MASTER

Visualization of model transformations in QVTo

van Dongen, M.

Award date:
2012

Link to publication
Visualization of Model Transformations in QVTo

Master’s thesis
Abstract

Model-driven software engineering (MDSE) is an engineering paradigm that is being more and more used as an alternative for traditional software engineering paradigms. In MDSE models play a central role in the development process, and model-to-model transformations (model transformations) are considered to be the heart and soul of MDSE. In the past years many model transformation languages have been introduced: ATL (ATLAS Transformation Language), Query/View/Transformation (QVT), Xtend, Kermeta (Kent Model Transformation Language), MOLA and YATL (Yet Another Transformation Language).

Traditional software engineering paradigms are supported by different traceability and debugging techniques and methods. However, since MDSE is a new paradigm, similar traceability and debugging techniques and methods are still rather limited in MDSE. Recent research on traceability and visualization of ATL transformations shows the benefits for debugging, coverage analysis and change analysis. For other transformation languages work is still to be done, as other languages require different traceability techniques and allow new visualization methods. This thesis is about visualization of model transformations in QVT-Operational (QVTo), which is one of the most widely-used and actively developed transformation languages nowadays.

We propose a method for the visualization of QVTo model transformations. In order to obtain this visualization, we introduce a hierarchical representation for QVTo transformations. Using this hierarchical representation in combination with the static and dynamic traceability information of a QVTo transformation, we visualize the relations between elements of the source and target models of a QVTo transformation.
Acknowledgements

This thesis is the result of my final master project of the master course Computer Science and Engineering at the Eindhoven University of Technology, Department of Mathematics and Computer Science. I have executed my master project within the Software Engineering and Technology Group under the supervision of Dr. A. Serebrenik.

I am thankful to my graduation supervisor, Alexander Serebrenik, for his guidance during my master project. Also, I would like to thank Mark van den Brand and Huub van de Wetering for taking place in my graduation committee. Furthermore, I would like to thank Luc Engelen and Marcel van Amstel for their feedback on the initial visualization methods. Besides, I want to thank Joep Ruiterman for his feedback on the grammar of my thesis.

Finally, I would like to thank my family and friends for their support and encouragements. Especially, I am grateful to my father, mother and sister for their supports over the years. Regretfully, for my father these words of gratitude come too late.

Maarten

Eindhoven, August 2012
# Table of contents

**Title**
Visualization of Model Transformations in QVTo

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Preliminaries</strong></td>
<td>6</td>
</tr>
<tr>
<td>2.1 Model-Driven Software Engineering</td>
<td>6</td>
</tr>
<tr>
<td>2.1.1 Domain-specific languages</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Model transformations</td>
<td>7</td>
</tr>
<tr>
<td>2.2 QVT Operational</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Model type definition</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Transformation initialization</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3 Mapping</td>
<td>10</td>
</tr>
<tr>
<td>2.2.4 Reuse of mappings</td>
<td>11</td>
</tr>
<tr>
<td>2.2.5 Resolving</td>
<td>13</td>
</tr>
<tr>
<td>2.2.6 Composing transformations</td>
<td>13</td>
</tr>
<tr>
<td>2.2.7 Intermediate classes and properties</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 Related work</strong></td>
<td>15</td>
</tr>
<tr>
<td>3.1 Traceability visualization</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Traceability of model transformations</td>
<td>17</td>
</tr>
<tr>
<td><strong>4 Design</strong></td>
<td>20</td>
</tr>
<tr>
<td>4.1 Requirements</td>
<td>20</td>
</tr>
<tr>
<td>4.1.1 Design choices</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Hierarchical interpretation of QVTo</td>
<td>21</td>
</tr>
<tr>
<td>4.3 Visualization method</td>
<td>24</td>
</tr>
<tr>
<td>4.3.1 Proposed visualizations</td>
<td>24</td>
</tr>
<tr>
<td>4.3.2 Final visualization</td>
<td>27</td>
</tr>
<tr>
<td><strong>5 Implementation</strong></td>
<td>30</td>
</tr>
<tr>
<td>5.1 Approach</td>
<td>30</td>
</tr>
<tr>
<td>5.2 Static traceability: Transformation analyzer</td>
<td>31</td>
</tr>
<tr>
<td>5.3 Dynamic traceability: the QVTo trace model</td>
<td>32</td>
</tr>
<tr>
<td>5.4 Building the trace tree</td>
<td>34</td>
</tr>
<tr>
<td>5.4.1 Processing a normal mapping</td>
<td>36</td>
</tr>
<tr>
<td>5.4.2 Processing a mapping with inheritance</td>
<td>36</td>
</tr>
<tr>
<td>5.4.3 Processing a mapping with merging</td>
<td>37</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Processing a mapping with disjunction</td>
</tr>
<tr>
<td>5.5</td>
<td>Generate TraceVis input file</td>
</tr>
<tr>
<td>5.6</td>
<td>Visualizing multiple transformations</td>
</tr>
<tr>
<td>6</td>
<td>Case study: UML model to relational database model</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>6.2</td>
<td>Visualizations</td>
</tr>
<tr>
<td>6.3</td>
<td>Conclusions</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Validation of requirements</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
</tr>
<tr>
<td>7.1</td>
<td>Future work</td>
</tr>
<tr>
<td>A</td>
<td>Transformation example: MMA to MMB</td>
</tr>
<tr>
<td>B</td>
<td>Transformation analyzer</td>
</tr>
<tr>
<td>C</td>
<td>Trace tree with only static trace information</td>
</tr>
<tr>
<td>D</td>
<td>Model transformation: UML to relational database</td>
</tr>
</tbody>
</table>
1 Introduction

Model-driven software engineering (MDSE, [1]) is one of the latest innovations in developing software. In MDSE the level of abstraction at which software is being developed is raised to a level where concepts in the domain in which the software has to be applied can be expressed effectively [2]. In MDSE the software is developed with the use of models that represent an abstraction of the system. One of the advantages of MDSE is that models are easier to verify and validate than source code, for example, it is much harder to check for deadlocks in source code than to check for deadlocks in the models [3].

One of the important techniques of MDSE is the model transformation. A model transformation transforms a set of source models into a set of target models. In MDSE, model transformation formalisms provide the mechanisms for manipulating and transforming models such that eventually these models can be used for purposes such as code generation, simulation and verification of software systems. Since MDSE is becoming more and more popular, model transformations also become more important. Because model transformations are becoming more frequently used, the need of traceability and debugging methods for model transformations has increased.

Traditional software development has various traceability and testing methods, for example requirement traceability, integration testing and unit testing. In MDSE these methods are still limited, which makes it more difficult to perform debugging in MDSE. Visualization of model transformations should improve the traceability of the transformation. One of the benefits of the traceability is that it makes debugging easier, for example the visualization can be used to see which parts of the transformation were used to generate an element in the target model. With visualization it is also easier to analyze the coverage of a transformation, this means to explore which parts of the source model are covered by the transformation and whether this includes all desired elements of the source model. Finally, with visualization it also becomes easier to check the impact of changes to the source model or the model transformation, by comparing the visualizations of the two versions.

Marcel van Amstel has introduced an approach to visualize ATL model transformations, [4]. Using this approach he was able to generate an input file for TraceVis to visualize the transformations. TraceVis is a tool to visualize the traceability across ordered hierarchies, [5].

The results of the visualization of ATL transformations are very promising. Therefore we try to extend this approach to visualize QVTo model transformations. The main difference between ATL and QVTo its style, ATL is declarative and QVTo is imperative. The imperative style of QVTo can be used to give a more detailed visualization of the model transformations. For example, in QVTo it is possible to make calls from a mapping to another mapping in the transformation. Moreover, it is also possible for a mapping to inherit functionality from another mapping in the transformation. Another aspect of QVTo which is interesting to visualize is the possibility for a transformation to extend the functionality of another transformation. It is interesting to try to visualize these aspects of QVTo in order to improve the understanding of the model transformation.
The remainder of this thesis is structured as follows. In Chapter 2, the preliminary information for this thesis is provided. This includes an introduction of the MDSE paradigm, and the syntax and semantics of the QVTo transformation language.

Chapter 3 describes the related work of this project. In the first section of this chapter the related work on traceability visualization is discussed. The second part of this chapter evaluates the existing work on traceability in model transformations.

In Chapter 4, the design of the developed approach is discussed. This is done by presenting a list of requirements for the visualization of model transformations. Followed by the introduction of a hierarchical representations for QVTo model transformations. The last section of this chapter discusses different possible visualization methods for a QVTo transformation, and presents the finally proposed method.

The implementation of the proposed approach is discussed in Chapter 5. In this chapter the traceability techniques are discussed, and how the traceability information is obtained and represented inside the hierarchical structure of Chapter 4. The last section of this chapter discusses how the input for the visualization tool is generated.

In Chapter 6, a case study is presented to validate the results of the visualization approach. For this case study the approach is applied on a model transformation to exhibit the different aspects of the visualization.

Finally, in Chapter 7 the conclusions of this master project are presented, and suggestions for future work are discussed.
2 Preliminaries

This section contains the preliminary information of this master project. Section 2.1 introduces the model driven software engineering paradigm, followed by explanation of the syntax and semantics of QVT Operational in Section 2.2.

2.1 Model-Driven Software Engineering

As long as software has been developed, people try to raise the level of abstraction in software development. In the early days of computing these abstractions resulted into programming languages like Assembly and early versions of Fortran, which saved programmers from writing complex machine code. This trend results in widely used modern object-oriented languages like C++, Java and C#.

Not all efforts to raise the level of abstraction were successful. One prominent effort was computer-aided software engineering (CASE) [6], which was introduced in the 1980s. CASE focused on developing methods and tools that enabled developers to express their software systems in graphical representations like, state machines, structure diagrams and dataflow diagrams. One goal of CASE was to enable more thorough analysis of programs which were less complex than analysis of programming code. Another goal was to visualize the implementation artifacts by the graphical representations to reduce the effort of coding and debugging. Although CASE attracted attention in research literature, it was not widely adopted in practice. One of the main problems of CASE was its inability to scale-up, for instance in complexity and size for industrial use. Furthermore it was limited in the range of application domains.

One of the latest efforts to raise the level of abstraction in software development has been the introduction of Model-Driven Software Engineering (MDSE) [2]. MDSE raises the level of abstraction by focussing on the software design rather than the implementation. In order to do so, MDSE consists of domain-specific models representing the software system. These languages are known as domain-specific languages. Another important aspect in MDSE is model transformation, which is used to transform the model representation of the system. In the next subsections domain-specific languages and model transformations are explained in more detail.

2.1.1 Domain-specific languages

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain” [7].
Since a DSL is specifically designed for a particular domain, the models that are created with the DSL can be understood by a domain expert. As an advantage, the domain experts can be involved more in the software development process. However, since DSLs are designed for a particular domain, their applicability is limited to this domain.

DSLs are described using metamodels, these models define the relationships between concepts in a domain and specify key semantics and constraints associated with these domain concepts. The models created with a DSL are instances of the DSL's metamodel. They describe elements of real-world objects and the relations between these objects. A metamodel is also a model which conforms to its own metamodel, a meta-metamodel. This results in the four-layered metamodeling architecture [8] depicted in Figure 2.1.

![Figure 2.1: The layered metamodeling architecture](image)

### 2.1.2 Model transformations

With model transformations an abstract model can be transformed into a more concrete model by adding more details or vice versa. A model transformation takes a set of source models and transforms them into a set of target models. Model transformations can also be used to transform a model into different formats, for example to transform the model into a format that is necessary for a particular application.

Using the layered architecture of Figure 2.1 it is also possible to create a layered architecture of a model transformation as depicted in Figure 2.2.

A model transformation is a model with its own metamodel, therefore it relates to the same architecture which is presented in Figure 2.1. It has a transformation definition at the M1 level.
which describes how elements of the source metamodel are related with elements of the target model. At the $M0$, level the run-time instance of the transformation definition is represented. This instance uses the input source model to generate the target model.

