

An Interactive Modeling Editor for QVT-Relations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christian DeTamble

Matrikelnummer 00827283

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Wien, 27. März 2018



Christian DeTamble

Manuel Wimmer

An Interactive Modeling Editor for QVT-Relations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christian DeTamble

Registration Number 00827283

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Manuel Wimmer

Vienna, 27th March, 2018



Christian DeTamble

Manuel Wimmer

Erklärung zur Verfassung der Arbeit

Christian DeTamble
Roitherstraße 17, 4802 Ebensee

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. März 2018



Christian DeTamble

Acknowledgements

I would first like to thank my advisor Assistant Prof. Manuel Wimmer, who gave me the opportunity to write this thesis at the Business Informatics Group at the Vienna University of Technology. He provided me with valuable input and directions to steer my work to.

I am also grateful to the experts at LieberLieber Software GmbH, who were involved in the evaluation part of this thesis. Specifically, my sincere thanks go to Daniel Siegl, Peter Lieber, Dr. Konrad Wieland, Philipp Kalenda and Hannes Dangel.

Finally, I am thankful to my parents, my sisters and Erwin, who made my Computer Science studies possible and unfailingly supported me throughout the past years.

Abstract

In traditional software engineering, artifacts are manually developed, based on a system specification in terms of *documenting* models. Models are abstractions of real-world concepts and environments. Model-driven Engineering (MDE) is a paradigm in which these models are considered the driving software artifacts, serving also purposes of code generation besides documenting the underlying system. In this context, the Object Management Group (OMG) suggests in their Model-driven Architecture (MDA) to standardize the models for them to have a maximized re-usability. Consequently, software engineers benefit from a positively influenced *productivity*. The topic of *model transformation* plays a central role in this context of MDE, as the involved models are specified on different levels of abstraction.

Since its publication in 2008, Query/View/Transformation-Relations (QVTr) claims to be the standard model transformation language (MTL) for the declarative specification of model transformations, and has been used as an enabling formalism. In consideration of productivity being a central goal in MDE, it is vital for tools and editors to maximize the usability of their implementing MTL.

However, taking into account the current state of the art in tools for QVTr, several shortcomings are revealed. First, the availability of matured tools is sparse, and furthermore, they have been developed with the goal to enable the underlying technology. Their design is not user-centered and, in particular, they lack from a poor level of automation and interactivity. In addition, we identify a lack of support for *short* feedback cycles, which significantly influences the usability of both the editor and implementing MTL. Finally, we consider the neglect of QVTr's concrete, graphical syntax in state of the art editors as unused potential for an increase in readability and traceability.

In the context of this thesis, we shed light on the impact of an increase in interactivity, automation, readability, traceability, the usage of QVTr's graphical syntax, and of short feedback cycles on the usability of QVTr. For this purpose, we propose a theoretical concept comprising techniques to push the modeling process towards a user-centered approach. The underlying key principles of our concept comprise the so called *outward modeling* style, a suggestion-driven process, interactive graphical model visualizations and the enforcement of conventions. To show the feasibility of our approach, we conduct user experiments in an industrial context at the LieberLieber Software GmbH company in Vienna, using a prototypical implementation.

Kurzfassung

Die traditionelle Software-Entwicklung (SE) sieht die manuelle Entwicklung von Artefakten vor, ausgehend von einer durch Modelle *dokumentierten* Spezifikation. Die Modelle dienen primär der Dokumentation und sind Abstraktionen der realen Welt. Die Modellgetriebene SE (MDE) ist ein Paradigma, in welchem diese Modelle als die primären Artefakte angesehen werden, und übernehmen neben der Dokumentation auch die Aufgabe der Code-Generierung. In diesem Kontext schlägt der Model-driven Architecture (MDA) Standard vor, diese Modelle zu standardisieren, um deren Wiederverwendbarkeit zu maximieren. In weitere Folge profitieren Software-Entwickler von einer maximierten *Produktivität*. Das Thema der *Modell-Transformation* spielt eine zentrale Rolle in der MDE, da involvierte Modelle auf separaten Abstraktionsebenen definiert sind. Seit der Publikation 2008 beansprucht die Modell-Transformationssprache (MTL) Query/View/Transformation Relations (QVTr) der Standard für die deklarative Spezifikation von Transformationen zu sein, und findet seitdem Verwendung in der Forschung. Da Produktivität eines der Ziele in der MDE ist, ist es für unterstützende Tools unerlässlich, die Benutzerfreundlichkeit der zugrundeliegenden MTL zu maximieren. In Anbetracht von derzeit verfügbaren Tools für QVTr offenbaren sich jedoch diverse Mängel. Die Verfügbarkeit von ausgereiften Tools ist spärlich, und haben primär das Ziel, die zugrundeliegende Technologie nutzbar zu machen. Das Design ist nicht Benutzer-zentriert und insbesondere herrscht ein Mangel an Automatisierung und Interaktivität. Zusätzlich fehlt die Unterstützung für *möglichst kurze* Feedback-Zyklen, was sich im Besonderen auf die Benutzerfreundlichkeit auswirkt. Zuletzt betrachten wir das Fehlen der konkreten, grafischen Notation von QVTr in den Tools als ungenütztes Potential für eine erhöhte Les- und Rückverfolgbarkeit. Im Kontext dieser Diplomarbeit wird ein Blick auf den Einfluss einer erhöhten Interaktivität, Automatisierung, Les- und Rückverfolgbarkeit, die Verwendung der grafischen Notation von QVTr, und von möglichst kurzen Feedback-Zyklen auf die Benutzerfreundlichkeit von QVTr geworfen. Zu diesem Zweck wird ein theoretisches Konzept vorgeschlagen, das Techniken umfasst, um den Modellierungsprozess Benutzer-zentriert zu implementieren. Die zugrundeliegenden Prinzipien umfassen den sogenannten *Outward* Modellierungsstil, einen Vorschlag-getriebenen Prozess, interaktive Visualisierungen für Modelle, und die Durchsetzung von Konventionen. Um die Realisierbarkeit unseres Ansatzes zu demonstrieren wurden mithilfe einer prototypischen Implementierung Experimente in einem industriellen Kontext bei der LieberLieber Software GmbH in Wien durchgeführt.

Contents

| | |
|--|-----------|
| Abstract | ix |
| Kurzfassung | xi |
| 1 Introduction | 3 |
| 1.1 Problem Statement | 4 |
| 1.2 Aim of the Work | 5 |
| 1.3 Contribution | 5 |
| 1.4 Methodological Approach | 5 |
| 1.5 Structure of the Work | 6 |
| 2 State of the Art | 7 |
| 2.1 Model Transformations | 8 |
| 2.2 Tool Support for QVTr | 8 |
| 2.3 Complementary Tool Support | 11 |
| 3 The QVTr Modeling Process | 15 |
| 3.1 Types of Model Transformations | 15 |
| 3.2 Model Transformation Pipeline | 17 |
| 3.3 Modeling Tasks | 18 |
| 3.4 Complementary Tasks | 27 |
| 3.5 Modeling Challenges | 30 |
| 3.6 Modeling Patterns | 32 |
| 3.7 Textual against Graphical Modeling | 36 |
| 4 A Concept for Productive Modeling with QVTr | 39 |
| 4.1 Design Decisions | 39 |
| 4.2 Preliminary Notation | 42 |
| 4.3 Automation | 44 |
| 4.4 Preventive Interactivity | 52 |
| 4.5 Corrective Interactivity | 61 |
| 4.6 Readability | 67 |
| 4.7 Traceability | 73 |

| | | |
|----------|---------------------------------------|------------|
| 5 | Evaluation | 77 |
| 5.1 | Prototypical Implementation | 77 |
| 5.2 | Experiment Plan | 78 |
| 5.3 | Results | 82 |
| 5.4 | Observations | 93 |
| 5.5 | Interpretation | 94 |
| 5.6 | Conclusion | 98 |
| 6 | Conclusion | 101 |
| 6.1 | Summary | 101 |
| 6.2 | Future Work | 104 |
| | List of Figures | 106 |
| | List of Tables | 108 |
| | Bibliography | 109 |
| A | Diagrams of Metamodels | 115 |
| B | User Experiment Exercise | 117 |
| C | User Experiment Questionnaire | 119 |

Introduction

In traditional software engineering, artifacts are manually developed, based on domain models that describe the software system to be built. The formal specification of these conceptual models is done using dedicated modeling languages (*e.g.* Entity-Relationship, UML, SysML), and serves for documentation and communication purposes.

Model-Driven Engineering (MDE) is a paradigm with the goal of standardizing these domain models for them to have a maximized re-usability, so that software engineers benefit from a positively influenced *productivity*. In order to standardize the reusability of software models, the OMG defines the separation into platform-independent (PIMs) from platform-specific models (PSMs) in their Model-Driven Architecture (MDA) [39]. According to this definition, a model for a relational database management system is a PSM, whereas a model for an UML class diagram is a PIM. Since PIMs and PSMs are defined on different abstraction levels there clearly exists the need for a mechanism to transform models from one abstraction level to another.

According to Kleppe et al. [23], a model transformation is defined as follows.

*“A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.”*

Figure 1.1 exemplary illustrates such model transformations. The UML diagram (PIM) on the left is used to generate a relational database schema (PSM) in a model-to-model transformation. In a subsequent step, the schema is transformed into SQL code in a model-to-code transformation.

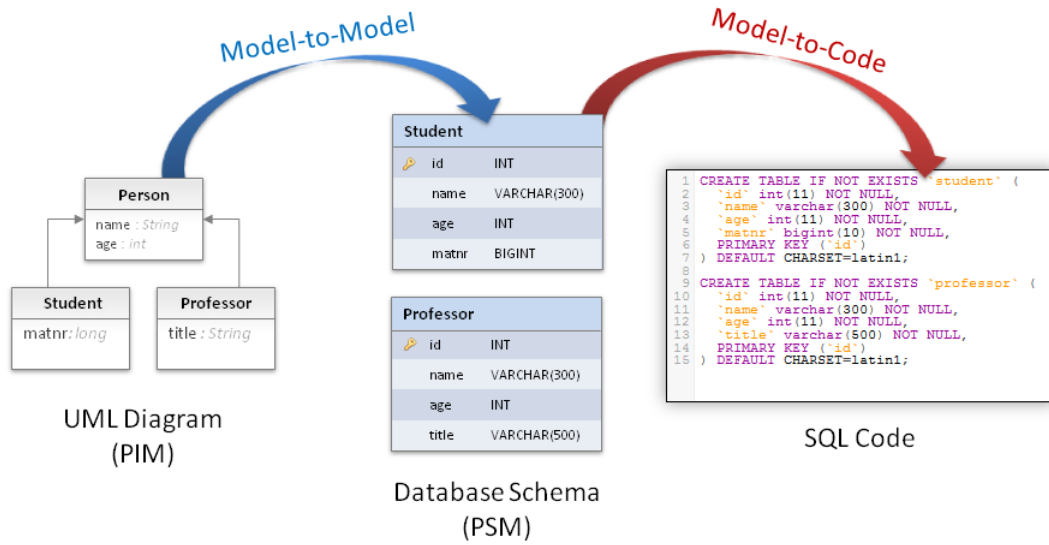


Figure 1.1: PIM to PSM model transformation

1.1 Problem Statement

Since its publication in 2008, Query/View/Transformation-Relations (QVTr) claims to be the standard [41] model transformation language (MTL) for declarative specification of model transformations. Since then, it has been used as an enabling formalism in academic research [7, 12, 28, 36, 45]. In consideration of productivity being a central goal in MDE, it is vital for tools and editors to minimize the required effort that a user has during the modeling process, and to maximize the usability of the used MTL. We observe the following shortcomings in state of the art tools for QVTr.

- The availability of *matured* tools is sparse. Specifically, we identify the tools Echo [33], medini QVT [34] and QVT Declarative [13] as such, opposing to rather prototypical implementations such as ModelMorf [2] and plug-ins for Enterprise Architect (EA) and the Visual Modeling and Transformation System (VMTS) [28].
- The focus of these 3 matured tools lies on enabling technology, but not primarily on usability. In this context, Strüber et al. [49] claim that a strong focus on usability for both modeling languages and implementing tools are a prerequisite to a broader adoption of MDE. What these 3 tools lack in particular is a high level of interactivity and automation. The modeling process in QVTr contains recurring, tedious tasks that potentially affect the productivity of users. To the best of our knowledge, a prototypical, usability-focused QVTr plug-in for Enterprise Architect is the only attempt yet to tackle this issue.
- We observe a lack of support in these editors for *short* feedback cycles.

- Finally, the specification of QVTr [41] also defines a concrete, graphical notation, which is unsupported in all of the matured tools. We consider this as unused potential for an increase in readability and traceability.

Based on these observations, we propose the following hypotheses.

- QVTr's usability benefits from an increase in interactivity and automation, since it decreases the modeling effort.
- QVTr's usability benefits from short feedback cycles.
- QVTr's readability and traceability benefits from using its graphical notation.

1.2 Aim of the Work

The aim of this thesis is to research the impact of an increase in interactivity, automation, readability, traceability, the usage of QVTr's graphical syntax, and of short feedback cycles on the usability of QVTr. For this purpose, we conduct user experiments in an industrial context at the LieberLieber Software GmbH company.

1.3 Contribution

The contribution of this thesis are answers to the following research questions.

- What are possibilities for interactive, automated, readable and traceable modeling?
- What are possibilities to accelerate feedback cycles?
- Which role does QVTr's graphical concrete syntax play concerning readability and traceability?
- How does our concept perform in practice, compared to a state of the art tool?
- In which context does our concept generate the most value?

1.4 Methodological Approach

The methodological approach to reach the listed contributions comprises the following steps.

- In the first part, state of the art tools for QVTr are discussed with respect to interactivity, automation, readability and traceability.

- The second part focuses on analyzing the modeling process when using QVTr to specify model transformations. We do this by means of a usability inspection method called *cognitive walkthrough* [17], and apply it on modeling the SimpleUML-ToSimpleRDBMS example transformation from the QVT standard specification.
- In the third part, we propose a theoretical concept that comprises techniques for implementing an increase in interactivity, automation, readability and traceability into the modeling process of QVTr. The concept is based on the insights gained in the previously conducted analyses of the modeling process and state of the art tools.
- In the next step, the previously defined concept is prototypically implemented using the JavaFX [43] programming language, since it is an established language for developing desktop applications with highly interactive user interfaces. For convenience reasons we will refer to this prototypical implementation using *QViT* as its name.
- Finally, we evaluate the impact of our concept in practice by conducting user experiments in an industrial context at the LieberLieber Software GmbH in Vienna.

1.5 Structure of the Work

The work done in the context of this thesis is reported in the following structure.

- Chapter 2 presents the state of the art in the topic of model transformations. This includes a discussion of state of the art tooling support for QVTr, and the research done in related work.
- In Chapter 3 we report on our analysis of the theoretical modeling process in QVTr. We begin with a clear definition of the type of model transformations that are of interest in the scope of this thesis. Then, we define a list of modeling tasks, challenges and patterns that occur throughout the process of modeling. A comparison of textual against graphical modeling concludes this chapter.
- In Chapter 4 we present our theoretical concept for productive modeling with QVTr, comprising a combination of dedicated techniques to increase the level automation, interactivity, readability and traceability.
- Chapter 5 reports the results of our conducted user experiments including an interpretation of the collected data. We also briefly describe the prototypical implementation our proposed concept, which has been used for the experiments.
- Chapter 6 summarizes the work done in this thesis, including the results and insights gained, and concludes this thesis with an outlook on future work.

State of the Art

Usability, the problem of attractive visual modeling and the user-experience of tools for MDE have recently come to the attention of academic research [3, 35, 49, 51]. What authors conclude in their work is that the adoption of MDE hinges on the usability of modeling languages and implementing tools, and that the usability is not inherent to the product itself, but is instead the result of the interaction with it.

In this context, Mens et al. [35] state that for editors to be useful and serve their practical purpose, they have to achieve a balance between *verbosity* and *conciseness*. What they mean by this statement is what code editors for major general-purpose programming languages such as Java or C# achieve with so called *code snippets*. On the one hand, the editor should expose as few functionality to the user at the same time as possible, in order to not be overwhelming or cluttered. On the other hand, this way, the modeling of complex transformations becomes tedious. A comparably low-verbose solution are code snippets, that the user inserts via dedicated context menus or ideally by keyboard shortcuts. Furthermore, the authors enumerate certain features that they consider a transformation tool to have to be usable. Among other points, this list includes CRUD¹ functionality, a suggestions mechanism when to apply a specific transformation, as well as support for modularization of transformations, such that their re-usability is maximized. Furthermore, they demand the need for automatic checking of syntactical correctness and completeness. The first case refers to the well-formedness of a transformation, whereas the latter property considers whether all elements in a source model have been related to a respective element in the target model. Next, the authors state that an editor should constantly analyze a transformation in order to maintain traceability links between the source and target side. Further features demanded by the authors are extensibility, interoperability, acceptance by a user community, performance and scalability, as well as the conformance to official standards.

¹create, read, update, delete

2.1 Model Transformations

Modeling languages for the specification of model transformations are defined on separate levels of abstraction. An abstract syntax (AS) defines the logical structure by means of a metamodel, where as a complementary concrete syntax (CN) defines its representation. The de-facto standard metamodeling language is the Meta Object Facility (MOF) [38], standardized by the OMG. The traditional approach to model transformations is the manual specification of a model transformation in the terms of the language's CN. As a prerequisite to that, users are required to have a deep understanding of the language's concepts, which are specified in the AS, while defining the actual model transformation using the CN. However, since users are typically more familiar with the CN of MTLs, their usability is significantly affected by how much effort it takes to understand the concepts encoded into the underlying metamodel(s) [20]. In order to aid users with this so called *concept hiding* [20] in metamodels, the approach of model transformation by-example (MTBE) has been suggested [6, 20, 21, 27, 55]. Unlike the traditional transformation by-modeling, this approach is result-driven as the specification of model transformations is done indirectly in terms of *examples* using the CN. The model transformation is then derived from this exemplary demonstration and can be applied to different input. Hence, users are not longer required to have a deep understanding of the AS of a MTL, as the knowledge in using the CN is sufficient. MTBE is comparable to the recording of replayable macros of user actions, executed in the context of user interfaces [31]. A similar approach is that of transformation by-demonstration (MTBD) [27, 50]. Like in MTBE, the model transformation is specified by the user in terms of examples. But the resulting model transformation is not derived from a post-modeling analysis, but instead from the set of recorded actions the user has performed during the modeling.

Both MTBE and MTBD are promising approaches to the topic of MDE, as a significant usability gain for the modeling process of model transformations is inherent to their underlying ideas. However, in the scope of this thesis we are interested in investigating on the problem of poor usability of *traditionally* applied MTLs, which causes the need for approaches like MTBE and MTBD in the first place.

2.2 Tool Support for QVTr

In this section, we present current state of the art tools for QVTr, and assess them with regards to their level of maturity, automation, interactivity, readability and traceability.

ModelMorf [2] is a proprietary execution engine for model transformation specified in QVTr. As it is implemented as a command-line tool, there is no GUI and no support for QVTr's graphical syntax. Models are read in the XMI format, and semantical error messages are provided after a transformation's execution. Support for the tool has been discontinued while still being in its beta-phase, but there is support for in-place transformations, OCL collection types and the possibility to generate trace output.

QVT Declarative (QVTd) [13] is a project in the context of the Eclipse Modeling Project, and has the aim to provide the capabilities for textual editing, parsing and debugging transformations specified in QVTc and QVTr for research purposes in the context of an Eclipse-based IDE. Although not explicitly targeted at novice users, there has been put effort into increasing the usability. First, QVTd supports syntactic, context-sensitive auto-complete functionality, that significantly eases the textual specification as users are suggested what to specify next at any point in time. Next, the initial configuration of execution profiles for specific QVTr transformation scripts is automated, as input models, output models, execution direction and intermediates are preselected for the user. Error analysis is also done by the editor, but only serve the purpose of error identification, as no QVTr-related quick fixes are suggested to the user. For an enhanced readability, QVTd offers the user to configure syntax highlighting, as for example the font style and color for comments, keywords or strings is changeable. Existing models and metamodels can be used in QVTd using the *XML Metadata Interchange* (XMI) and the *EMF Ecore* data format respectively. Finally, the editor also supports clickable relation calls, increasing the traceability of relations. The possibility for transformation execution has been introduced with the release of Eclipse Neon in June 2016 [59], and there is ongoing work [57–59] with the aim to optimize the execution’s performance.

medini QVT (medQVT) [22] is an Eclipse-based IDE built upon the Eclipse Modeling Framework (EMF) for model-to-model transformations using QVTr, developed by IKV++ technologies. It features the textual modeling, execution, debugging of transformations, and has also support for incremental updates. Its execution engine is open-source and thus is integrable into external editors. The editor provides a concise, dedicated modeling and execution environment for QVTr, featuring refactoring techniques and the display of errors. Similar to QVTd, no applicable quick fixes to identified errors are suggested to the user. medQVT features auto-complete for the process of textual modeling, but is unfortunately subject to indeterministic behavior as no suggestions are made at certain points in time. Syntax highlighting (*e.g.* keywords and strings) is supported, but is not configurable by the user, and like in QVTd, relation calls are clickable. Before modeling, metamodels have to be globally imported using a dedicated configuration window. medQVT has also support for expressions formulated in the Object Constraint Language (OCL) [40].

Enterprise Architect (EA) [48] is an integrated development environment for system design, analysis and source code generation developed by Sparx Systems. 11 years ago, the LieberLieber Software GmbH [1] company has developed a plug-in for EA to enable the editing of QVTr transformations using its graphical syntax. The idea was to improve the usability of the modeling process by (*i*) stemming it from a textual to a graphical level, and (*ii*) by means of user-guidance. In particular, the user is assisted with suggestions derived through metamodel analysis. However, there currently is no support for the execution of the created model transformations and thus also no debugging is possible.

Echo [9, 32, 33] is an EMF-based tool that provides means for repairing models inconsistencies with their underlying metamodels, built upon the Alloy language. Metamodels

are imported in the Ecore format, and models in the XMI format. The user specifies the consistency rules using QVTr and OCL, and is presented the models in a visual way for an enhanced readability. The tool is user-focused, as it does not only detect consistency errors, but also provides quick fixes to repair them. Echo always suggest the minimal set of repair operations needed to resolve the inconsistencies. In this context, the user is free to chose from a graph-edit or an operation-based distance for applying the quick fixes. What distinguished the editor from the aforementioned ones is that the user is presented with dedicated, diagram-like previews for all possible quick fixes applicable to resolve a specific error. That way, the user is given control of the model repair process, thus leading to a high degree of interactivity.

Lengyel et al. [28] have developed a QVTr plugin for their Visual Modeling and Transformation System (VMTS) to define, execute and validate model transformations visually in QVTr. Figure 2.1 illustrates an example transformation modeled with the plug-in. Since the target group of their editor comprises engineers without a background in model engineering the authors wanted to choose a *declarative* model transformation language – namely QVTr. Furthermore their editor allows the definition of high-level validation constraints using OCL (Object Constraint Language) statements. Unfortunately, recent contact with the authors at the University of Technology and Economics Budapest has revealed that the support for QVTr in the current version of VMTS has been discontinued five years ago. We were therefore not able to include the plug-in into our tools assessment. One of the reasons for the discontinued support is the lack of visual control flow in QVTr, which is inherent to its declarative nature. Also, the authors claim that the concept of when and where clauses is tedious to use for branching scenarios. Furthermore, the authors mentioned that the implementation of the imperative OCL part was challenging.

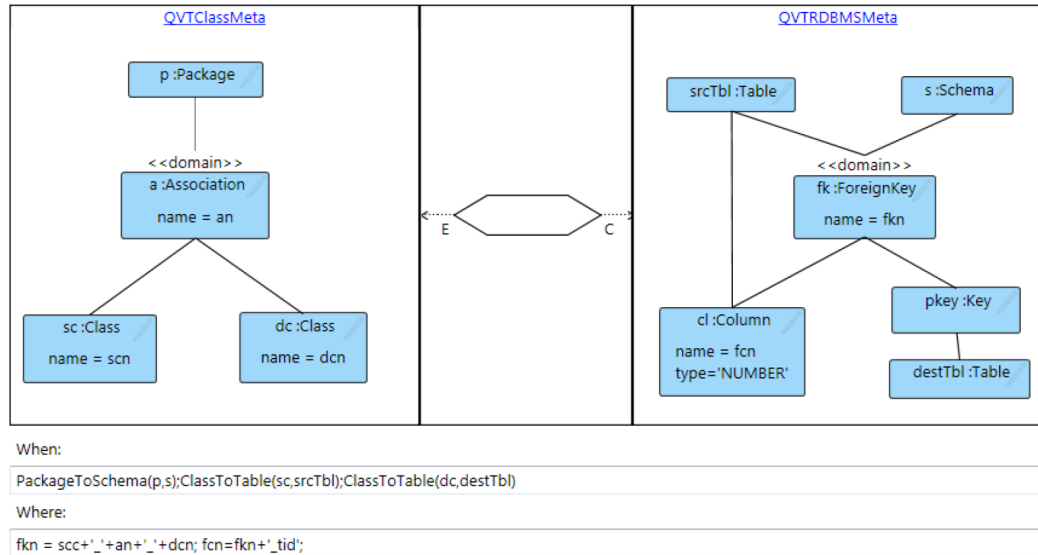


Figure 2.1: Example of a QVTr relation modeled with the VMTS Plug-In [28]

2.3 Complementary Tool Support

In contrast to the modeling and execution of transformations, there are editors for the modeling of *models and metamodels*, as described in the following.

The *Eclipse Modeling Framework* (EMF) [52] is a (meta)model framework upon which tools can be built for modeling, validation, or code-generation. Complementary, the Graphical Modeling Framework (GMF) [53] is a generative system for graphical editors based on EMF.

EcoreTools [14] is a graphical diagram editor and analyzer for models specified in the Ecore format, which is the meta-metamodel of EMF. It features the editing of class diagrams, a constraint validation view and an overview for package dependencies. The editor is convenient for modeling the metamodels needed as input for model-to-model transformations in other, dedicated editors.

Topcased [54] is an Eclipse RCP application that uses EcoreTools to enable the graphical editing of Ecore models and UML diagrams as well as the generation of metamodels in the Ecore format.

Schütze et al. [47] investigate the tooling support for OCL debugging and come to the conclusion that a step-by-step debugger does not yet exist, but that it would be a significant contribution since step-by-step debugging is one of the most-wanted features in the academic community. The authors follow a methodological approach by first defining requirements and features that an OCL debugger should implement. For instance, step-by-step debugging using breakpoints, the visual designation of visited OCL statements, conditional breakpoints or the support for debugger watches. The authors describe their implementation for Dresden OCL, an open-source and Eclipse-based OCL tool and compare it with existing OCL tools based on their defined requirements. They conclude their comparisons with the finding that tree-based debugger views are insufficient for more complex OCL statements that make use of iterators.

