]->map ClassToTable();
    uml.objects[Association]->map AssocToFKey();
  }
  mapping Package :: PackageToSchema () : Schema {...}
  mapping Class :: ClassToTable () : Table {
    when {
      self.kind = 'Persistent';
      self.namespace <> null;
      self.namespace.resolveoneIn(PackageToSchema) <> null;
    }
    population {
      self.map AttributeToColumn(result);
      result.schema := self.namespace.resolveoneIn(PackageToSchema);
      result.name := self.name;
      var cl := object Column {
        name := self.name + '_tid';
        type := 'NUMBER';
      };
      result.column += cl;
      result.hasKey := object Key {
        name := self.name + '_pk';
        column := cl;
      }
    }
  }
  mapping Association :: AssocToFKey () : ForeignKey {...}
}
```

4.4 When Clause and Source Domains

In Relations, statements and OCL constraints in the source domains and in the when clause are used for filtering candidate models from the source domains. This is done by assigning objects and values to bound variables of a source domain. The purpose of unbound variables is to temporarily store values for the reuse in other domains of the relation, which e.g. allows for adopting a value from a source to the target domain. OCL constraints over the relation domains that are compliant with the QVT specification are supported by our algorithm.

Translating the When Clause. The when clause of a relation references other relations to represent preconditions of that relation. For each reference, the algorithm generates a call to the respective OM mapping. The sorting algorithm ensures that the called mapping has been executed before the calling mapping.

The execution semantics of Relations performs a pattern matching of the passed variables to the model elements for which the referenced relation holds. In OM, *resolve* expressions are used to perform such pattern matching. An appropriate *resolveIn* expression is generated in the *population* body of OM if a set of objects is passed; otherwise, a *resolveOneIn* expression is generated. This can be seen with the variable *s* in the relation call *PackageToSchema(p,s)* in the when clause (cp. lines 8 and 25 of Listing 1.1). As *s* is assigned to the bound variable *namespace* in the source domain, the *resolveIn* expression is performed on the respective variable *self.namespace* in OM (cp. line 18 of Listing 1.3).

Translating the Source Domain. An assignment to a bound variable according to pattern matching semantics filters model elements from the candidate models. Therefore, a respective condition is generated in the *when* block of OM. If a single value or object is assigned, the statement is adopted straightforward. In the example, the variable *kind* is used to filter all classes having that variable set to the value *'Persistent'*. This is translated into the operational statement *self.kind='Persistent'* (cp. line 12 of Listing 1.3). If a set is assigned to such a variable, an *xselect* condition is generated in OM instead. That *xselect* iterates over the candidate models and uses a condition that corresponds to the assigned set. The algorithm also considers cases that are not covered by the example such as multiple assignments to the same bound variable, which are translated into one combined expression using the logical *and* operator.

Assignments to unbound variables according to pattern matching semantics are not translated directly. Whenever such a variable is used at another place in the relation, the assigned value is used by the translation algorithm instead of the variable itself. This eliminates those variables in OM. The variable *cn* in lines 10 and 14 of Listing 1.1 gives an example. It is used to store the value of the attribute *name* of a *Class* and assign it to the variable *name* of the respective *Table*. The translation of such an assignment affects the target domain.

Furthermore, each variable bound to an object template must not be null. Therefore, respective conditions are generated in the *when* block of OM.

4.5 Where Clause and Target Domain

For candidate models that do match in the Relations source domain, the respective target models are generated according to the statements and OCL constraints in the where clause and the target domain. The respective model elements are created, changed, or deleted. If a target model does not exist, it is created from scratch.

Translating the Where Clause. In contrast to the when clause, a relation reference in the where clause represents a postcondition of the relation. Such a reference is directly translated into an invocation of the respective OM mapping at the beginning of the *population* block; cp. *AttributeToColumn(c,t)* in line 29 of Listing 1.1. Here, the passed variable *c* in the where clause represents the source domain and is therefore translated to the *self* attribute in OM. For the passed variable *t*, the algorithm generates the *result* attribute.

Translating the Target Domain. Variable assignments that modify the target model still remain to be translated. An example is given by the variable *cn* in lines 10 and 14 of Listing 1.1. In the source domain *uml*, the root variable *c* is represented by the variable *self* in OM. The attribute *name* is assigned to the variable *cn*. In the target domain *rdbms*, the root variable *t* is represented by the variable *result* in OM and the value of *cn* is assigned to the target variable *name*. The algorithm generates the respective assignment *result.name:=self.name* in the *population* body of OM (cp. line 19 of Listing 1.3).

If the assigned value occurs within an object template in the source and in the target domain, the translation is more complicated as the assignment happens within a set of objects. In this case, an appropriate *xcollect* expression is generated in OM and the += operator instead of := is used. That *xcollect* adds for each object in the source domain a respective object in the target domain.

In either case, an object expression is generated whenever an object template is used and its bound variable is bound for the first time. Thus, a new object must be instantiated in the imperative environment, which is e.g. the case with the variable *c1* in line 20 of Listing 1.3.

4.6 Updates of Existing Target Models

Updates of existing target models are automatically supported by the Relations semantics. In OM updates must be specified in the transformation explicitly. Model elements in Relations are uniquely identified by a set attributes specified by *key* expressions (cp. line 2 of Listing 1.1). For this purpose, the algorithm generates queries in OM that search for those model elements in the target model which have the same values for the identifying attributes as the respective model elements in the source model. These queries are performed before model element instantiation. The result object of an OM mapping is initialized with the result of the respective query as illustrated in Listing 1.4. If no respective model element is found, a new instance is created in the implicit instantiation section.

Listing 1.4. Updating an existing target model in QVT Operational Mappings

```
1 query findTable(name: String, schema: Schema): Table {
2   rdbms.objects()[Table]->xselectOne(t | t.name = name and t.schema = schema);
  }
4 mapping Class::ClassToTable(): Table {
    init {
6     result := findTable(self.name, self.namespace.resolveone(Schema));
    }
8 }
```

Relations also supports the deletion of model elements which are no longer valid. In OM the deletion of model elements must also be specified explicitly, which is not a trivial task. One approach is to delete all objects from the target model that cannot be found in the trace data of the transformation execution after all mappings have been executed (cp. Listing 1.5). However, there are issues with regard to object expressions as they do not generate trace data according

to the QVT specification [12]. Object expressions could be realized as mappings that do generate trace data. This again involves issues since the transformations would increase in length, for example.

Listing 1.5. Deleting objects in QVT Operational Mappings

```
main() {
  uml.objects()[Package]->map PackageToSchema();
  uml.objects()[Class]->map ClassToTable();
  uml.objects()[Association]->map AssocToFKey();
  rdbms.objects()->xselect(obj | obj.invresolve(true) = null)->forEach(obj){
    dest.removeElement(obj);
  };
}
```

A second approach is tagging all model elements that should not be deleted, which applies to model elements that are bound by the queries and that are newly created. A effective implementation of that approach depends on the concrete transformations and is not further regarded in this paper.

5 Implementation and Evaluation

In this section we present the implementation of the compiler and evaluate it with respect to the experience gained in the UML to RDBMS example.

5.1 Implementation

In order to demonstrate our translation approach, we developed an implementation of our algorithm as Eclipse plugin under the GNU General Public License [14]. The compiler is called QVT-Rel2Op and performs a translation from Relations to OM as described in Sections 3 and 4.

For this purpose, the compiler frontend takes two inputs: the Relations transformation as a textfile and the respective metamodels as emof models. A parser [13] generates a representation of the Relations transformation as emof model, which is passed to the compiler backend. In the backend an oAW workflow controls the further steps of the translation. The translation logic is implemented in Java and subsequently generates the respective OM transformation as emof model. A code generator and a beautifier generate a textual representation of that emof model and return an OM textfile as the result of the compilation.

The compiler implements important features of the Relations language. These are transformations, modeltypes, relations, domains, when clauses, where clauses, pattern matching, restriction expressions, keys, and object creation. However, some restrictions are made to the relational transformation. The compiler only allows two non-primitive domains, one source and one target domain. The check-only mode of Relations is not supported. For each binding of the root variable of the source domain, only one binding in the target domain is allowed.

5.2 Evaluation

Besides some other small examples, the UML to RDBMS transformation was taken to evaluate our translation approach and implementation. For this purpose, we executed a series of Relations transformations with ModelMorf [15], which is an execution engine for Relations. We then used our compiler to generate the respective OM transformation and executed the resulting transformation with SmartQVT [6], which is an execution engine for OM.

As SmartQVT does not support *resolveIn* expressions, a minor modification of the translation was required. For this purpose, the compiler offers a compatibility mode that generates appropriate *resolve* expressions instead, which are supported by SmartQVT. This works fine if all OM mappings return different object types. Finally, we compared the results of both transformations to each other and observed that the generated OM transformation returns the same results as the Relations transformation. This indicates that our algorithm correctly translates the relational execution semantics into the imperative environment.

We also compared our translation approach and implementation to others. As described in [10], there exist model transformation compilers for imperative model transformations. Thereby, languages like ATL or OM are mapped onto the ATL VM language [5,10], which serves as a basis for the execution of imperative model transformations. Other implementations compile model transformations into Java code. SmartQVT [6] generates Java code to execute OM transformations. [1] compiles model transformations defined by a combination of graph transformation and abstract state machine rules into transformer plugins for the EJB 3.0 platform. [17] provides an overview and comparison of further graph-based approaches compiling transformation rules into native executable code (Java, C, C++). Higher-order model transformations are also an elegant way to specify the semantics of model transformation languages [3]; the QVT specification [12] e.g. describes a translation of the Relations to the Core language. Other objectives of higher-order model transformation are to refactor and improve model transformations, increase the performance of model transformations, and maintain or upgrade model transformations [3,16].

6 Conclusions

In this paper we presented a higher-order model transformation that takes Relations model transformations as input and produces OM model transformations as output. Our implementation is a first realization of translating QVT declarative specifications into QVT operational specifications. Hence, it provides the basis for realizing the development of hybrid model transformations with QVT. Translating Relations into OM and not the other way round seems to be the natural way of realizing a hybrid approach for two reasons: first, all features of the declarative language can be translated into the imperative language without restrictions, which is not the case for the other direction [11]; second, hybrid approaches normally use declarative relations first, which are manually refined into operational rules later on [16].

Our translation allows developers to specify the 'easy' things in Relations and extend and execute their transformations as OM. It saves them implementing update functionality in OM code and gives them means to specify bidirectional transformations instead of several unidirectional OM transformations. This is especially beneficial in synchronization and conformance checking scenarios. The generated code can be executed on an optimized OM engine and developers can use tool support that is available for OM (editors, debuggers, profilers, etc.).

As future work, we will apply our approach and implementation to further transformations and case studies in order to gather more experience and address further scenarios. Moreover, we will realize further concepts of the Relations language such as in-place updates or support for multiple source domains.

References

1. Balogh, A., Varró, G., Varró, D., Pataricza, A.: Compiling model transformations to EJB3-specific transformer plugins. In: 21st ACM SAC, pp. 1288–1295 (2006)
2. Bézivin, J.: On the unification power of models. Software and System Modeling 4(2), 171–188 (2005)
3. Bézivin, J., et al.: Model Transformations? Transformation Models! In *9th MoDELS Conference*. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
5. Eclipse Project, A.T.L.: Use Case - QVT to ATL Virtual Machine Compiler, `http://www.eclipse.org/m2m/atl/usecases/QVT2ATLVM/`
6. France Telecom R&D. SmartQVT, `http://smartqvt.elibel.tm.fr/`
7. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. In: 1st MetaModelling for MDA Workshop, pp. 178–197 (2003)
8. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
9. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
10. Jouault, F., Kurtev, I.: On the architectural alignment of ATL and QVT. In: 21st ACM Symposium on Applied Computing, pp. 1188–1195. ACM Press, New York (2006)
11. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. Science of Computer Programming 68(3), 114–137 (2007)
12. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Final Adopted Specification. ptc/07-07-07 (July 2007)
13. Sourceforge. QVT Relations Parser, `http://sourceforge.net/projects/qvtparser/`.
14. Sourceforge. QVT Relations to Operational Mappings (2007), `http://sourceforge.net/projects/qvtrel2op/`
15. TRDDC. ModelMorf, `http://www.tcs-trddc.com/ModelMorf/`
16. Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: 7th UML Conference. LNCS, pp. 290–304 (2004)
17. Varró, G., Schurr, A., Varró, D.: Benchmarking for Graph Transformation. In: IEEE Symposium on VL/HCC, pp. 79–88. IEEE, Los Alamitos (2005)