Several model transformation languages have been proposed so far, each with their own advantages and disadvantages. Examples are: ATLAS Transformation Language ([9]), Query/View/Transformation, Xtend ([10]), Kent Model Transformation Language ([11]), MOLA ([12]) and Yet Another Transformation Language ([13]). Because of its imperative style, making it interesting to visualize, this project focuses on the QVT Operational language. QVT Operational is part of Query/View/Transformation and described in Section 2.2.

![Figure 2.2: The transformation architecture](image)

---

8 Visualization of Model Transformations in QVTo
2.2 QVT Operational

Query/View/Transformation (QVT, [14]) is a set of model transformation languages defined by the Object Management Group. The QVT standard consists of three transformation languages, QVT-Core, QVT-Relations and QVT-Operational (QVTo). The first two are declarative languages, while QVTo is an imperative language. The remainder of this section introduces the QVTo language, including the constructs frequently used and important for the visualization of the transformation.

The examples that are used in this section are inspired by the QVTo tutorial presented in [15] and [14].

2.2.1 Model type definition

QVTo uses model types as references with metamodels. In Listing 2.1 two examples of model type definition are given. The model type definition consists of two parts: the name and the reference to the metamodel definition. Referring can be done by pointing to the Ecore-file in the Eclipse workspace. This is done by referring to platform:/resource/ followed by the location of the file in the Eclipse workspace. Another way of referring to a metamodel is by using the Uniformed Resource Identifier (URI) of the metamodel. In order to use this type of reference the metamodel should be registered in Eclipse.

Listing 2.1: References to metamodels.

```
1 // Location specific reference.
2 modeltype MMA uses "platform:/resource/MMA/MMA.ecore";
3
4 // Declaration using a package namespace URI.
5 modeltype MMB uses "http://mmb/1.0";
```

2.2.2 Transformation initialization

The transformation is initialized with the transformation statement, consisting of a name for the transformation, the input and output models. Besides the in and the out models it is also possible to define an inout model to update a model, this is also called an in-place transformation.

Listing 2.2: Transformation initialization and main function.

```
transformation MMATransformation (in Source:MMA, out Target:MMB); 

// The entry point for the execution of the transformation.
main () {
    Source.rootObjects () [MMA::Automaton] -> map toAutomaton();
}
```

Listing 2.2 shows an example of a transformation initialization and the entry point of the transformation. The main function of the transformation is the entry point of a transformation. From this main function the transformation is started. In Listing 2.2 the root objects of the source model of type Automaton are collected and the mapping toAutomaton is invoked for these objects. In Section 2.2.3 the term mapping is explained in detail.
2.2.3 Mapping

The transformation in QVTo is executed with the use of mappings. A mapping is used to transform an object of a given type into an object of another type.

When a mapping is invoked, its name is preceded by one of the keywords map or xmap. The difference between them is when the mapping fails, with map the mapping will return a null-value, while when xmap is used an exception will be raised.

Listing 2.3: Example of a mapping.

```java
mapping in Rectangle::fromRectangle(in name:String) : Location
when { self.Name.equalsIgnoreCase("a"); }
{
  init
  { log("Start of fromRect()"); }
  population
  { result.Name := self.Name + "_" + name;
    result.InitialState := self.IsInitial;
    result.Shape := "Rectangle";
    result.Dimension := self.Sides;
  }
  end
  { log("End of fromRect()"); }
}
```

In Listing 2.3, an example of a mapping is given. The mapping keyword is followed by the unique signature of the mapping. The signature of a mapping consists of the input object, the mapping name, optional parameters and the output object. In the example the input object is 'Rectangle', this object can be preceded by the direction keyword in or inout. When the direction is in, the keyword can be omitted. Between the parentheses, after the mapping name, the optional parameters of the mapping are defined. These parameters have the same direction options as the input object. The last element in the mapping signature is the definition of the output object type.

The optional pre- and postcondition of a mapping is respectively defined with the keywords when and where followed by a conditional expression. The body of a mapping is enclosed in curly brackets and consists of three parts, the init, population and end sections. The init section is optional and contains a set of expressions to be executed before instantiation of the output parameters. The population section contains the expressions to set the inout and output parameters. The population keyword and related parentheses can be omitted. Finally, the end section is used for expressions to perform the computations before leaving the mapping operation.
2.2.4 Reuse of mappings

The QVTo language offers three ways to reuse mappings. The following subsections discuss the three types that are available.

Inheritance

The first type of mapping reuse in QVTo is to inherit from another mapping. Imagine a metamodel in which there is an abstract type defined with a couple of subtypes that inherit from this abstract type. In QVTo it is possible to define a mapping to transform the abstract supertype and other mappings to transform specific attributes of the subtypes, where the first mapping is inherited by the other mappings. This is useful because the transformation of the elements of the supertype are defined once and are not repeated inside the mapping operations for every subtype. Also, invoking a mapping to transform a subtype will automatically invoke the mapping that transforms the elements that are inherited from the supertype.

Listing 2.4: Example: inheritance between mappings.

```qvto
mapping State :: Updates () : Location {
    Name := self.Name;
    InitialState := self.IsInitial;
}

mapping Rectangle :: fromRectangle () : Location
    inherits State :: Updates {
    init {
        log("This part is executed before Updates()");
    }
    Shape := "Rectangle";
    Dimension := self.Sides;
}
```

In Listing 2.4 an example is given of inheritance in QVTo, here `State` is the abstract supertype and `Rectangle` is its subtype. The execution of a mapping that uses inheritance differs from the normal execution of a mapping. In the case where the mapping `fromRectangle` is invoked it will execute the `init` section of this mapping first. After that, the complete inherited mapping `Updates` is executed and finally the remainder of `fromRectangle` is executed with the result of `Updates`.

When inheriting from another mapping, the signatures should be conforming to the inherited mapping. This means that the use of inheritance is only possible when the input type of the inheriting mapping is either identical to the input type of the inherited mapping or is a subtype of the input type of the inherited mapping. The same applies to the output types. These should be either identical to the output type of the inherited mapping or the output type of the inheriting mapping has to be a subtype of the output type of the inherited mapping.

Merging

Another type of mapping reuse in QVTo is merging. This can be used to invoke additional mappings depending on the values of the elements of the input object. Listing 2.5 shows an example of this type of mapping reuse. After the mapping signature of `Updates`, the keyword `merges` indicates the merging of an ordered list of mappings, in this case `UpdateInit` and `UpdateNormal`. When invoking the mapping `Updates`, the complete mapping `Updates` is executed first, including the possible `end` section. Then the ordered list of merged mappings is executed in sequence,
as long as their guard condition holds. In case of Listing 2.5 this means that either UpdateInit or UpdateNormal is executed, but it is also possible to execute multiple merged mappings.

Listing 2.5: Example: merging of mappings.

```plaintext
class State::Updates( )

    merges State::UpdateInit, State::UpdateNormal { 
        InitialState := self.IsInitial;
    }

class State::UpdateInit( )

    when {self.IsInitial} { 
        Name := "init_" + self.Name;
    }

class State::UpdateNormal( )

    when {!self.IsInitial} { 
        Name := "normal_" + self.Name;
    }
```

Disjunction

The third way to compose mappings is by means of the disjunction of mappings. In Listing 2.6 an example is shown. The keyword disjuncts indicates that the mapping toLocation contains a disjunction between mappings. The body of toLocation is empty because that part of a disjunction mapping is never reached. After the keyword disjuncts there is an ordered list of mappings. When invoking toLocation, the first mapping from this list whose guard condition holds will be executed. In case there is no mapping in the list which guard condition holds, the mapping toLocation returns ‘null’.

Listing 2.6: Example: disjunction between mappings.

```plaintext
class State::toLocation( )

disjuncts Rectangle::fromRectangle, Ellipse::fromEllipse { }

class Rectangle::fromRectangle( )

    inherits State::Updates {
        ...
    }

class Ellipse::fromEllipse( )

    inherits State::Updates {
        ...
    }
```

The disjunction construct is particularly useful to transform a set of elements with an abstract type containing elements of different subtypes. For example, in Listing 2.6 the type State is abstract and has two subtypes Rectangle and Ellipse. When transforming a set that contains elements of both types it is sufficient to invoke the mapping toLocation.
2.2.5 Resolving

An example of the resolution construct of QVTo is given in Listing 2.7. This construct is used when a source object is transformed to a target object and later on during the transformation another source object has a reference to this source object. In this case it should not transform the object again, with resolving it is possible to retrieve the already created target object.

Listing 2.7: Example: resolving of already transformed objects.

```
mapping Edge::toTransition() : Transition
{
    Event := self.Action;
    sourceLocation := self.container().oclAsType(State).resolveone(Location);
    targetLocation := self.targetState.resolveone(Location);
}
```

The mapping of Listing 2.7 transforms an Edge to a Transition. The Edge is contained by a state and has a reference to a target state, but these states have already been transformed to locations. However, the Transition has to refer to a sourceLocation and a targetLocation. In order to fulfill this reference resolveone is used to retrieve the Location object that is already created from the State objects.

2.2.6 Composing transformations

Besides mapping composition QVTo also offers the possibility to compose transformations. Listing 2.8 gives an example. There are two techniques to compose a transformation; access and extends. With access the import behaves as a traditional package import, whereas extends combines package import and class inheritance.

Listing 2.8: Example: transformation composition.

```
transformation CompleteUml2Rdbms(in uml:UML, out rdbms:RDBMS) access transformation UmlCleaning(inout UML), extends transformation Uml2Rdbms(in UML, out RDBMS);
main() {
    var tmp : UML = uml.copy();
    var retcode := (new UmlCleaning(tmp))−transform();
    . . .
}
```

In Listing 2.8 the extended transformation Uml2Rdbms requires the input UML to be a “clean” model without redundant associations. This is why the accessed transformation UmlCleaning is used to perform an in-place transformation on the UML to make it suitable for Uml2Rdbms.
2.2.7 Intermediate classes and properties

In a QVTo transformation it is possible to define intermediate classes and properties. These classes and properties can be used to make certain parts of the transformation easier. In Listing 2.9 an example of both is given, first an intermediate class is defined and on the second line this class is added as an intermediate property to the type Class. The intermediate classes and properties can be used in the entire transformation, however, they do not appear in the target model as they are temporarily.

Listing 2.9: Example: intermediate classes and properties.

```java
intermediate class LeafAttribute {
    name : String;
    kind : String;
    attr : Attribute;
}

intermediate property Class::leafAttributes : Sequence(LeafAttribute);
```
3 Related work

The related work on the subject of this thesis is discussed in this chapter. In Section 3.1 related work on traceability visualization is presented. In the second section, Section 3.2 the related work on the traceability in model transformations is discussed.

3.1 Traceability visualization

A lot of research has been done on the subject of traceability in computer science, especially focused on the software engineering domain. As a result of the research on traceability different software maintenance tools have been developed that support users in better understanding and changing existing software systems [16]. Such tools provide the user with a wealth of information about the software, for example information about the dependencies between the user requirements of a software project. Often traceability links are needed to support such tasks. In [17] and [18] three ways are presented to represent traceability links: matrices, cross-references and graph based models (see Figure 3.1).

Figure 3.1: Examples of three basic traceability link visualizations [18].

In Figure 3.1a an example is given of a traceability matrix. In the traceability matrix the horizontal and vertical dimension list the items that can be linked. The links between items are represented by the entries in the matrix [17]. The traceability matrix is easy to understand, which is an advantage. However, when dealing with a lot of artifacts traceability matrices can become very large and unreadable [18]. Another limitation of a traceability matrix is that it is difficult to follow a trace across a couple of links, since an item is represented by a column or a row instead of a single item. This requires the user to jump to different places in the matrix to follow one trace [18].

A second method to represent traceability links are cross-references, as depicted in Figure 3.1b. Cross-references are usually easy to understand since they can be expressed using natural language [17]. However, the representation of traceability links with cross-references is very narrow, the user only sees incoming and/or incoming links for just one artifact. Also with many references from and to one single artifact it is not always obvious which one is relevant for the particular trace the user is trying to follow [18].
The third common method for traceability visualization is the graph based representation, shown in Figure 3.1c. Every node in the graph represents an artifact of the system, the directed edges between nodes indicate the relations between the artifacts. When the number of nodes and edges is limited, the graph based representation can easily be used to follow a single trace of references. Another advantage of this representation is its ability to represent traceability links with arity higher than two [17]. However, this representation is less suitable when the number of nodes and edges increases a lot. This can result in visual clutter and reduces the ability to analyze traceability [18].

Figure 3.2: From Marcus et al. [19], cross-referencing method based on colors.

Numerous derivative methods have been developed from these three common methods for traceability visualization. One of them is proposed by Marcus et al. in [19]. It represents various parts of artifacts with small rectangles, as shown in Figure 3.2. The colors of the different parts are used to visualize the relations, i.e. parts with the same color are related. This visualization method is relatively simple and intuitive. However, as the number of artifacts and dependencies increases this method becomes less suitable, because one cannot distinguish hundreds of colors. Also it is not possible to visualize directions of relations.

Figure 3.3: Visualization of relations between hierarchical ordered data using hierarchical edge bundling [5].
Another visualization method to represent traceability is proposed by Van Ravensteijn [5], as shown in Figure 3.3. This method is particularly suitable to visualize traceability between hierarchical ordered data and is already successfully applied to software projects [20]. The artifacts are represented with vertical columns, containing various subparts. Traces are represented with lines between the related parts.

The approach of Ravensteijn is suitable for large datasets since the relations are drawn using the hierarchical edge bundling technique [21]. This reduces the visual clutter as the number of trace links increases. However, with this method the visualization of trace links between elements of the same column leads to visual clutter [5].

Bavota et al. presented a traceability tool called TraceME. This tool supports the software engineer by capturing traceability links between different types of artifacts [22]. The tool uses information retrieval techniques to find the traceability links between items of the different artifacts. It is possible to give feedback on the results which are proposed by the tool and this feedback will be used to improve future recoveries of traceability links. However, the visualization of the traceability links in TraceME is limited since it uses a graph based representation with poor scalability.

### 3.2 Traceability of model transformations

In [23] Galvão and Goknil present a survey of traceability approaches in MDSE. They introduce three traceability categories: requirement-driven approaches, modeling approaches and transformation approaches. The first category, requirement-driven approaches, contains traceability approaches that follow the life of requirements during the software development process. The modeling approaches category contains techniques that are interesting when wanting to know how metamodels and models are involved in the tracing process. Finally, the transformation approaches use the model transformation mechanisms for generating trace information. The survey of Galvão and Goknil shows that tool support is crucial to automate the traceability in MDE. Also the representation of trace information plays a crucial role in achieving the benefits of applying traceability techniques [23].

Von Pilgrim et al. introduces a technique to represent traceability among a chain of model transformations [24]. Their visualization uses a three-dimensional space and represents the models as two-dimensional planes in this space. Elements of different models are connected to each other using lines (Figure 3.4). An advantage of this approach is that the representation of the models in the two-dimensional planes equals the representation when developing the models. However, this approach is not scalable for larger models or longer transformation chains. Moreover, the three-dimensional space introduces overlapping problems which may affect the clarity of the visualization [25].
In [26] a traceability framework is proposed for the Kermeta language, which is model-oriented programming language. The framework consists of a metamodel able to store the traces between source and target objects. They extend the model transformation to generate a model which contains the trace information. Visualization of the trace information is not part of the traceability framework, which is a disadvantage of the framework [23].

Van Amstel et al. proposed an approach to visualize ATL transformation by adding tracing information to the transformation [27]. This approach is depicted in Figure 3.5. It uses an higher-order ATL transformation to add the tracing information to the ATL transformation that needs to be visualized. This part is called ‘Trace adder’ in Figure 3.5. The higher-order ATL transformation results in the so-called ‘Augmented model transformation’, and thus execution of this transformation by the ‘Transformation engine’ produces a trace model in addition to the target models.

Figure 3.5: Approach for the visualization of ATL transformations [27].
The next step in the process is to generate a TraceVis input file based on the source, target and trace models. Figure 3.6 shows an example of the visualization of an ATL transformation with TraceVis. In this figure the left and right columns represent respectively the source and target model, while the middle column represents the transformation itself. The lines between the columns represent the trace links between elements of the source models and the transformation, and between the transformation and elements of the target models.

Figure 3.6: Example of the model transformation visualization from [27].
4 Design

4.1 Requirements

From the previous chapters we know which aspects of traceability are important to make it useful. One of the important aspects is the tool support to automate the generation of the trace information [23]. The further the generation of trace information is automated, the lesser the amount of work that needs to be done, thus making traceability easier to apply. Obviously, the representation of the trace information also plays a crucial role in achieving the benefits of applying traceability techniques [18, 23], since the representation of the trace information determines the usability of it.

The study of related work on traceability visualization and traceability of model transformation results in the following list of requirements for visualization of QVTo transformations:

**REQ1** The visualization tool should support visualization of the entire chain of trace links from a source model element to target model elements.

**REQ2** The visualization tool should include the visualization of internal calls (Definition 4.1.1) between mapping operations.

**REQ3** The visualization tool should automate the generation of the trace information.

**REQ4** The visualization tool should support visualization of the QVTo specification as described in Section 2.2 i.e. this excludes helper operations.

**REQ5** The visualization tool should support visualization of a chain of model transformations (Definition 4.1.2).

**REQ6** The visualization tool should support scalability for large model transformations and chains of transformations.

In order to clarify the requirements the following definitions are introduced:

**Definition 4.1.1.** The notion of internal call is defined as a mapping invocation which is not explicitly defined in the body of a mapping. This includes mapping invocations that are initiated by mappings using inheritance, disjunction or merging of mappings.

**Definition 4.1.2.** The notion of chain of model transformations is defined as a set of model transformations that have been executed in a specified unique order. During the execution of this set of transformations the target model of each preceding transformation is used as source model for the subsequent one.
4.1.1 Design choices

Based on the conclusions of the research on traceability visualization in Section 3.1, the design choice is made to use TraceVis for the visualization of QVTo transformations. This choice is supported by the results in [27], which show the suitability of TraceVis to visualize model transformations. This also makes it possible to define requirements that are based on the features of TraceVis:

**REQ7** The TraceVis timeline-function should be used to allow step-by-step visualization of the model transformation.

4.2 Hierarchical interpretation of QVTo

TraceVis is designed to visualize the traceability across dynamically ordered hierarchies [5]. In the Eclipse Modeling Framework (EMF, [28]) models are already defined in tree-structures, i.e. one can use them as an ‘hierarchy’ in TraceVis. Since the QVTo transformations is one of the ‘hierarchies’ that we want to include in our visualization we have to introduce an hierarchical interpretation for it.

Based on the QVTo syntax described in Section 2.2, we define hierarchical dependencies in a QVTo transformation. When analyzing a QVTo transformation top-down we can see that it consists of two types of operations, these are an **entry operation** and a set of **mapping operations**. The entry operation is more commonly known as the **main** operation of the transformation, this operation is used to call one or more mapping operations.

For the set of mapping operations it is also possible to define an hierarchical dependency. Although a mapping operation is more complex than the entry operation, since a mapping operation has more options. Unlike the entry operation, a mapping operation is not limited to make only calls to one or more other mapping operations. As explained in Section 2.2.4 a mapping operation may reuse other mappings defined in the transformation. Besides a set of called mappings, this means that a mapping can also be associated with a set of inherited and/or merged mappings. A special case applies when the mapping provides a disjunction between a set of mappings, since a mapping which contains a disjunction does not have a **body** section. This means that this type of mapping cannot have a set of called, inherited or merged mappings.
From this information we can define the following hierarchical structure for a QVTo transformation:

1. The transformation has an entry operation
   (a) The entry operation is the main operation
      - The main operation contains one or more called mappings
2. The transformation has a set of mapping operations
   (a) Mapping has no further relation with other mappings
   (b) Mapping is a disjunction between a set of mappings
   (c) Mapping has a relation with other mappings and one or more of the following cases applies:
      - Mapping has a set of one or more called mappings
      - Mapping has a set of one or more inherited mappings
      - Mapping has a set of one or more merged mappings

Although a QVTo transformation has only one entry operation the item $1a$ is added in the structure above for the sake of symmetry. This item $1a$ ensures that the main operation and the mapping operations are on the same level in the hierarchical structure.

In Listing 4.1 an example of a QVTo transformation is shown to demonstrate how this transformation is hierarchically interpreted. This transformation needs an input model with a root object of type $A$ and a set of elements of abstract type $B$, these elements can be either of subtype $B1$ or of subtype $B2$. The target model of the transformation has a root object of type $X$ and a set of attributes of type $Y$.

![Hierarchical representation of the QVTo code.](Figure 4.1)

The transformation starts with the main operation on line 6. For all root objects of type $A$ of the source model, the mapping fromA2X is invoked. In the fromA2X mapping on line 12 the fromB2Y mapping is called for all members of elements. Since this fromB2Y mapping is a disjunction, either mapping SubB1 or mapping SubB2 is invoked for members of elements, depending on their type (which is $B1$ or $B2$). Finally, on lines 20 and 25 it is defined that both mappings, SubB1 and SubB2, inherit the mapping Updates.
Using the defined hierarchical structure and Listing 4.1 we can create an hierarchical tree-structure of the QVTo code as shown in Figure 4.1. The top level of Figure 4.1 shows the complete ‘Transformation’ with the ‘Entry Operation’ and the ‘Mapping Operation’ set at the second level. At the third level of the figure the operations of the transformation of Listing 4.1 are shown. The fourth level of Figure 4.1 shows the type of relations that operations have with the other mappings. The bottom level contains the related mappings which are references to the third level mappings. Using this hierarchical structure the transformation of Listing 4.1 can be represented as a hierarchy in a TraceVis visualization.
4.3 Visualization method

The main challenge when visualizing a QVTo transformation is to correctly visualize the language constructs of QVTo. For example, Section 2.2 shows that the syntax of QVTo contains constructs for inheritance and disjunction of mappings. These constructs each have their specific execution pattern. In order to create a visualization that correctly shows the traceability among the model object these specific execution patterns should be considered during the creation of a visualization method.

As stated before, an important difference between QVTo and ATL is the fact that QVTo has an imperative style, while ATL has a declarative style. The imperative style of QVTo has the advantage that it gives insight into the order in which transformation steps are executed. Another effect of the imperative style is the possibility to invoke other mappings from within a certain mapping. The advantage of these internal calls is that they allow more detailed traces between source and target elements. These advantages give the possibility to create more detailed visualization of QVTo transformation compared to the ATL visualizations of Van Amstel et al. in [27].

4.3.1 Proposed visualizations

In this section two methods to visualize a QVTo transformation are discussed. The metamodels depicted in Figure 4.2 and Figure 4.3 are used to create models of automata as shown in Figure 4.4. The QVTo transformation for models of metamodel A to metamodel B can be found in Listing D.1 in Appendix A.

![Figure 4.2: Metamodel A](image)

24 Visualization of Model Transformations in QVTo
In Figure 4.5 the first visualization of the transformation of the model from Figure 4.4 is shown. In this figure the leftmost column represents the source model and the rightmost column shows the target model. In the middle the transformation from source to target model is visualized. The traces between elements of the source model and target model are depicted by the lines between the columns. The relations between the source model column and the transformation column visualize the input of the transformation. The relations between the transformation column and the target model column visualize the output of the transformation. The lateral relations between elements of the transformation column show the internal calls between mappings.
In order to follow the trace of the transformation of a Rectangle state in the source model into a Location in the target model, the green elements and lines in Figure 4.5 should be followed. Inside the mapping toAutomaton on line 14 of Listing D.1 for all states of the input automaton the mapping toLocation is invoked. This is visualized by the relation between the element of the source model and the top green element of the transformation column. The next step is to invoke the toLocation of the actual mapping. This is visualized by the green relation between the top green element and the middle green element of the transformation column. Since toLocation is a disjunction between mappings the first mapping from the ordered list on line 26 of Listing D.1 for which the guard condition holds, is invoked. Thus, in the case of a Rectangle object the mapping fromRectangle is invoked, this is visualized by the green relation between the middle green element and the bottom green element of the transformation column. The last step of this trace is the green relation from the transformation column to the Location object in the target model.

The lateral relations are the major drawback of this first visualization method in Figure 4.5. In TraceVis lateral relations quickly cause visual clutter and thus it is recommended to avoid these type of relations [5]. Moreover, in TraceVis lateral relations are automatically drawn on both sides of the transformation column, since a column is mirrored when it is between two other columns in TraceVis.

The desire to avoid the lateral relations results in the second type of visualization which is shown in Figure 4.6. For this visualization the transformation column has been duplicated and the internal relations (i.e. lateral relations in Figure 4.5) are now visualized between the two transformation columns.
Following the trace of the transformation of the *Rectangle* object into a *Location* object has become different. The first and last step of the trace remained the same compared to the visualization in Figure 4.5. The transitive closure property is applied on the relations between the two transformation columns. The top green element in the left transformation column is related with the middle green element and related with the bottom green elements. Because of the transitive closure property this also means that the middle element is related with the bottom element. The removal of the lateral relations results in less cluttering. Compared to the visualization in Figure 4.5 the visualization in Figure 4.6 is therefore clearer. However, one of the disadvantages of the visualization as shown in Figure 4.6 is that the transitive closure makes it impossible to trace the order of consecutive internal calls. Another problem is the visual difference between the two transformation columns: although both columns represent the same transformation, they look differently. The reason for this problem is that TraceVis changes the column layout depending on which elements are used in a relation. For example the *toAutomaton* element in the left transformation column of Figure 4.6 is used in a relation, and thus it is drawn different compared to the *toAutomaton* element in the right transformation column.

### 4.3.2 Final visualization

Both visualization methods have been discussed with experts in the model transformation domain, Marcel van Amstel and Luc Engelen. They gave valuable feedback on the visualization which is used to create the final visualization method. The duplication of the transformation columns was considered to have a positive effect on the clarity of the visualization. However, the use of transitive closure for the relations between the two transformation columns was not perceived as being intuitive. Another comment on the visualizations was the relation between the transformation column and the produced element in the target model. In the two visualizations of Figure 4.5 and Figure 4.6 the mapping invocation is related to the produced target element. Instead, it was proposed to relate the produced target element with the actual mapping of the transformation were the element is created. When considering Figure 4.5 and Figure 4.6 the bottom green element in the right transformation column should be related to the green element in the right column, instead of the relation between the top green element in the transformation column and the element in the target column.
In Figure 4.7 the final visualization method is shown. For this visualization the hierarchical edge bundling has been turned off, since the effect of this technique is limited for small visualizations. Without the hierarchical edge bundling, the relations are depicted as straight lines. In this visualization the consecutive internal transformation calls are visualized with a chain of relations between the two transformation columns, i.e. from the left transformation column to the right transformation column and vice versa, instead of the transitive closure that is used in Figure 4.6. This automatically shows the order in which a serie of calls is executed, since for every invocation of a transformation element a unique instance child is added to the transformation element in the transformation columns. The only problem occurs when the chain ends in the left transformation column, then a ‘silent step’ is made to the right transformation column to make it possible to reach the target model column. In Figure 4.7 these silent steps are depicted by the horizontal relations between the two transformation columns.

Another improvement to the visualization which was proposed by one of the experts was to use the timeline feature of TraceVis to allow to visualize the transformation step-by-step. Because of the imperative style of QVTo, the order of consecutive steps in the transformation can be derived. This information can be used to add timestamps to the relations between the different columns. In Figure 4.7 and Figure 4.8 the timeline is shown at the bottom of the visualization, the ticks on the timeline show when changes are made to the visualization. The visualization in Figure 4.7 has the slider of the timeline at the rightmost position, this shows the complete visualization. While in Figure 4.8 the slider is around the middle of the timeline and visualization only shows the first steps of the transformation.
Figure 4.8: Example of the use of the TraceVis timeline function for step-by-step visualization.
5 Implementation

5.1 Approach

When developing tooling, the creation of TraceVis input should be automated as much as possible. Initially the aim was to copy the approach that is used for the visualization of ATL model transformations which is presented in [27] and depicted in Figure 3.5. This approach uses three steps to generate the visualization of the ATL transformation. In the first step a higher-order transformation is used to create an augmented ATL transformation. The second step is to execute the augmented model transformation. In addition to performing the normal model transformation the augmented transformation also produces a trace model of the transformation. This trace model contains trace links between elements of the source model and elements of the target model. The last step is generating an input file for TraceVis based on the source, target and trace models.

This approach to ATL transformation visualization is not applicable to QVTo model transformations: it is currently not possible to define a higher-order model transformation that has an executable QVTo transformation as its target model. However, when executing a QVTo transformation it is possible to enable an option to save a QVTo trace model from the execution of the transformation. Unfortunately, this trace model only contains an ordered list of the executed mappings with their references to elements of the source and target models. This means that this trace model does not contain information about the internal calls between mappings and usage of disjunction, inheritance and merging of mappings. Therefore it is necessary to retrieve this missing information in another way.

![Figure 5.1: Approach for the tooling.](image-url)
In Figure 5.1 the approach to retrieve the information that is needed to generate TraceVis input files is depicted. This approach starts with two steps that can be executed in parallel. One of these steps consists in executing the model transformation and saving the QVTo trace model. The dashed-line arrows from the QVTo trace model to the input and output models indicates that the trace model contains the relations between the elements of the input and output models. The other step is analyzing the model transformation itself to retrieve the static information about the internal calls between mappings and the disjunction, inheritance and merging of mappings. This step is performed by the ‘Transformation analyzer’ in Figure 5.1. The last step is generating a TraceVis input file based on the input, QVTo trace and output models and the analysis of the QVTo transformation.

In order to create traceability of QVTo transformation following the requirements in Section 4.1 the approach of Figure 5.1 combines static traceability information and dynamic traceability information, obtained respectively from the Transformation analyzer and the QVTo trace model. These techniques are explained in the Section 5.2 and Section 5.4.4.

In the approach of Figure 5.1 the interpretation of the source and target models by the ‘TraceVis input file generator’ is reused from the ATL visualization tooling developed by Marcel van Amstel. The hierarchical interpretation of the source and target models has been reused since ATL and QVTo both require models defined using the Eclipse Modeling Framework.

### 5.2 Static traceability: Transformation analyzer

One of the blocks in Figure 5.1 represents the ‘Transformation analyzer’. The transformation analyzer is the part of the traceability which performs static traceability on QVTo model transformations. The static traceability focusses on the trace information which is unaffected by the input of a transformation, this trace information is called ‘static’ since it is equal for every possible execution of the transformation. With static traceability the structure of the QVTo transformation is traced, this includes information about the calls between mappings and the use of disjunction, inheritance and merging of mappings. The traceability that is executed by the Transformation analyzer only considers the defined QVTo transformation itself.

The Transformation analyzer is a QVTo transformation itself, the implementation of this transformation can be found in Appendix B. The Transformation analyzer uses the fact that a QVTo model transformation conforms to a metamodel itself, this means that inside a transformation the QVTo transformation can be used as a model. In other words the QVTo transformation that the user wants to visualize can be used as the input for the Transformation analyzer. Inside the analyzer all mapping operations of the model transformation are identified, also the potential relations with other mappings are considered (i.e. mapping calls, disjunction, inheritance and merging). Using this information the target model is generated. This target model conforms to the metamodel that is shown in Figure 5.2. Using this target model it is possible to trace back the internal relations and dependencies between the mappings of the transformation.
5.3 Dynamic traceability: the QVTo trace model

In order to complete the traceability for the visualization of an executed QVTo transformation, the static traceability information is combined with dynamic traceability information. It is possible to save a QVTo trace model of the execution of a QVTo model transformation. This trace model is created by the QVTo engine during the execution of the transformation. For every execution of a mapping a trace record is generated, containing four attributes: the executed mapping, input elements, parameters and output elements.

As an example Table 5.1 shows the content of the QVTo trace model for the execution of the transformation in Appendix A on the model of Figure 4.4 in Chapter 4. The first column contains the executed mappings of the transformation and the second and fourth column respectively contain the input and output elements of the transformation. The third column shows the parameters of the mapping operations, in this example there are no optional parameters used by any mapping in the table.
To illustrate the limitation of the trace information of Table 5.1 consider the first two entries of the table: the executions of the mappings `toAutomaton` and `fromRectangle`. Line 5 of Listing 5.1 shows that for all elements of `self.states` the `toLocation` mapping is invoked, however this mapping is not mentioned in Table 5.1. This is because invoking a disjuncting mapping only the executed mapping from the disjuncts list that is mentioned in the QVTo trace model; in this case the `fromRectangle` mapping. This means that in order to visualize the disjunction correctly this dependency between `toLocation` and `fromRectangle` has to be retrieved from the static traceability of the transformation.

Also, on line 2 of Listing 5.2 it is specified that `fromRectangle` inherits from `Updates`. The trace model of Table 5.1 shows immediately after the trace record of the `fromRectangle` mapping a record is defined for the execution of `Updates`. In this case the static traceability of the transformation also has to be retrieved and visualize this inheritance dependency.

Table 5.1: Content the QVTo trace model for the execution of the transformation of Listing D.1 with the model of Figure 4.4.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Input elements</th>
<th>Parameters</th>
<th>Output elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. toAutomaton</td>
<td>Automaton (MMA)</td>
<td>-</td>
<td>Automaton (MMB)</td>
</tr>
<tr>
<td>2. fromRectangle</td>
<td>Rectangle S1</td>
<td>-</td>
<td>Location S1</td>
</tr>
<tr>
<td>3. Updates</td>
<td>State S1</td>
<td>-</td>
<td>Location S1</td>
</tr>
<tr>
<td>4. fromEllipse2</td>
<td>Ellipse S2</td>
<td>-</td>
<td>Location S1</td>
</tr>
<tr>
<td>5. Updates</td>
<td>State S2</td>
<td>-</td>
<td>Location S2</td>
</tr>
<tr>
<td>6. fromEllipse2</td>
<td>Ellipse S3</td>
<td>-</td>
<td>Location S3</td>
</tr>
<tr>
<td>7. Updates</td>
<td>State S3</td>
<td>-</td>
<td>Location S3</td>
</tr>
<tr>
<td>8. toTransition</td>
<td>Edge A</td>
<td>-</td>
<td>Transition A</td>
</tr>
<tr>
<td>9. toTransition</td>
<td>Edge C</td>
<td>-</td>
<td>Transition C</td>
</tr>
<tr>
<td>10. toTransition</td>
<td>Edge B</td>
<td>-</td>
<td>Transition B</td>
</tr>
</tbody>
</table>
5.4 Building the trace tree

The `TraceVis input file generator` of Figure 5.1 combines the static and dynamic traceability information to build a trace tree. The trace tree is a data structure conforming the hierarchical interpretation for a QVTo transformation defined in Section 4.2. In Figure 5.3, the implementation of the trace tree is shown, the `TraceTree` object is used to store nodes of the trace tree.

![Data structure for the TraceTree](image)

The `TraceTree` has an attribute `label` that can be used to give the node a name. The value of this attribute is displayed in the final visualization with `TraceVis`. With the `parent` attribute it is possible to have a reference to the parent node in the trace tree, when the element is at the root of the trace tree the `parent` attribute is a `null` value. The attribute `branches` contains a list of tree nodes that are direct children of the `TraceTree` element. The `nodeType` attribute defines the type of the node, this attribute is used to ensure the hierarchy of the trace tree. Each node has lists for the incoming and outgoing links, `sourceElements` contains URI's that refer to elements of the source model and `targetElements` contains URI's that refer to elements of the target model. In order to visualize the internal calls of the transformation both lists can also contain references to other elements of the trace tree.

In Figure 5.4, the hierarchy between the different values of the `NodeType` enumeration is shown, this hierarchy is based on the design in Section 4.2. This hierarchy shows that the root of the trace tree has to have the value `Transformation` as its `NodeType`. The children of the `Transformation` node have to be `OperationType`, this is used to make a distinction between the entry operation of the transformation and the set of mapping operations. Below the node with type `OperationType` the children have to be of type `OperationName`. At this level the actual operations are defined. An `OperationName` node can have two types of children, namely `Instance` and `SubOperationType`. In case of the `Instance` type the node represents a single invoke of the operation of its parent node.
Nodes with type SubOperationType are used to make a distinction between the type of reference that the parent operation node has with other operations in the transformation. Possible values for a SubOperationType node are ‘init’, ‘population’, ‘inherits’, ‘disjuncts’ or ‘merges’. Every node with type SubOperationName contains the name of a related operation in the transformation. The children of a SubOperationName node are again of the type Instance and represent a single call to the related operation.

In order to build the trace tree in the ‘TraceVis input file generator’ (of Figure 5.1), first the static trace of the ‘Transformation analyzer’ is used to build a tree that contains the structure of the QVTo transformation. This intermediate trace tree already contains all the nodes of the final trace tree, except for the Instance nodes since these are generated during the processing of the dynamic trace of the ‘QVTo trace model’. Figure C.1 in Appendix C shows the intermediate trace tree with the NodeType values for the transformation in Appendix A with only the static trace information.

To finalize the trace tree the information of the dynamic trace is added to the trace tree. This means that Instance nodes are added that represent the actual invokes of the mappings operations and contain the references to elements of the source model and elements of the target model.

For the creation of the Instance nodes an ordered list of trace records is used. This list is extracted from the ‘QVTo trace model’ and in Table 5.1 an example of such a list is shown. The ordering of this list is based on the order in which the mappings have been invoked during the transformation.

Beside this ordered list of trace records there is also a stack with the ‘active mappings’. An active mapping is a mapping of an already processed trace records for which it is possible to invoke another mapping. This stack of active mappings is used when determining from which mapping a certain mapping is invoked. The stack is initialized with one element which references to the entry operation of the transformation, the main operation, since from this entry operation the first mapping of the transformation is invoked.

Figure 5.4: Hierarchy between the NodeType values.
5.4.1 Processing a normal mapping

The first trace record of Table 5.1 that has to be added to the trace tree is the invocation of the toAutomaton mapping. Figure 5.5 shows the Instance nodes that are added for the invocation of toAutomaton. The first step consists in selecting the first node of the stack, this is the main operation. For this main operation it is checked whether the operation has a call to the toAutomaton mapping. Since this is the case the Instance node \( n \) is added as child of the node that indicates the invocation of the toAutomaton mapping for an element of the source model.

![Figure 5.5: Insertion of Instance nodes for a normal mapping invocation.](image)

The next step is to search for the toAutomaton node in the ‘Mapping operation’ subtree. When this node is found the Instance node \( n' \) is added as child of the toAutomaton node. The last step is to add the trace links. Node \( n \) has an incoming link from the source model, and an outgoing link to node \( n' \). This means that node \( n' \) automatically has an incoming link from node \( n \), and it also has an outgoing link to the target model. After the trace information for the first trace record of Table 5.1 is added to the tree, the toAutomaton mapping is added to the stack of active mappings. This means that for the second trace record of the table it will first be checked whether the toAutomaton mapping contains a call to that mapping operation, if this is not the case the toAutomaton mapping is removed from the stack.

5.4.2 Processing a mapping with inheritance

In Figure 5.6 the insertion of Instance nodes in case of an inheritance is shown. When the fromRectangle mapping is invoked, normally the instance node is added like described in Section 5.4.1. However the fromRectangle node used inheritance, and from Section 2.2.4 we know that the inherited mapping is executed after init section of fromRectangle. In Table 5.1 it is shown that every trace record for the fromRectangle and fromEllipse2 mappings is followed by a record for the Updates mapping. This means that first the Instance node \( n \) is added to represent the call to the Updates mapping, then the Instance node \( n' \) is added as child of the Updates node. Finally, the Instance node \( n'' \) is added to the fromRectangle node.
Figure 5.6: Insertion of Instance nodes in case of an inheritance.

The links are from the invocation of fromRectangle to node \( n \), and from node \( n \) to node \( n' \). From node \( n' \) the link to node \( n'' \) is created, and from \( n'' \) there is a relation with the target model. The reason to create a link from the invocation of fromRectangle to node \( n \) instead of node \( n'' \) is because of the fact that the Updates mapping is executed between the init and population section of the fromRectangle mapping. This means that it is possible to use the result of Updates in the remainder of the fromRectangle mapping.

5.4.3 Processing a mapping with merging

In case of a merge of mappings the insertion of Instance nodes is similar to the case of inheritance. The only difference is that the links between the Instance nodes are different. If in Figure 5.6 the 'inherits' would be replaced by 'merges' the incoming invocation of fromRectangle is linked with the Instance node \( n'' \). The outgoing link of node \( n'' \) would be linked with the node \( n \), and node \( n \) is linked to node \( n' \). The outgoing link of \( n' \) goes to node \( n'' \) and the node \( n'' \) is again linked with the target model.

5.4.4 Processing a mapping with disjunction

Another special case for the insertion of Instance nodes applies to traces that include mappings that use disjunction. Section 5.4.1 states that after the first trace record of Table 5.1 is processed, the stack of active mappings contains the toAutomaton mapping on top. For the second trace record of Table 5.1 it is checked whether toAutomaton contains a invocation of fromRectangle. This is not the case although it does call a mapping that disjuncts the fromRectangle mapping.
From we know that there are no trace records for disjunctive mappings, this means that the \textit{toLocation} mapping is never mentioned in the list of trace records. From the static traceability information we know that \textit{toLocation} contains a disjunction that includes the \textit{fromRectangle} mapping, this information is used to insert the correct \textit{Instance} nodes. First the \textit{Instance} node $n$ is added as child of the \textit{toLocation} invocation, there is an \textit{Instance} node $n'$ as child of \textit{fromRectangle} in the disjunction list, and the \textit{Instance} node $n''$ is added to the node that represents the \textit{fromRectangle} mapping. The source model is linked with the node $n$, the outgoing link of $n$ goes to node $n'$. This node $n'$ is linked with node $n''$, and the node $n''$ is linked with the target model.

5.5 Generate TraceVis input file

When the trace tree is completed, the actual TraceVis input file can be generated. The data has to be stored following the XML format. In this format there are two sections, namely stages and relations.

Inside the stages section the hierarchical ordered sets are defined by stage tags. The elements of the hierarchical ordered set are defined by node tags inside the corresponding stage tag. For a node tag the following attributes can be defined: the name, parent node and priority level. The relations section contains relation tags that represent the relations between nodes that are defined inside the stages section. For each relation tag the following attributes can be defined: the type of relation, source node and target node. The complete definition of the TraceVis input format can be found in [5].

In Section 4.3 the design choice was made to visualize the transformation using two transformation columns in TraceVis, thus two stage tags. As a result of this choice, prior to the generation of the XML data the trace tree is duplicated into two identical trace trees. In both trace trees links are removed in such a way that the left tree contains the incoming links from the source model, and the right tree has the outgoing links to the target model.
The internal links of the transformation are distributed between the left and right trace tree, such that every link for the left tree ends in the right tree and vice versa. When a chain of consecutive internal links ends in the left tree it is not possible to define a relation from the element in the left tree to an element in the target model. TraceVis does not allow a relation between two stages that are not next to each other. As mentioned in Section 4.3.2 this is solved by the addition of a silent step between the element in the left tree and the element in the right tree that represents the same transformation instance.

Figure 5.8: Generated TraceVis visualization for the example in Figure 4.4 and transformation in Appendix A.

The generation of the TraceVis input file consists of two steps, first the stage tags representing the source and target models and the transformation are created. For the two stages that represent the transformation, the priority attribute of the node tags is used to distinguish the different node types of the corresponding elements in the trace tree. In Figure 5.8 this results in the different coloring of the element in the two middle stages of the visualization that represent the transformation. The second step consists of generating the relation tags between the stages. The type attribute of the relation tag is used to distinguish the relations, there are three types: from source model to transformation, relations inside the transformation, from transformation to target model and the silent relations from the left transformation stage to the right transformation stage. Each relation type has its own color in Figure 5.8. The XML data for the source and target models of the transformation is generated in the same way as it has been done for the ATL visualization.
5.6 Visualizing multiple transformations

With the implementation described in this chapter it is also possible to visualize a chain of model transformations (see Definition 4.1.2). In this case the target model of each preceding transformation is used as source model for the subsequent transformation in the visualization. Moreover, for every transformation, contained in the chain of consecutive model transformation, the Transformation analyzer has to be executed. In Figure 5.9 shows an example of a visualization that contains two consecutive model transformations. Between the two pairs of transformations columns the model is visualized which is the target model of the first transformation, and the source model of the second transformation.

Figure 5.9: Example of the visualization of two consecutive model transformations.
6 Case study: UML model to relational database model

For this case study one of the examples of the Query/View/Transformation Specification document [14] is used to demonstrate the visualization techniques. The purpose of this case study is to show the different aspects of the visualization on a relatively large QVTo transformation to demonstrate the scalability of this visualization approach.

6.1 Introduction

The transformation that is used for this case study transforms a given UML model into a relational database model. This transformation generates the target relational database model based on the information that is collected from the UML diagram. In Table 6.1 the statistics of the transformation and the input source model are presented.

<table>
<thead>
<tr>
<th>Number of mapping operations</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of lines of code</td>
<td>387</td>
</tr>
<tr>
<td>Number of objects in the source model</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 6.1: Statistics of the transformation and source model.

In Figure 6.1 the UML diagram is shown that is used as source model for the transformation of this case study. The diagram shows the design for a laptop shop. In this shop a laptop is defined by a Configuration object. This Configuration consists of a LaptopModel, CPU, Memory, HardDisk and GraphicsCard component. These five components are all based on the abstract Component object, this implies that the components are identified by their componentId.

There are two types of orders defined in the UML diagram, the ClientOrder and the ConfigurationOrder. The ClientOrder is used to store information about an order of a specific configuration by a client, this information includes the order status, shipment type and payment status of the order. With the object ConfigurationOrder the orders to the manufacturers for new configurations are stored. Moreover, the Manufacturer and Client classes are used to store information about the suppliers and clients of the laptop shop.
The generated relational database model based on the source UML model is depicted in Figure 6.2. This model shows that every class of the diagram in Figure 6.1 is transformed into a database table. The attributes of the classes have been transformed into fields of the tables. Furthermore, the status and shipmentType enumerations of the ClientOrder class in the UML model are transformed to a normal columns in the ClientOrder table. Moreover, the object references in the UML model are transformed to foreign key relations in the database model, for example in case of the ConfigurationOrder table. The ConfigurationOrder object in the UML consists of a Manufacturer and Configuration object. For the transformation to the ConfigurationOrder table these objects have been replaced by the manufacturer_manufacturerId and configuration_configurationId columns respectively, with foreign key relations to the Manufacturer and Configuration tables.
In Figure 6.1 the Client and Manufacturer classes use the Address datatype to store the address information of a customer. For the transformation into a database table this address information has been added to both tables. Furthermore, the UML model contains the abstract Component class which defines the general information of the five subtypes: LaptopModel, CPU, Memory, HardDisk and GraphicsCard. In the relational database table these subtypes have each been transformed to a table and the attributes of the abstract supertype Component have been added as columns to these tables.

The model transformation that is used to transform the UML model into the relational database model can be found in Appendix D. In general, the transformation transforms the persistent classes of the UML model into database tables, i.e. the classes: Client, Manufacturer, ClientOrder, ConfigurationOrder, Configuration, CPU, Memory, LaptopModel, HardDisk and GraphicsCard. Basic type attributes of a persistent class are transformed into a column in the database model, these basic types are strings, integers and boolean values.

The transformation of attributes that refer to other instances inside the UML diagram is more complex. If an attribute of a persistent class A refers to an instance of a persistent class B the...
identifying element of that class \( B \) is added to table of \( A \) and a foreign key to table \( B \) that is transformed from that persistent class \( B \). For example when looking at table ClientOrder in Figure 5.2, this table has the client_clientId column which contains the identifier of the Client table and a foreign key relation on the client_clientId column.

References in a class to non-persistent classes, i.e. Address, and enumerations, i.e. Order-Status and ShipmentType, are different. For example, the persistent Client class contains an attribute that refers to an instance of the non-persistent Address class. The table that is created for the Client class contains a column for every attribute of the Address class. In case a class contains an attribute that refers to an enumeration, the table representation of that class contains a column to store the value of the enumeration.

6.2 Visualizations

This section shows the visualizations of the transformation from the UML model of Figure 6.1 to the target relational database model of Figure 6.2. The figures in this section demonstrate how the visualizations can be used to observe the different aspects of the model transformation.

In Figure 6.3 the complete visualization of the transformation is shown. In this figure the left column represents the source UML model and the right column represents the target relational database model, the two columns in the middle represents the QVTo transformation that has been executed on the source model.

The visualization of Figure 6.3 shows trace links of different colors. These colors indicate the type of the relation, Figure 6.4 shows the legend with the meaning of each color. There are four types of relations that can be distinguished in the visualization: internal relations, input relations, output relations and silent relations. This last type, the silent relations, is only used when the chain of internal relations for a transformation trace ends in the left transformation column. In that case a silent relation is used to reach the equivalent object of the transformation in the right column to be able to reach the elements in the target model that are created during this transformation trace.
In order to maintain a good overview of the transformation the hierarchical edge bundling of TraceVis is used in the visualizations. With the edge bundling in Figure 6.3 it can immediately be observed that the incoming links of the target model can roughly be divided into ten groups, corresponding to the ten database tables of the model in Figure 6.2. When selecting one of these groups, as shown in Figure 6.6, it can be seen that this highlights the \texttt{ClientOrder} table in the target model. It is also visualized that in the source model there are three classes highlighted, these are the classes \texttt{Client}, \texttt{Configuration} and \texttt{ClientOrder} itself. The classes \texttt{Client} and \texttt{Configuration} from the source model have a relation with the \texttt{ClientOrder} table in the target model since the \texttt{ClientOrder} class contains refers to these two classes (see Figure 6.1).

TraceVis offers the option ‘Hide Unselected’, using this option on the visualization of Figure 6.5 the visualization of Figure 6.7 can be obtained. By only visualizing the relevant trace links and elements in the source and target models, it is possible to observe the details of the creation of the \texttt{ClientOrder} table. In Figure 6.6 it is, for example, possible to see the names of the different columns of the \texttt{ClientOrder} table. Moreover, this visualization makes it easier to study the traces of every single element that is related to the \texttt{ClientOrder} table in the source and target model. However, a disadvantage of this visualization is that the symmetry between the two transformation columns is lost, because when a element is selected in the left transformation column, the corresponding element in the right transformation column is not selected. Thus, it is hidden in the right transformation column and the symmetry between the left and right column is affected.
Figure 6.6: Only visualizing the traces that are related to the generation of the `ClientOrder` table.

With TraceVis it is also possible to hide one or more columns. In order to retain a correct visualization it is not recommended to only hide one transformation column, since that gives a visualization with lateral relations that does not give a correct representation of the model transformation. Moreover, lateral relations are considered to have a negative affect on the ‘readability’ of the visualization as is discussed in Section 4.3.1. For future work it is an option to block the possibility to hide only one of the transformation columns. In Figure 6.7 a visualization is shown where both transformation columns have been hidden. This gives a more clear view of which source and target model elements are related with each other.

Figure 6.7: Hiding the transformation columns to visualize the links between source and target elements directly.
In Figure 6.8 it is shown how the timeline of TraceVis can be used to step-by-step walk through the transformation process. The visualization of Figure 6.8a shows the state of the transformation before the creation of the ClientOrder table in the target model. At that moment the selected table in the target model, which represents the ClientOrder table, is not connected with any element of the transformation. In Figure 6.8b the state after the creation of the ClientOrder table in the target model can be observed. This visualization shows which elements of the source model are related to the ClientOrder table in the target model. The bottom set of highlighted element of the source model represent the ClientOrder class, the two highlighted elements at the top represent the Client and Configuration classes. The Client and Configuration classes are used during the generation of the ClientOrder table since both classes are related with the ClientOrder class. This relation is transformed as a foreign key relation in the target model.
6.3 Conclusions

This case study shows how the visualizations of the transformation from UML model to relational database model can be used to analyze the transformation process. With the visualizations in TraceVis it is possible to observe the entire transformation, as presented in Figure 6.3. However it is also possible to limit the visualization to only those parts that the user is interested in, as is done for Figure 6.7. This shows that the visualization method is also suitable for large model transformations. When the number of trace links in the visualization increases, the hierarchical edge bundling of TraceVis still makes it possible to analyze the transformation process from an high level. However, with the visualization in TraceVis it is also possible to obtain a better insight into certain details of a large transformation by hiding the irrelevant parts of the source and target models and the transformation itself in the visualization.

6.3.1 Validation of requirements

In this section the requirements of Section 4.1 are verified on the visualization of the model transformation used in this case study. For each requirement it is described how the example of the case study verifies the requirement.

REQ1 The visualization tool should support visualization of the entire chain of trace links from a source model element to target model elements.

REQ2 The visualization tool should include the visualization of internal calls (Definition 4.1.1) between mapping operations.

On the left side of Figure 6.9 the CPU class of the source model is selected, on the right side the CPU table and the cpu_componentId column of the Configuration table in the target model are highlighted. The relations between the source model and transformation are highlighted, and the relations between the transformation and target model are highlighted. Moreover, the internal calls between the elements of the transformation that are used for the transformation of the CPU class are highlighted as well. This validates the requirements REQ1 and REQ2.

Figure 6.9: Visualization that verifies requirements REQ1 and REQ2.
REQ3 The visualization tool should automate the generation of the trace information.

The trace information that is required for the visualization that is shown in this case study is obtained in two steps. The first step is to execute the UML model to relational database model transformation on the UML model, depicted in Figure 6.1. During this first step the QVTo trace model, which contains the dynamic trace information, is obtained. Next, the transformation analysis is performed on the model transformation itself. This step generated the static trace information, and thus completes the generation of the trace information.

REQ4 The visualization tool should support visualization of the QVTo specification as described in Section 2.2, i.e. this excludes helper operations.

This requirement is verified with two examples. First an example of a part of the QVTo specification which is contained in the visualization tool.

Figure 6.10 shows a part of the transformation that is used for this case study. The figure shows the transformation from an attribute of a class into a column of a table. In Listing 6.1 the used mappings are presented. During the transformation of a class into a table the persistentClass2table invokes the class2columns mapping in order to transform the attributes of a class into columns. This class2columns mapping uses the init section to invoke the dataType2columns mapping, which invokes primitiveAttributes2columns mapping. From this primitiveAttributes2columns mapping the primitiveAttribute2column mapping is invoked that actually generates the column. In Figure 6.10 this trace of internal mapping calls is visualized with the relations between the two transformation columns.
Listing 6.1: Part of the transformation of Appendix D which is used in the visualization of Figure 6.10.

```plaintext
1 mapping UML::Class::persistentClass2table() : RDB::Table
2  when { self.isPersistent() }
3 {
4   name := self.name;
5   columns := self.map class2columns(self)->sortedBy(name);
6   primaryKey := self.map class2primaryKey();
7   foreignKeys := self.attributes.resolveIn(
8     UML::Property::relationshipAttribute2foreignKey,
9     RDB::constraints::ForeignKey)->asOrderedSet();
10 }
11
12 mapping UML::Class::class2columns(targetClass: UML::Class) : OrderedSet(RDB::TableColumn) {
13  init {
14    result := self.map dataType2columns(targetClass)->
15     union(self.map generalizations2columns(targetClass))->asOrderedSet()
16  }
17 }
18
19 mapping UML::DataType::dataType2columns(in targetType: UML::DataType) : OrderedSet(RDB::TableColumn) {
20  init {
21    result := self.attributes->map primitiveAttribute2column(targetType)->
22      union(self.map enumerations2columns(targetType))->
23      union(self.map relationshipAttributes2columns(targetType))->
24      union(self.map associationAttributes2columns(targetType))->asOrderedSet()
25  }
26 }
27
28 mapping UML::DataType::primitiveAttributes2columns(in targetType: UML::DataType) : OrderedSet(RDB::TableColumn) {
29  init {
30    result := self.attributes->map primitiveAttribute2column(targetType)->
31      asOrderedSet()
32  }
33 }
34
35 mapping UML::Property::primitiveAttribute2column(in targetType: UML::DataType) : RDB::TableColumn
36  when { self.isPrimitive() }
37  {
38    isPrimaryKey := self.isPrimaryKey();
39    name := self.name;
40    type := object RDB::datatypes::PrimitiveDataType { name := umlPrimitive2rdbPrimitive(self.type.name); };
41  }
```

50 Visualization of Model Transformations in QVTo
In Figure 6.11 the generated CPU table in the target model is shown. This figure shows that the datatypes of the are not related with any transformation element. The definition of the primitiveAttribute2column mapping, that creates the columns in the target model, is presented in Listing 6.2.

This listing also shows the definition of the umlPrimitive2rdbPrimitive helper, this helper transforms a datatype of the UML model into a datatype of the relational database model. However, the QVTo trace model only contains the invocations of mapping operations, and thus the invocations of helper operations are not traceable.

Listing 6.2: Mapping and helper that create the column with primitive datatypes in the target model.

```java
mapping UML::Property::primitiveAttribute2column (in targetType : UML::DataType) : RDB::TableColumn
    when { self.isPrimitive() }
    {
        isPrimaryKey := self.isPrimaryKey();
        name := self.name;
        type := object RDB::datatypes::PrimitiveDataType { name := umlPrimitive2rdbPrimitive(self.type.name); };
    }

helper umlPrimitive2rdbPrimitive (in name : String) : String {
    return if name = 'String' then 'varchar' else
    if name = 'Boolean' then 'int' else
    if name = 'Integer' then 'int' else
        name
    endif
    endif
    endif
}

Figure 6.11: The primitive datatypes in the target model are not related with the transformation elements that created them.
```
REQ5 The visualization tool should support visualization of a *chain of model transformations* (Definition 4.1.2).

The transformation which is presented in this case study consists of only one model transformation. Thus, this requirement cannot be verified with this case study. However, in Section 5.6 it is shown that it is possible to visualize multiple consecutive model transformations.

REQ6 The visualization tool should support scalability for large model transformations and chains of transformations.

In this case study we have seen that the hierarchical edge bundling of TraceVis is useful when visualizing large model transformations. The visualization of Figure 6.3 contains the complete transformation, which includes many trace links. However, with the bundling it is still possible to identify groups of trace links that are related to the same element in the source or target model. For example, it can clearly been seen that the trace links that are related with the target model are bundled in ten groups. These groups can be related to each of the tables in Figure 6.2. Moreover, in Figure 6.6 and Figure 6.7 it is shown that how it is possible to further investigate a certain part of the total visualization, and thus a part of the complete transformation.

REQ7 The TraceVis timeline-function should be used to allow step-by-step visualization of the model transformation.

The use of the TraceVis timeline-function is demonstrated in Figure 6.8. Every indent on the timeline at the bottom of Figure 6.8a and Figure 6.8b represents a single step in the transformation. Every relation that is added between the elements of the visualization is considered as a step in the transformation, and thus corresponds to a indent on the timeline. By moving the slider from the left side of the timeline to the right side it is possible to step-by-step visualize the model transformation, and thus requirement REQ7 is fulfilled.
7 Conclusion

In this thesis we have presented an approach for the visualization of QVTo model transformations. This approach is based on the visualization of ATL transformations which is presented in [27]. Using the characteristics of QVTo the approach of [27] is modified and extended in this project.

For the visualization of QVTo model transformations the visualization tool TraceVis is used. TraceVis is a tool for the visualization of hierarchical ordered data sets [5]. In order to visualize the QVTo model transformation, first, the source and target model of the transformation are represented as hierarchical ordered data in TraceVis. Next, static traceability is performed on the QVTo transformation by a transformation analyzer. This transformation analyzer obtains the structure of the model transformation and creates a hierarchical representation of the QVTo transformation.

The traces between source model and transformation, and transformation and target model are added to the visualization using the dynamic traceability information. This dynamic traceability information is obtained from the QVTo trace model which is created during the execution of the QVTo model transformation. Using the QVTo language specification [14] and the transformation structure, that is obtained by the transformation analyzer, also the internal calls between elements of the QVTo transformation are included in the visualization. The QVTo trace model contains the order in which the elements of the source model have been transformed. This order is used during the generation of the visualization in TraceVis, and makes it possible to step-by-step visualize the QVTo transformation using the timeline function of TraceVis.

A case study was performed, providing details on how this visualization approach achieves in representing the QVTo model transformation. It shows how a model transformation is visualized and how this visualization can be used to analyze different aspects of the transformation. For example, it is shown how the visualization can quickly give an overview of the complete transformation. Also, the visualization allows to zoom in on specific details of the transformation for further analysis of certain parts of the transformation.
7.1 Future work

Currently, there is no support for the visualization of invocation of the helper operations that are used in a QVTo transformation. This is caused by the QVTo trace model which is created during the execution of the QVTo transformation. This trace model, which is used to obtain the dynamic traceability information, does not consider the invocations of helper operations. The transformation analyzer, which obtains the static traceability information, can already collect the required information about the helper operations of a QVTo transformation. For future work it could be considered to include the invocations of the helper operations in the dynamic traceability information. This could be done by extending the standard QVTo trace model in order to include the invocations of helper operations. The Java implementation of this standard QVTo trace model is available since the CVS repository of QVTo is publicly available [29].

Another direction for future work focusses on the visualization tool that is used for this project, TraceVis. Currently, TraceVis is not modified for the visualization of model transformations, this can lead to a couple of problems. For example, TraceVis contains the feature to hide individual columns in the visualization. However, the visualization of a QVTo transformation uses two columns in the visualization to represent the complete transformation. With TraceVis it is possible to hide only one of these columns which results in an incorrect visualization of the transformation from source into target model, since only half of the transformation is visualized. A possible solution could be to make it possible to define strong couplings between columns. Moreover, when a column is hidden that is strongly coupled with another column, that other column should also be hidden. This would avoid invalid visualizations of a transformation, since it would be impossible to hide half of the transformation. It would also be useful to define a standard color scheme in TraceVis for nodes and relations when visualizing a model transformation. Currently, the coloring of nodes and relations in the visualization is defined manually.

Further automation of the creation of traceability information is also recommended based on the experience from this project. Currently, one has to execute the model transformation one wants to visualize and perform the transformation analyzer on this transformation before one can generate a TraceVis input file. It would be desirable to reduce the steps that have to be performed to obtain the TraceVis input file. This could be done by combining the transformation analyzer and the TraceVis input generator. For the transformation analyzer step a QVTo transformation is executed on the model transformation that needs to be visualized. It should be possible to execute this transformation inside the TraceVis input generator, and thus remove the extra step of executing the transformation analysis manually.
Bibliography


A Transformation example: MMA to MMB

Listing A.1: Transformation example: MMA to MMB.

```java
modeltype MMA uses "http://mma/1.0";
modeltype MMB uses "http://mmb/1.0";

transformation MMATransformation (in Source: MMA, out Target: MMB);

main() {
    Source.rootObjects()[MMA::Automaton] -> map toAutomaton();
}

mapping MMA::Automaton::toAutomaton() : MMB::Automaton
    { self.Name.startsWith("A"); }
    { Name := self.Name;
      locations := self.states -> map toLocation();
      transitions := self.states.outgoingEdges -> map toTransition();
      log("Name of the Automaton: " + Name);
    }

mapping Edge::toTransition() : Transition {
    Event := self.Action;
    sourceLocation := self.container().oclAsType(State).resolveOne(Location);
    targetLocation := self.targetState.resolveOne(Location);
}

mapping State::toLocation() : Location
    disjuncts Rectangle::fromRectangle, Ellipse::fromEllipse1, Ellipse::fromEllipse2 {
    log("This log is never reached, disjunct mappings don’t use their body.")
}

mapping Rectangle::fromRectangle() : Location
    inherits State::Updates {
    Shape := "Rectangle";
    Dimension := self.Sides;
}

mapping Ellipse::fromEllipse1() : Location
    inherits State::Updates
    when { 1 == 2 }
    { /* Since the condition doesn’t hold (1 is obviously not equal to 2),
      this mapping won’t be executed for a Ellipse object. */
    }

mapping Ellipse::fromEllipse2() : Location
```

57 Visualization of Model Transformations in QVTo
inherits State::Updates {
    Shape := "Ellipse";
    Dimension := self.Radius;
}

class mapping State::Updates() : Location {
    Name := self.Name;
    InitialState := self.IsInitial;
}
Listing B.1: Transformation analyzer

```java
modeltype qvt "strict" uses "http://www.eclipse.org/qvt/1.0.0/Operational/Expressions";
modeltype TA uses "http://trafoanalyze2/1.0";
modeltype ecore uses "http://www.eclipse.org/emf/2002/Ecore";
modeltype eimpocl uses "http://www.eclipse.org/qvt/1.0/ImperativeOCL";
modeltype ecoreocl uses "http://www.eclipse.org/ocl/1.1.0/Ecore";

transformation TrafoAnalyze (in Source:qvt, out Target:TA) {
    main () {
        Source.rootObjects [qvto::OperationalTransformation] -> xmap analyze () ;
    }
}

mapping qvto::OperationalTransformation::analyze () : Transformation {
    result.Name := self.name ;
    result.models := self.modelParameter -> map toModel () ;
    result.operations := self.eOperations [qvto::ImperativeOperation] ->
        toOperation () ;
}

mapping qvto::ModelParameter::toModel () : Model {
    result.Name := self.name ;
    result.Type := self.eType.name ;
    result.Direction := getDirection (self.kind) ;
}

mapping qvto::ImperativeOperation::toOperation () : TA::Operation {
    disjuncts qvto::EntryOperation::fromEntry,
        qvto::MappingOperation::fromMapping
}

mapping qvto::EntryOperation::fromEntry () : TA::EntryOperation {
    var res:OrderedSet (OrderedSet (String)) := self.getCalledMapping () ;
    result.bodyCallingOperations := res -> at (2) ;
}

mapping qvto::MappingOperation::fromMapping () : TA::MappingOperation {
    result.Name := self.name ;
    result.input := self.context ->
        map toMappingElement () -> asSequence () -> first () ;
    result.parameters := self.eParameters -> map toParameter () ;
    result.output := self._result -> map toMappingElement () ;
    var res:OrderedSet (OrderedSet (String)) := self.getCalledMapping () ;
}
```

Visualization of Model Transformations in QVTo
result.initCallingOperations := res->at(1);
result.bodyCallingOperations := res->at(2);
result.disjuncted := self.disjunct.name;
result.merged := self.merged.name;
result.inherited := self.inherited.name;
self.getCalledMapping();

helper qvto::ImperativeOperation::getCalledMapping() : OrderedSet(OrderedSet(String)) { 
  var res : OrderedSet(OrderedSet(String));
  var initCalls : OrderedSet(String);
  var bodyCalls : OrderedSet(String);
  var res2 : OrderedSet(qvto::MappingCallExp);
  var res3 : OrderedSet(qvto::ImperativeCallExp);
  self.allSubobjectsOfType(qvto::MappingCallExp) [qvto::MappingCallExp] ->
  forEach(i) {
    i.referredOperation[qvto::MappingOperation] ->forEach(j) {
      var temp:Element := i;
      var isInBody:Boolean := false;
      while(temp.container() != null) {
        temp := temp.container();
        if(temp.oclIsTypeOf(ConstructorBody) or temp.oclIsTypeOf(EntryOperation)) then {
          isInBody := true;
        }
      }
      if(isInBody) then {
        bodyCalls := bodyCalls->append(j.name);
      }
      else {
        initCalls := initCalls->append(j.name);
      }
    }
  }
  self.allSubobjectsOfType(qvto::ImperativeCallExp) [qvto::ImperativeCallExp] ->
  forEach(i) {
    i.referredOperation[qvto::Helper] ->forEach(j) {
      var temp:Element := i;
      var isInBody:Boolean := false;
      while(temp.container() != null) {
        temp := temp.container();
        if(temp.oclIsTypeOf(ConstructorBody) or temp.oclIsTypeOf(EntryOperation)) then {
          isInBody := true;
        }
      }
      if(isInBody) then {
        }
    }
  }
if (isInBody) then {
    bodyCalls := bodyCalls ->append(j.name);
} else {
    initCalls := initCalls ->append(j.name);
} endif;
};
res := res ->append(initCalls);
res := res ->append(bodyCalls);

return res;
}

mapping qvto::VarParameter::toMappingElement() : MappingElement {
    result.Name := self.name;
    result.Direction := getDirection(self.kind);
    result.Type := self.eType.name;
}

mapping ecore::EParameter::toParameter() : MappingElement {
    result.Name := self.name;
    result.Type := self.eType.name;
}

helper getDirection(dir:qvto::DirectionKind) : Direction {
    var res:Direction;

    if (dir.repr().equalsIgnoreCase("in")) then {
        res := Direction::IN;
    }
    else
        if (dir.repr().equalsIgnoreCase("inout")) then {
            res := Direction::INOUT;
        }
        else
            if (dir.repr().equalsIgnoreCase("out")) then {
                res := Direction::OUT;
            }
        endif
    endif
}

return res;
}
C Trace tree with only static trace information

Figure C.1: Trace tree for the transformation in Appendix A with only static trace information, with the NodeType values of Figure 5.4
Model transformation: UML to relational database

Listing D.1: Transformation: UML to relational database.

```java
/*
 * The SimpleUML to RDB Sample demonstrates how to use QVT transformations for
 * transforming platform independent model to platform specific model.
 * It also demonstrates the following basic features of QVT language:
 * helper queries, mapping guards, and resolution operations.
 * Sample model pim.simpleuml is included to be used as an input for the
 * transformation.
 */

/*
 * Two modeltypes are declared. The http NS URIs correspond to those used to
 * register the Ecore models in the environment. Alternatively, a workspace metamodel may
 * be used in conjunction with mappings defined in the project properties.
 */
modeltype UML uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/simpleuml';
modeltype RDB uses 'http://www.eclipse.org/qvt/1.0.0/Operational/examples/rdb';

/*
 * The transformation signature declares that a UML modeltype is required as
 * input, while an RDB modeltype is produced as an output. The UML modeltype is referenced as
 * `uml` throughout the transformation definition, while no name is needed for the output RDB
 * modeltype. Note that OCL type and namespace notation are used in operational QVT (: and ::
 * respectively).
 */
transformation Simpleuml_To_Rdb(in uml : UML, out RDB);

/*
 * The main entry point of the transformation. The `uml` reference to the
 * input UML modeltype instance is used to collect all rootObjects() of type Model. The
 * rootObjects() operation is available on all QVT Model objects (extents) and returns those objects
 * found at the
*/
```
* root of the input model. The [UML::Model] statement following the call to
  rootObjects() is shorthand notation for the imperative select (xselect) construct where
  the condition is
* a type expression, which effectively performs a oclIsKindOf(UML::Model)
  with type recasting
* as a sequence.
* The invocation of the model2RDBModel() mapping is done using an \( \rightarrow \) operator
  which is a
* shorthand notation for the imperative collect (xcollect) construct.
  Alternatively, it could
* be written as uml.rootObjects()[UML::Model]\( \rightarrow \)xcollect(a | a.map
  model2RDBModel());
*/
main() {
  uml.rootObjects()[UML::Model]\( \rightarrow \)map model2RDBModel();
}
/*
* This mapping returns an RDB::Model instance from the UML::Model passed from
  main(). The name
* attributes map directly using the OCL assignment operator. The RDB Model
  has a collection
* of schemas populated by the package2schemas() mapping. The Sequence of RDB
  ::Schema objects
* that is returned by the mapping is converted into the required OrderedSet
  using the OCL operation
* asOrderedSet().
* This mapping has no init or end section, leaving the body as an implicit
  population section.
*/
mapping UML::Model::model2RDBModel() : RDB::Model {
  name := self.name;
  schemas := self.map package2schemas()\( \rightarrow \)asOrderedSet();
}
/*
* This mapping recursively invokes the package2schema() mapping to produce a
  Sequence of
* RDB Schema objects from a UML Package and its subpackages. Note the use of
  OCL union()
* and flatten() operations to produce a single flattened Sequence of Schema
  objects.
* There is no population section in this mapping, but an init section that
  assigns the
* Sequence of returned objects to the result. Alternatively, the statement
  below could
* have been used in the schemas assignment of the mapping above.
*/
mapping UML::Package::package2schemas() : Sequence(RDB::Schema) {
  init {
    result := self.map package2schema()\( \rightarrow \)asSequence()\( \rightarrow \)
      union(self.getSubpackages()\( \rightarrow \)map package2schemas()\( \rightarrow \)flatten());
  }
}
/* This mapping creates an RDB Schema object from a UML Package. It includes a when clause to verify the passed Package contains persistent classes. Look below to see the logic employed within the hasPersistentClasses() query. */

class package2schema

when { self.hasPersistentClasses() } {
  name := self.name;
  elements := self.ownedElements[UML::Class]->map persistentClass2table() asOrderedSet();
}

/*
This mapping produces an RDB Table object from a UML persistent Class object. The when clause uses the isPersistent() query below to see if the Class has a 'persistent' string as one of its stereotype strings.

Again, the name attributes map directly. The Class is mapped to a set of TableColumn objects using the class2columns() mapping, the results of which are sorted by name using the OCL sortedBy() operation. The primaryKey is set using the class2primaryKey() mapping, while foreignKeys are set using a resolveIn function. This will allow us to resolve RDB ForeignKey objects created using the relationshipAttribute2foreignKey() mapping for each of the Class attributes.
*/

class persistentClass2table

when { self.isPersistent() } {
  name := self.name;
  columns := self.map class2columns(self)->sortedBy(name);
  primaryKey := self.map class2primaryKey();
  foreignKeys := self.attributes.resolveIn(
    UML::Property::relationshipAttribute2foreignKey,
    RDB::constraints::ForeignKey)->asOrderedSet();
}

/* A PrimaryKey object is created from a Class by prefixing the name with 'PK' and resolving one (the first) Table created from the Class in order to obtain its primary key columns using a query. */
/*
 * mapping UML::Class::class2primaryKey() : RDB::constraints::PrimaryKey {
 *   name := 'PK' + self.name;
 *   includedColumns := self.resolveOneIn(UML::Class::persistentClass2table, RDB::Table).getPrimaryKeyColumns()
 * }
 */

/*
 * This mapping will create an OrderedSet of RDB TableColumn objects from a UML Class object.
 * Similar to package2schemas(), this mapping has no population section, but just an init
 * that assigns the result based on the union of type mappings from the Class and its
generalizations (Class extends DataType).
 * Note that this mapping is defined for type UML::Class and also takes a UML::Class named
targetClass as a parameter. This pattern is used in several places within this
transformation definition to account for how generalization in the UML model is
mapped to columns in the RDB model. As properties and inherited properties are
flattened into columns, the combined use of 'self' and 'target' parameter represent a
way to allow multiple copies of columns for a given property source, as subsequent
mapping invocations retrieve the same resulting columns from the trace model.
*/

mapping UML::Class::class2columns(targetClass : UML::Class) : OrderedSet(RDB:: TableColumn) {
   init {
      result := self.map dataType2columns(targetClass) ->
               union(self.map generalizations2columns(targetClass))->asOrderedSet()
   }
}

/*
 * For the passed DataType, an OrderedSet of TableColumn objects is created. Again, the result
 * is assigned within the init block to the union of several attribute to column mappings.
 */

mapping UML::DataType::dataType2columns(in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
   init {
      result := self.map primitiveAttributes2columns(targetType) ->
               union(self.map enumerationAttributes2columns(targetType)) ->
               union(self.map relationshipAttributes2columns(targetType)) ->
               union(self.map associationAttributes2columns(targetType))->asOrderedSet()
   }
}
This mapping creates an OrderedSet of TableColumn objects from a DataType object.
The mapping declares three input parameters, including a prefix string and primary key.
The `init` section uses the result keyword with mapping invocation, as we've seen before.
What's new in this mapping is the use of an object definition within a `collect` operation.
Here, the OrderedSet of TableColumn objects returned from the `dataType2columns()` mapping is filtered to select only those marked as primary keys, which in turn are used within the context of the `collect` where those matching the TableColumn created using object are returned.

```
mapping UML::DataType::dataType2primaryKeyColumns (in prefix : String, in leaveIsPrimaryKey : Boolean, in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.map dataType2columns(self) -> select(isPrimaryKey) ->
                      collect(c | object RDB::TableColumn {
                         name := prefix + '_' + c.name;
                         domain := c.domain;
                         type := object RDB::datatypes::PrimitiveDataType {
                            name := c.type.name;
                        };
                         isPrimaryKey := leaveIsPrimaryKey
                      }) -> asOrderedSet();
    }
}
```

This mapping returns an OrderedSet of TableColumn objects by invoking the `primitiveAttribute2column()` mapping for each attribute of the DataType.

```
mapping UML::DataType::primitiveAttributes2columns (in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.attributes -> map primitiveAttribute2column(targetType) ->
                      asOrderedSet();
    }
}
```

This mapping creates a TableColumn from a Property when the `isPrimitive()` query returns true. The isPrimaryKey and name mappings are straightforward, while the type reference is created using the object keyword to create a new `PrimitiveDataType` with name initialized to the result of the query.

```
mapping UML::Property::primitiveAttribute2column (in targetType : UML::DataType) : RDB::TableColumn
    when { self.isPrimitive() }
```
isPrimaryKey := self.isPrimaryKey();
name := self.name;
type := object RDB::datatypes::PrimitiveDataType { name := umlPrimitive2rdbPrimitive(self.type.name); }

/* This mapping returns an OrderedSet of TableColumn objects by invoking the
 * enumerationAttribute2column() mapping for each attribute of the DataType.
 */

mapping UML::DataType::enumerationAttributes2columns(in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
  init {
    result := self.attributes->map enumerationAttribute2column(targetType)->asOrderedSet()
  }
}

/* This mapping creates a TableColumn from a Property when the isEnumeration() query
 * returns true. The isPrimaryKey and name mappings are straightforward, while
 * the type reference is created using the object keyword to create a new
 * PrimitiveDataType with name initialized to 'int'.
 */

mapping UML::Property::enumerationAttribute2column(in targetType : UML::DataType) : RDB::TableColumn
when { self.isEnumeration() }
{
  isPrimaryKey := self.isPrimaryKey();
  name := self.name;
  type := object RDB::datatypes::PrimitiveDataType { name := 'int'; }
}

/* This mapping creates an OrderedSet of TableColumn objects from relationship
 * attributes. The check for if the DataType is a relationship is performed
 * in the when clause of the invoked relationshipAttribute2foreignKey mapping.
 */

mapping UML::DataType::relationshipAttributes2columns(in targetType : UML::DataType) : OrderedSet(RDB::TableColumn) {
  init {
    result := self.attributes->map relationshipAttribute2foreignKey(targetType )-->
    collect(includedColumns)->asOrderedSet();
  }
}

/* This mapping creates a ForeignKey object from a DataType that returns true
 * from
 * the isRelationship() query in the when clause. The name is prefixed with 'FK'.
 * The includedColumns collection is populated using the
 * dataType2primaryKeyColumns()
 * mapping on the Property type reference cast to DataType.

68 Visualization of Model Transformations in QVTo
The referredUC reference uses resolveoneIn, but with the late modifier. This causes the resolution to happen at the end of the transformation, thereby avoiding a second pass to resolve objects that may not have been created during execution at this point.

```java
mapping UML::Property::relationshipAttribute2foreignKey (in targetType: UML::DataType) : RDB::constraints::ForeignKey
when { self.isRelationship() }
{
    name := 'FK' + self.name;
    includedColumns := self.type.asDataType().map dataType2primaryKeyColumns;
    referredUC := self.type.late resolveoneIn(UML::Class::class2primaryKey, RDB::constraints::PrimaryKey);
}

/*
 * This mapping produces an OrderedSet of TableColumn objects from dataType attributes that return true from the isAssociation() query. The TableColumn objects are created using a call to the dataType2columns() mapping.
 */

mapping UML::DataType::associationAttributes2columns (targetType: UML::DataType) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.attributes[isAssociation()]--
            collect(type.asDataType()-->map dataType2primaryColumns(targetType))-->
            asOrderedSet();
    }
}

/*
 * This mapping returns an OrderedSet of TableColumn objects from a Class using the generalizations of the class and the class2columns() mapping.
 */

mapping UML::Class::generalizations2columns (targetClass: UML::Class) : OrderedSet(RDB::TableColumn) {
    init {
        result := self.generalizations.general--map class2columns(targetClass)-->
            flatten()--asOrderedSet();
    }
}

/*
 * This query returns an OrderedSet of Package objects from a Package’s ownedElements collection that are of type UML::Package using shorthand xselect notation.
 */

query UML::Package::getSubpackages() : OrderedSet(UML::Package) {
    return self.ownedElements[UML::Package]--asOrderedSet();
}

69 Visualization of Model Transformations in QVTo
/*
 * This query performs a type cast from a UML Type to a UML DataType.
 */

query UML::Type::asDataType() : UML::DataType {
  return self.oclAsType(UML::DataType)
}

/*
 * This query returns true if the list of string stereotypes includes one equal to 'primaryKey'.
 */

query UML::Property::isPrimaryKey() : Boolean {
  return self.stereotype->includes('primaryKey')
}

/*
 * This query returns true if the list of string stereotypes includes one equal to 'identifying'.
 */

query UML::Property::isIdentifying() : Boolean {
  return self.stereotype->includes('identifying')
}

/*
 * This query returns true if the type attribute of the Property conforms to the UML PrimitiveType.
 */

query UML::Property::isPrimitive() : Boolean {
  return self.type.oclIsKindOf(UML::PrimitiveType)
}

/*
 * This query returns true if the type attribute of the Property conforms to the UML Enumeration.
 */

query UML::Property::isEnumeration() : Boolean {
  return self.type.oclIsKindOf(UML::Enumeration)
}

/*
 * This query returns true if the type attribute of the Property conforms to the UML DataType and returns true from the isPersistent() query.
 */

query UML::Property::isRelationship() : Boolean {
  return self.type.oclIsKindOf(UML::DataType) and self.type.isPersistent()
}

/*
 * This query returns true if the type attribute of the Property conforms to the UML DataType and returns false from the isPersistent() query.
 */

query UML::Property::isAssociation() : Boolean {
  return self.type.oclIsKindOf(UML::DataType) and not self.type.isPersistent()
}

/*
 * This query returns an OrderedSet of TableColumn objects from those columns where isPrimaryKey returns true.
 */
```
query RDB::Table::getPrimaryKeyColumns() : OrderedSet(RDB::TableColumn) {
    return self.columns->select(isPrimaryKey)
}

/*
 * This query returns true if the list of string stereotypes includes one
 * equal to 'persistent'.
 */
query UML::ModelElement::isPersistent() : Boolean {
    return self.stereotype->includes('persistent')
}

/*
 * This query examines the contents of a Package to determine if there exists
 * at least one Class that returns true for the isPersistent() query.
 */
query UML::Package::hasPersistentClasses() : Boolean {
    return self.ownedElements->exists(
        let c : UML::Class = oclAsType(UML::Class) in
        c.oclIsUndefined() implies c.isPersistent())
}

/*
 * This helper returns the RDB primitive string corresponding to the passed
 * UML primitive string. This helper produces no side effects and could be
 * written as a query, alternatively.
 */
helper umlPrimitive2rdbPrimitive(in name : String) : String {
    return if name = 'String' then 'varchar' else
       if name = 'Boolean' then 'int' else
          if name = 'Integer' then 'int' else
             name
          endif
       endif
    endif
}
```

71 Visualization of Model Transformations in QVTo