Li et al. [29, 30] present an approach to map the selection criteria of QVTr queries to XPath expressions. The goal is to enable the implementation of such queries as XSLT functions and thereby combine the visual attractiveness of the graphical syntax and the expressiveness of the textual syntax. They also present their work on a visual editor, shown in Figure 2.2, that facilitates the specification of queries using an adapted version of QVTr's graphical syntax.

Rentschler et al. [44] propose an interactive, visual analysis tool for Eclipse to ease the maintenance of QVT-Operational (QVTo) transformations. In particular, the tool adds a navigable dependency graph which is synchronized with the state of Eclipse's text editor. They motivate the need for their approach by explaining how error-prone the manual modification of textual QVTo transformations is (*e.g.* using search/replace techniques) and that a high level of expertise in QVTo is needed to do so. In their approach, the model dependencies are statically analyzed and then visualized in an interactive, navigable graph view. The authors also deal with the problem of information

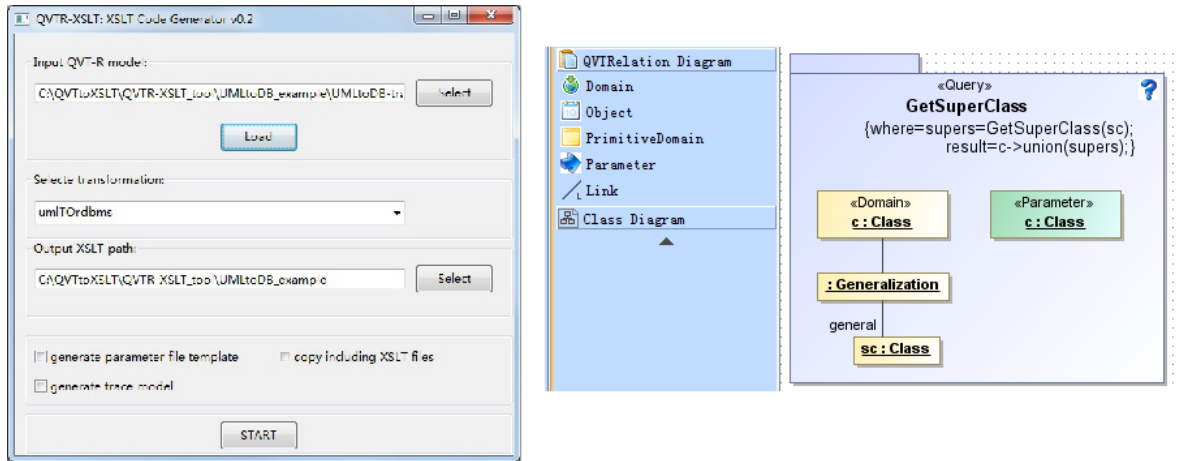


Figure 2.2: An editor to define model queries in a visual way using an adapted version of QVTr’s graphical syntax [29,30]

overload in their graph due to significant numbers of dependencies. In order to mitigate this problem and to decrease the search space, they propose dedicated filter criteria depending on the particular maintenance task that should be carried out. For example, one filter only shows elements that are in direct context to the currently selected unit. In their conclusion, Rentschler et al. claim that their approach significantly improved maintenance efficiency and productivity.

A comparative study proposed by Samimi-Dehkordi et al. [26] in October 2016 states that TGG, which is similar to QVTr, already enjoys comprehensive tool support (MoTE, TGG Interpreter, eMoflon), whereas only a single matured tool (medQVT) exists for QVTr. As can be seen in Figure 2.3 QVTr not only lacks *available tools*, but is also considered to lack *simplicity*, measured by the size of the language’s metamodel(s). The modeling process in the only matured tool medQVT is rather static and hardly offers guidance for a transformation developer. This makes it difficult especially for novice users of QVTr to explore capabilities and limitations of the modeling language. Furthermore, the modeling is based on QVTr’s textual syntax, which, in particular, makes a visual analysis of relation dependencies an inconvenient task. Understandability and scalability are features that hinge on *visual* cues. The work in this thesis contributes to the exploration of the interrelation between the usability of QVTr and the interactivity of the used modeling process. In this context, interactivity is defined as the level of guidance that a user receives throughout the modeling process of a transformation.

Willink [56] motivates the need for a standardized metamodel in order to be able to re-use OCL implementations in tools that make use of OCL-based languages. The author also shows that it is possible to realize static model analysis for QVTr in a textual editor using these standardized OCL implementations.

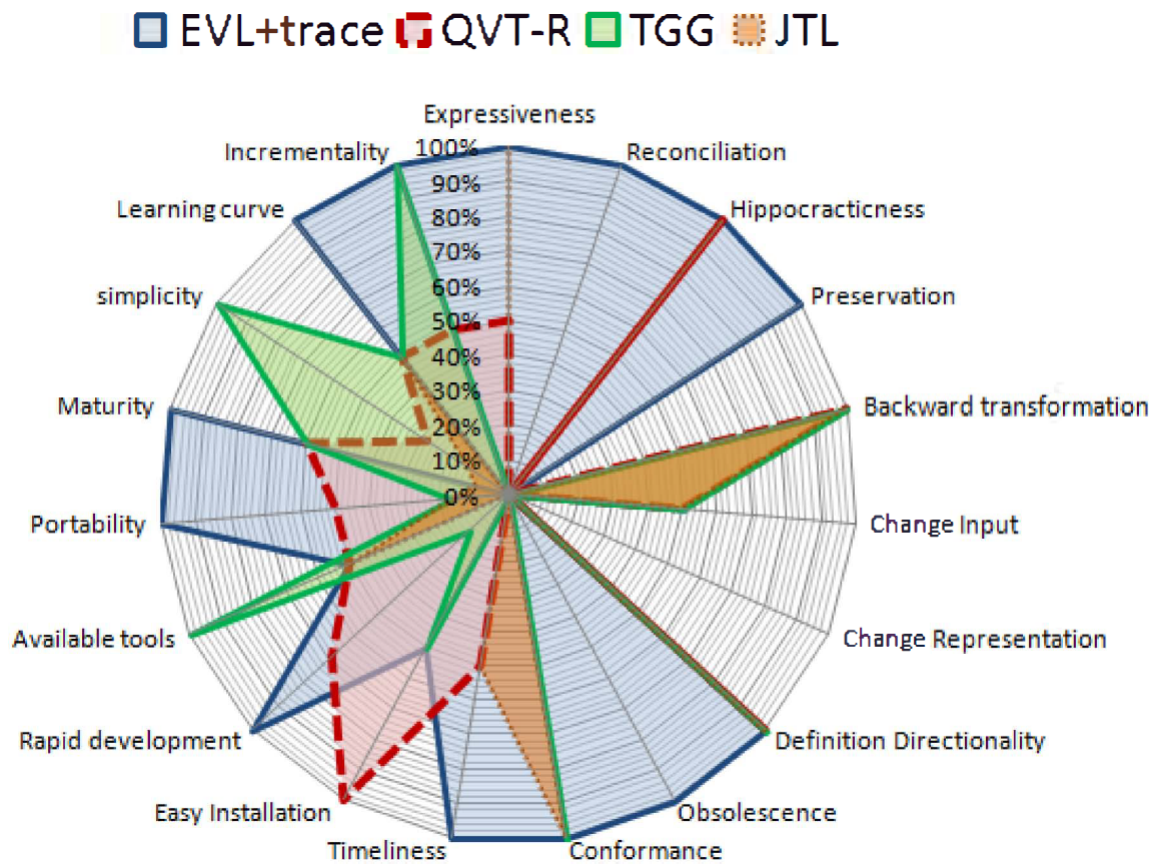


Figure 2.3: Visual comparison of bidirectional MTLs [26]

The QVTr Modeling Process

After our discussion of state of the art tools in the previous chapter, we report on our conducted analysis of the *theoretical* process of specifying model transformations with QVTr in this chapter. Our motivation for conducting this analysis manifests itself in (i) the fact that we have to know how to *manually* model a transformation *before* we are able to automate it. And (ii), in order to design interactivity-increasing techniques, we are required to have a deeper understanding of the problem of model transformation per-se. This includes the definition of model transformation types, a model transformation pipeline, modeling tasks, modeling challenges, and modeling patterns. We have bundled the gained insights from this analysis into a theoretical concept, which is presented in the subsequent chapter. Finally, we elaborate on the different effects that the type of notation used in an editor has on the modeling process. Informally, this chapter reports on *what* can be done to increase usability and productivity, whereas the next chapter focuses on the concrete *how* aspects to do so.

3.1 Types of Model Transformations

Before we are able to elaborate on possible improvements to the process of modeling, we first have to define a clear baseline, and thus delimit the type of model transformations, that we consider in the scope of this thesis. Model transformations have been categorized in academic research [10, 11, 35] by the means of certain features like being in-place, or to which representation the source model shall be transformed to. In the following list, we define which type of model transformations we consider.

Approach Classification. Approaches to model transformations can be classified into being directly-manipulative, imperative, declarative, hybrid or graph-based [11]. In the first form, model transformations are specified by targeting an application programming interface (API) using a programming language such as Java. For instance, by means of

the Java Metadata Interface (JMI) it is possible to modify UML models without the use of a dedicated MTL. The strength of imperative MTLs (*e.g.* QVT Operational) is that of an explicit control flow, where as declarative MTLs have the advantage of low verbosity. With QVTr we consider a declarative MTL. Hybrid MTLs (*e.g.* Atlas Transformation Language [19]) aim to combine the advantages of imperative and declarative languages. Finally, graph-based MTLs (*e.g.* Triple Graph Grammars [46], Henshin [4]) are similar to declarative MTLs as models are put into relation from a logical, graph-based perspective.

Declarative against Operational. With QVTr we consider a declarative model transformation language, meaning that a user specifies the *what* rather than the *how*. Operational languages are also referred to as imperative languages. We also note that functional and logical languages fall into the category of declarative languages.

Executability. The purpose of MTLs is primarily to provide a formalism to *specify* model transformations, whereas the *execution* thereof is not necessarily an aimed goal. With QVTr however, we consider besides the modeling aspects also the specification of executable transformations in this thesis.

Transformation Target. With QVTr we are considering a so called *model-to-model* MTL. In this group of MTLs, the transformation's goal is to instantiate a model of the target metamodel [11]. In contrast, the so called *model-to-text* MTLs serve for purposes of generating another representation of the inputted source model. In this thesis, we consider QVTr model transformations, which by their definition take as input a target metamodel, and generate a target model, as well as a trace model in each execution.

Amount of Input & Output Parameters. Since the work in thesis is about model *transformations*, we do not consider so called “checkonly” transformations, that do not generate a target model.

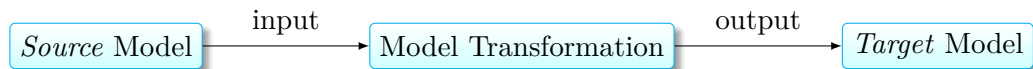


Figure 3.1: Transformations of our interest have exactly one source & target model

For reasons of practicality, we consider only transformations that have exactly one input and exactly one output model, as illustrated in Figure 3.1. In terms of QVTr, we only consider transformations that are executed in the so called “enforce” mode. An analysis that incorporates the support for multiple input and output models is subject to future research, since we consider the groundwork to be done beforehand.

Endogenous & Exogenous. The involved models in exogenous transformations are specified in separate MTLs, whereas in endogenous transformations, the same MTL is used over all models [35]. Since we aim to elaborate on QVTr specifically, we consider the latter type of transformations in the scope of this thesis.

Levels of Abstraction. In this thesis, we primarily consider so called *horizontal* transformations [35], where the degree of abstraction from the source to target model stays the same.

Out-Place & In-Place. Mens et al. [35] state that it is possible to further distinguish endogenous transformations into so called in-place and out-place transformations. In the first case, the number of models that the transformation operates on is equal to exactly one, whereas multiple models are involved in the latter case.

Directionality. In this thesis, we are concerned with transformations, that generate a single target model from a single source model. This type of transformation is unidirectional, as it is executed in only one direction, as Figure 3.2 depicts.



Figure 3.2: The execution direction of our interest is from source to target

In contrast, multidirectional transformations are typically used for model synchronization, as there is no restriction put on the respective execution direction [11].

Source/Target Relationship. Regarding the input and output models, we emphasize that we only consider transformations, where the source and target models are two separate models, which are not the one and the same.

Unconsidered Concepts. Those concepts related to QVTr that are outside of this thesis' scope are Black-Box operations such as integration of OCL expressions, change-propagation, collection types (such as Set, Bag, Sequence and OrderedSet), and negated object templates.

3.2 Model Transformation Pipeline

Figure 3.3 illustrates the big picture of model transformations that we consider in the context of this thesis. We note that only transformations are considered, that have exactly one source and exactly one target model, as depicted in the pipeline figure. First, we have the transformation *definition* (TD), which has to be manually modeled by the user. This transformation has to conform to the syntactic rules specified by the QVTr language, as well as to the used metamodels on both the source and target side. Although we have two separate instances of metamodels (the models themselves), we allow that both the source and target model refer to the same metamodel. Finally, we have an execution engine, that (i) takes as input the source model. (ii) The engine reads the defined TD along with the source metamodel it uses, and selects certain elements in the source model, according to the selection queries contained in the TD. In terms of QVTr, all “checkonly” domain patterns are evaluated against the source model.

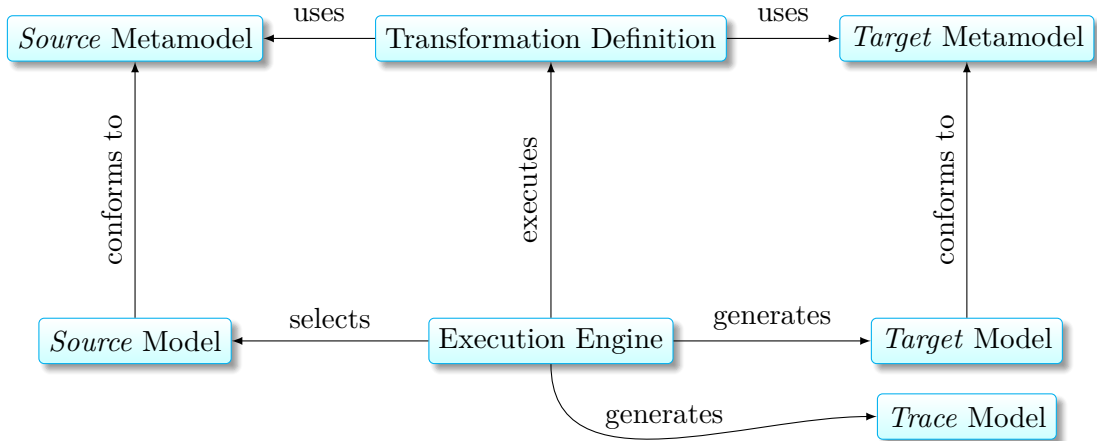


Figure 3.3: The Model Transformation Pipeline [11]

After this selection has been done, the execution engine generates a target model taking as input these selected elements from the source model, and the generative patterns contained in the TD. Again, from the perspective of QVTr, the “enforce” domain patterns are evaluated towards the selection from the source model. Then, besides a target model, also a *trace* model is generated in parallel. The trace model is a model that includes information about the links between the selected elements in the source model, and their correspondent elements that have been generated in the target model. Finally, with the generation of both the target and trace models, the execution of the transformation ends. Furthermore, we recognize that for a successful passing through the pipeline, the syntactic correctness of the TD, and the source model are required as necessary prerequisites. We point out that an editor may or not be implemented in a way such that it is separated from its execution engine. In the context of this thesis, we chose for this separation since we are interested in the specificities of the *modeling* process, and take a functional, usable execution as granted. In other words, the process of designing execution engines is outside of this thesis’ scope. Similarly, we also assume syntactically and semantically correct metamodels to be given. In particular, we will refer to two specific metamodels throughout this thesis, as means to provide examples of QVTr scripts. For this purpose, we have decided to use the **SimpleUML** and **SimpleRDBMS** metamodels, as defined in the QVT standard specification [41]. The respective diagrams for both metamodels can be found in Figures A.1, A.2 in the appendix.

3.3 Modeling Tasks

In order to gain a deeper understanding of the process of modeling in QVTr, we examine the **SimpleUMLToSimpleRDBMS** example transformation, defined in the QVTr specification [41], and define common tasks that we encounter throughout the process.

For this to achieve, we utilize a usability inspection method known as *cognitive walk-through* [17]. Using this method, we first define a set of tasks that the user has to perform, in order to successfully model the mentioned transformation. We then walk through these steps one-by-one, while asking ourselves if our conducted process of modeling so far has potential for improvement. If so, we document our thoughts for later incorporation. The concrete techniques to cope with the challenges we encounter in this process are reported in the next chapter, where we present our theoretical concept for productive modeling with QVTr.

Figure 3.4 outlines the required tasks we define for the modeling process of QVTr. We note that although relations have a containment relationship to transformations, we consider the creation of relations in isolation, and not to be a subtask of that of creating transformations. Hence, we have defined 3 main tasks, *i.e.* the creation of transformations, the creation of relations, and complementary tasks, which go hand-in-hand with the former two. This choice of separation is reflected in Figure 3.4 as the size of the rectangles suggest the level of complexity of the respective task.

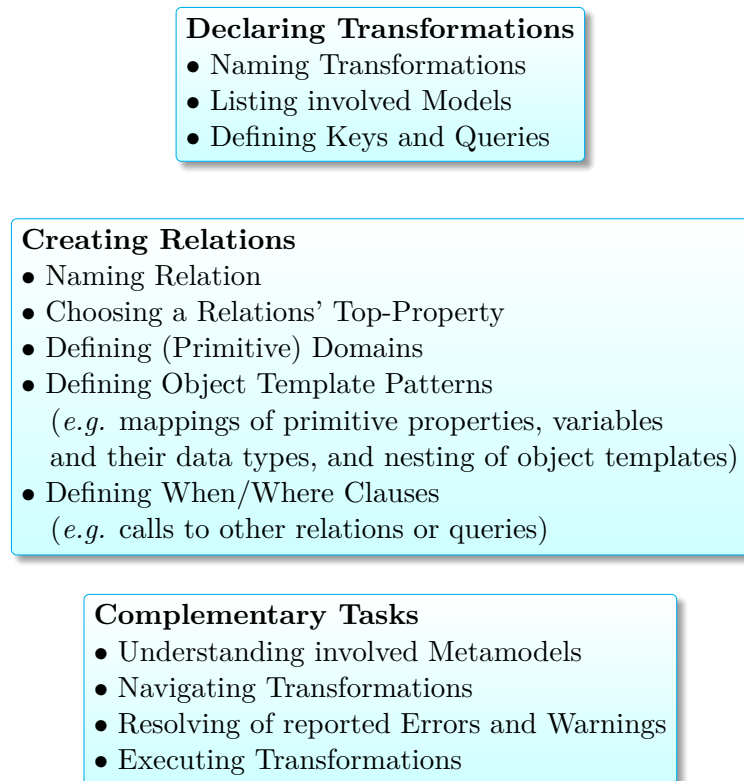


Figure 3.4: Categorization of tasks in the modeling process of QVTr

3.3.1 Declaring Transformations

Before we are able to model any relations, we have to first define their “housing” in terms of a transformation declaration. We distinguish between a transformation’s header and body. The first segment comprises its name, a list of involved models, keys and queries. The body on the other hand contains the list of relations. We note that no definite order is defined over this list.

Naming Transformations

At first sight, naming a transformation seems to be a trivial task, which is not worth being analyzed for usability potential. However, there are certain quirks to it. First of all, the name of elements such as transformations, relations, or object templates have to conform to the invalidation rules defined in the QVTr standard, where the names of these elements are referred to as *identifiers*. The specified list of reserved keywords in QVTr comprises `checkonly`, `domain`, `enforce`, `extends`, `implementedby`, `import`, `key`, `overrides`, `primitive`, `query`, `relation`, `top`. The standard states that identifiers are not allowed to be equal to either of the listed reserved keywords. In order to escape these terms in names where their usage can not be avoided, the standard suggests to prepend an underscore character ‘`_`’, and to enquote the name with single quotation marks. For example, instead of `key`, the usage of `__'key'` is recommended, as it conforms to the so called underscore-prefixed-string-literal escape from OCL. Although not explicitly specified in the standard, there are well-established best-practices for naming in state of the art programming languages. That is, we consider valid identifiers to be case-sensitive, not to start with numerical characters, nor to contain any special or whitespace characters. These restrictions are supposed to be consistently followed during the creation of the transformation, and have the goal to increase the overall code quality. But there are further conventions that potentially push further towards the same goal. So far, we have only considered syntactic rules for names, but there are also semantic rules to be considered. For example, it is conceivable to encode properties of the transformation directly into its name for an enhanced readability and traceability. For instance, consider the transformation declaration defined as follows.

```
transformation UMLToRDBMS (uml:SimpleUML, rdbms:SimpleRDBMS) { ... }
```

The name “UMLToRDBMS” encodes two types of information. (i) The terms “UML” and “RDBMS” refer to the involved metamodels. (ii) The binding term “To” reports on the execution direction of the transformation. Note, that a term such as “From” could also be used as binding word, but may decrease readability, since a name of such form implies a right-to-left reading direction, opposed to the left-to-right direction given by the rest of the QVTr script.

Listing involved Models

The next step of declaring a transformation is result-driven, as it is specified which (source) model the transformation shall transform into which (target) model as a result. As can be seen in the following script example, it is conceivable to reflect the transformation’s execution direction not only in its name, but also with the order of the model list. If an editor takes the effort to enforce such a convention, then it is even possible to automatically derive the list of models from the transformation’s name, while the user types it in. As a required prerequisite to this however, a list of available metamodels has to be given as input.

```
transformation UMLToRDBMS (uml:SimpleUML, rdbms:SimpleRDBMS) { ... }
```

Another point to consider is the naming of the models. Ideally, an editor also enforces a certain convention also in this case, to be able to apply automation here as well. A conceivable heuristic takes the transformation’s name and the set of metamodels involved in the model list as input, finds the longest common substring (*e.g.* [16]) by comparing them, and uses the lower-case version of that substring as the name for the models. For example, the heuristic would then derive “uml” from “UML” and “SimpleUML”, and “rdbms” from “RDBMS” and “SimpleRDBMS”.

Defining Keys

Similar to the concept of keys in relational database management systems, the purpose of keys in QVTr is to prevent duplicate entries – in our case in the target model. A key, according to the definition in the standard, is “*a set of properties that uniquely identifies an object instance of the class in a model*”. Multiple keys may be defined for this purpose. In order to define a key, the user has to choose a type from one of the involved metamodels, and select a list of primitive properties, that are defined in this type. For instance, a key of the form **key** Column { name }; would ensure that no two columns exist in the target model, that share the same name. An editor may also check for duplicate keys, as a key is uniquely identified by its type and an (unordered) list of primitive properties. For types that have a containment relation to a containing type, an editor may offer to select so called “non-navigable opposite role” besides other primitive properties. For example, such a role is necessary to express that an **Attribute** is uniquely identified by its name and its owning type, which is **Class** in this case. The following script shows an example, of how such a key is defined in QVTr.

```
1  transformation UMLToRDBMS (uml:SimpleUML, rdbms:SimpleRDBMS) {
2      key Attribute {name, opposite(Class.attribute)};
3      ...
4  }
```

Defining Queries

Queries serve the purpose of utility functions in QVTr. For instance, a query can be used to transform string representations of data types from one platform to another.

```
1  query PrimitiveTypeToSQLType(primitiveType:String) : String {  
2      if (primitiveType = 'INTEGER' )  
3      then 'NUMBER'  
4      else if (primitiveType = 'BOOLEAN' )  
5          then 'BOOLEAN'  
6          else 'VARCHAR'  
7      endif  
8      endif  
9  }
```

Figure 3.5: Example query to convert QVTr data types into SQL format

Figure 3.5 illustrates an example, in which primitive data types of QVTr, namely INTEGER, BOOLEAN, STRING, are converted into their corresponding SQL types, which are NUMBER, BOOLEAN, VARCHAR respectively. In order to define a query, the user has to (i) provide a query name, (ii) a list of input parameters, (iii) a return data type, and (iv) finally the actual business logic of the query. This is closely related to the way functions and methods are defined in common general purpose languages such as Java or C#. One consideration that editors could take into account for the process of defining queries is to view them as black-boxes. This means, that users first have to define the interface they want to work with when invoking the query. In particular, the user begins with defining the query’s header, *i.e.* the name, list of input parameters, and the return type. We identify potential for automation in this process, depending on the contextual information that an editor has access to. For example, if the query is defined by the user *just-in-time* while defining a predicate in a when/where clause, chance is that an editor is able to automatically derive the return data type from that expression.

```
1  relation PrimitiveAttributeToColumn {  
2      sqltype : String;  
3      ...  
4      where { sqltype = *; }  
5  }
```

Figure 3.6: *Just-in-time* query definition

Consider Figure 3.6, where the \star symbol denotes the user’s current caret position in the process of creating the `PrimitiveAttributeToColumn` relation. An editor, that offers the possibility to define a new query at this point has access to the contextual information that

the query’s return type has to be compatible with that of `String`. This context-sensitive technique is applicable for all kinds of primitive and complex types. However, we notice that in the case of relation calls, no such type derivation is necessary, since the return type of relation calls defaults to `Boolean`.

```

1  relation PrimitiveAttributeToColumn {
2      sqltype, pn : String;
3      ...
4      where { sqltype = PrimitiveTypeToSqlType(pn, ★); }
5  }
```

Figure 3.7: *Just-in-time* parameter definition

Similarly, this technique is applicable to the derivation of data types for input parameters of a query. Consider Figure 3.7, where an editor is able to derive, that the data type of the first parameter of the `PrimitiveTypeToSqlType` query has to be compatible with `String`.

Finally, there is potential for automation in the providing of default names for both the query’s name, and that of the parameters. Since in the first case, an editor has to semantically understand the purpose of the query using only the information available from its invoking relation, a pragmatic heuristic to that is the suggestion of generic names such as “query1” or “PrimitiveAttributeToColumnQuery”. In the latter case, the names of the variables that are handed over to the query in its containing relation can be directly reused. For instance, in the example depicted in Figure 3.7, an editor can use “pn” as the name of the query’s first parameter, without negatively affecting the readability.

3.3.2 Creating Relations

The creation of relations can be considered a subtask of creating transformations. Hence, our previous usage of the term *declaration* instead of *definition*. In QVTr, the source side of a relation *selects* elements, whereas elements are being *generated* on the target side. In this sense, a user has to mentally perform a mapping-driven process, where relations are its output.

Naming Relations

Like with the declaration of a transformation, one of the first tasks a user has to perform is the naming of the relation to be created. Similarly, an editor may establish a convention that encodes certain properties of the relation into its name. For example, it is conceivable to communicate the mapped domain types with names of the form “PackageToSchema”, “ClassToTable”, or “AttributeToColumn”. Again, the “To” term encodes the direction of the relation, and thus determines its source and target side.

Choosing a Relation's Top-Property

The decision of whether a relation shall be top or not is a non-trivial task, which significantly influences the output in the generated target model. In order for a user to make an *educated* decision, the whole set of defined relations in the transformation has to be taken into account. To support this process, editors may provide the user with graphical visualizations of the call hierarchy and dependencies among relations, and also perform some dependency analysis automatically. An editor is able to analyze the dependencies by considering already existing relation calls in when/where clauses of defined relations. Another point that editors have to consider is that of to which value the top-property shall be set to by default. In the case of relations being created from scratch, a default value of **false** may quickly turn out to be tedious for users, since they have to manually opt-in for the top-property in order to integrate the relation into the transformation's call hierarchy, and thus enabling the relation to affect the generated output model.

Defining (Primitive) Domains

With means of relations, the user declaratively specifies which elements in the source model shall be transformed into which elements in the target model. The answer to this question already defines the domain patterns of the respective relations. Domains can also be seen as entry-points to the previously defined models of a relation [11]. Informally, we can consider them as the calling parameters of a function. Hence, domains represent the set of input parameters, that a relation call has to provide. We distinguish between domain patterns for the source and target side.

In total, for defining a domain, the user has to (i) provide the respective source/target side, (ii) the type, (iii) the model, and (iv) the domain's name. Assuming that the list of models in the containing relation encodes its execution direction as described before, an editor may let the user select certain types of the involved metamodels, and automatically derive whether they should be located on the source or target side using this information. For example, from "PackageToSchema" as the relation's name, a heuristic may derive a checkonly domain `uml p:Package`, and an enforce domain `rdbms s:Schema`. We notice that this heuristic uses generic default names for the domains, such as "p" and "s" to hint to their respective types in the metamodel.

However, editors shall use such an heuristic with caution, since it is allowed for different relations to have the same types of domains, while having different names. An example for this is the `AttributeToColumn` relation in the `SimpleUML` metamodel, that has domains of the types `Class` and `Table`, similarly to the domains of the `PrimitiveAttributeToColumn` and `ComplexAttributeToColumn` relations.

The role of *primitive* domains in QVTr is to provide means for a relation to take additional parameters as input. Therefore, the name of such domains is closely tied to the semantic meaning of the parameter's usage within the relation. For example, in the

UmlToRdbms transformation shown in the standard specification, primitive domains of the form `primitive domain prefix : String` are frequently used.

```

1  relation PrimitiveAttributeToColumn {
2      an, cn : String;
3      primitive domain prefix : String
4      where {
5          cn = if (prefix = " ") then an else prefix + ' ' + an endif ;
6      }
7  }
```

Figure 3.8: Example usage of a primitive domain

Figure 3.8 illustrates, how a primitive domain is used as calling parameter of the `PrimitiveAttributeToColumn` relation, to concatenate together the name of a `Class` object with a given prefix.

Defining Object Templates (OTs)

The effects of object templates are sensitive to the kind of domain, in which hierarchy they are defined in. That is, on the source side, OTs further *restrict* the selection criteria, that the complete pattern represents. Under a target side domain however, OTs further *refine* the generative pattern. Informally, the more OTs a user defines on the source side, the less elements are being selected. In contrast, the more OTs a user defines on the target side, the more elements are being generated. A user has to be aware of this difference, in order to effectively define OTs. The challenge in defining OTs for the source side is to find the balance between a too strict and too weak selection pattern. From a perspective of effectiveness, it is crucial for users to carry a strong understanding of the underlying metamodel, in order to extend OTs with compatible OTs. We identify potential for automation in this context, since an editor may provide non-committal suggestions of extending OTs to the user derived by means of static metamodel analysis. Besides defining the blueprint of models via raw OTs, they can also hold primitive properties. A common usage is the mapping of primitive properties using variables. We also identify potential for automation here, since an editor may also offer suggestions for the mapping of common primitive properties, taking into account multiple OTs, that the user has selected beforehand. Since QVTr is a declarative language, the mapping of the values from source-to-target is done implicitly with variables that have the same name. Figure 3.9 illustrates this, as the primitive `name` property is mapped from the `p` domain on the source side, to the `s` domain on the target side.

Defining When/Where Clauses

When and where clauses allow the user to specify pre- and postconditions to a specific relation. A precondition may be the call to another relation, that has to be executed

```

1  top relation PackageToSchema {
2      pn : String;
3      checkonly domain uml p : Package { name = pn };
4      enforce domain rdbms s : Schema { name = pn };
5  }

```

Figure 3.9: Mapping of the primitive name property with OTs

before. An example to this is the binding of unbound containment relations, as illustrated in Figure 3.10.

```

1  top relation PackageToSchema {
2      checkonly domain uml p : Package {};
3      enforce domain rdbms s : Schema {};
4  }
5  top relation ClassToTable {
6      checkonly domain uml c : Class { namespace = p : Package{}; }
7      enforce domain rdbms t : Table { schema = s : Schema{}; }
8      when { PackageToSchema(p, s); }
9  }

```

Figure 3.10: Binding of an unbound containment relation

In this example, the `ClassToTable` relation specifies two nested domains, each referencing a parent type, *i.e.* `Package` in the case of `Class`, and `Schema` for the `Table` type. Another usecase for when and where clauses is that of calling other relations. With the concept of relation calls, in either a when or where clause, the user establishes a call hierarchy among the relations of a transformation. Thus, the execution order of relations is defined. For a relation to be invoked by an execution engine, it has to either be defined as a postcondition in the where clause of a calling relation, or defined to be a top relation. Besides relation calls, where clauses are also used to bind unbound variables, which are defined in an enforce domain.

```

1  top relation PackageToSchema {
2      checkonly domain uml p : Package {};
3      enforce domain rdbms s : Schema { name = pn };
4      where { pn = 'schema1' ; }
5  }

```

Figure 3.11: Binding of an unbound containment relation

Consider the example shown in Figure 3.11. Without the assignment of `pn` in the `where` clause of the `PackageToSchema` relation, the value of the primitive `name` property of the `s` domain of kind “enforce” would be unbound.

Hence, the execution engine would have no means to determine which value to assign the name of the schema element to, which is being generated in the target model. We also note that the usage of OTs can be categorized into different modeling patterns, such as branching, merging and mapping. Such patterns are explained in more detail later in the section about modeling patterns.

Further usecases for when and where clauses are that of invoking queries or the integration of black-box operations, such as OCL expressions, where the latter one is outside of this thesis’ scope.

3.4 Complementary Tasks

The tasks described so far are purely functional, but the modeling process also requires the user to perform non-functional tasks of complementary nature. In this context, we distinguish between the understanding of metamodels, navigation through transformations, the incorporation of reported errors, and the execution of transformations themselves.

3.4.1 Understanding involved Metamodels

In informal terms, metamodels provide the user with the “building blocks” needed to specify transformations. While examining through the modeling process of our example transformation, we noticed that a significant amount of time is spent with understanding the involved metamodels, their types and primitive properties. This is especially the case when defining new OTs, since the extension thereof underlies the rules of type compatibility as defined in the respective metamodel. The need for an understanding of the metamodel comes apparent when, for example, the user intends to define the containment relation of the `Class` domain in the `ClassToTable` relation, as depicted in Figure 3.10. Using an editor that does not provide the information, that the relation to `Package` is established through the primitive `namespace` property, the user has to spend effort with manually consulting the metamodel documentation of the `SimpleRDBMS` metamodel. The same applies to complex properties, *i.e.* the types of OTs with which a certain OT can be extended with.

3.4.2 Navigating Transformations

A task that is closely related to the implementing editor is that of navigation. We identify multiple forms of it, including *(i)* the problem of having an overview of defined relations, *(ii)* understanding the execution order, *(iii)* identifying variable occurrences, *(iv)* understanding the nesting of object templates, *(v)* identifying unused relations, and *(vi)* defining entry points for the execution of the transformation. Each of these tasks require navigation through the various different views editors provide the user with.

Facing the variety of navigational tasks, we identify that the degree of how navigable a transformation is, significantly depends on the navigation capabilities of the used editor.

Considering the model transformation pipeline and its implications on the amount of information needed to be presented to the user, we identify that the proper design of navigable views for an editor is a non-trivial task. Editors have to find the balance between a clear, uncluttered layout and the level of accessibility to the editor's modeling functionalities. In addition, editors have to find a way to present dedicated views for the modeling, execution and analysis of transformations. For an editor to provide short feedback cycles, users have to be able to easily navigate from the modeling to the execution view, and back again. And there are even further types of views that users may find useful throughout the modeling process. That is, editors may provide dedicated, editable views for both the graphical and the textual syntax of QVTr.

3.4.3 Resolving reported Errors

One of the major benefits of using a dedicated modeling editor is that of a reduced analyzation effort, as the editor performs static and dynamic error analysis and reports the results to the user in the form of visually represented errors. In contrast to static (metamodel) analysis, dynamic (model) analysis takes the so-far modeled transformation into account. We identify that the resolving of an error consists of *(i)* perceiving its existence as the editor indicates this with error icons or similar means, *(ii)* understanding what the problem is, *(iii)* understanding the cause which includes locating the involved elements, *(iv)* developing a solution to it, and *(v)* validating the error's successful resolving. Regarding *(i)*, editors may use specific colors, icons or text in a dedicated errors view, or the status bar of the main window. For case *(ii)*, editors may provide both a concise and a comprehensive, textual problem description. Concerning *(iii)*, the process of debugging is an established technique to understand causes of errors. However, debugging techniques are outside of the scope of this thesis, since it is a user-driven technique. Instead, we are interested in ways of implementing editor-driven *user guidance*, such that the required modeling effort for users is minimized. For example, visual hints like underlined or highlighted text, or elements on a diagram canvas help to identify location and identity of the error's involved actors. That way, involved actors to a problem are easily identifiable, which drastically reduces the search space for error-related parts of the transformation. In the case of *(iv)*, editors may provide a textual description of a suggested solution to the problem. Furthermore, it is even conceivable to offer the application of this solution in a one-button-click-to-apply fashion. It is also conceivable to even provide multiple of such easy-to-apply solutions, and ease the user's decision to choose one of them by letting the user peek into previews of the respective effects on the transformation. Finally for *(v)*, editors may also visually indicate with success notifications, that a specific error has been successfully resolved. For example, a message like "Successfully resolved the unbound containment in the ClassToTable relation." could be decently presented to the user. Appropriate means of presenting this information without interfering with the modeling process are push notifications, animated progressbars in a status bar, or a

fade-out animation of the error from its list in the dedicated errors view.

3.4.4 Transformation Execution

Another task that is strongly tied to the implementing editor is that of executing the modeled transformation. A common approach to this is the usage of configurable profiles, which have to be manually set up by the user initially, and can then be reused throughout the subsequent modeling process. Such profiles contain all information that an execution engine needs, in order to perform the transformation. For our purposes, we derive this information directly from our defined transformation pipeline in Figure 3.3. Hence, we define the following information for such an execution profile to have.

- The file locations to all involved *metamodels*.
- The file location to a *source* model.
- The *transformation* to be executed.
- The *execution direction* of the transformation.
- The file locations to where the *target* and *trace* models shall be generated to.
- (Optionally, a file location to store logging output to.)

We also require the metamodels, source-model and the transformation to be in a data format, that the execution engine supports. Popular data formats are the *Extensible Markup Language* (XML) [62] format for trace models, the *XML Metadata Interchange* (XMI) [42] format for source models, the *EMF Ecore* [52] format for metamodels, and the textual format of QVT for the transformation script itself. We identify the following challenges for editors to overcome. First, if the editor caches the metamodels for performance reasons, they may become out-of-sync when externally updated. The same applies to the source and target models. Editors may attach file change listeners to the respective file locations, and inform the user about a possible data loss, if not incorporating the updated file. Second, editors that target external execution engines which are not part of the editor itself, depend on the level of integration the execution engine provides. This comes apparent, if an execution engine only stores the generated output directly to disk, and not to memory. This way, editors have to take the detour of waiting for the files to be successfully written, and reload the generated file after each execution. For editors with performance constraints, this is an important, architectural point to consider. Finally, we identify significant usability potential in the way and behavior of an editor to provide and present the generated output to the user. Conceivable points to consider are instant-presentation after each click on a dedicated “generate” button, automatic re-generation after each change a user applies to the transformation, the incorporation of trace links into the target models’ visualization, or the heuristic positioning of the diagram elements, in the case of such a visualization.

3.5 Modeling Challenges

In this section, we provide a non-exhaustive list of common modeling challenges that the user encounters during the modeling process of QVTr.

3.5.1 Relation Dependencies

The first modeling challenge in our list is the establishment of a correct call hierarchy among relations such that the desired output is generated correctly. This involves the sub-challenges of *(i)* understanding dependencies, *(ii)* the decision of which relations should be top, and *(iii)* what the order of execution over all relations is. Unfortunately, there is no bullet-proof method to answer these questions, but we can nevertheless provide the following heuristics to head towards the right direction.

- A high degree of unrelated top relations, *i.e.* they have no relation calls defined in when clauses between them, commonly results in multiple unconnected parts in the target model.
- On the one hand, it is conceivable that relations, which match types of higher hierarchy in their respective metamodel, are the first to be executed. On the other hand, the inverse conclusion of “the more specific, the later in the order” should be drawn with caution, because such relations can also be modeled as stand-alone *top* relations, which are then generating independent parts by themselves.
- Dependencies that are already established by existing relation calls in when and where clauses, provide considerable hints to which relations may be chosen to be top, and which not. In the next chapter of this thesis, we introduce a technique to automate this in an editor.
- Given a transformation, that already produces the desired target model, it is conceivable to analyze the call hierarchy for *unused* relations. A purging heuristic may be implemented for this purpose, as it identifies non-top relations, that are also not called from elsewhere via a relation call within a where clause.

3.5.2 Result Validation

Another modeling challenge the user has to cope with is that of validating that the generated output conforms to the expected output. This includes the *(i)* execution of the transformation, *(ii)* the analyzation of the generated result, and *(iii)* the identification of changes needed to adapt the transformation to push it towards the expected result. In the field of software engineering, this process is also referred to as the incorporation of so called *feedback cycles*, which ideally are as short as possible in terms of time, and available as early as possible. The duration of such cycles hinges on the support for them in the respective editor. For example, editors may auto-update the target model after each change made to the transformation.

Another consideration is how the relative changes after each new transformation execution shall be visualized and presented to the user. Fade-in animations or similar means potentially enhance the traceability of performed actions and their effects for the user, and ensure a minimized time and effort to gain modeling feedback from the editor.

3.5.3 Type Completeness

This challenge refers to the detection of unused types in the involved metamodels. In particular, we consider a type unused in the scope of a transformation, if it does not occur as the type of an OT in any relation in the transformation. The effect of an unused type is possible data loss from source-to-target. In the other extreme case, types might be covered multiple times and, unless the involved relations are not defined in a way such that they select *disjoint* sets in the source model, redundant parts may be generated in the target model.

3.5.4 Traceability

Another challenge in the modeling process regards the traceability of the generated output. Specifically, the task of understanding which patterns in the transformation are responsible for the generation of which elements in the target model, based on which selection in the source model.

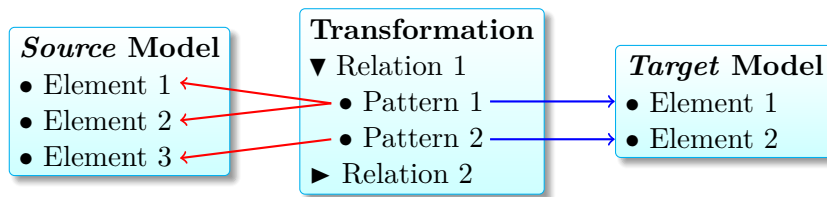


Figure 3.12: Traceability links between the source and target model

Figure 3.12 visualizes the source (red) and target (blue) links, that a user has to be able to make, in order to trace back the elements in the generated output. We identify that the way editors visualize these links significantly influences this form of traceability. In addition, there are further forms of traceability, like that of tracing back all occurrences of a variable or a specific string literal in the scope of a relation.

3.5.5 Naming

As described before, the naming of elements such as transformations, relations, object templates and variables is, like in traditional software development, a common problem which is tackled by using standardizations, conventions and best-practices. Common errors are the usage of reserved keywords or invalid characters like special or whitespace characters within names. The level of how an editor communicates these invalidation rules to the user significantly influences the complexity of this modeling challenge.

Another point to consider is that the naming of elements can significantly be automated by an implementing editor. We introduce techniques to do so in the next chapter of this thesis.

3.6 Modeling Patterns

Throughout the modeling process in our example transformation, we have identified different modeling patterns of how variable assignments and object templates can be used to achieve different kinds of goals. In the following, we list the patterns we identify for the modeling process of QVTr.

3.6.1 Mapping

A common pattern is that of a 1:1 mapping. There are different variations to this, may it be *Type-To-Type*, *OT-To-OT* or *Primitive-Property-To-Primitive-Property*. Independent from the case, the goal is always to match semantically equivalent pairs from the source to the target model.

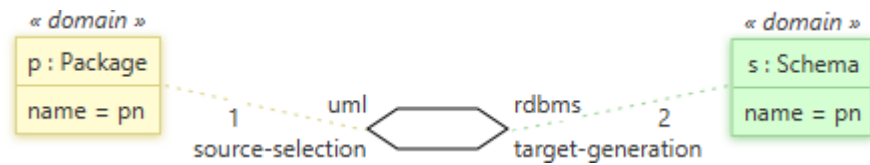


Figure 3.13: Example of a 1:1 mapping with the PackageToSchema relation

The precedent case for a mapping pattern in our examined example transformation is that of the PackageToSchema relation, as shown in Figure 3.13.

3.6.2 Branching

To understand the problem of branching, we consider Figure 3.14, in which we have two elements of type *Class*, and where one also has an *Attribute* element. If we want to treat *Classes* differently, in case they have attributes, we have to use the branching modeling pattern. A conceivable usecase for this would be when all *Classes* shall be transformed to *Tables*, and *Attributes* shall be *Columns* that are contained in the *Table* elements. This is done by defining a so called *branching relation* of name *AttributeToColumn*, in which another relation *PrimitiveAttributeToColumn* is invoked by means of a relation call in the *where* clause. Figures 3.15 and 3.16 illustrate these relations. It is conceivable to add further relations in a similar way such as to support cases like super classes.

3.6.3 Enriching

We identify a variation of the mapping pattern as to when there is one input element, but multiple output elements.

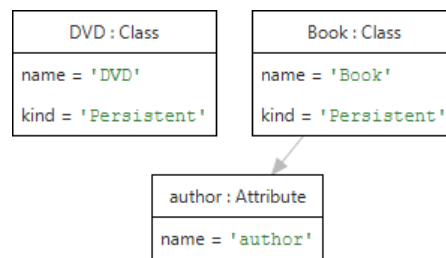


Figure 3.14: Example of the branching modeling pattern

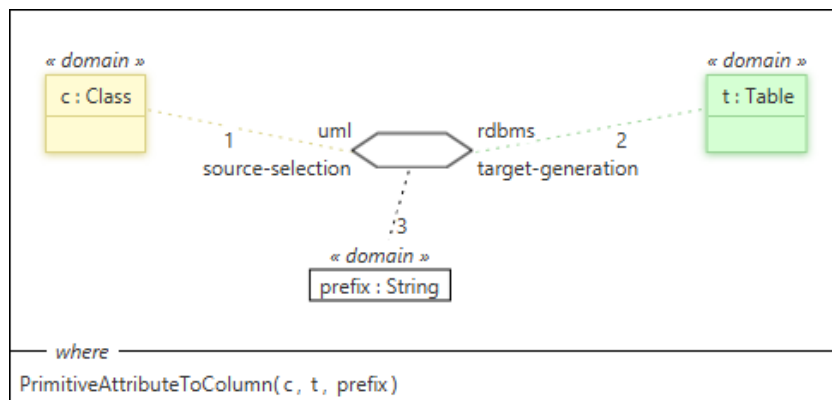


Figure 3.15: The AttributeToColumn relation in our example

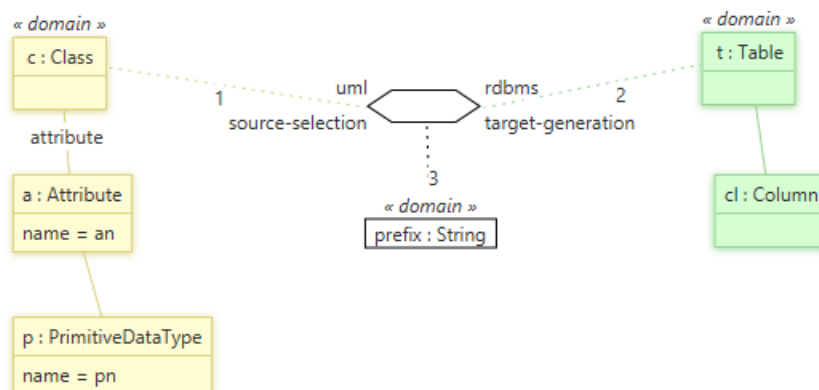


Figure 3.16: The PrimitiveAttributeToColumn relation in our example

One could also refer to the enrichment pattern as a 1: n mapping. Figure 3.17 illustrates an example, as for all **Classes** not only **Tables** are generated, but also a primary key **Column** that holds *id* values.

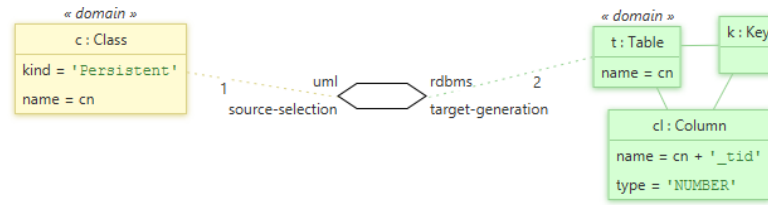


Figure 3.17: Example of a model enrichment

3.6.4 Merging and Hierarchy Flattening

Contrary to the enriching pattern, we refer to a $n:1$ mapping as the *merging* pattern. Figure 3.18 depicts an example, as a single Table is generated for Classes, that have a parent relation to a super element of the same type.

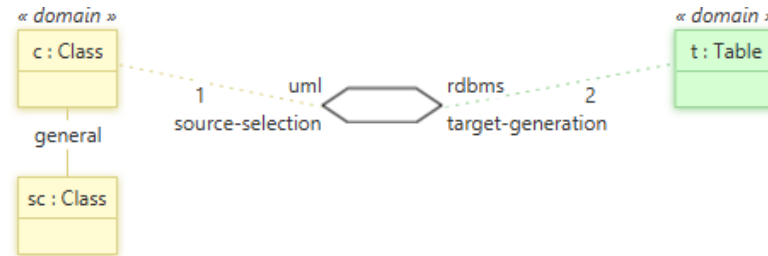


Figure 3.18: Example of the merging pattern

3.6.5 Binding

So far we have considered different variations of the mapping pattern. Another pattern is that of *binding* unbound containment relations or variables. We have already discussed the former usecase in Figure 3.10. An example for an unbound variable is that of an expression of the form `name = pn` within the hierarchy of an “enforce” domain pattern, where `pn` does not have any other occurrences within the same relation. This variable could be bound by adding the expression `pn = 'packageName'` to the where clauses of the respective relation.

3.6.6 Shared Parent

In the case of metamodels defining containment relations among their types, the pattern of a so called *shared parent* occurs. This challenge is concerned with the question of how to achieve it, that certain elements are contained in the same, shared parent element. In the example shown in Figure 3.19, two elements `Book:Table` and `DVD:Table` have been generated by the `ClassToTable` relation, as illustrated in Figure 3.11. Although the user has specified that for all elements of type `Class`, which have a relation to an element of type `Package`, a corresponding `Table` element should be created contained in a parent element of type `Schema`, both objects still have their own parents defined.

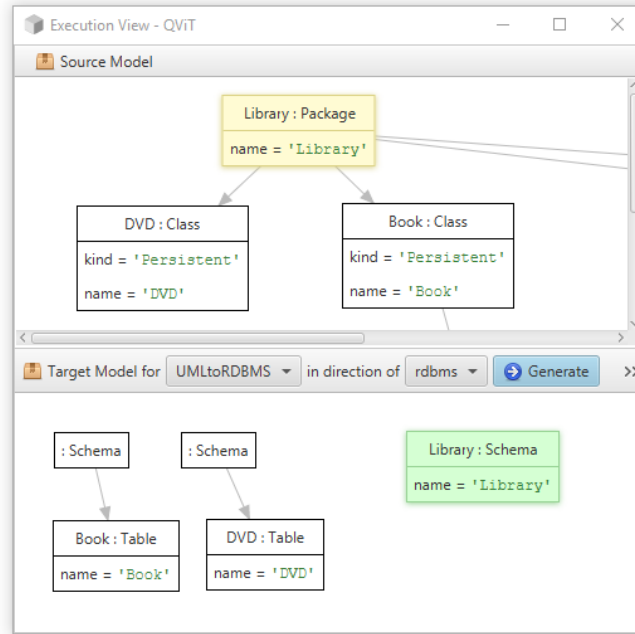


Figure 3.19: Example of the shared parent pattern

The solution to that problem is the definition of a relation call of the form `PackageToSchema(p, s)` in the `ClassToTable` relation, which would bind the `p` and `s` OTs.

3.6.7 Inward & Outward Modeling

We identify an interesting difference in the direction towards which modeling is performed to. In particular, we distinguish between the *inward* and *outward* modeling style, where the difference between them is merely the order in which diagram elements are created. The former case refers to the way modeling is done in traditional diagram editors. The user selects and drags specific elements onto the diagram canvas, and connects them together *afterwards*. In contrast, the idea of outward modeling is to let the user select already existing elements on the diagram, and to offer context-sensitive suggestions of new elements, that the user can add by simply clicking on it. This entails that diagrams are required to have at least one selectable diagram element at any time. That way, the diagrams subsequently grow from the inside to the outside. One can see a relation between the way relations grow “from-in-to-outside”, and the way users have to think when successively strengthening a selection pattern for the source model. We notice that this technique is not limited to the use of the GN, as possible suggestions for extending OTs may also be provided using auto-complete drop-downs, in the case of editors based on the TN.

3.7 Textual against Graphical Modeling

We have also analyzed the differences in the modeling process when using the TN or GN.

3.7.1 Standard Specification

In order to analyze the differences between the TN and GN, we begin with describing what the QVTr standard specifies for editors based on the GN, by the means of Figure 3.20.

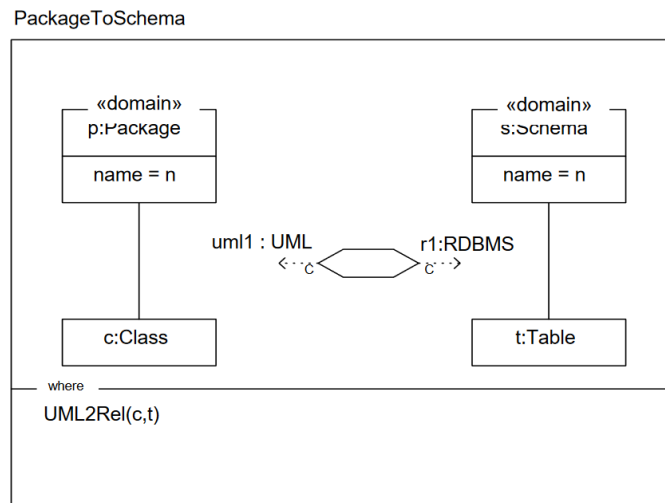


Figure 3.20: The standardized visual representation of relations using the GN

First, a new hexagon-shaped symbol is introduced to represent the interface between the related domains in each relation. This symbol also has information attached to it, that comprises (i) which models the domains are based on, (ii) the metamodels the models are based on, and (iii) the relation’s execution direction with a “c” (*i.e.* “checkonly”) indicating the source side, and “e” (*i.e.* “enforce”) identifying the respective domain to be on the target side. The standard also specifies, that the relation itself is represented in a box shape, having separate sections for the object templates, and when/where clauses. In Figure 3.20, the where clause `UML2Rel(c,t)` expresses that the `UML2Rel` relation is invoked after the successful execution of the `PackageToSchema` relation, and that it is given the object templates `c` and `t` as call parameters. The representation of an object template is also box-shaped, and divided into two parts. The upper section states the name and metamodel type, whereas statements concerning primitive properties are located in the lower one. Domains, which are a special form of OTs, are indicated as such by “«domain»” attached to the top border of their respective box. The nesting of OTs is done by connecting the respective pair of boxes with a solid, straight line.

3.7.2 Well-Formedness

Using the TN, there is potential for erroneous modeling of a transformation with regards to its well-formedness. Using appropriate bracket characters, the user has to manually ensure this type of syntactic correctness, and to properly define the scopes of transformations, relations, object templates and when/where clauses. Using the GN in editors, the effort users have to spend in this context can be automated to some extent, as the user defines relations and object templates by means of *closed units*, which are the atomic building blocks available for modeling. Concerning expressions in when/where clauses, or for primitive properties, there is also some degree of automation, that an editor can implement. For example, defined variables can also be treated as closed units that do not have to be typed out manually by the user, but instead be *selected* from a drop-down or similar interface widget.

3.7.3 Condensity & Spatial Arrangement

Using the TN enables a more condensed representation of a transformation, compared to using the GN. On the other hand however, the possibilities for spatial arrangement are limited to that of inserting empty lines and indentation. Users of editors based on the GN are given more freedom of visually configuring object templates on the diagram canvas, and thus are enabled to adapt the visualization to their needs. Due to a more coarse-grained visualization using the GN, editors may also consider to separate the respective visualizations of relations into different views. This can, for example, be achieved using a tabbed diagram canvas, where each relation is represented in a dedicated tab. Using the TN however suggests that all relations have to be contained in a single file, which is not necessarily worse, but may still affect its readability, depending on the user.

3.7.4 Reading Direction

An interesting point to consider is that of different reading directions depending on the notation used. In the case of the TN, the information is presented to the user in a vertical way, whereas the GN suggests a horizontal reading direction. We assume that this is also due to the form of the standardized hexagon symbol, and the example visualizations in the standard document.

3.7.5 Object Template Refinement

Another considerable difference lies in the way object templates are extended with further object templates. Using the TN, extending of OTs is done by nesting, whereas when using the GN, the boxes representing OTs on the diagram canvas stay on the same level, and are instead just connected to each other with solid lines.

3.7.6 Explicit against Implicit Property Mapping

Between the TN and GN, we identify a difference in suitability for the explicit mapping of primitive properties. We define an implicit mapping between primitive properties to be established, if a variable of the same name is part of the respective assignment expression. For instance in Figure 3.13, the primitive name property of the `Package` is *implicitly* mapped to the property of the same name by means of the `pn` variable. We define an explicit way of mapping to be a procedure, where users first select a set of object templates, and then get suggested a list of mappings they can *explicitly* apply. Due to the broader-grained layout of clickable elements, we identify an advantage for the GN, due to bigger touch-areas, and thus a more fault-tolerant selection process. However, it is also conceivable to implement the technique of explicit property mapping in editors based on the TN.

3.7.7 Variable Management

Finally, we identify potential for automation in the way the declaration of variables is managed in an editor. Using the TN may potentially imply more effort, since the user has to manually declare variables, decide on appropriate data types, and group them together within the scope of the respective relation, as shown in the following script.

```
1  relation PrimitiveAttributeToColumn {  
2      an, pn, cn, sqltype : String;  
3      ...  
4  }
```

Using the GN, an *on-the-fly* approach is conceivable, where the declaration of the variables and their grouping is automated, and hence not of concern of the user anymore.

A Concept for Productive Modeling with QVTr

In this chapter, we present our theoretical concept that comprises techniques and strategies to implement an increase in automation, interactivity, readability and traceability into the modeling process of QVTr.

4.1 Design Decisions

Before presenting the concrete techniques of our concept, we present the rationale behind certain design decision that constitute the foundation of our concept. These principles have been chosen based on the modeling process analysis in the previous chapter.

Graphical Syntax. We decide to build our concept on top of QVTr graphical syntax, as we consider it to be promising with regards to its adaptability to the user's needs, its suitability for explicit property mapping, automatable variable management, and a possibly enhanced readability. In addition, the GN allows us to provide the user with closed units that form the building blocks for modeling, while eliminating the need for manually ensuring well-formedness.

Outward Modelig. The usage of the outward modeling style is to some extent implied when using the GN. For instance, it helps to reduce clutter in the GUI of editors, since no toolbox, that would allow for the dragging of elements onto the diagram, is required. Instead, the object templates to add to the diagram are presented in a pop-over view *on-demand* as certain already existing OTs are selected by the user. Furthermore, the way diagrams grow “from-in-to-outside” possibly correlates with the way users think when successively strengthening a selection pattern for the source model of a specific relation.

Suggestion-driven Modeling Process. Consider the following scenario. The user wants to extend an object template, but (i) does not know the metamodel, and (ii) does not know which types are compatible, forcing the user to manually refer to the metamodel documentation. This “foggy set” scenario reveals that for executing a valid action, the user has to first be aware of the total set of possible actions and second, has to filter out those which lead to errors. Being presented with all valid object templates to choose from repeatedly during the modeling process, the user interactively learns about the underlying metamodel. Additionally, keyword filtering speeds up the time it takes for the user to find the desired type, as the search space is gradually narrowed down as the specified keyword is typed in by the user.

A modeling process where the creation/adding of new elements is done by means of offered suggestions avoids situations of “foggy sets” and reduces the users’ effort as already formulated elements only have to be *selected*, and not created by the users themselves. In addition, we consider a suggestion-driven approach to be a promising technique for the preventive reduction of errors. This is due to the fact that the total set of offered suggestions restricts the user to only these possibilities, which have already been approved to be valid by the editor.

Visual Result Validation. We lay the foundation for *short* feedback cycles in the modeling process with visualizations of the generated target and trace models, such that the time users have to spend on validating the result with the expected one is minimized. In particular, users need to be able to (i) quickly execute the transformation, (ii) view it in its graphical representation, (iii) easily trace back the generated elements by means of a visualized trace model, and (iv) validate if it conforms to the expected output. If not, users are able to change the transformation respectively, and to repeat the process. In order to achieve this process for users, a combination of the following techniques is required.

1. We minimize the time until users are able to initially execute the modeled transformation by means of automatically created execution profiles that represent the input to the underlying execution engine. In addition, it is conceivable to establish a system that detects changes to the transformation made by the user, and regenerates the transformation each time anew.
2. The visualization of the target model in a diagram-like manner entails the automatic positioning of elements on the canvas. In this context, the hierarchy of types and associations among the elements has to be taken into account. Being able to visually analyze the generated output ensures a minimized time for users to manually perform the result validation.
3. We directly integrate the visualization of the links between source and target elements defined in the trace model into the target model visualization by means of a color encoding. That is, in the scope of a specific relation, OTs that are responsible for the selection of certain elements in the source model are highlighted in a yellow-ish color, and so are the elements in the source model visualization.

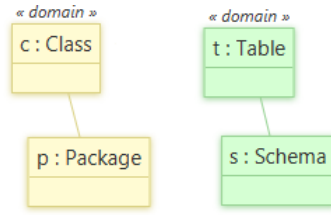


Figure 4.1: Color encoding representing the generated trace model

Similarly, OTs responsible for the generation of specific elements in the target model are, just as these elements in the target model, designated in a green-ish color.

Conventions and their Enforcement. We identify significant potential for an increase in automation of the modeling process with the enforcement of certain conventions on the user. For example, the domain patterns of a relation could potentially be automatically derived from a relation’s name of the form “PackageToSchema”. A convention that enforces the format for relation names enables the editor to automatically derive the domain patterns. Further scenarios where conventions are conceivable to be applied are default names for variables, or the capitalization of names in OTs. The challenge however is how such conventions can be subtly suggested to the users in an editor, without interfering with their intentions or getting tedious and annoying. One possibility to do so is to let an editor present suggestions to the user in that specific format. That way it is suggested to the user to follow that convention. For the scenario where the user enters a relation’s name in a differing format, an editor may derive the new format and ask the user if that new format should now be considered the new default format for suggestions in the future. That way, an editor automatically adapts to the user’s needs.

Multiple Ways of Achievement. Users learn about a subject by answering their own questions themselves. Hence, it is vital for an implementing editor to not block users by trying to monolithically enforce behaviors to achieve certain tasks on them. Instead, an editor that provides multiple ways for achieving the same goal is more likely to accommodate with the intentions and internalized behaviors that vary among users.

Fault Tolerance. In order to encourage the user’s explorative behavior during the modeling process, the strategy of fault tolerance is beneficial. A common technique to achieve this is the implementation of an undo/redo system, where each action that can be executed by the user has defined an inverse action, which reverses the effects carried out by the original action. These actions are then managed with a stack data structure to establish a FIFO ordering among the executed actions.

Usability Heuristics. We base our concept on certain heuristics as suggested by Nielsen [37]. In particular, we chose the heuristics about communicating the current status of the system, an undo/redo system, consistency, error prevention, recognition rather than recall, hot-key accelerators, a clear and minimalistic GUI, and explaining errors descriptions.

4.2 Preliminary Notation

In order to provide a concise description of the techniques that our concept comprises, we introduce the following notation.

- A project p is a tuple (\hat{M}, T) , where \hat{M} is a set of metamodels, and T is a set of transformations.
- A transformation $\tau \in T$ is defined as a function f that takes as input a set of models M , a set of keys K , a set of queries Q , a set of relations R , a source model s , an execution direction $d \in M$ and returns a target model s' .

$$\tau : f_{(M,K,Q,R,s,d)} = s'$$

- Each model $m \in M$ represents an instantiation of a metamodel $\hat{m} \in \hat{M}$.
- A metamodel \hat{m} consists of types t .
- \hat{m}_t denotes the metamodel of a type t .

We define the following utility functions.

- $\text{kind}(d) \in \{\text{checkonly}, \text{enforce}\}$ denotes the type of a domain d .
- $\text{type}(x)$ denotes the type of an object template or domain x .
- $\text{dom}(x)$ denotes the domain of an object template x , and may return x if it already is a domain.
- $\text{sup}(t)$ denotes the super type of type t , or \emptyset if there is no super type.
- $\text{isAbstract}(t)$ evaluates to \top if a type t is abstract, \perp otherwise.
- $\text{isTop}(r)$ evaluates to \top if a relation t is top, \perp otherwise.
- $\text{name}(x)$ denotes the name of an element x .
- $\text{toLower}(a)$ denotes the lower-case version of a string x .
- $\text{startsWith}(a, b)$ evaluates to \top if the string a starts with the string b .

We define the following operator overloads.

- $x[i]$ denotes the i -th character of a string x .
- \in denotes element-inclusion in sets, as well as substring relations.

- $+$ denotes numerical addition, as well as string concatenation.
- $t_1 \subseteq t_2$ denotes that the two types t_1, t_2 are compatible with each other, meaning that t_1 is more specific than t_2 . In other words, t_1 is either equal to, or is an extending type of t_2 .
- $r_1 < r_2$ denotes that relation r_1 is a pre-condition to r_2 .
- $r_1 > r_2$ denotes that relation r_1 is a post-condition to r_2 .

We define the following functions, that return sets of elements as output.

- $\text{MOD}(\tau)$ denotes the set of models in a transformation τ .
- $\text{OTS}(r)$ denotes the set of all object templates defined in a relation r .
- $\text{VAR}(r)$ denotes the set of all variables defined in a relation r .
- $\text{DOM}(r)$ denotes the ordered list of domains in a relation r (excluding primitive domains).
- $\text{PRIMDOM}(r)$ denotes the ordered list of primitive domains in a relation r (excluding normal domains).
- $\text{REL}(\tau)$ denotes the set of relations in a transformation τ .
- $\text{TYP}(\hat{m})$ denotes the set of all types in a metamodel \hat{m} .
- $\text{SUP}(t)$ denotes the set of all super types of a type t .
- $\text{REF}(t)$ denotes the set of all references to other types of a type t .
- $\text{PARM}(x)$ denotes the ordered list of parameters of a query x or a relation call x .
- $\text{WHEN}(r)$ denotes the set of all when clauses of a relation r .
- $\text{WHERE}(r)$ denotes the set of all where clauses of a relation r .

4.3 Automation

We start with techniques to increase the degree of automation in the modeling process of QVTr, such that the user's required effort is decreased.

4.3.1 Derivation of Model Declarations

The declaration of a transformation requires the user to provide a name, and a list of model declarations. Under the assumption that a set \hat{M} of metamodels is given and that the user inputs the name of a transformation τ in a format that encodes the involved metamodels, there are 2 ways of automation conceivable. (i) M can either be constructed by parsing a given transformation name, or (ii) the transformation's name can be generated from a given M . The first approach carries higher entropy since a transformation name may not only contain names of metamodels, but also terms that hint to the execution direction of τ . However, the transformation direction is undefined over the elements in M , leaving no option to encode this information into a generated transformation name. Let I be the string that the user inputs for the name of τ . Then we construct a set of models M for τ as follows.

$$\forall \hat{m} \in \hat{M} \exists m \in M : \text{name}(m) = \text{toLower}(\text{name}(\hat{m})) \iff \text{name}(\hat{m}) \in I \quad (4.1)$$

For instance, let $I = \text{"UMLToRDBMS"}$ and $\hat{M} = \{\text{UML}, \text{RDBMS}\}$. Applying construction rule 4.1 on I and \hat{M} yields the following declaration for τ .

transformation UMLToRDBMS (uml:UML, rdbms:RDBMS) { ... }

In order to not interfere with the editors widget to manage the models (*e.g.* a table or list), the described technique for automatic derivation of model declarations may be disabled as soon as that widget is not empty anymore.

Additionally, as conventions are favored in our concept, we introduce the following constraint for the ordering of model declarations. This constraint applies to model declarations whether added manually by the user, or automatically by the described derivation technique. We assume that I has an order of the form $(x_1, \dots, x_k, \star, x_{k+1}, \dots, x_n)$, where $x_i, 1 \leq i \leq n$ is a substring in I that uniquely identifies a metamodel $\hat{m} \in \hat{M}$, and \star is a substring that either identifies its LHS or its RHS to be the source-side of τ .

The reason for enforcing this order is to eliminate confusion about the two transformation directions that, on the one hand, $\text{name}(\tau)$ implies and, on the other hand, the order of models in M implies. For instance, in a transformation declaration that is defined by

transformation UMLToRDBMS (uml:UML, rdbms:RDBMS) { ... }

the directions implied by $\text{name}(\tau)$ and the ordering in M are the same. In contrast, in the declaration that is defined by

transformation UMLFromRDBMS (rdbms:RDBMS, uml:UML) { ... }

the directions are inconsistent. The definition $\star \in \{\text{“To”}, \text{“From”}\}$ is conceivable, depending on which convention over the direction of transformations an editor intends to enforce. However, we favor the usage of “To” to be the binding word, since it does not conflict with the reading direction of the rest of the QVTr script. A possibility to advise a convention’s usage to the user are auto-completing suggestions (see Section 4.4.1 for details), which follow the chosen convention.

4.3.2 Derivation of Domain Patterns

The concept of deriving model declarations from a transformation’s name is also applicable between domain templates and a relation’s name. This time, let I be the string the user inputs for a relation name. Then we construct the set D of domain templates as follows.

$$\begin{aligned} \forall \hat{m} \in \hat{M} \ \forall t \in \text{TYP}(\hat{m}) \ \exists d \in D : \\ \text{type}(d) = t \ \wedge \ \text{name}(d) = \text{toLower}(t[0]) \\ \iff \text{name}(t) \in I \end{aligned} \quad (4.2)$$

Additionally, we assume I to have an order of the form $(x_1, x_2, \dots, x_{k-1}, \star, x_{k+1}, \dots, x_n)$ where x_i is a substring in I that uniquely identifies a metamodel $\hat{m}_i \in \hat{M}$, and \star is a substring that identifies its LHS as the source-side. Then $\forall d \in D$ we derive whether $\text{kind}(d)$ is either “enforce” or “checkonly” as follows. Let $d \in D$ be an arbitrary domain template, and i be the position in I , where $\hat{m}_{\text{type}(d)} = x_i$, then we set

$$\text{kind}(d) = \begin{cases} \text{checkonly}, & \text{if } i < k \\ \text{enforce}, & \text{otherwise.} \end{cases}$$

For example, let $I = \text{“PackageToSchema”}$ and $\hat{M} = \{\text{UML}, \text{RDBMS}\}$. Applying construction rule 4.2 on I and \hat{M} yields:

```

1  relation PackageToSchema {
2      checkonly domain p : UML::Package { ... }
3      enforce domain s : RDBMS::Schema { ... }
4  }
```

In order to resolve a name collision between two domains d_1, d_2 in the scope of a relation r , we append the cardinality defined by $|\{d \mid d \in \text{DOM}(r) \setminus \{d_2\} \wedge \text{name}(d) = \text{name}(d_2)\}|$ to $\text{name}(d_2)$. For example, if $\text{name}(d_1) = \text{name}(d_2) = \text{“s”}$ represents the name collision, then we append $|\{d_1, d_2\} \setminus \{d_2\}| = |\{d_1\}| = 1$ to the name of d_2 , which then yields $\text{name}(d_2) = \text{“s1”}$.

4.3.3 Default Top Property

When creating a new relation, we set its top-property always to **true** by default. Our rationale behind this decision is that manually opting-in for the top-property blocks the user from instantly seeing the effect of the created relation in the transformation's execution. For this to achieve, the user has to manually set the relation to be top using the dedicated view in the editor. Instead, if the relation is already top by default, the transformation can be executed instantly, and effort in terms of time is reduced for users.

4.3.4 Automatic Parameter Selection

We automate the initial selection of parameters in relation calls or queries in our concept. In the workflow of creating relation calls and queries, the relation or query to call has to be specified before defining the call parameters. Call parameters can either be *new* literals (primitive types such as strings or integers) or (primitive) domains and variables that *already exist* in the relation, that the respective relation call or query is created in. Our general approach for the automatic parameter selection is as follows. Since in all cases, the data type of the parameter must match the declared type of the slot where the parameter belongs to, we automate the parameter selection by first generating a list of parameters of a compatible type in the scope of the respective relation. Then, we simply select the first element in that list as default. In the following, we distinguish between the parameter selection for relation calls, and queries.

Relation Calls

For the preselection of a call parameter, for each parameter slot in a relation call rc , where r_1 is the calling relation and r_2 the called relation, we first construct a set L_{parm} for each parameter in rc containing suitable object templates or variables from r_1 , as shown in construction rule 4.3.

$$\begin{aligned} \forall p \in \text{PARAM}(rc) \exists L_{parm} &= L_{ot} \cup L_{var}, \\ \forall d \in \text{DOM}(r_2) \exists ot \in L_{ot} &\iff ot \in \text{OTS}(r_1) \wedge \text{type}(ot) = \text{type}(d), \quad (4.3) \\ \forall d_{prim} \in \text{PRIMDOM}(r_2) \exists v \in L_{var} &\iff v \in \text{VAR}(r_1) \wedge \text{type}(v) = \text{type}(d_{prim}) \end{aligned}$$

For the preselection, the first element in each L_{parm} is automatically selected.

Queries

The procedure of preselecting parameters for queries is similar to that of relation calls. Let r be a relation that calls a query q . Then we construct the L_{parm} -lists as follows.

$$\begin{aligned} \forall p \in \text{PARAM}(q) \exists L_{parm} &= \{ot \mid ot \in \text{OTS}(r) \wedge \text{type}(ot) = \text{type}(p)\} \cup \\ &\quad \{v \mid v \in \text{VAR}(r) \wedge \text{type}(v) = \text{type}(p)\} \end{aligned} \quad (4.4)$$

4.3.5 Default Names

Some elements in QVTr require to be manually named by the user. In particular, these elements are (primitive) domains, models, variables, relations, queries, object templates and transformations. The setting of a default name can be automated for each of these. In our concept, we enforce the following conventions for default names.

Generic Names. The name of queries and primitive domains is sensitive to their purpose in the containing where clause. Hence, no other convention but that of generic default names of the form “q1” or “primdom2” is possible.

Type-driven Names. Names for object templates, domains and variables have to be defined in a way such that they hint to their underlying type. For instance, an object template of the type `UML::Package` would have the name “pkg” or even just “p”, and an integer variable would have the name “numberOfVertices”.

Metamodel-driven Names. Names for models and transformations have to be defined in a way such that they include the names of involved metamodels. Furthermore transformation names include the “To” binding word, which hints to the transformation’s execution direction.

Domain-driven Names. Finally, names for relations have to be defined in a way such that they relate to their defined domains. For example, the relation “Package-ToSchema” suggests that it contains a checkonly domain of type `UML::Package`, and an enforce domain of type `RDBMS::Schema`.

In order to prevent name collisions, we can make use of the technique mentioned in Subsection 4.3.2, which is to append the number of collisions to one of the names.

4.3.6 Typed Expression Containers

When creating a new variable in an existing expression, its data type has to be compatible with the expression’s type. A common technique for this to achieve is backtracking. For example, in the expression `prefix = cn` the `cn` variable has to be backtracked, only to find out that it holds the name of an object template of the type `UML::Class` in a string. In contrast, in our concept we introduce the notion of typed containers for expressions. The idea is that the container’s type is set preemptively, and thus dictates the type to which its contained expression evaluates to. Hence, only variables can be created and concatenated in the expression, which type matches with that of the container. That way, the complexity of an algorithm to derive a new variable’s type is in $\Theta(1)$.

4.3.7 Automatic Execution Profile Creation

An execution profile is the set of configured input parameters needed to execute a QVTr transformation. According to the transformation pipeline, the needed input parameters to generate both a target model s' and a trace model m^{traces} comprise

- a source model s ,
- a transformation τ ,
- and a domain $d \in \text{MOD}(\tau)$, which defines τ 's execution direction.

In state of the art editors for QVTr, such profiles have to be created manually by the user. In our concept, we use heuristics to suggest a default profile based on actions carried out by the user while modeling the transformation. The goal is to minimize the effort the user has with manual configuration. As this heuristically determined profile represents a suggestion merely to ease the *initial* execution of the transformation, the user is still free to adapt the profile as needed afterwards. We use the following heuristics to automatically determine all of the aforementioned required input parameters for the generation of a default execution profile.

Source Model. Editors may enforces a convention, where all source models have to be manually put into a specific directory by the user. This directory may be project-specific or be defined globally in the editor. All the source models located in this directory form a list S of suggestions to be selected for the execution profile. In order to minimize the list, those models $s \in S$ are excluded where the underlying metamodel \hat{s} is either undefined in the currently edited transformation τ , or is not on τ 's source-side. Formally, this exclusion constraint is defined as follows.

$$s \notin S \iff \nexists m \in \text{MOD}(\tau) : \hat{m} = \hat{s} \vee \\ \forall r \in \text{REL}(\tau) \nexists d \in \text{DOM}(r) : \text{kind}(d) = \text{checkonly} \wedge \text{type}(d) \in \text{TYP}(\hat{s})$$

However, since this behavior is platform-dependent, it is not considered in our concept. Instead, our used technique intends that an editor only remembers previously imported source models for each project individually, and re-loads them automatically.

Transformation. For automatically choosing the transformation τ to be executed in the profile, we heuristically select that transformation which currently has the focus in the editor. Since according to explanations in Section 4.6 to the definition and outlaying of views, the user's focus is concentrated on a tabbed diagram canvas. Hence, the entity represented in the currently opened tab in the canvas determines the focus. For editors that allow the arrangement of multiple tabs side-by-side, that tab is determining the focus in which the keyboard cursor remains. In case the focused entity is not a transformation itself, but instead a relation, we directly select this relation's transformation. Using this heuristic also means that the transformation in the execution profile is subject to frequent change as the user navigates from transformation to transformation using the sidebar.

Execution Direction. A conceivable heuristic to determine the execution direction of a transformation is to choose that one model $m \in \text{MOD}(\tau)$ having the most domains

of kind “enforce”. If there is no distinct domain of such kind, we try to derive the direction from the transformation’s name. We do this by finding a model $m \in \tau$, where $\text{name}(m)$ occurs as substring in $\text{name}(\tau)$ after the “To” binding word.

Output Directories. Regarding the output directories where to store s' and m^{traces} models to, it is convenient to let the editor manage their location on disk without requiring the user to specify any storage locations manually. Plus, an editor may offer the option to override this storage location on-demand. A conceivable usecase scenario would be when the user intends to integrate the output models into an external transformation pipeline. However, since the management of output directories is platform-dependent, we do not consider them in our concept.

4.3.8 Automatic Target Model Generation

For short feedback cycles in the process of result validation, we propose the (silent) automatic re-generation of the target model after every change made to the transformation. In order to automatically kick off the transformation’s execution after each change the user made to the transformation, an editor has to be able to constantly monitor for and detect such changes. We achieve this in our concept with the encapsulation of all modifying actions users can perform to a transformation into units of undoable actions. In this context, we distinguish between optimistic and pessimistic generation. In the first case, a running generation process may be interrupted as the user requests a new one. In the later case, the editor does not allow to fire a new generation as long as a previously requested one is still running, *e.g.* by disabling the respective interface element to do so.

4.3.9 Automatic Vertex Positioning (AVP)

To some extent, we are able to automate the positioning of elements, so called *vertices*, that represent QVTr elements on the diagram. In Figure 4.2 we introduce the boxmodel used for representing vertices in our concept.

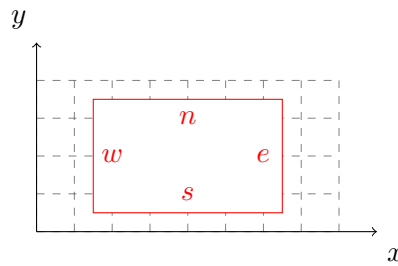


Figure 4.2: The boxmodel of AVP

As can be seen, we make use of cardinal directions, as for example $v.n$ denotes the northern, top boundary of the rectangle that represents an arbitrary vertex v .

Throughout the modeling process, such an AVP is needed in the following cases.

- The user interactively creates a new relation by defining a source and a target domain. In this case, the vertices representing the two domains have to be automatically positioned on the diagram canvas.

Algorithm 4.1: The naive recursive AVP algorithm

- 1: $d_1 \leftarrow \text{DOM}(r)[0]$, $d_2 \leftarrow \text{DOM}(r)[1]$;
 - 2: $hGap \leftarrow$ an arbitrary positive number;
 - 3: position d_1 in the upper left corner
 - 4: recursively position all direct children of d_1 below it, using BFS;
 - 5: $hex.n \leftarrow d_1.n \wedge hex.w \leftarrow (\text{right-most child of } d_1).e + hGap$;
 - 6: $\forall p_i \in \text{PRIMDOM}(r) : p_i.n \leftarrow \min(hex.n, p_{i-1}.n) + (i \text{ is odd ? } -hGap : hGap)$;
 - 7: $d_2.e \leftarrow hex.w \wedge d_2.n \leftarrow hex.n$;
 - 8: recursively position all direct children of d_2 below it, using BFS;
 - 9: position all $d_i \in \text{DOM}(r), i > 2$ similarly beneath the south-most vertex so far;
-

Algorithm 4.1 outlines the naive approach to the positioning of vertices that represent domains and their descendant OTs. It begins with the positioning of the first domain. Then, all children are recursively positioned using breadth-first-search (BFS), *i.e.* *layer-by-layer* having the anchor to $d_1.w$. Then the hexagon symbol, abbreviated with *hex*, is positioned right next to the right-most child of d_1 , and vertically on the same level of d_1 . Optionally, primitive domains are positioned underneath the *hex* vertex in a “wobbly” manner, as odd elements are slightly positioned westwards, and eastwards otherwise. Finally, d_2 is positioned right next to *hex*, and the same recursive BFS procedure is used for positioning the children of d_2 . An inherent caveat to the algorithm is the fact that it is globally uninformed, and thus vertex overlapping is possible. If there are more than two domains, the remaining ones are positioned similarly starting from the most south vertex so far.

This is opposed to a comparably moderate time complexity of $O(|V| + |E|)$. Figure 4.3 illustrates the positioning the algorithm outputs for the **AssocToFKey** relation.

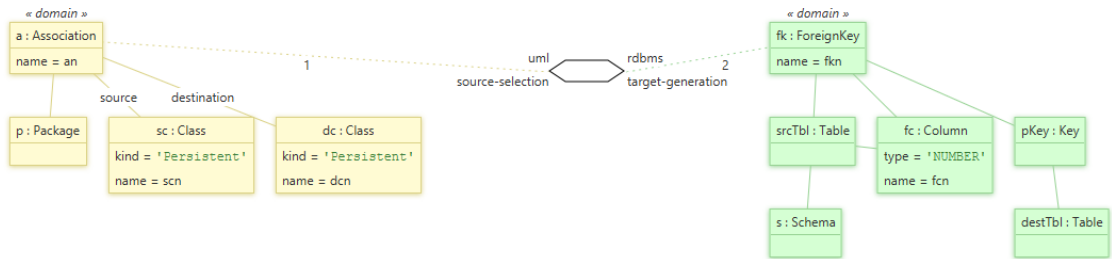


Figure 4.3: Example of the general, naive AVP algorithm

- After executing a so called *One-Click-Extension* (OCEs) (see Section 4.4.6), a new vertex representing the new object template is added to the diagram canvas, and has to be automatically positioned. For this purpose, we use an algorithm similar

to line 6 in Algorithm 4.1, such that new OTs are added in a vertical, “wobbly” manner beneath its parent OT, as shown in Figure 4.4.

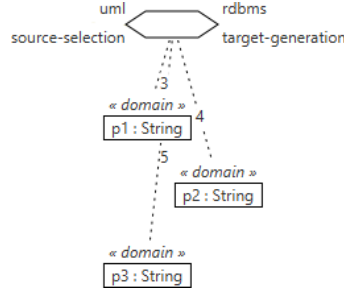


Figure 4.4: Example of the AVP algorithm for OCEs

- The vertices in a dependency graph visualization (see Section 4.7) have to be positioned automatically. For this purpose, we use Algorithm 4.2.

Algorithm 4.2: AVP algorithm for dependency graph visualizations

- 1: $rs \leftarrow REL(\tau)$
 - 2: sort rs by hierarchy in descending order;
/ top relation > relation
more relation calls in when clauses > less relation calls in when clauses */*
 - 3: position $rs[0]$ in the upper left corner;
 - 4: position all $r_i \in rs \setminus \{rs[0]\}$ where $isTop(r_i) = \top$;
/ those with a “when” relation call in a row below $rs[0]$,
otherwise right next to each other on same row as $rs[0]$ */*
 - 5: position the remaining non-top relations row-by-row below the lowest row so far;
/ the less relation calls a relation has, the more it is positioned to the left */*
-

The algorithm accounts for the call hierarchy of a transformation, as it favors top relations to be above non-top relations, and heuristically positions relations with less relation calls to the left side of the diagram. Figure 4.5 depicts the positioning of the algorithm for our SimpleUMLToSimpleRDBMS transformation.

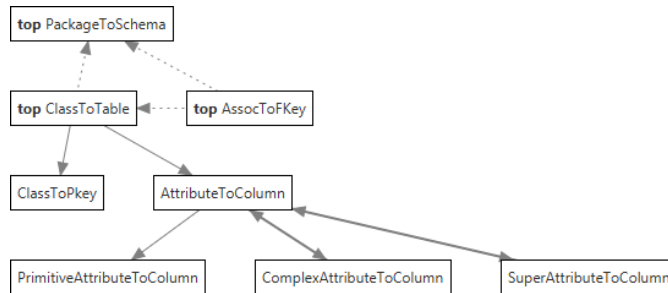


Figure 4.5: Example of the AVP algorithm for dependency graph visualizations

- The vertices that represent the objects in the source/target model visualizations have to be automatically positioned. For this purpose, we use Algorithm 4.3, which, starting from a root element, recursively positions all child elements of the respective parent element in a single row, sorted by their names in ascending order.

Algorithm 4.3: AVP algorithm for visualizations of source/target models

- 1: $v \leftarrow$ the root element;
 - 2: $L = \{c \mid c \text{ is a direct child of } v \text{ and has not been positioned yet}\}$;
 - 3: sort L by $\text{name}(\star)$ in ascending order;
 - 4: position all elements $c \in L$ in a single row below v ;
 - 5: mark all elements $c \in L$ to be positioned;
 - 6: repeat lines 1 to 4 recursively for each element $c \in L$;
-

4.3.10 Automatic Variable Management

Due to our design decision of using QVTr's GN for the modeling process, we are able to automate the declaration, grouping and purging of variables in the scope of a specific relation. We achieve this automation by defining a list L_{var} for each relation, including variables v that have a name and data type associated with them. For the automatic declaration of the variables in the TN, an algorithm is able to simply loop over the variables and print them out line-by-line using a format such as $\text{name}(v) : \text{type}(v)$. In order to minimize the verbosity of this generated list, an algorithm may group together the variables by type using the format defined by $\text{name}(v_1), \text{name}(v_2), \dots, \text{name}(v_n) : \text{type}(v_1)$, where $\text{type}(v_1) = \text{type}(v_2) = \dots = \text{type}(v_n)$. Finally, those $v \in L_{var}$ may eventually be purged from the list, if they do not have any occurrences in the respective relation anymore. This is done by simply searching for occurrences in the object template hierarchies established by the domains of the relation, including expressions of primitive properties, as well as occurrences in when/where clauses.

4.4 Preventive Interactivity

We continue proposing our concept with techniques that interactively provide the users with suggestions to choose from, while restricting them at the same for the purpose of error *prevention*. Because the suggestions have been generated by the editor, they comprise the set of available user actions, and are approved in the sense that they do not lead to errors when chosen by the user.

4.4.1 Auto-Completion

Whenever the user is asked to name certain elements in QVTr, an editor can not only set a default name, but also offer non-committal alternatives to choose from. In particular, such suggestions may be provided when naming (i) transformations, or (ii) relations. The technique for auto-completion is to first generate an extensive list of suggestions that

successively shrinks as the user types in parts of the name. That way, an editor responds to the user in an interactive way, while defensively enforcing name conventions. Let I be the string that the user inputs and S the set of name suggestions. Initially, when I equals the empty string, we construct S as follows.

1. $S = \{\text{name}(\hat{m}_1) + \text{"To"} + \text{name}(\hat{m}_2) \mid \hat{m}_1, \hat{m}_2 \in \hat{M} \wedge \hat{m}_1 \neq \hat{m}_2\}$ which is the cartesian product of \hat{M} with itself. For example, let $\hat{M} = \{\text{UML}, \text{RDBMS}\}$, then we generate $S = \{\text{"UMLToRDBMS"}, \text{"RDBMSToUML"}\}$.
2. $S = \{\text{name}(c) \mid c \in \text{TYP}(\hat{m}) \wedge m \in \text{MOD}(t)\}$. The user may either instantly choose one $s \in S$ or manually type in I .

In the second case, a scenario may occur where I is a substring of one $s \in S$. For example, if $I = \text{"U"}$ and $S = \{\text{"UML"}, \text{"RDBMS"}\}$. Given this condition, we can set $S = S \cup \{s \mid \neg \text{startsWith}(s, I)\}$. This yields only such suggestions that complete the user's input string. As the user types further, the case where $\exists s \in S : s = I$ may eventually arrive. For instance, when $I = \text{"UML"}$ and $S = \{\text{"UML"}, \text{"RDBMS"}\}$. In this case, we can set $S = \{s_1 + \text{"To"} + s_2 \mid s_1 \in S \wedge s_2 \in S \wedge s_1 = I\}$. Notice that S covers the special case where $s_2 = s_1$ for endogenous transformations like "GeometryToGeometry".

4.4.2 Naming Conventions

In Section 4.3.5 we already discussed which elements in QVTr require the user to provide a name for, and how a default name could be automatically set. However, the user may still adapt this default name to a different format or change it completely. In order to not let the user alone in either of these scenarios, an editor may offer alternative names in a dropdown list for the user to choose from. For example, consider the variable name for the primitive `name` property in an object template of type `UML::Package`. Conceivable names are "pn", "pkgName", "packageName", "pname". All of these names correspond to a specific format. One way of enforcing a format is to assume a specific format by default, and when the user selects a name of a different format, to ask the user in a subtle pop-over if that format should be the new default. As the editor interactively adapts itself to the user's naming convention, it is also able to set default names of the desired format in the future. That way, the recurring and tedious task of adapting names to a specific convention is prevented which reduces the modeling effort. An editor may freely choose the scope of these default formats. For example, a naming convention may differ across projects, but could also be set globally in the editor.

4.4.3 Common Property Mappings (CPM)

Since QVTr is a declarative modeling language, the mapping of primitive properties across object templates is done implicitly by using variables. For example, to map the name of an object template of type `UML::Package` to another of type `RDBMS::Schema`, both object templates would hold a primitive property of the form `name = pn`. For this

to achieve, a user has to manually create the same primitive property twice. What can be done instead is to first let the user select a set of object templates, to find primitive properties that the object templates in the set have in common, and to display them as applicable suggestions in a “one-click-to-apply-fashion”.

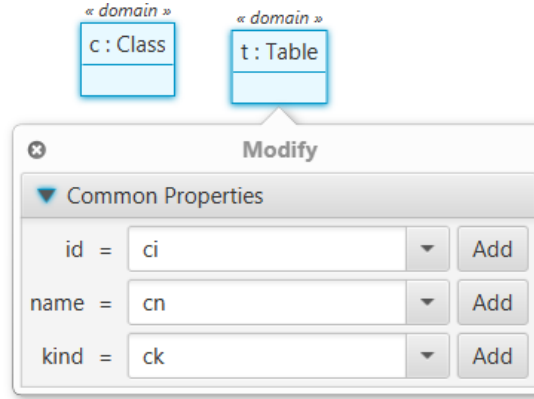


Figure 4.6: Example of a common property mapping

Figure 4.6 illustrates an example of such suggestions, in which the two OTs `c:Class` and `t:Table` have been simultaneously selected. Note that the user is still free to edit the variable’s name before applying the mapping. We construct the set P of common properties between two distinct object templates o_1 and o_2 as follows.

$$P = \{p_1 \mid p_1 \in o_1 \wedge p_2 \in o_2 \wedge \text{name}(p_1) = \text{name}(p_2) \wedge \text{type}(p_1) = \text{type}(p_2)\}$$

Note that o_1 and o_2 may also contain inherited properties from their super types in the metamodel.

4.4.4 Prediction of new Relations

An interesting way of minimizing the modeling effort is to predict and suggest the base structure of whole relations. This not only has the advantage of sparing the user to model the raw structure of relations manually, but it also aids the user to decide which types should be related to which types. The notion behind the prediction algorithm is to statically analyze the involved metamodels of a transformation, and to compute the similarities of all possible pairs of types. Thus, this strategy of static metamodel analysis is only applicable for exogenous transformations. Using a certain threshold value for the similarity, the algorithm returns only those pairs of types that would be meaningful to be related to in a new relation. We distinguish between static and dynamic similarity measures. Static ones only consider the underlying metamodels, whereas a dynamic measure also takes an already modeled transformation into account. The following list describes the similarity measures used in our concept.

Hierarchy Similarity. We define the hierarchy similarity σ^{hry} of two types t_1, t_2 to be the normalized edit distance (*e.g.* the Levenshtein distance [63]) of their hierarchy strings. The Levenshtein distance between the first i characters of a string s_1 and the first j characters of a string s_2 is defined as follows.

$$\text{lev}_{(i,j)}(s_1, s_2) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0 \\ \min \begin{cases} \text{lev}_{(i-1,j)}(s_1, s_2) + 1 \\ \text{lev}_{(i,j-1)}(s_1, s_2) + 1 \\ \text{lev}_{(i-1,j-1)}(s_1, s_2) + 1_{(s_1[i] \neq s_2[j])} \end{cases} & \text{otherwise} \end{cases} \quad (4.5)$$

Examples:

$$\begin{aligned} \text{lev}_{(4,6)}(\text{"Class"}, \text{"Column"}) &= 4 \\ \text{lev}_{(4,4)}(\text{"Class"}, \text{"Clazz"}) &= 2 \\ \text{lev}_{(7,6)}(\text{"Package"}, \text{"Schema"}) &= 6 \\ \text{lev}_{(1,1)}(\text{"x"}, \text{"x"}) &= 0 \end{aligned}$$

The hierarchy string s^{hry} of a type $t \in \text{TYP}(\hat{m})$ is constructed as follows.

$$s^{hry}(t) = \begin{cases} \text{""} & \text{if } t \text{ is } \emptyset \\ \text{"e"} + s^{hry}(\text{super}(t)) & \text{if } \text{super}(t) \text{ is abstract} \\ \text{"n"} + s^{hry}(\text{super}(t)) & \text{otherwise} \end{cases} \quad (4.6)$$

Examples:

$$\begin{aligned} s^{hry}(\text{RDBMS::Schema}) &= \text{"e"} \\ s^{hry}(\text{UML::PrimitiveDataType}) &= s^{hry}(\text{UML::Class}) = \text{"eee"} \\ s^{hry}(\text{UML::Classifier}) &= \text{"ee"} \\ s^{hry}(\text{UML::ModelElement}) &= \text{""} \end{aligned}$$

Finally, we define the hierarchy similarity σ^{hry} between two types t_1, t_2 as follows.

$$\sigma_{(t_1, t_2)}^{hry} = 1 - \frac{\text{lev}_{|n_1|, |n_2|}(n_1, n_2)}{\max(|n_1|, |n_2|)} \quad (4.7)$$

$$\begin{aligned} n_1 &= \text{name}(t_1) \\ n_2 &= \text{name}(t_2) \end{aligned}$$

| | t_1 | t_2 | σ^{hry} |
|-----------|------------------|---------------------|----------------|
| Examples: | UML::Class | UML::Class | 1.0 |
| | UML::Package | RDBMS::Schema | 1.0 |
| | UML::Association | RDBMS::Key | $1/2$ |
| | UML::Class | RDBMS::Column | $1/3$ |
| | UML::Attribute | RDBMS::ModelElement | 0 |

References Similarity. We define the references similarity σ^{ref} of two types t_1, t_2 as the sum of three different similarities: (i) number of references, (ii) number of references that are containments, and (iii) occurrences of multiplicities. Let R_1 be the set of references to other types in t_1 and assume R_2 likewise, then σ^{ref} is defined as follows.

$$\sigma^{ref} = \frac{\text{normalize}(|R_1|, |R_2|) + \text{normalize}(|R_{1(c)}|, |R_{2(c)}|) + \sigma^{mult}}{3}, \quad (4.8)$$

$$\sigma^{mult} = \begin{cases} 0 & \text{if } |D_{max}^{mult}| = 0 \\ \frac{\sum_{i=0}^{|D_{max}^{mult}|} \text{normalize}(D_{max}^{mult}(i), D_{min}^{mult}(i))}{|D_{max}^{mult}|} & \text{otherwise} \end{cases},$$

$$D_{max}^{mult} = \begin{cases} D_1^{mult} & \text{if } |D_1^{mult}| > |D_2^{mult}| \\ D_2^{mult} & \text{otherwise} \end{cases},$$

$$D^{mult}[k \rightarrow v](x)^1 = \begin{cases} v & \text{if } x = k \\ 0 & \text{otherwise} \end{cases},$$

$$\text{normalize}(x, y) = \begin{cases} 1 & \text{if } x = 0 \wedge y = 0 \\ 1 - \frac{\text{abs}(x - y)}{\max(x, y)} & \text{otherwise} \end{cases},$$

$$R_{(c)} \subset R = \{r \mid r \in R, \text{isContainment}(r)\}$$

| | t_1 | t_2 | σ^{ref} |
|-----------|----------------|---------------------|----------------|
| Examples: | UML::Class | UML::Class | 1.0 |
| | UML::Package | RDBMS::Schema | 1.0 |
| | UML::Class | RDBMS::Column | $\frac{2}{3}$ |
| | UML::Attribute | RDBMS::ModelElement | $\frac{1}{3}$ |

Abstractness Similarity. We define the abstractness similarity σ^{abt} of two types t_1, t_2 as follows.

$$\sigma_{(t_1, t_2)}^{abt} = \begin{cases} 1 & \text{if } \text{isAbstract}(t_1) = \text{isAbstract}(t_2) \\ 0 & \text{otherwise} \end{cases}$$

¹ D^{mult} is a dictionary, which assigns a value v to each unique key k .

| | t_1 | t_2 | σ^{abt} |
|-----------|-----------------|---------------------|----------------|
| Examples: | UML::Class | RDBMS::Column | 1.0 |
| | UML::Package | RDBMS::Schema | 1.0 |
| | UML::Class | UML::Class | 1.0 |
| | UML::Attribute | RDBMS::ModelElement | 0 |
| | UML::Classifier | RDBMS::ModelElement | 0 |

Primitive Properties Similarity. We define the primitive properties similarity σ^{prop} of two types t_1, t_2 as follows. Let P_1 be the set of primitive properties of type t_1 and assume P_2 likewise. Note that P_1, P_2 also include inherited properties.

$$\sigma^{prop} = 1 - \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|} \quad (4.9)$$

| | t_1 | t_2 | σ^{prop} |
|-----------|----------------|---------------------|-----------------|
| Examples: | UML::Package | RDBMS::Schema | 1.0 |
| | UML::Attribute | RDBMS::ModelElement | 1.0 |
| | UML::Class | UML::Class | 1.0 |
| | UML::Class | RDBMS::Column | $3/4$ |

One-Click-Extensions Similarity. All of the aforementioned similarity measures are static, since they only consider properties of the underlying metamodels of t_1 and t_2 . In contrast, a dynamic metric would also take a transformation τ into account yielding different similarities for two types, depending on the contained relations in τ . The idea is to assign a higher similarity to two types t_1, t_2 , if there already exists a relation with types t'_1, t'_2 as domains, where t'_1, t'_2 are direct neighbors of t_1, t_2 . For instance, if τ already contains a relation **ClassToTable**, then the creation of a **PackageToSchema** relation is considered to have a higher probability, compared to the scenario where **ClassToTable** does not exist. Formally, we define σ^{oce} as follows.

$$\sigma_{(t_1, t_2)}^{oce} = \begin{cases} 1 & \text{if } \exists r \in \text{REL}(\tau) \exists d_1, d_2 \in \text{DOM}(r) : \text{areMapping}(d_1, d_2) \\ 0 & \text{otherwise} \end{cases} \quad (4.10)$$

$$\begin{aligned} \text{areMapping}(d_1, d_2) = & \text{isOCEof}(\text{type}(d_1), t_1) \\ & \wedge \text{isOCEof}(\text{type}(d_2), t_2) \\ & \wedge \text{kind}(d_1) = \text{checkonly} \\ & \wedge \text{kind}(d_2) = \text{enforce} \end{aligned} \quad (4.11)$$

$$\text{isOCEof}(t_1, t_2) = \begin{cases} \top & \text{if } t_1 \in \text{OCE}(t_2) \\ \perp & \text{otherwise} \end{cases} \quad (4.12)$$

$$\text{OCE}(t) = \{u \mid u \in \text{REF}(t) \wedge \neg \text{isAbstract}(u)\} \cup \text{EXT}(t, \hat{m}_t) \quad (4.13)$$

$$\text{EXT}(t, \hat{m}) = \{t_{\hat{m}} \in \text{TYP}(\hat{m}) \mid t \in \text{SUP}(t_{\hat{m}})\} \quad (4.14)$$

Total Weighted Similarity. Since certain measures are more expressive than others, we suggest the usage of a total weighted similarity, that is computed as follows.

$$\sigma_{(t_1, t_2)} = \sum \sigma_{(t_1, t_2)}^i \cdot \kappa \quad (4.15)$$

For the selection of feasible κ values we take into account the following considerations. Since $\forall i \in \{hry, ref, abt, prop, oce\}$ it holds that $\sum \kappa^i = 1.0$, *i.e.* for a balanced weighting we have $1/5$ for each κ . However, since σ^{ref} combines 3 different similarity metrics, we consider its entropy to be 3 times higher than that of σ^{hry} . Hence, $\kappa^{ref} = 3 \cdot \kappa^{hry}$ which yields $\kappa^{hry} = 2/5 \cdot 25\% = 1/10$ and $\kappa^{ref} = 2/5 \cdot 75\% = 3/10$. This leaves a total of $3/5$ to distribute over the remaining values. For the remaining two static metrics we define $\kappa^{abt} = \kappa^{prop}$ since we do not consider either one to be of more importance than the other. Since σ^{oce} is significantly depending on the actual τ , we recommend a lower impact on the overall similarity compared to σ^{abt} and σ^{prop} . For instance, a proportion of 1:7 ensures that σ^{oce} is only slightly steering the similarity to one of two directions rather than making up for a large part of it. The proportion of 1:7 yields $\kappa^{abt} = \kappa^{prop} = 7 \cdot \kappa^{oce}$ and taking into account the aforementioned calculations $\kappa^{abt} = \kappa^{prop} = 0.28$ and $\kappa^{oce} = 0.04$.

In summary, we suggest to set $\kappa^{ref} = 3/10$,

$$\kappa^{hry} = 1/10,$$

$$\kappa^{abt} = \kappa^{prop} = 0.28,$$

$$\kappa^{oce} = 0.04.$$

Examples:

| t_1 | t_2 | σ^{hry} | σ^{ref} | σ^{abt} | σ^p | σ^{oce} | σ |
|-------------------|---------------|----------------|----------------|----------------|------------|----------------|----------|
| UML::Package | RDBMS::Schema | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 96% |
| UML::Class | RDBMS::Table | $1/3$ | 0.7 | 1.0 | 1.0 | 0.0 | 80.4% |
| UML::Attribute | RDBMS::Column | 1.0 | 0.5 | 1.0 | $3/4$ | 0.0 | 75.7% |
| UML::ModelElement | RDBMS::Table | 0.0 | $1/3$ | 0.0 | 1.0 | 0.0 | 38% |

Under the assumption that a relation **ClassToTable** already exists, $\sigma^{oce} = 1$ for a relation **PackageToSchema** resulting in a total similarity of 1.0 which is an increase of 100%. Under the assumption that a relation **PackageToSchema** already exists, $\sigma^{oce} = 1$ for a relation **ClassToTable** resulting in a total similarity of 0.844 which is an increase of roughly 5%.

| t_1 | t_2 | σ^{hry} | σ^{ref} | σ^{abt} | σ^p | σ^{oce} | σ |
|--------------|---------------|----------------|----------------|----------------|------------|----------------|----------|
| UML::Package | RDBMS::Schema | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 84.4% |

4.4.5 Concatenations

In order to add interactivity to the process of specifying expressions (RHS) with their assignments (LHS), we suggest to view expressions as concatenations that are located in a typed expression container. That is, the user first specifies the type of the container and then defines the variable assignment and formulates the expression. That way, an editor is able to interactively propose suggestions that conform to the chosen type. Expressions are formulated for primitive properties, variable assignments and in when/where clauses. We distinguish the following cases to specify the type of the container. (i) The expression dictates the type of the concatenation, *i.e.* the user first specifies the expression on the RHS, and afterwards the variable to assign the expression's result to on the LHS. Since in this case, the data type of the expression's result defines the variable's type, an editor is able to suggest existing variables to the user that are of that specific type and in the scope of the relation. (ii) The assignment dictates the type of the concatenation, *i.e.* the user first specifies the variable (*i.e.* the name and the data type) on the RHS, and afterwards the expression on the LHS. In this case, when suggesting the user with possible options to concatenate with the expression, an editor is able to minimize the set of these suggestions to those that have a type that is compatible with the variable's type.

4.4.6 Derivation of extending Object Templates aka One-Click-Extensions (OCEs)

A recurring task when modeling relations is the specification of object templates. In QVTr, object templates hold primitive properties and other object templates. Since the type of an object template defines which primitive properties or other object templates are allowed to be added, a high level of metamodel understanding is required to avoid type incompatibility errors. We introduce the concept of so called *One-Click-Extensions* (OCEs) for the extension of existing object templates that, on the one hand, help to prevent type incompatibilities, and, on the other hand, provide an interactive way of gaining a deeper metamodel understanding. The underlying idea of OCEs is to enforce outward modeling by (i) letting the user select an object template, and (ii) to suggest a list of possible object templates to attach to the selection, which are applicable in a one-click fashion. This is significantly contrary to the way in which elements are created and connected to each other in traditional diagram editors. There, the modeling is practiced inwards as elements are dragged on a canvas first, and linked together afterwards. Figure 4.7 illustrates the intended workflow of using OCEs. We now define how suggestions for OCEs are constructed. Let ot be an arbitrary object template that the user selects. Then, we construct the set E of OCEs using construction rules 4.13 and 4.14, with the only difference that the $REF(type(ot))$ function in the former rule has to be replaced with a variation, such that it also returns all references of all super types of ot along the hierarchy. The user benefits from OCEs since they allow for a quick and safe way of exploring the types defined in a specific metamodel. However, one limitation is that the user only peeks ahead to applicable types *one layer at a time* and not further. For the case, where a whole path of object templates has to be created before reaching the

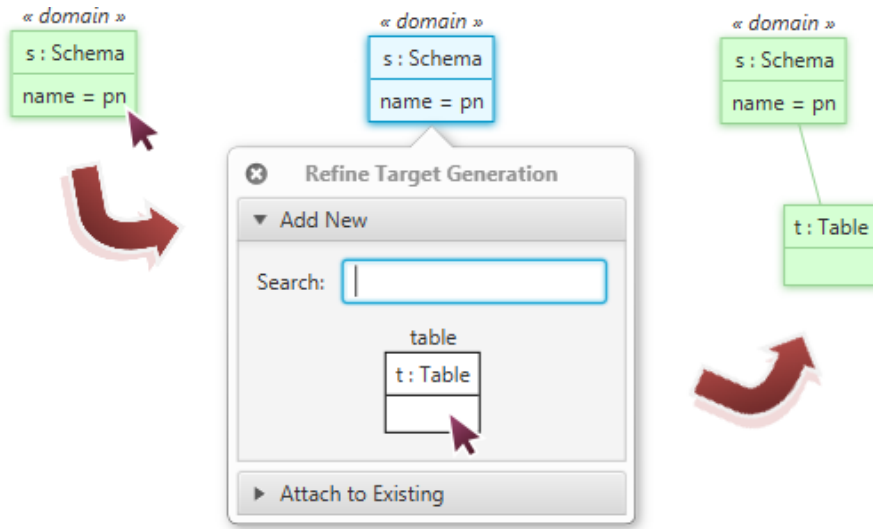


Figure 4.7: The workflow of how One-Click-Extensions (OCE) are used

desired type, OCEs are somewhat limiting. For such usecases, we suggest the usage of “path-pickers” instead. The idea is to not only allow the user to select a single object template to add to the relation, but a whole path of object templates at once. However, a definite formal description of such a path-picker is outside of this thesis’ scope.

4.4.7 References to other Object Templates

OTs can be referenced using their names as variables. In the following QVTr script, the object template *cl* is referenced by the object template *k*.

```

1  enforce domain rdbms t : RDBMS::Table {
2      column = cl : RDBMS::Column {
3          name = cn + 'tid' ,
4          type = 'NUMBER'
5      },
6      key = k : RDBMS::Key {
7          column = cl
8      }
9  };

```

When creating a reference to another object template, an editor may interactively suggest other OTs to reference to. Using either of GN or TS, we do this in our concept by graying out all OTs of the relation with incompatible types, as shown in Figure 4.8. That way, the user is visually directed to those OTs that can be validly referenced to, and errors are being prevented.

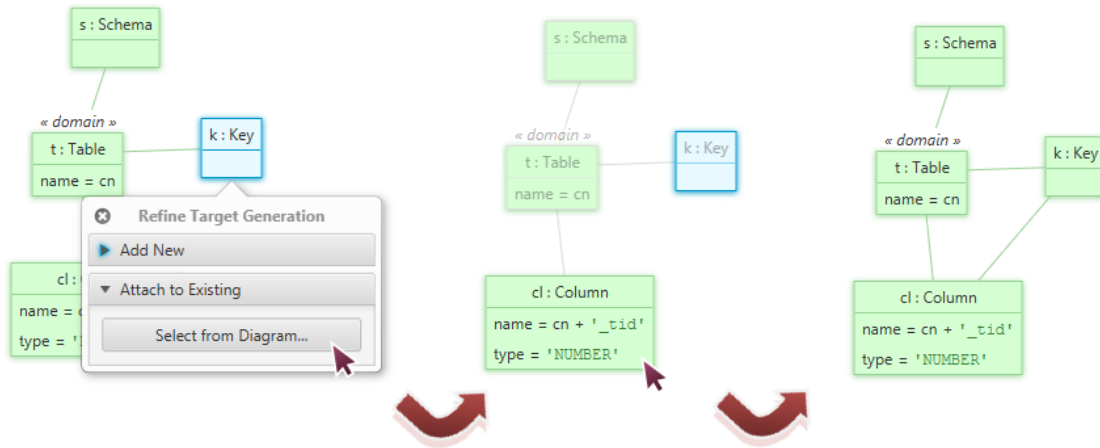


Figure 4.8: The workflow of referencing OTs to other OTs

Since the viewport of the diagram in an editor may not be big enough to display all possible targets to select for an OT referencing, we suggest the usage of dedicated visual indicators, that point in the direction of these selectable OTs on the canvas. For instance consider Figure 4.9, in which an icon, placed along the diagram viewport's boundary, indicates that two more OTs to select can be found in a south-east direction.

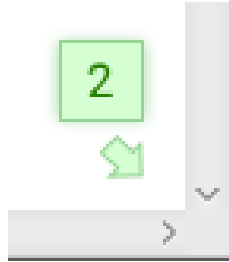


Figure 4.9: Visual indicator that points to OTs, which are out-of-sight in the viewport

4.5 Corrective Interactivity

Unlike the techniques described so far, the techniques introduced in this section aim at guiding the user interactively to solve *already occurred* problems. We distinguish two kinds of problems, according to their level of severity. *Errors* are crucial problems, which require immediate attention since they prevent a transformation from being executed. *Warnings* however are problems that may potentially result in unexpected results when executing the transformation, but they per-se do not stop the transformation from being executed. Each error and warning is presented to the user along with textual descriptions of the problem itself, the involved transformations and relations, and of the suggested solution to be applied in a one-click fashion.

4.5.1 Errors

We consider *errors* to be violations to the *syntactical correctness* of a transformation.

Sanity Checks

We define the following rules in order to check for general sanity of the transformation.

| Sanity Rule | Problem Description & Suggested Solution |
|---|---|
| $\forall p(\hat{M}, T) : \hat{M} > 0$ | Each project has at least one metamodel. <i>Fix:</i> Import at least one metamodel into p . |
| $\forall p(\hat{M}, T) : T > 0$ | Each project has at least one transformation. <i>Fix:</i> Create a new transformation in p . |
| $\forall \tau \in T : \text{MOD}(\tau) > 0$ | Each transformation has at least one model. <i>Fix:</i> Define at least one model declaration in τ . |
| $\forall \tau \in T : \text{REL}(\tau) > 0$ | Each transformation has at least one relation. <i>Fix:</i> Create or predict a new relation in p . |
| $\forall \tau \in T \exists r \in \text{REL}(\tau) : \text{isTop}(r)$ | Each transformation has at least one top relation. <i>Fix:</i> Manually set an arbitrary $r \in \text{REL}(\tau)$ to be top, or let the editor automatically suggest them. |
| $\forall r \in \text{REL}(\tau) : \text{DOM}(r) > 0$ | Each relation has at least one domain. <i>Fix:</i> Define at least one domain pattern in r . |

Invalid Relation Calls and Queries

A relation call to a non-top relation is only allowed in the *where* clause of an arbitrary relation. That is, a non-top relation can only be a *post*-condition to another relation. Also, queries are only allowed to be called from where clauses. Similarly, a relation call to a top-relation is only allowed in the *when* clause of an arbitrary relation. That is, a top relation can only be a *pre*-condition to another relation. The suggested solution to these problems is to either (i) toggle the top-property of the called relation, or to (ii) delete the respective relation call. In the case of queries, only the deletion can be offered to the user.

Invalid Relation Call Parameters

As described before, the concept of a relation call in QVTr is similar to that of a function call in traditional procedural programming languages. Hence, it becomes clear that the type of each OT defined as a parameter in a relation call must be compatible with the type of the domain, that the OT corresponds to. In addition, the number of call parameters must *exactly* match the number of domains of the called relation. Formally, let r_1 be the calling relation and r_2 the called relation. Assume that the relation call also has a list of parameters P of the form (p_1, p_2, \dots, p_n) where $\forall p_i \in \text{OTE}(r_1)$ and $n = |\text{DOM}(r_2)|$. For the parameter list to be valid, it must hold that $\forall i, 1 \leq i \leq n$ it holds that $\text{type}(p_i) \subseteq \text{type}(d_i)$ where d_i denotes the i -th domain in $\text{DOM}(r_2)$. The

suggested solutions to the problem is (i) the deletion of the relation call, or (ii) to let the editor automatically chose the parameters again by means of automatic parameter selection described in Section 4.3.4.

Invalid Names

As described before, names that contain a reserved keyword are suggested to be prepended with an underscore character “_”, and also be enquoted with single quotation marks. Our concept however suggests a slightly stricter set of rules for names, which is similar to those of state of the art programming languages such as Java. Let I be a trimmed² string, representing a name. Then, I disqualifies as a valid name in the following cases.

- I equals the empty string.
- I contains a whitespace character, detected by the regular expression `\\s+`.
- The first character of I represents a numeric value. In particular, a character is considered numeric if it matches with the regular expression `[-+]?\d*\.\d+`.
- I contains a special character with the exception of the underscore character “_”. In particular, I contains special characters if it matches with the regular expression `[^a-zA-Z0-9_]`.
- I equals a reserved keyword, as defined in Section 3.3.1.

The procedure to convert an invalid name to a valid name directly follows from the stated constraints. First, all whitespace characters are replaced with an underscore character. Then, if the name starts with a numeric value, that number is replaced with another underscore character. Finally, an underscore character is prepended to each occurrence of a reserved keyword and a special character in the name.

4.5.2 Unbound Enforce Variables

Variables that appear in the context of an enforce domain are required to have a value assigned/bound to it. Variable assignments can either occur in object templates that are children of a checkonly domain, or in a where clause. The transformation can't be executed as long as variables are unbound within an enforce domain. For example, consider the following relation, in which the `pn` variable is unbound.

Possible solutions to resolve this issue are to either bind the variable in the `p` domain with

```
checkonly domain uml p : UML::Package { name = pn }
```

²no leading or trailing whitespace characters

```
1  top relation PackageToSchema {
2    pn : String;
3    checkonly domain uml p : UML::Package {};
4    enforce domain rdbms s : RDBMS::Schema { name = pn };
5  }
```

or as a where clause like presented in the following example.

```
1  top relation PackageToSchema {
2    pn : String;
3    ...
4    enforce domain rdbms s : RDBMS::Schema { name = pn };
5    where { name = 'schemaName' }
6  }
```

4.5.3 Warnings

Similar to errors that refer to violations of the syntactic correctness of transformations, *warnings* refer to possible infringements of their *semantic* correctness. Unlike errors, warnings do not prevent a transformation from being executed. However, the generated output may not conform to the expected one. Warnings have the goal to only steer the user's attention to possible bugs in the transformation, but can still be ignored as such.

Unbound Containment Relations

We have a containment between two types, if the type t_1 of an object template ot has a containment relation to the type t_2 of its parent in the underlying metamodel. That is, instances of type t_2 are contained by instances of type t_1 . Furthermore, a containment is unbound, as long as ot is not used as a parameter in a relation call in a when clause in the respective relation r_1 . In the following example, p and s are both unbound as their types `Uml::Package` and `Rdbms::Schema` contain the types `Uml::Class` and `Rdbms::Package` of their parent OTs.

```
1  top relation ClassToTable {
2    checkonly domain uml c : UML::Class {
3      namespace = p : UML::Package {}
4    };
5    enforce domain rdbms t : RDBMS::Table {
6      schema = s : RDBMS::Schema{}
7    };
8  }
```

The suggested solution to resolve such unbound containments is to offer the creation of a relation call to another relation r_2 that takes as input all of the t_2 types. That is, under the assumption that r_1 contains two unbound containments, one in a checkonly and the other one in an enforce domain, another relation r_2 qualifies as a pre-condition to r_1 (and thus can be called in a where clause) if the following constraints hold. Let $(ot_1, ot_2, \dots, ot_{k-1}, ot_k, ot_{k+1}, \dots, ot_n)$ be the list of object templates involved in unbound containments in r_1 , where $\forall ot_i, 1 \leq i \leq k$ it holds that $\text{kind}(\text{dom}(ot_i)) = \text{checkonly}$, and $\forall ot_j, k+1 \leq j \leq n$ it holds that $\text{kind}(\text{dom}(ot_j)) = \text{enforce}$. Then, an arbitrary relation $r_2, r_1 \neq r_2$ qualifies as a precondition to r_1 (i.e. $r_1 < r_2$) if it holds that $\forall ot_i, 1 \leq i \leq n, \exists d_i \in \text{DOM}(r_2)$ such that $\text{type}(d_i) = \text{type}(ot_i)$. Note that the order of ot_i has to match to the order of domains d_i .

If such a relation r_2 could be found, an editor may offer the automatic creation of a relation call to it. For instance, under the assumption that a relation `PackageToSchema` exists, the following example illustrates how the unbound containments described in the previous example are bound by such a relation call in the when clause of the relation.

```

1  top relation PackageToSchema {
2    checkonly domain uml p : UML::Package {}
3    enforce domain rdbms s : RDBMS::Schema {}
4  }
5  top relation ClassToTable {
6    checkonly domain uml c : UML::Class {
7      namespace = p : UML::Package {}
8    };
9    enforce domain rdbms t : RDBMS::Table {
10     schema = s : RDBMS::Schema {}
11   };
12   when { PackageToSchema(p, s); }
13 }
```

Abstract Type Instantiations

Abstract types serve to form the structure of a metamodel. For example, they can be used to group together a set of properties that inheriting types have in common. Another example usage would be the separation into multiple type branches, if such a distinction is desired. Similar to the concept of abstractness in object-oriented programming we consider the instantiation of abstract types in object templates and domains as mistakes made by the user in our concept. The suggested solution to resolve an instantiation of an abstract type t is to let the user change the type to a non-abstract one, while staying in the scope of \hat{m}_t .

Unused Relations

In our concept, a relation r is considered to be unused if it holds that

1. $\neg \text{isTop}(r)$,
2. and $\nexists r', r \neq r'$ that calls r in one of its where clauses.

Possible solutions to resolve an unused relation is to either delete it, or to set its top property to \top .

Cycles and Deadlocks

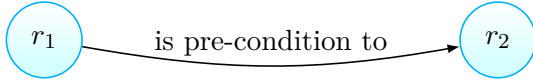
For the detection of cyclic dependencies we introduce the concept of *dependency graphs*. A dependency graph is a directed graph $G_\tau = (V, E)$ of a transformation τ , each vertex $v \in V$ represents a relation $r \in \text{REL}(\tau)$ and each edge $e \in E = (v_1, v_2)$ from v_1 to v_2 denotes that the relation represented by v_1 is a pre-condition to the relation represented by v_2 .

In order to construct a dependency graph $G_\tau = (V, E)$ for a given transformation τ , we define the following construction rules.

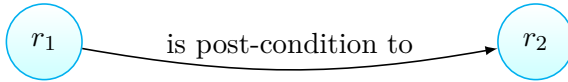
1. $\forall r \in \text{REL}(\tau) : \exists v \in V$.



2. $\forall r \in \text{REL}(\tau) \ \forall rc_{(r_1, r_2)} \in \text{WHEN}(r) : \exists e \in E = (v_1, v_2)$ such that v_1 represents r_1 and v_2 represents r_2 .



3. $\forall r \in \text{REL}(\tau) \ \forall rc_{(r_1, r_2)} \in \text{WHERE}(r) : \exists e \in E = (v_1, v_2)$ such that v_1 represents r_1 and v_2 represents r_2 .



For the sake of simplicity, we introduce the notation $r_1 < r_2 > r_3$ to express that r_1 is a pre-condition to r_2 , and that r_2 is a post-condition to r_3 .

We define a cycle of length n to be a path in G_τ of the form $r_1 < r_2 < \dots < r_n < r_1, r_i \neq r_j, i \neq j, 1 \leq i, j \leq n$, where the parameters in all involved relation calls stay unmodified. For example, let a relation call from relation r^{caller} to relation r^{called} be defined as a pair $(r^{\text{caller}}, r^{\text{called}})$, and assume a parameter list $P = (ot \mid ot \in \text{OT}(r^{\text{caller}}))$. Then, the relation call has modified parameters, if $\exists p \in P$ such that $p \notin \text{DOM}(r^{\text{called}})$.

Note that by our definition, a deadlock is just a cycle of length 2.

The suggested solution to break a cycle is to select an arbitrary pair (r_i, r_j) from the cycle path where $r_i < r_j$, and to remove the dependency between them by deleting the respective relation call to r_j in r_i .

4.5.4 Suggestion of Top Relations

A dependency graph can not only be used to detect and break cyclic dependencies, but also to heuristically determine which relations of a transformations should be top, and which should not. A relation is a potential candidate for a top relation if its corresponding vertex in the dependency graph either has only incoming “when” edges, or has no incoming edges at all. Formally, we define these constraints as follows. Let $r \in \text{REL}(\tau)$ be an arbitrary relation, and v be its corresponding vertex in the dependency graph. Then we set the top-property of r as follows.

$$\text{isTop}(r) = \begin{cases} \top & \text{if } \deg^-(v) = 0 \vee \forall (r_1, r) \in E : \exists rc_{(r_1, r)} \in \text{WHERE}(r) \\ \perp & \text{otherwise} \end{cases} \quad (4.16)$$

Applied to our SimpleUmlToSimpleRdbms transformation, this algorithm identifies the relations ClassToTable, PackageToSchema, and AssocToFKey to be top, where as the rest of the relations is identified to be non-top.

4.5.5 Metamodel Coverage

A property that we have not considered yet is syntactical *completeness*, which refers to the ability of an editor to check if the types of elements in the source model have a corresponding element defined in the target model.

A type t is considered to be unused in the scope of a transformation τ , if $\nexists r \in \text{REL}(\tau)$ such that $\nexists d \in \text{DOM}(r)$ where $\exists ot \in \text{OTE}(d) : \text{type}(ot) = t \wedge \neg \text{isAbstract}(t) \wedge (\text{kind}(\text{dom}(ot)) = \text{checkonly} \vee \text{enforce})$. Note that we can restrict the search for unused types to either checkonly or enforce domains with the last constraint.

4.6 Readability

We define the readability of a subject s to be a variable that depends on the way information is visualized and to what extent this way corresponds with the user. The visualization of subjects has certain degrees of freedom that we separate into two types. First, we consider properties that are inherent to text (*e.g.* font type, size, weight and style), and second, the spatial configuration of text. We notice that these degrees of freedom are significantly influenced by the type of notation being used for the visualization. In the case of QVTr, a GN and a TN is defined. However, readability is subjective to the user and, in particular, on the level of experience with the subject. Under the assumption

that proper syntax highlighting is available, textually viewing QVTr transformations is likely to be considered more readable than viewing it graphically, especially for advanced users of the language. In contrast, novice users possibly benefit from an approach based on the GN since the information to consume is presented in a less dense and thus overwhelming way. Our concept includes the following techniques to increase the readability.

Definition and Outlaying of Views. Since the GUI of an editor is the link between the user and the transformation to be edited or executed, it is worth spending thoughts on how its appearance and structure can contribute to the readability. A challenging part of designing an editor for QVTr is to decide *which parts* of the model transformation pipeline should be presented to the user *at what time* using *which form* of visualization. An editor that has the goal to enable the user to *model* and *execute* transformations at all, has to at minimum provide navigable and editable views for transformations and their containing relations, as well as a way for the user to select source models to execute the transformation on. For this basic functionality, the following views are required.

1. View for navigating through the contents of transformations.
2. View for editing transformations and relations.

However, this minimal toolset is limited in terms of providing *productive* modeling, since it does not allow for short feedback cycles. Since our concept incorporates the idea of a visual result validation as described in Section 4.1, we require the following views in addition.

3. Visualization view for the *source* model.
4. Visualization view for the *target* model.
5. Visualization view for the *trace* model.

After the required views that an editor has to provide have been defined, the editor's designer has to decide on the layout of these views. Since a total of 5 views pushes the limits of current state of the art screen resolutions to their limit, we combine views 1. to 2. in a single *main view*, and outsource views 3. to 5. into a separate *execution view* to minimize clutter in the GUI. Since our concept is based on the GN of QVTr, view 2. is represented by a tabbed diagram canvas. Additionally, we house view 1. in a sidebar to the left, which can be collapsed as needed. As can be seen in Figure 4.10 the main view is laid out in a way which ensures that the user's focus concentrates on the diagram canvas.

The concrete layout is separated into 3 parts, described as follows.

- A title bar that grants access to functions for managing projects, creating or importing transformation and relations and for opening additional views of the project.

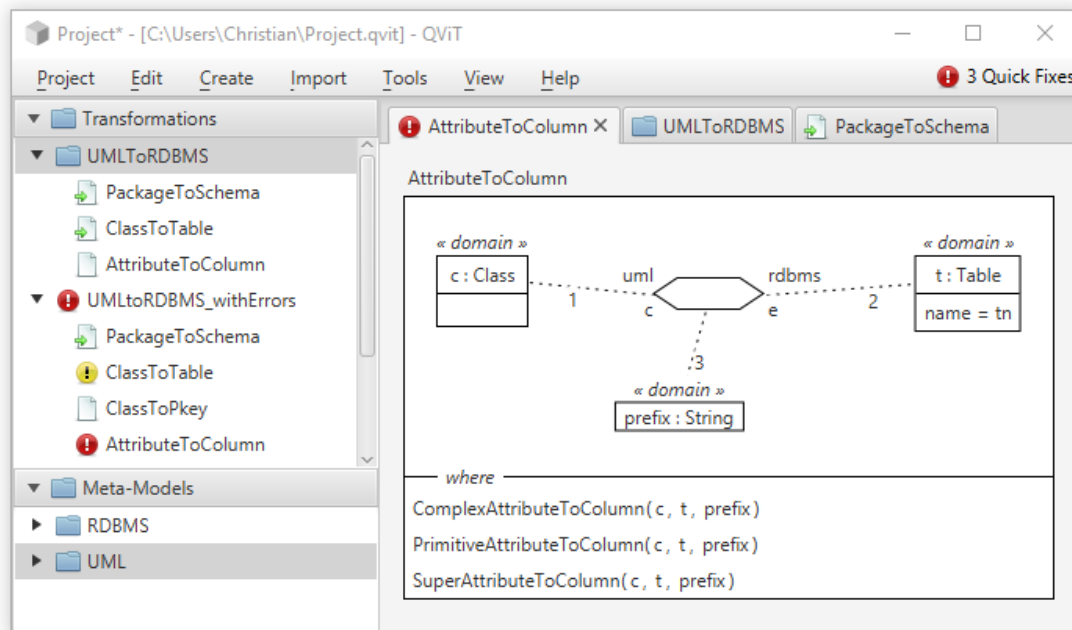


Figure 4.10: A clear and focused layout for the required main view

- A sidebar that lists the contents of the currently opened project and allows for its navigation.
- The tabbed diagram canvas that allows for editing relations and viewing transformations.

Now that also the outlaying of the chosen views is defined in our concept, the used notation for each view has to be decided on. Since QVTr has both a textual and a graphical syntax, editors may choose to support either one, or both of them. In our concept, we suggest the *editing* of transformations using the GN, whereas an optional view for *viewing* the transformation is provided using the TN. We choose for the optionality of the TN view to further concentrate the user's focus on the diagram canvas.

Another important aspect that the designer of an editor has to decide on is the level of edit granularity in the diagram canvas. The challenge here is to find a balance between 2 extremes such that the readability is not affected negatively. The first extreme is a static visualization, where the vertices are not editable directly on the canvas, but instead in separate dialog windows. The other extreme is a highly dynamic canvas that integrates editable widgets such that vertices are directly edited on the canvas itself. In our concept, we choose an approach where the structure of the relation is edited on the canvas itself, but individual vertices are only editable in dedicated dialog windows. This is due to the idea of presenting the *outline* of a relation in the canvas, that a user can arrange in a way that is desired and most readable. For the detailed editing of object templates or

predicates in the when/where clauses, the editing context switches to dedicated dialog windows, to avoid information overloading and to strengthen the user's focus on a single task at a time. For emphasizing this context switch, we also gray out the underlying windows of dialogs.

| Id | Subject to visualize | Visualized in | Notation | Editable? |
|----|----------------------|----------------|----------|-----------|
| 1. | Navigation | Main View | TN | No |
| 2. | Transformations | Main View | GN, TN | Yes |
| 3. | Relations | Main View | GN, TN | Yes |
| 4. | <i>Source</i> Model | Execution View | GN | No |
| 5. | <i>Target</i> Model | Execution View | GN | No |
| 6. | <i>Trace</i> Model | Execution View | GN | No |

Table 4.1: Required views in our concept

Table 4.1 summarizes the required views.

Syntax Highlighting. Syntax highlighting is a well-established strategy for increasing the readability of source code among state of the art textual editors for traditional programming languages such as Java or the family of C languages. The idea is to minimize the time it takes users to understand the semantics of code by letting them consume the code with dedicated colors and styles for keywords, literals or comments. For example, Figure 4.11 illustrates different colors and font weights depending on the data type of values.

| | |
|---------------|---------|
| « domain » | |
| a : | Area |
| name = | 'area1' |
| numVertices = | 3.0 |
| isHidden = | false |

Figure 4.11: Syntax highlighting for QVTr's GN

For the textual notation of QVTr, Figure 4.12 illustrates a conceivable styling.

Domain Numbering. The QVTr standard does not specify how the GN should communicate the order of domains in a relation. Clearly, there is a need for a user to find out about the order, since the order of parameters in relation calls is sensitive to it. Otherwise, a relation call with parameters to another relation that has an undefined domain ordering is non-deterministic. Users of editors based on the TN can rely on the order being established by the reading order, which typically is top-down. Considering the GN however, the reading order is not a reliable option anymore, since the vertices can be positioned by the user as desired. To establish a visual ordering that is independent of

```

1  -- This relation maps packages to schemas.
2  top relation PackageToSchema {
3      sn : String;
4      checkonly domain uml p : UML::Package {};
5      where { sn = 'schema1'; }
6  }

```

Figure 4.12: Syntax highlighting for QVTr's TN

the vertices' positions, we put a number indicating its position to each limb that connects a domain vertex to the hexagon symbol. Figure 4.13 illustrates this form of domain numbering.

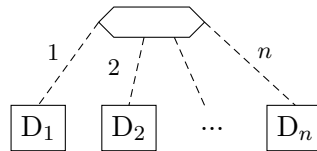


Figure 4.13: Visualizing the order of domains in a relation with numbering

Domain Kind Sensitive Captions (DKSC). Taking into account novice users of QVTr, which are not yet familiar with QVTr's terminology, we suggest the use of so called DKSC. These could potentially make the use of OTs clearer, *i.e.* “checkonly” OTs are *selecting* elements from a source model, whereas OTs under an enforce domain *generate* elements in a target model. Figure 4.14 illustrates an example of DKSC, as the terms “source-selection” and “target-generation” are used to indicate the respective domain kind.

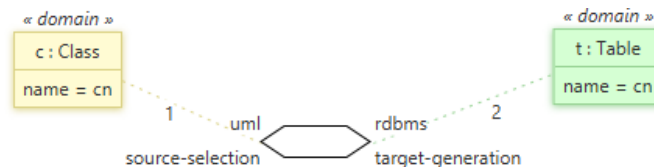


Figure 4.14: An example of Domain Kind Sensitive Captions (DKSC)

Top Property Indicator. Furthermore, the QVTr standard does not specify how top relations should be visually marked as such. To increase the readability in this context, the term “top” is put beside the relation's name, using a bold font weight. Figure 4.15 illustrates that the `PackageToSchema` relation is a top relation, whereas the `ClassToTable` relation in Figure 4.16 is not.

Landing Page. While not specifically targeted at an increased readability, our concepts includes the use of a landing page, when first starting the editor. On this welcoming

Top PackageToSchema

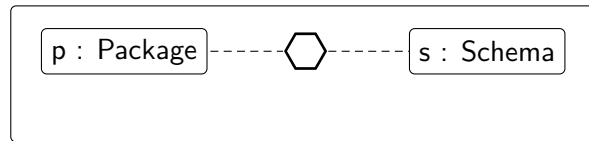


Figure 4.15: An indicator for the top property beside the name of a relation

ClassToTable

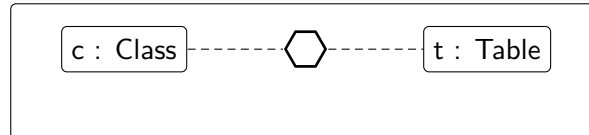


Figure 4.16: No indicator for the top property beside the name of a relation

page, the user finds buttons that suggest the next required actions necessary to start with modeling. As can be seen in Figure 4.17, we first display a single “New Transformation” button, and after there has been one created in the current project, we also display “New Relation” and “Predict new Relation” in addition.

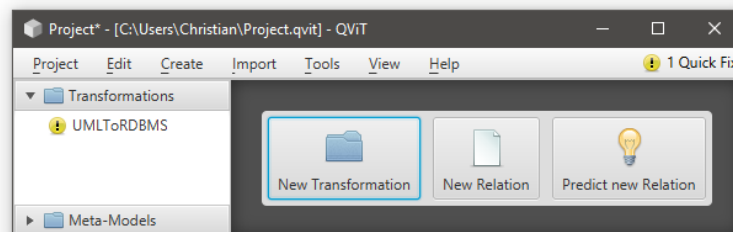












Figure 4.17: A landing page communicates necessary actions to the user

Icons. The entities a user has to manage in QVTr comprise transformations, relations, metamodels and their types. For each of them, the usage of dedicated icons allows for the following advantages.

- The icons  and  suggest a parent/child relation, based on the directory/file analogy. Since there is such a parent/child relationship between transformations and relations, and between metamodels and types, it is conceivable to use folder and file icons respectively. The locations of usage may expand from the project explorer in the sidebar to the icons used in tabs, or in dialogs. On the one hand, such an analogy is effective since it allows users to instantly understand the relationships, behaviors and properties of new, yet unknown entities. However, a bad analogy may also mislead the user and cause for misconceptions.

- In addition, the usage of icons for these entities increases their recognition value. This is especially useful in dialog windows, as the user faster recognizes to which entity a dialog refers to, as the icon is indicating if it is about a transformation, a relation, or a metamodel.
- Subtle details in an icon can hint to specific *states* of entities. In QVTr this is especially useful for indicating the top-property of relations. In our concept, top relations are marked as such using  as icon, as opposed to non-top relations being represented with  as an icon.
- Another usage of icons that the user can benefit from significantly is that of indicating error states. In our concept, we distinguish between errors using , warnings using  and the absence thereof using  as icons. The GUI may indicate the error state of the currently opened project at a central place. To make it easier to locate the sources of errors and warnings, it is conceivable to directly replace the icons of affected transformations and relations with those of an error or a warning.
- Finally, icons are conveniently used to suggest the outcome of certain actions that a user can execute by clicking on a specific button. Examples for this are the  icon for deletions, the  icon for copying, or  that suggest the execution of utility tools like the suggestion of top relations or the prediction of new relations.

In each case, the *consistent* usage of the same icons for the same elements is crucial, since otherwise the aforementioned recognition values would be negated again.

4.7 Traceability

Considering the complexity of the transformation pipeline, traceability plays a central role for the usability. We define traceability as a subject's ability to be traced by the user. Our concept includes the following techniques, which aim to increase the traceability of QVTr and the generated target model.

Trace Model Visualization. First, we consider the traceability of the objects in a generated target model to their related objects in the respective source model, according to a certain relation in the transformation. The idea is to visually highlight those objects in the source model, that were queried against and also those objects that were generated in the target model, according to a specific relation.

For instance, consider the following `AttributeToColumn` relation, which maps each class with an attribute to a column in a table.

As can be seen in Figures 4.19 and 4.20 the respective elements are visually highlighted using a certain color encoding. Being able to inspect a target model in such a per-relation way allows the user for an increased traceability.

Occurrence Highlighting. A way of increasing the intra-relational traceability is to visually highlight all occurrences of a certain variable or object template in the scope of

```

1  relation AttributeToColumn {
2    an : String;
3    checkonly domain uml c : UML::Class {
4      attribute = a : UML::Attribute { name = an; }
5    };
6    enforce domain rdbms t : RDBMS::Table {
7      column = cl : RDBMS::Column { name = an; }
8    };
9  }

```

Figure 4.18: A relation that maps classes with attributes to columns in tables.

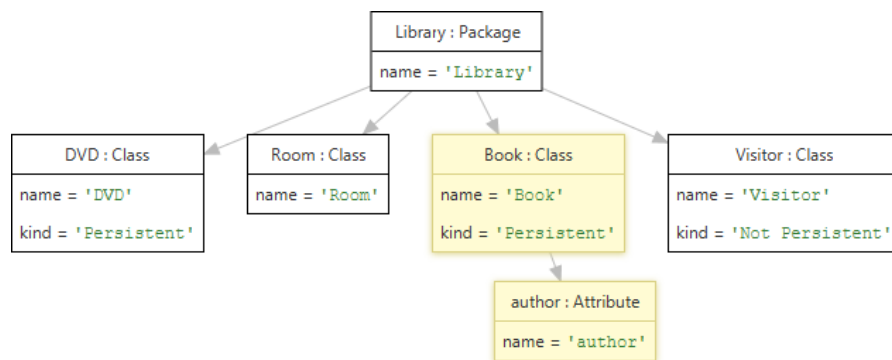


Figure 4.19: Objects *selected* by the AttributeToColumn relation in a *source* model

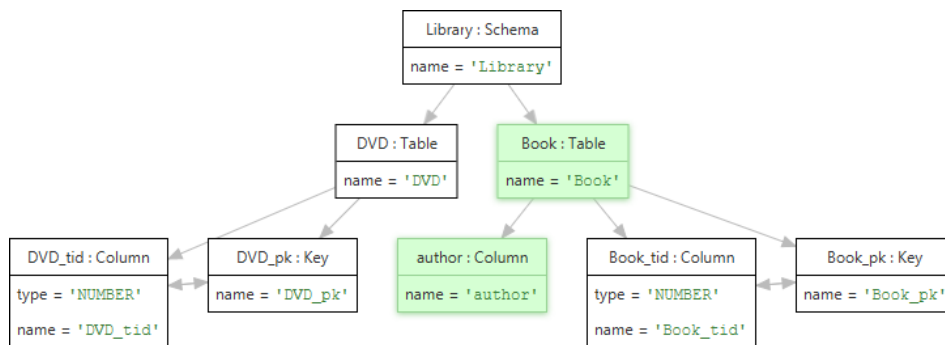


Figure 4.20: Objects *generated* by the AttributeToColumn relation in a *target* model

a single relation. That way, a user is able to quickly find out about the *location* thereof. An established way of achieving this in text editors is to decently change the background color of all occurrences of a variable, as soon as the user puts the cursor on it. For editors that implement QVTr's GN, figures 4.21, 4.22 illustrate how this can also be done using a mouse-hover behavior.

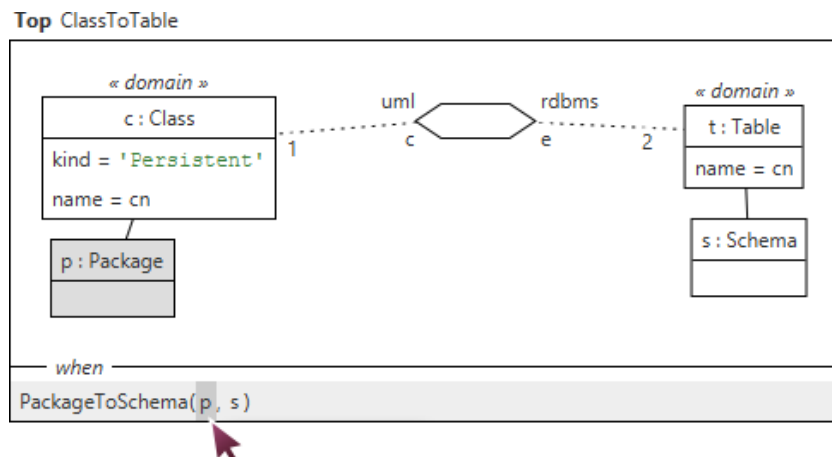
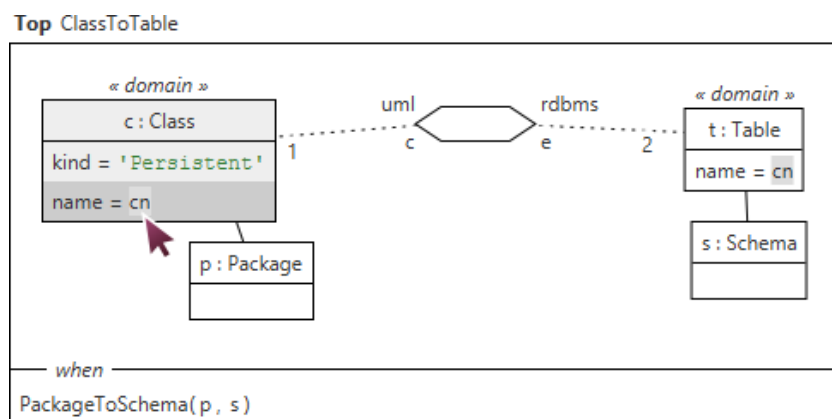
Figure 4.21: Occurrences of the *p* object template are highlighted on mouse-hoverFigure 4.22: Occurrences of the *cn* variable are highlighted on mouse-hover

Figure 4.22 illustrates how occurrence highlighting increases the traceability of the value of variables used in enforce domains. That way a user is able to quickly determine that, in this example, the *cn* variable holds the name of persistent classes.

Navigable Relation Calls. A way to increase the inter-relational traceability is to allow for navigable relation calls. The idea is to visually indicate the relation call as a clickable hyperlink, as soon as the user hovers over it, to suggest the navigation capabilities.

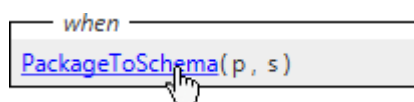


Figure 4.23: Hyperlink-behavior for relation calls

Dependency Graph Visualization. Viewing a transformation in textual form turns out to be tedious only for finding out about the call dependencies among the relations. It is conceivable to offer the user a dedicated view showing the hierarchy among relations only. Figure 4.24 shows how such overviews are defined in our concept.

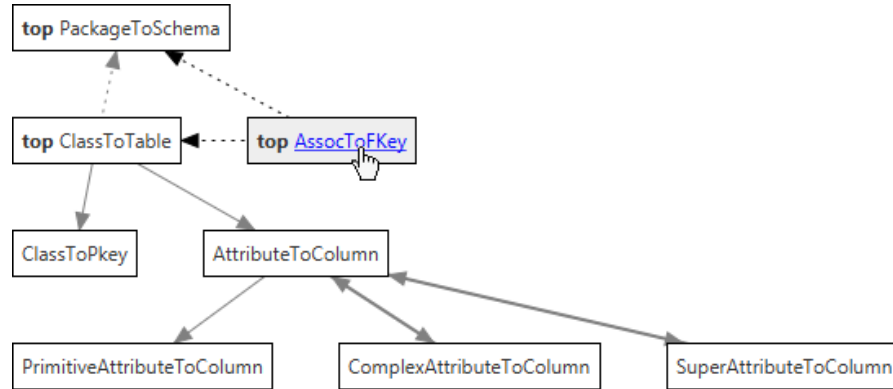


Figure 4.24: Overview of the call hierarchy among multiple relations

Each relation is represented by a node holding both the top property and the relation's name. The nodes are navigable, as a mouse-click on them would open the respective relation in a new tab in the tabbed diagram canvas. A relation is a pre-condition to another, if a dashed arrow is pointing to it. Similarly, a relation constitutes a post-condition to another, if it has a solid arrow pointing to it. Note that the node arrangement is done in a top-down order, to visually indicate the hierarchy among the relations.

Evaluation

The theoretically formalized concept in Chapter 4 provides conceivable means to reason about the first 3 of our 5 research questions in this thesis. For the remaining 2 questions, we are interested in how that theoretic concept performs against state of the art editors for QVTr *in practice*. At this point, we like to emphasize our intention of not comparing actual implementations, but instead the enabled modeling approaches against each other. Hence, in this chapter, we report on the empirical study that we have conducted to achieve this comparison.

5.1 Prototypical Implementation

In order to be able to evaluate our proposed theoretical concept, we have prototypically implemented it using the JavaFX [43] programming language, since it is an established language for developing desktop applications with highly interactive user interfaces. In this context, we have incorporated the ten usability heuristics proposed by Nielsen [37]. For completion's sake we would like to note that the non-functional requirement *execution performance* has been considered of secondary importance during the implementation of our prototype. In the following, we describe the components of which our QViT editor consists of.

Diagram Canvas. We have implemented a diagram canvas, that contains *draggable* and *selectable* vertices for domains, the hexagon symbol, and object templates. The box that represents a relation automatically resizes as the user drags vertices around. For the purpose of providing OCEs and CPMs, we make use of a pop-over that appears as soon as the user selects certain vertices on the canvas.

Parser & Serializer. QViT contains parsers for the Ecore, the textual QVT and the XMI data format, and is also able to store the such imported (meta-)models and transformations back to disk in their respective formats using dedicated serializers.

Undo/Redo System. We have encapsulated all actions that a user can perform to modify a transformation into closed units, that are undoable and redoable. Different stacks of executed actions are defined in the context of the main view and dialogs.

Detail Dialogs & Alerts. The dialogs in which the user is able to edit the details of projects, transformations, relations and object templates appear in a modal fashion over the main view. In addition, modal alerts are raised for the purpose of informing the user about success, info and error messages.

Additional Utility Views. QViT contains a dedicated *execution view* in which the imported source model and the generated target and trace models are visualized in draggable diagrams. In addition, a dedicated *text view* shows the currently focused transformation in its textual notation.

Quick Fixes. We have implemented the techniques concerning preventive interactivity by means of quick fixes, accessible over the title bar in the main view. A clickable hyperlink shows the number of currently detected errors and warnings, and a click on the hyperlink opens a pop-over, containing a detailed list of all found problems. As soon as the user selects one item from this list, the description of the problem, the involved actors and of the suggested solution is presented. This pop-over is also the location where the quick fixes are applied by a single button click.

Logging System. Finally, we have implemented a system that logs user's mouse and keyboard input, as well as all executed actions that modify a transformation.

5.2 Experiment Plan

The planning of our experiments is inspired by the work of Wohlin et al. [61] for experimentation in software engineering. According to them, an empirical study is mainly *exploratory* or *explanatory*. In the first type, the research is primarily done by observing the participant while putting minimum effort to control the test environment. This passive way of conducting a study is suitable when to gain new perspectives on the object of study, since each participant possess different understandings and beliefs. In contrast, the latter type focuses on the discovery of relationships between specific and well defined causes and effects. Hence, a much more controlled test set-up is required in this case. This type of study is useful when the impact of a well documented modification to a specific method should be evaluated. In the context of this thesis, we choose a hybrid approach rather than favoring one type over the other. This is because we are not only interested in *finding out* about the usability effects of our concept, but also in understanding their causes. For this to achieve, we have to observe and interpret the reactions, intentions, emotions and (non-)verbal behavior of the participants during the experiment. Since our aim is to identify the context in which our concept generates the most value, we conduct the experiments under controlled conditions with MDE practitioners in an *industrial* environment. In particular, we were granted to perform the experiment at the LieberLieber Software GmbH company in Vienna.

5.2.1 Scope

In the first step, we define the scope of the experiment, in order to answer *why* the experiment is conducted. For this to achieve, we formulate the goal definition of our experiment using the *goal definition framework* [61] as follows.

The goal of our empirical study is to analyze *our proposed concept* for the purpose of *identifying the context*¹ *in which it generates the most value* with respect to its *usability* from the point of view of *MDE practitioners* in the context of *a software company with a professional interest in MDE*.

5.2.2 Planning

The second step aims to define *how* the experiment is conducted.

Selection of Context and Participants. We conduct our experiment in the context of an MDE-based software company consisting of professional MDE practitioners with a software engineering background. Our rationale for this selection is based on the actual professional interest of these participants, as opposed to students which possibly are indifferent towards the experiment. Furthermore, since in our experiment we are interested in evaluating the *first-time* usability of our concept, they were selected such that their knowledge level with both editors were equally balanced. This has been done by self-assessment of the participants. We also required participants to have an understanding of modeling tools in general.

Comparative Design. Since we want to evaluate the capability of our proposed concept to contribute to an increased usability of using QVTr, we have decided to use a comparative design for our experiments. Hence, in order to measure an increase in usability, we have to first define a baseline towards which the comparison should relate to. A state of the art editor for QVTr is an appropriate candidate for such a baseline. In order to ensure that the experiment does not compare the capabilities of two editors rather than two theoretical approaches, we must ensure functional equivalence. In other words, we have to select a state of the art tool, such that the functionality provided by QViT is a subset of that of the selected tool. This subset comprises the *modeling* and *execution* of model transformations. Taking this considerations into account, the Eclipse-based state of the art editor *medQVT* is an appropriate choice for the comparison. For an increased fairness, we have also implemented QViT in a way, such that it can easily interface with different execution engines. Hence, we were able to make both editors use the same execution engine for a fair comparison. To further increase the fairness of the comparison, the participants were required to have equal experience with both editors, which mainly was less to no experience at all.

¹The users and their environment.

Task to be achieved. The experimental testing of two approaches also requires the definition of a certain model transformation exercise. We have decided to provide participants with pre-defined source and target models as input to the model transformation task.

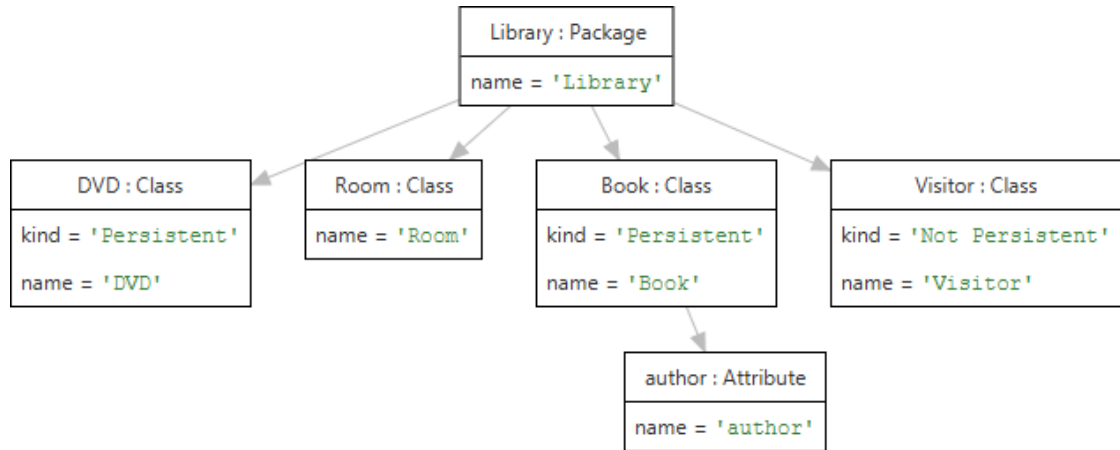


Figure 5.1: The source model to be transformed

The task to achieve is to create a QVTr transformation, that transforms the given source model in Figure 5.1 into a given target model, that can be seen in Figure 5.2.

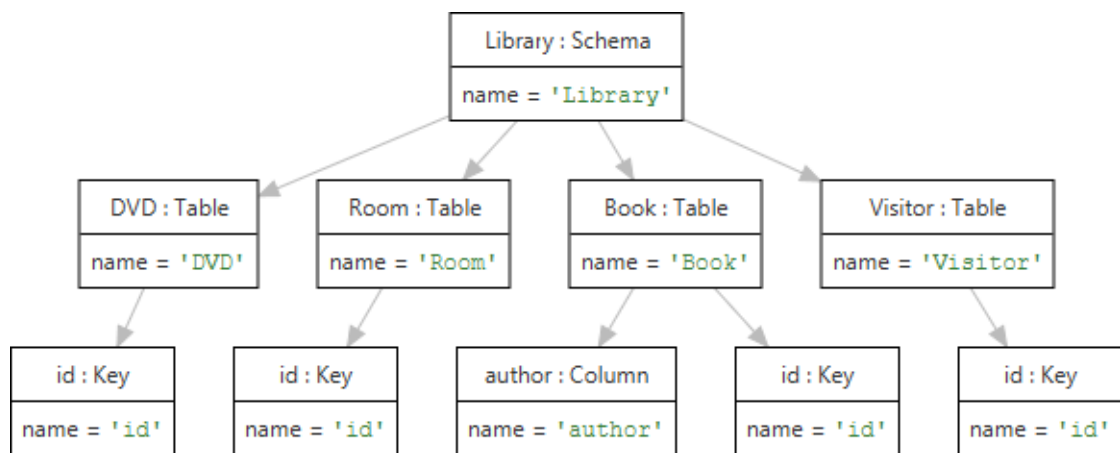


Figure 5.2: The target model to generate from the source model

The printed-out hard-copy that was handed out to the participants during the experiments can be found in the appendix of this thesis.

The example solution to our defined task is illustrated in the following QVTr script.

```

1  transformation UMLToRDBMS (uml:UML, rdbms:RDBMS) {
2
3      top relation PackageToSchema {
4          pn : String;
5          checkonly domain uml p : UML :: Package { name = pn };
6          enforce domain rdbms s : RDBMS :: Schema { name = pn };
7      }
8
9      top relation ClassToTable {
10         cn : String;
11         checkonly domain uml c : UML :: Class {
12             name = cn,
13             namespace = p : UML :: Package {}
14         };
15         enforce domain rdbms t : RDBMS :: Table {
16             name = cn,
17             schema = s : RDBMS :: Schema {},
18             _key = k : RDBMS :: Key { name = 'id' }
19         };
20         when { PackageToSchema(p, s); }
21         where { AttributeToColumn(c, t); }
22     }
23
24     relation AttributeToColumn {
25         an : String;
26         checkonly domain uml c : UML :: Class {
27             attribute = a : UML :: Attribute { name = an }
28         };
29         enforce domain rdbms t : RDBMS :: Table {
30             column = c1 : RDBMS :: Column { name = an }
31         };
32     }
33 }

```

We have designed this exercise in a way, such that it covers the lion's share of commonly used QVTr concepts which are listed in the following.

- creation of transformations, relations, model declarations, domains
- one-to-one mappings, and nested object templates
- when and where clauses, and relation calls
- primitive properties, and variables

Variables Selection. The independent variables that are not under our control during the experiments are the experience levels of participants in QVTr, medQVT and QViT. However, as stated earlier, we have required participants to have balanced experience with both editors using self-assessment. In contrast, the dependent variables in our experiments are the effectiveness, efficiency and satisfaction of participants, as defined by ISO 9241-11 [18].

Data Collection. We have setup a system that collected quantitative data during the experiments. First of all, the computer screens of the test machines have been recorded including the movement of the mouse cursor. Privacy of the collected data has been ensured by only recording video and no audio material. We have chosen for the screen recording, since it enables the post-test derivation of the intentions that the user had during the usage of the tested software. In addition, we have implemented QViT in a way such that it logs each occurred event, key-stroke and mouse-click, including the time of occurrence and the (x, y) -coordinates in the later case. Examples of logged events are when the user's focus switches from the main view into the execution view, or into a modal dialog. In order to collect qualitative data, we have designed a questionnaire with closed questions targeting our hypotheses. The questionnaire has been implemented as an online webpage, to ensure easy and fast usage during the experiment.

5.2.3 Execution

Consequently, the next step after the experiment planing was its actual execution. The experiments were conducted off-line, in 2 separate sessions at the LieberLieber Software GmbH, which is a medium-sized company in Vienna focusing on products for MDE. In the beginning, the participants were told to test out a new tool for QVTr that has been developed with a focus on usability. However, they were not informed about *how* the usability should be achieved, *i.e.* about our hypotheses from Chapter 1. After a quick introduction to the two syntaxes of the QVTr modeling language, the participants were told that their task during the experiment is to compare two approaches against each other. (i) Textual modeling with medQVT against (ii) graphical modeling with QViT. They were not told that the first approach hardly offers interactivity and automation compared to the second approach. After this introduction, they were exposed to the approach provided by medQVT, as they were demonstrated the modeling of an example transformation in a think-aloud style. After that, the participants were asked to solve the same task using QViT, and they were told that they would have the chance to give feedback about the comparison in an interview and a questionnaire afterwards.

5.3 Results

In this section, we descriptively present visualizations and textual reports of the data that has been collected during the two sessions of our empirical study. The interpretation of the presented data follows in the subsequent section. If not explicitly described otherwise, the visualizations show the *averaged* data over all participants.

5.3.1 Time Distributions

First, we present the data that shows how the participants spent their time during their experiment sessions.

- **Time spent in Views**

Figure 5.3 illustrates in which views the participants have spent their time. As mentioned in Chapter 4, we distinguish between

- the main view, that contains the project explorer and the tabbed diagram canvas,
- the execution view, containing the visualizations of the source, target and trace model,
- the text view, containing a textual representation of the currently edited transformation,
- and in modal dialogs, used for detailed editing of projects, transformations, relations and object templates.

- **Time spent in Dialogs**

Figure 5.4 zooms in on the distribution over specific dialogs.

- **Time spent with Modeling Tasks**

Figure 5.5 depicts for which modeling tasks the participants have spent their time. The various different modeling tasks have been defined in Chapter 3.

5.3.2 Input Distributions

We have also recorded each mouse-click and key-stroke along with the time of occurrence during the experiment.

- **User Input Distribution**

Figure 5.6 compares the overall occurrence of mouse input in the form of mouse-clicks (primary and secondary mouse buttons) against that of keyboard input.

- **Mouse Input Distributions**

Figure 5.7 shows the distribution of mouse input per view.

- **Keyboard Distributions**

Similar to the previous, Figure 5.8 shows the distribution of keyboard input per view.

- **Mouse/Keyboard Distributions**

Figure 5.9 illustrates in which view either the mouse or the keyboard has been the dominant form of input.

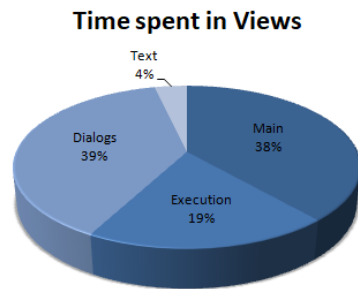


Figure 5.3: Time spent in Views

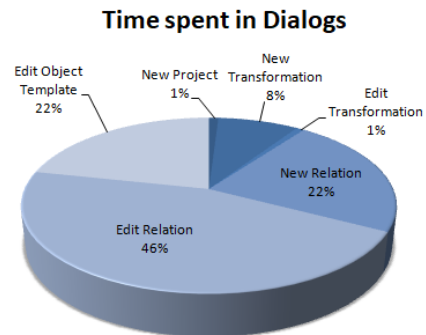


Figure 5.4: Time spent in Dialogs

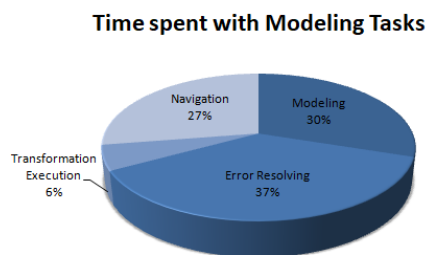


Figure 5.5: Time spent with Modeling Tasks

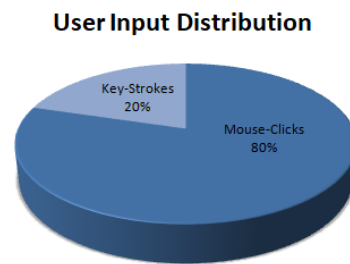


Figure 5.6: User Input Distribution

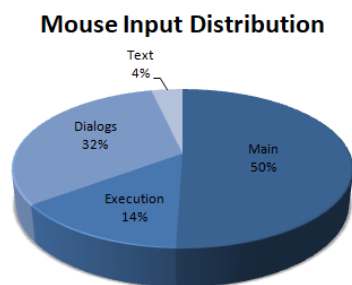


Figure 5.7: Mouse Input Distribution

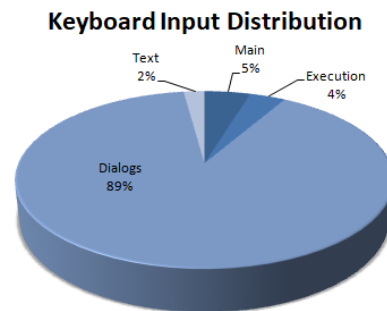


Figure 5.8: Keyboard Input Distribution

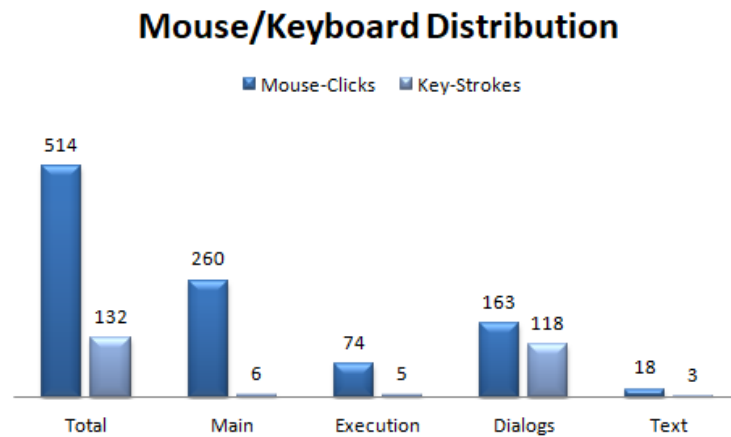


Figure 5.9: Mouse/Keyboard Distribution

5.3.3 Mouse Input Agglomerations

With the use of *Heatmapper* [5], we can visualize those spatial locations on the views, where agglomerations of mouse-click input occurred. In order to ensure the validity of these heatmaps, we have disabled window resizing in QViT during the experiments. Window positioning however was still enabled.

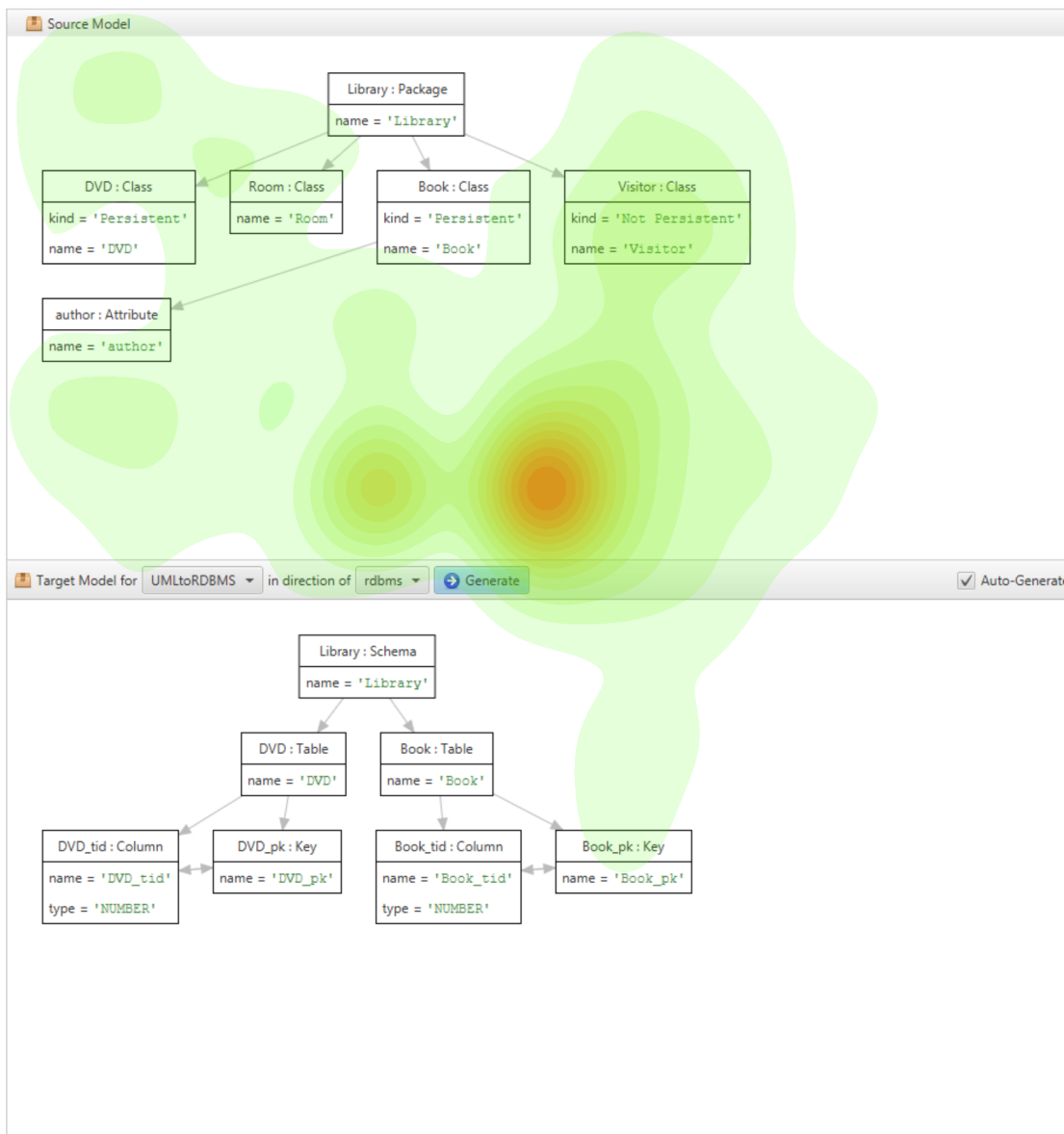


Figure 5.10: Heatmap over the Execution View



Figure 5.11: Heatmap over the Text View

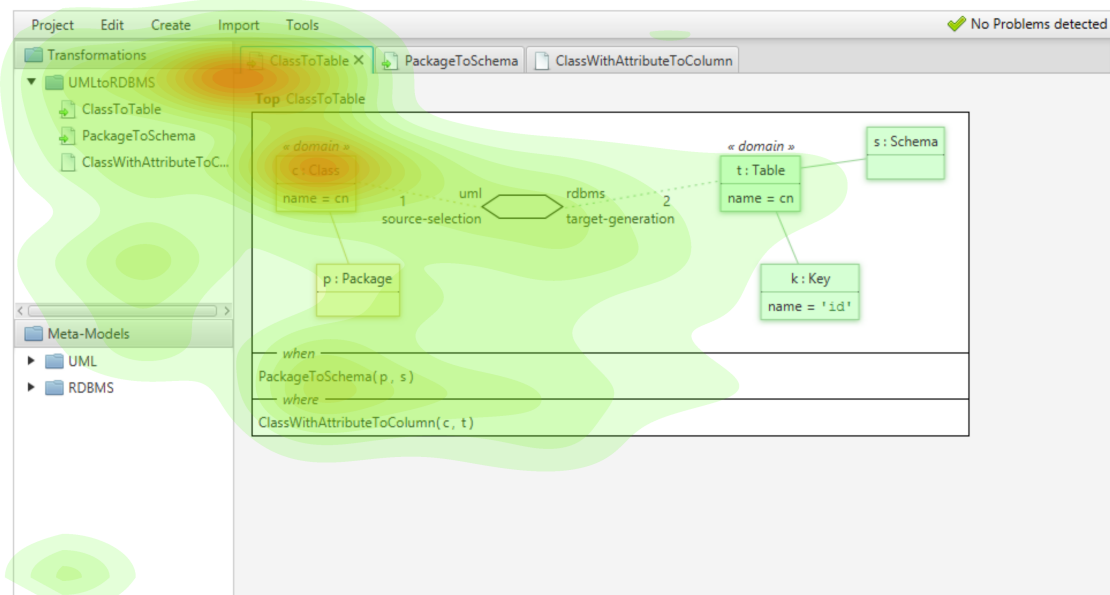


Figure 5.12: Heatmap over the Main View

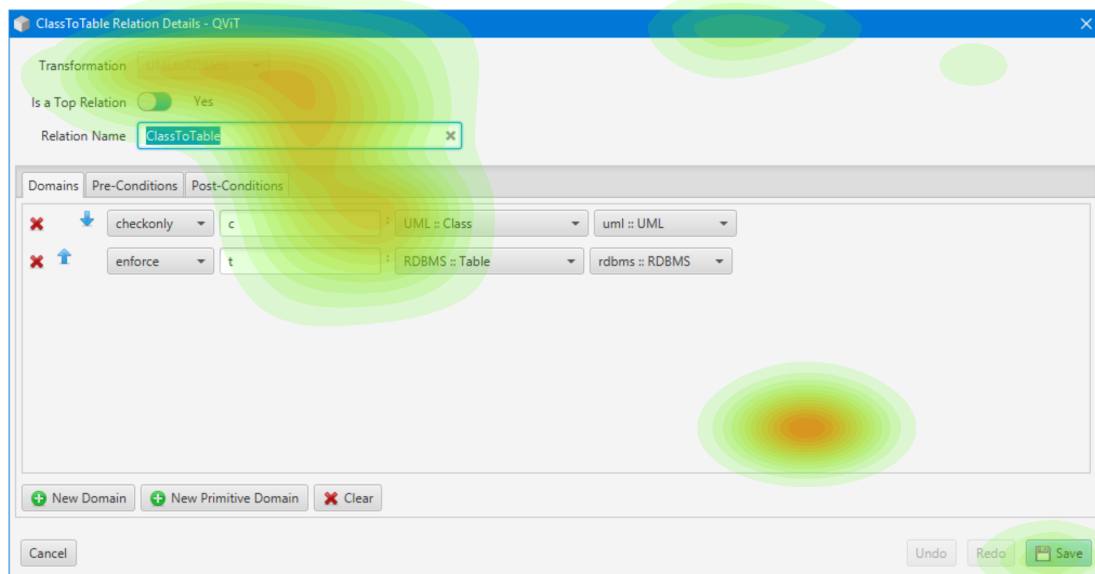


Figure 5.13: Heatmap over the Relation Details Dialog

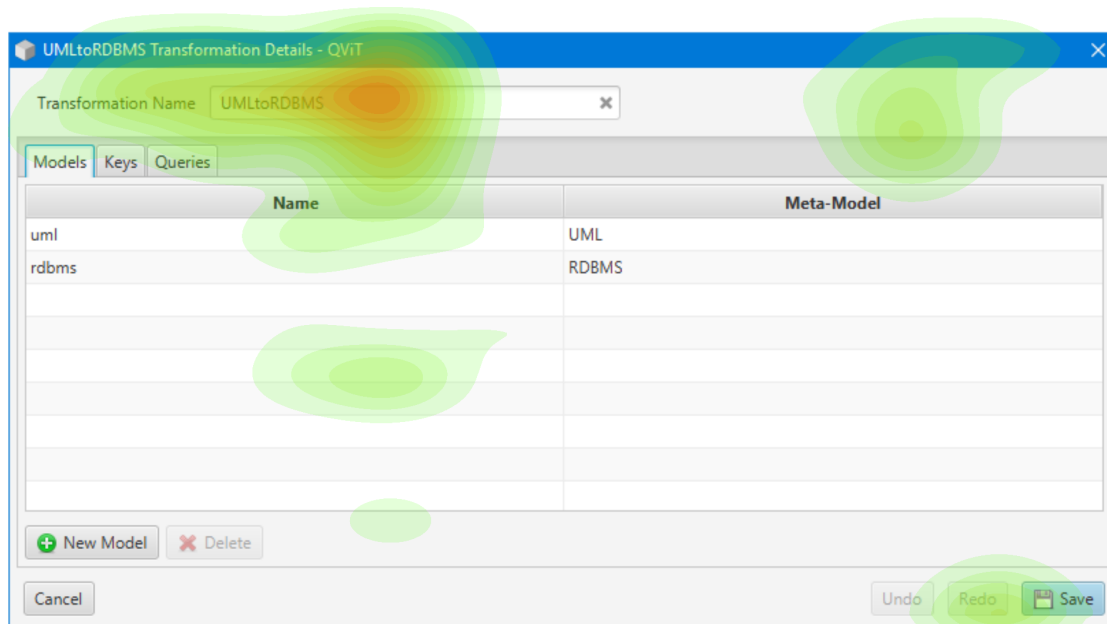


Figure 5.14: Heatmap over the Transformation Details Dialog

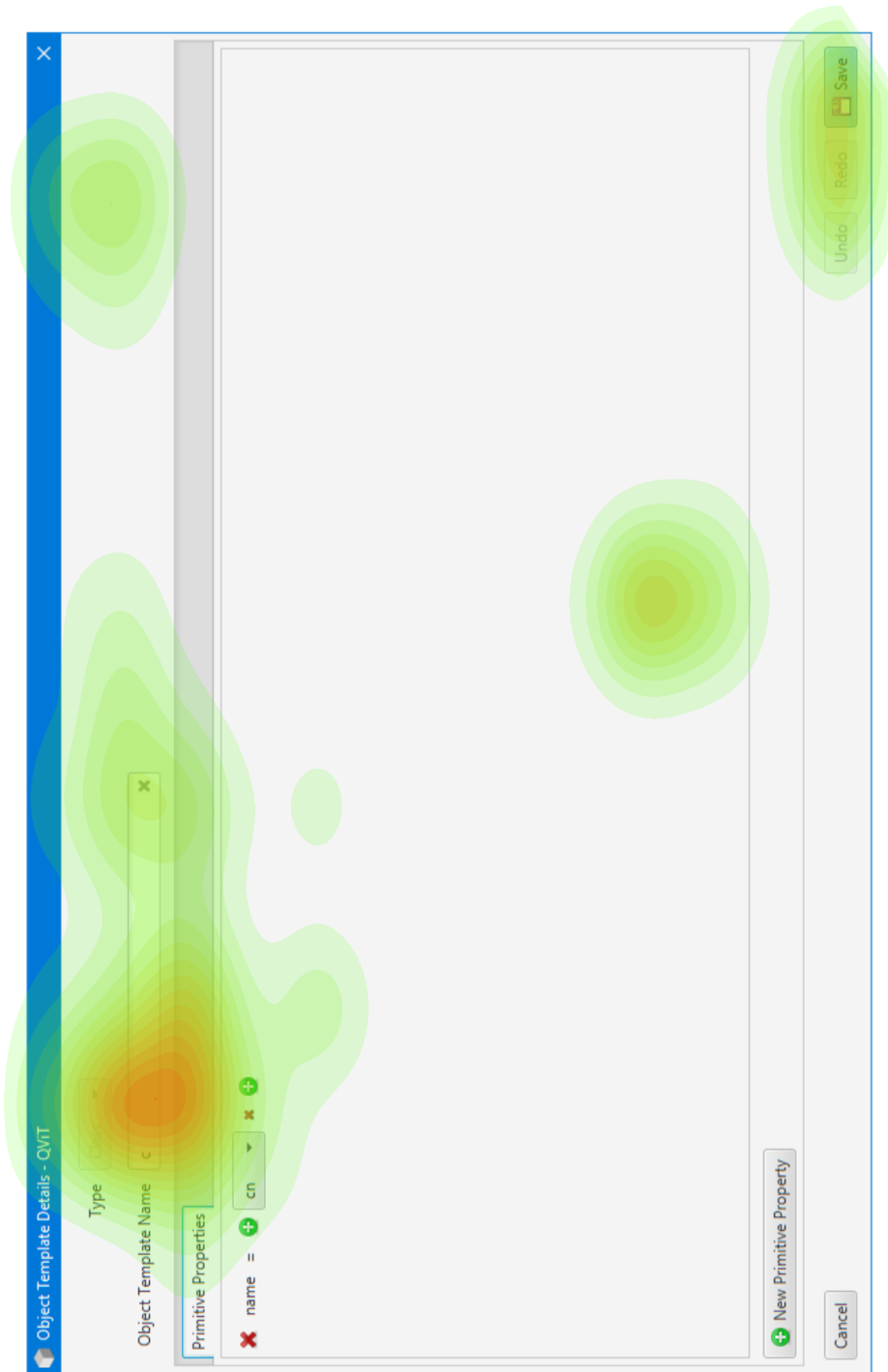


Figure 5.15: Heatmap over the Object Template Details Dialog

5.3.4 Timelines

Figures 5.16, 5.17 illustrate the distributions of view-usage and user-input over time, using the data of different participants.

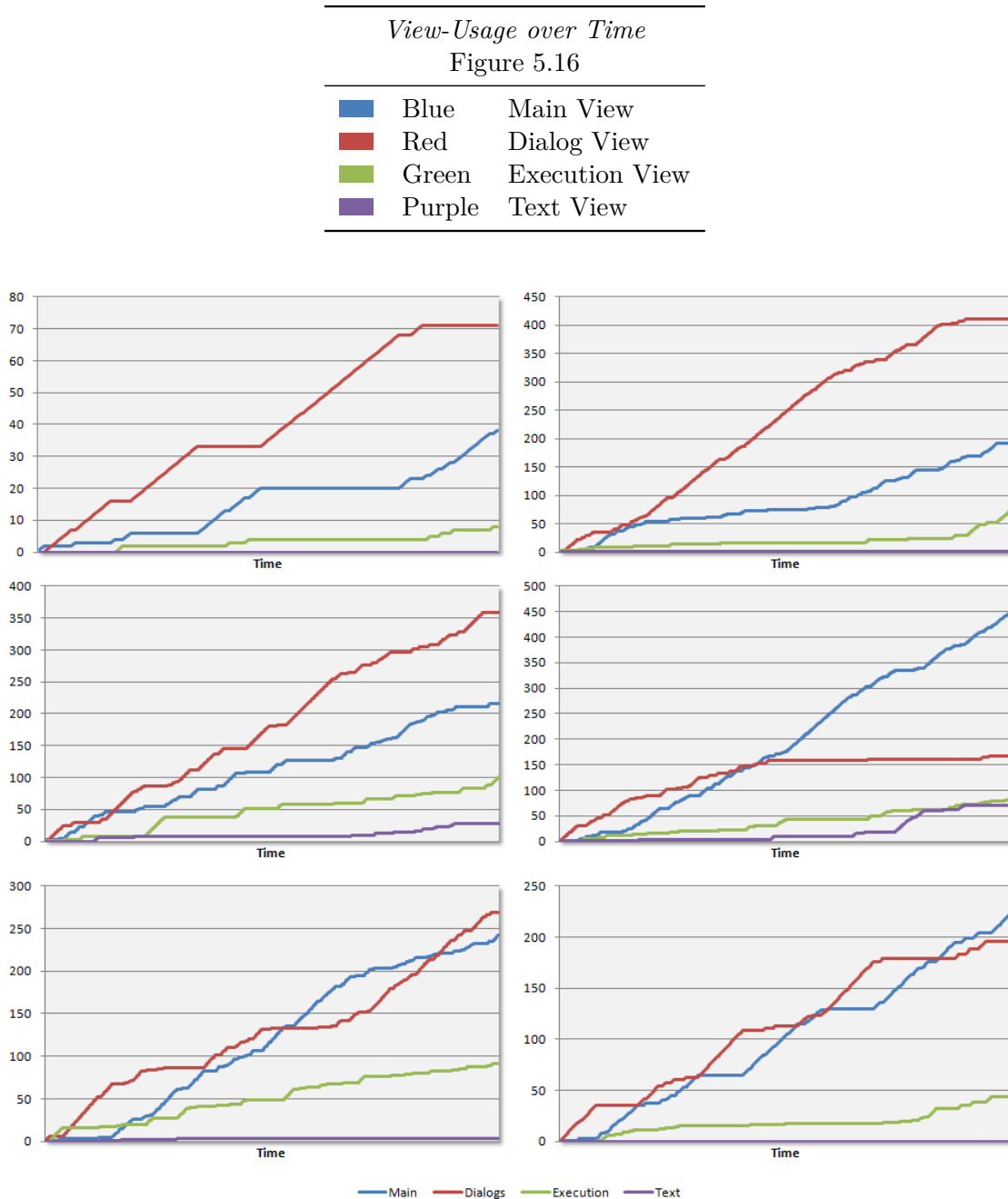


Figure 5.16: View-usage over time of different participants

User Input over Time
Figure 5.17

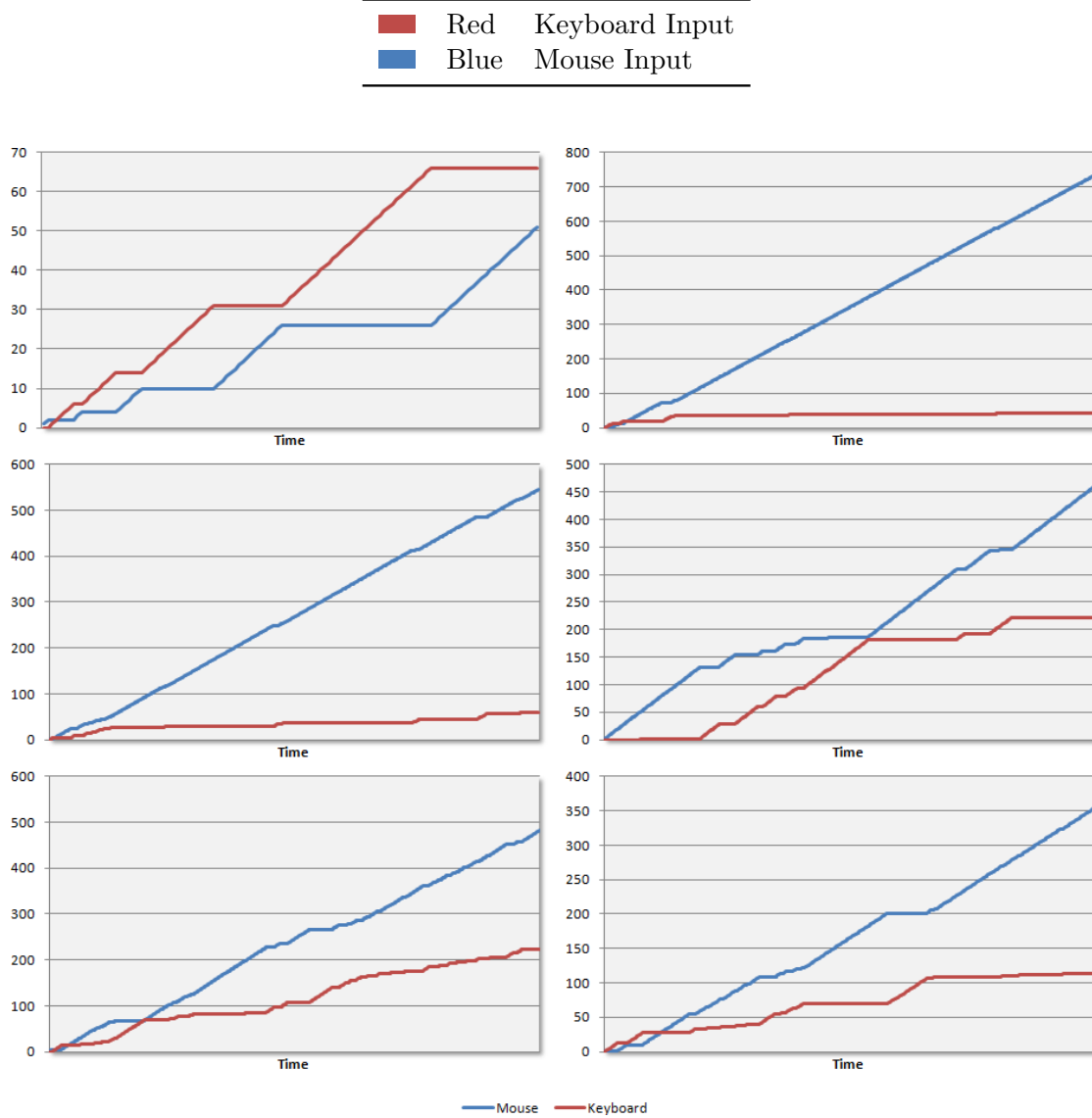


Figure 5.17: Cumulative user-input over time of different participants


































| | <i>What decreased the modeling effort in QViT?</i> | <i>Yes</i> | | <i>No</i> |
|-----|--|------------|---|-----------|
| 14. | Model Derivation from Transformation names | 100% |  | 0% |
| 15. | Domain Derivation from Relation names | 100% |  | 0% |
| 16. | Suggestion of Default Names | 100% |  | 0% |
| 17. | Automatic parameter-selection in Relation calls | 100% |  | 0% |
| 18. | Automatic configuration of the execution profile | 100% |  | 0% |
| 19. | