Designing and Describing QVTo Model Transformations

Ulyana Tikhonova and Tim Willemse
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Keywords: Model Driven Engineering, Model Transformation, QVTo, Documentation, Software Design.

Abstract: Model transformations are the key technology of MDE that allows for software development using models as first-class artifacts. While there exist a number of languages that are specifically designed for programming model transformations, in practice, designing and maintaining model transformations still poses challenges. In this paper we demonstrate how mathematical notation of set theory and functions can be used for informal description and design of QVTo model transformations. We align the mathematical notation with the QVTo concepts, and use this notation to apply two design principles of developing QVTo transformations: structural decomposition and chain of model transformations.

1 INTRODUCTION

Model transformations are the key technology of Model Driven Engineering (MDE), as they allow for software development using models as first-class artifacts. Employing model transformations as the key development technology poses challenges typical for software engineering, such as design, documentation, and maintenance. To address these challenges one needs to be able to describe model transformations in an unambiguous and clear way.

Nowadays, there exist a number of languages specifically devoted for implementing model transformations, such as: ATL, QVT family of languages, ETL, etc. These languages can be viewed as DSLs for model transformations. Consequently, a notation for designing and documenting programs written in such languages should be specifically tailored to describing model transformations, thus disqualifying general purpose notations such as UML. In the current practice there exist no specific notation for describing model transformations (except those provided by the model transformation languages themselves). The common approach for documenting and/or explaining model transformations is to use concrete examples of their inputs and the corresponding outputs. Clearly, this approach provides an incomplete picture of the transformation and fails to give an overview of the transformation design.

In this paper we focus on QVT-Operational (QVTo), that was introduced as part of the MOF (Meta Object Facility) standard (omg, 2011). QVTo comprises both high-level concepts specific for model transformation development, such as constructs of the Object Constraint Language (OCL), traceability, inheritance of subroutines; and low-level concepts of the imperative languages, such as loops, conditional statements, explicit invocation of subroutines. Consequently, using QVTo requires strong language expertise. However, in practice QVTo is used by engineers who are experts in their own domains but typically have little to no computer science training and lack the required language expertise.

Following our experience of developing and maintaining a complex QVTo model transformation and of teaching QVTo to students, we propose to adopt the mathematical notation of functions and set theory as a notation-independent approach for documenting, explaining, and designing QVTo model transformations. Typically such mathematical concepts are familiar to most engineers. In this paper we demonstrate how this notation can be aligned with the QVTo concepts, and thus can be used for explaining QVTo concepts and for documenting model transformations (Section 3). Moreover, using this notation we formulate two common design principles of developing model transformations and show how the corresponding organization structure and information flow can be described (Section 4). We discuss and assess the proposed approach based on interviews with QVTo practitioners in Section 5. Related work is discussed in Section 2. Conclusions and directions for future work are given in Section 6.

Tikhonova U. and Willemse T..
Designing and Describing QVTo Model Transformations.
DOI: 10.5220/0005556004010406
In *Proceedings of the 10th International Conference on Software Engineering and Applications* (ICSOFT-EA-2015), pages 401-406
ISBN: 978-989-758-114-4

2 RELATED WORK

The broad problem of supporting the whole lifecycle of the model transformation development with a proper description notation is raised by Guerra et al. in (Guerra et al., 2013). In this study they propose a family of different visual languages for various phases of the development process: requirements, analysis (testing), architectural design, mappings overview and their detailed design. Although the authors state that the resulting diagrams should guide the construction of the software artifacts, there is no discussion in the paper on how to design model transformations. We use a single mathematical notation for describing three of the listed phases: architectural design (transformation chains), mappings overview (function signatures) and their detailed design (formulas).

The earlier works on a notation for describing and designing model transformations, such as (Etien et al., 2007) and (Rahim and Mansoor, 2008), propose graphical representations that are based on UML class diagrams. The major disadvantage of such approaches is the difficulty to describe the organizational structure of a transformation and the information flow through this structure from source to target models. This challenge is addressed in the visual notation of the MOLA transformation language (Kalnins et al., 2004), that combines 'structured flowcharts' for describing a transformation algorithm with (model) patterns for defining input and output of a transformation. Though MOLA aims for describing model transformations in an 'easy readable way', it introduces a number of specific visual means, which might require certain experience from a user.

The existing studies that specify model transformations using mathematical formalisms, *e.g.* (Idani et al., 2013), aim for formal analysis of transformations rather than for designing and maintaining them. Such research is orthogonal to ours, but it is interesting to see if they can be combined.

3 NOTATION

A model transformation takes as an input one or more source models and generates as an output one or more target models. An algorithm for such generation is defined in terms of source and target metamodels. In particular, the smallest units of a model transformation (*mappings* in QVTo) are defined in terms of *classes* and *associations* specified in the source and target metamodels. In order to provide a description of a model transformation using the mathematical notation of set theory, we need to specify how we refer to the metamodel concepts in this notation.

In the context of MDE, a metamodel is usually defined using the MOF standard. This standard can be seen as a subset of UML class diagrams. Following a common approach when applying set theory to describe class diagrams, we will interpret classes introduced by a metamodel as sets of objects that instantiate this class. For instance, the example metamodel depicted in Figure 1 introduces the sets *StateMachine*, *State*, *Transition*, *CompositeState*, etc. To refer to the associated classes, we use the dot notation common in object-oriented languages (including QVTo); for instance $s.states$ where $s \in StateMachine$.

In general, a model transformation defines a relation between a set of source models and a set of target models (Czarnecki and Helsen, 2006). A QVTo mapping is more restrictive, realizing a function that maps one or more source model objects into one or more target model objects. Thus, we can denote a QVTo mapping as a function from a set representing the source class to a set representing the target class. For example, the simple mapping depicted in the line 1 of Listing 1 can be viewed as a function with the following signature:

$$CloneTransition : Transition \rightarrow Transition \quad (1)$$

Mappings with multiple inputs and/or outputs can be treated in the same way, using Cartesian products of sets to indicate these inputs and/or outputs; *e.g.* the QVTo mappings of lines 2 and 3 in Listing 1 can be represented as follows:

$$MultTrans : \begin{array}{l} Transition \times State \\ \rightarrow Transition \end{array} \quad (2)$$

$$MultState : State \times State \rightarrow State \quad (3)$$

QVTo also allows a collection of objects to be used as an input and/or output of a mapping (line 4 in Listing 1). To indicate this we use the powerset $\mathbb{P}(S)$ of a set S:

$$MultiplyStateMachines : \begin{array}{l} \mathbb{P}(StateMachine) \\ \rightarrow StateMachine \end{array} \quad (4)$$

Function signatures allow us to concisely capture the purpose of a mapping in terms of source and target metamodels. To describe what a mapping actually does, we need to describe how elements of target objects are calculated from the elements of source objects. For this, we use formulas of the following form:

$$\begin{array}{l} CloneTransition(self) : \\ \quad trigger = \bigcup_{t \in self.trigger} \{CloneTrigger(t)\}, \\ \quad guard = CloneConstraint(self.guard), \\ \quad effect = \bigcup_{e \in self.effect} \{CloneBehavior(e)\}, \\ \quad ... \end{array} \quad (5)$$

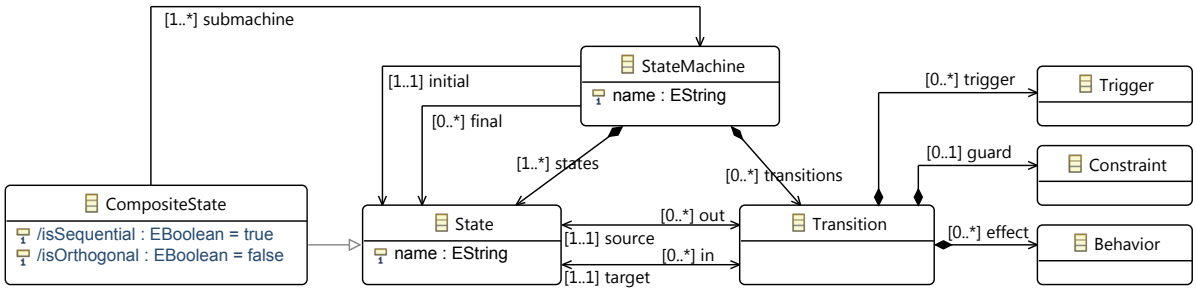

Figure 1: Metamodel of the UML state machine.

```
1 mapping Transition :: CloneTransition() : Transition {...}
2 mapping Transition :: MultTrans(in state : State) : Transition {...}
3 mapping Tuple(s1: State, s2: State) :: MultState() : State {...}
4 mapping Set(StateMachine) :: MultiplyStateMachines() : StateMachine {...}
```

Listing 1: Mapping declarations of the example QVTo transformation.

```
1 mapping Transition :: CloneTransition()
2                     : Transition {
3  result.trigger := self.trigger ->
4                    map CloneTrigger();
5  result.guard := self.guard.
6                  map CloneConstraint();
7  result.effect := self.effect ->
8                    map CloneBehavior();
9  ... }
```

Listing 2: QVTo code of the *CloneTransition* mapping.

Formula 5 describes the QVTo mapping depicted in Listing 2. It shows how each element of the target object is calculated by the invocation of other mappings on elements of the source object. We denote a calculation that is performed on a collection by a quantified union over elements of the collection (lines 1 and 3 of the formula correspond to lines 3-4 and 7-8 of the listing). The invocation of a mapping is indicated by the application of the corresponding function, such as *CloneConstraint*(*self.guard*).

4 DESIGN OF MODEL TRANSFORMATIONS

Designing model transformations is a challenging part of the model transformation development process. When designing a model transformation one must answer questions such as 'what are the mappings that constitute a model transformation?' and 'how are these mappings related with each other, *i.e.* invoked by each other?'. In practice, there exist a number of design principles that developers can follow when creating their model transformations (Lano and Rahimi, 2014). In this section we demonstrate how the presented mathematical notation can facilitate application of some of these design principles.

4.1 Structural Decomposition of Model Transformations

Model transformations construct target models from source models. Such models typically consist of objects connected with each other by associations. Therefore, mappings that constitute a model transformation should construct both target objects and the associations between them. In QVTo each mapping constructs a new object via assigning values to its properties, *i.e.* by constructing objects it is associated with. This observation is captured by the following principle for designing model transformations:

> mappings should be related to (invoked by) each other in the same way as the objects that they construct are associated with (composed of) each other.

In other words, the structure of a model transformation follows the structure defined by the target metamodel. In (Gerpheide et al., 2014), this was identified as one of the best practices for understandability and maintainability of transformations. To apply this principle in our design process we take the following steps:

1. We identify the inputs and outputs of the transformation and define their structures,
2. We establish the correspondence between the elements in input and output structures,
3. We capture these correspondences in signatures of the constituent mappings,

4. We decompose the transformation into constituent mappings.

In what follows, we show how these steps help design a mapping that takes two state machines and constructs a state machine representing their product. The source and target metamodel of this transformation is depicted in Figure 1. The function signature of the mapping is as follows.

$$\begin{aligned} &\mathit{MultiplyTwoSTMs} : \mathit{StateMachine} \times \\ &\mathit{StateMachine} \rightarrow \mathit{StateMachine} \end{aligned} \tag{6}$$

The task of writing such a transformation may not seem challenging, taking into account that we know the definition of a product of two state machines. However, it might be not so obvious how to implement this transformation in QVTo code.

First, we identify the structure of the inputs and outputs. For this we 'rewrite' each of the classes in the initial function signature with the classes that are associated with this one. The structure of the class *StateMachine* is determined by a tuple of classes referenced through the associations *states*, *transitions*, *initial*, and *final*. From this point of view, the class *StateMachine* can be seen as the relation $\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{Transition}) \times \mathit{State} \times \mathbb{P}(\mathit{State})$.[1] Thus, as a result of rewriting signature (6), we obtain the following:

$$\begin{aligned} &(\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{Transition}) \times \mathit{State} \times \mathbb{P}(\mathit{State})) \times \\ &(\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{Transition}) \times \mathit{State} \times \mathbb{P}(\mathit{State})) \rightarrow \\ &\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{Transition}) \times \mathit{State} \times \mathbb{P}(\mathit{State}) \end{aligned} \tag{7}$$

Second, we connect elements in input to output structures, i.e. we strive towards breaking down the structure in the formula. To this end, we redistribute the inputs on the left of the arrow and the outputs on the right of the arrow according to our understanding of *which input elements are required to construct which output elements*. For example, we know that the states of the resulting state machine are constructed from the states of the input machines. We capture this by grouping the input collections of *State*s and the output collection of *State*s in a separate *sub-signature* in Formula (8).

$$\begin{aligned} &(\,\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{State}) \rightarrow \mathbb{P}(\mathit{State})\,) \times \\ &(\,\mathbb{P}(\mathit{Transition}) \times \mathbb{P}(\mathit{Transition}) \times \\ &\quad \mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{State})) \rightarrow \mathbb{P}(\mathit{Transition})\,) \times \\ &(\,\mathit{State} \times \mathit{State} \rightarrow \mathit{State}\,) \times \\ &(\,\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{State}) \rightarrow \mathbb{P}(\mathit{State})\,) \end{aligned} \tag{8}$$

We do the same for the output transitions, the initial state, and the final states. The output transitions depend both on the input transitions and on the input states, thus we duplicate the input state collections on the left of the arrow for the second sub-signature. The resulting initial state is composed of the input initial states, see the third sub-signature in Formula (8). The same holds for the collection of final states, see the fourth sub-signature in Formula (8).

Third, we derive signatures of the constituent mappings. Each sub-signature, underlying the signature obtained in the previous step, captures a correspondence between source and target objects. We decompose the mapping *MultiplyTwoSTMs* into invocations of the constituent mappings according to these correspondences.

To derive signatures of the constituent mappings, we consider each sub-signature separately. The first sub-signature of Formula (8) is

$$\mathbb{P}(\mathit{State}) \times \mathbb{P}(\mathit{State}) \rightarrow \mathbb{P}(\mathit{State}) \tag{9}$$

In QVTo a $\mathbb{P}$-symbol that appears evenly on both sides of an arrow corresponds to the invocation of a mapping on an input collection and assigning the result of this mapping to an output collection. If the mapping operates on elements of the collections rather than on the collections, then we can reduce the corresponding collection symbols $\mathbb{P}$. Mathematically, such a reduction of $\mathbb{P}$-symbols corresponds to an *inverse of function lifting*. After applying this technique to signature (9), we derive the following underlying signature:

$$\mathit{MultState} : \mathit{State} \times \mathit{State} \rightarrow \mathit{State} \tag{10}$$

The resulting mapping *MultState* constructs a state of the target state machine as a pair of states from the two source machines. The initial and final states are constructed in the same way. Thus, the signature of *MultState* can be derived from the first, third, and fourth sub-signatures of formula (8).

Each output transition is a copy of an input transition duplicated as many times as its *source* and *target* states have been duplicated to construct the output states. As each state of an input machine is paired with all states of the other input machine, we conclude that each transition of an input machine is paired with all states of the other input machine. This knowledge is captured by regrouping the corresponding collections in the second sub-signature of formula (8):

$$\begin{aligned} &(\mathbb{P}(\mathit{Transition}) \times \mathbb{P}(\mathit{State})) \times \\ &(\mathbb{P}(\mathit{Transition}) \times \mathbb{P}(\mathit{State})) \rightarrow \mathbb{P}(\mathit{Transition}) \end{aligned} \tag{11}$$

The described pairing of transitions with states is applied symmetrically to both input machines. The target collection of transitions is constructed as the *union* of the two resulting collections. Therefore, we reduce signature (11) to the following underlying signature:

$$\mathbb{P}(\mathit{Transition}) \times \mathbb{P}(\mathit{State}) \rightarrow \mathbb{P}(\mathit{Transition}) \tag{12}$$

[1] Here we use the powerset $\mathbb{P}(S)$ of a set S to indicate that multiplicity of an association end has an upper bound greater than one.

```
mapping Tuple(m1: StateMachine, m2: StateMachine)::MultiplyTwoSTMs() : StateMachine
{
  result.states := m1.states -> collect(s1 | m2.states -> collect(s2 |
                       Tuple{st1=s1, st2=s2}.map MultState()));

  result.transitions := m1.transitions -> collect(t | m2.states -> collect(s |
                          t.map MultTrans(s)) )
            -> union ( m2.transitions -> collect(t | m1.states -> collect(s |
                          t.map MultTrans(s))) );

  result.initial:=Tuple{st1=m1.initial,st2=m2.initial}.map MultState();
  result.final := m1.final -> collect( f1 | m2.final -> collect( f2 |
                       Tuple{st1=f1, st2=f2}.map MultState()));
}
```

Listing 3: QVTo code of the *MultiplyTwoSTMs* transformation.

Observing that we can once more use a reverse function lifting, we finally arrive at the following signature for a constituent mapping:

$$MultTrans : Transition \times State \rightarrow Transition \quad (13)$$

Finally, we describe the implementation of a mapping in a formula using the notation we introduced in Section 3.

$$\begin{aligned}
&MultiplyTwoSTMs(m1,m2):\\
&states = \bigcup_{s1\in m1.states} \bigcup_{s2\in m2.states} \{MultState(s1,s2)\},\\
&transitions =\\
&\quad \bigcup_{t\in m1.transitions} \bigcup_{s\in m2.states} \{MultTrans(t,s)\} \cup\\
&\quad \bigcup_{t\in m2.transitions} \bigcup_{s\in m1.states} \{MultTrans(t,s)\},\\
&initial = MultState(m1.initial, m2.initial),\\
&final = \bigcup_{f1\in m1.final} \bigcup_{f2\in m2.final} \{MultState(f1,f2)\}
\end{aligned} \quad (14)$$

After we have come up with the design of the mapping and captured it in a formula, we *write the QVTo code according to this formula*. Listing 3 corresponds to formula (14).

4.2 Chaining Model Transformations

The principle described in the previous section is difficult to apply if source and target models have very different structures; for example, if mappings that match source and target elements are scattered over the metamodels structures, or if there are mutual dependencies between constructed objects. This type of situation is described in the literature as *structure clash* (Jackson, 2002), or *semantic gap* (van Amstel et al., 2008). Such design difficulty is usually solved using a *chain of model transformations*. In this way, structure clash is managed in a chain of intermediate steps each of which with a minimal clash.

```
helper CompositeState::Flatten()
        : states: Set(State),
          transitions: Set(Transition)
{
  return self.submachine ->
        map Orthogonal2Sequential().
        map SubstituteMachine(self._in,
                              self._out);
}
```

Listing 4: QVTo code of the *Flatten* transformation.

A chain of model transformations can be described using *function composition*: $f(m) = (f_1 \circ f_2)(m)$, where a model transformation $f : A \rightarrow B$ has a too wide gap between structures A and B, and therefore is split into model transformations $f_1 : A \rightarrow C$ and $f_2 : C \rightarrow B$. To construct a function composition, i.e. to connect intermediate steps into a chain of model transformation, we propose to use *currying*.

The example model transformation depicted in Listing 4 flattens a state machine by removing its composite states and replacing them with equivalent sets of simple states and transitions. This transformation consists of two steps: (1) transform orthogonal (parallel) composite state into a sequential composite state, (2) substitute the submachine of the composite state into the parent machine. The following formula describes the chain of these two steps using function composition and currying:

$$\begin{aligned}
&Flatten(state) =\\
&(Orthogonal2Sequential(state.submachine)\\
&\circ SubstituteMachine(state.in, state.out))
\end{aligned} \quad (15)$$

5 DISCUSSION & VALIDATION

Above we demonstrated how the mathematical notation of set theory can be used to describe QVTo mappings, their signatures and implementations, to derive organizational structure of mappings, and to describe the information flow in chains of model transformations. In this paper, we did not discuss QVTo constructs such as inheritance of mappings, imperative statements, OCL expressions, recursion, etc. We were, however, able to describe such concepts in a larger application (for instance, inheritance of mappings can be described using union of functions; and OCL expressions can be represented as predicates). While the proposed approach for describing and designing model transformations still requires further investigation and experiments, we have the following early validation results:

1. we applied the proposed approach when developing, documenting, and refactoring two different model transformations;
2. feedback provided by QVTo practitioners on our proposal (we performed three semi-structured interviews with developers from three different affiliations and different engineering domains).

Based on our experience and on the feedback of the QVTo practitioners, we conclude that:

- there is a need for a notation for documenting and designing model transformations;
- the description of QVTo mappings in the form of formulas is clear and understandable (also to engineers with no computer science background);
- the design principles can assist in the creative process of designing model transformations;
- designing a model transformation in the form of formulas before coding it in QVTo improves readability and understandability of the resulting code.

In addition to these results, we conclude that for the successful application of the proposed notation the following questions require further investigation:

- whether the proposed notation restricts the resulting design of a model transformation;
- what are the limits of the proposed notation, whether all essential situations can be described.

6 CONCLUSION

In this paper we showed how the mathematical notation of set theory and functions can be used to explain QVTo concepts, to facilitate the design process of model transformations, and to document model transformations. The resulting formulas give an overview of the organizational structure and the information flow of a transformation in an unambiguous and concise way. To assess the applicability of this approach, we applied it when designing, developing, and refactoring two model transformations[2], one of which is a complex transformation bridging a wide semantic gap. Moreover, we performed interviews with QVTo practitioners. While the results of these experiments and interviews are positive, the approach requires further investigation and experiments. In addition to the research questions listed in Section 5, we aim to examine if the proposed approach is language independent and can be used for developing in other model transformation languages, *e.g.* ATL.

REFERENCES

(2011). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. OMG. Version 1.1.

Czarnecki, K. and Helsen, S. (2006). Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645.

Etien, A., Dumoulin, C., and Renaux, E. (2007). Towards a Unified Notation to Represent Model Transformation. Research Report 6187, INRIA.

Gerpheide, C. M., Schiffelers, R. R. H., and Serebrenik, A. (2014). A Bottom-Up Quality Model for QVTo. In *QUATIC*, pages 85–94. IEEE.

Guerra, E., de Lara, J., Kolovos, D., Paige, R., and dos Santos, O. (2013). Engineering model transformations with transML. *Software and Systems Modeling*, 12(3):555–577.

Idani, A., Ledru, Y., and Anwar, A. (2013). A Rigorous Reasoning about Model Transformations Using the B Method. In *BPMDS*, pages 426–440.

Jackson, M. (2002). JSP in Perspective. In Broy, M. and Denert, E., editors, *Software Pioneers*, pages 480–493. Springer-Verlag New York, Inc.

Kalnins, A., Barzdins, J., and Celms, E. (2004). Model Transformation Language MOLA. In *MDAFA*, pages 62–76.

Lano, K. and Rahimi, S. K. (2014). Model-transformation design patterns. *IEEE Trans. Software Eng.*, 40(12):1224–1259.

Rahim, L. A. and Mansoor, S. B. R. S. (2008). Proposed Design Notation for Model Transformation. In *ASWEC*, pages 589–598. IEEE Computer Society.

van Amstel, M., van den Brand, M. G. J., Protic, Z., and Verhoeff, T. (2008). Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap? In *ICMT*.

[2]The source code and the description of the transformation used as an example in this paper are available online at code.google.com/p/qvto-flatten-stm/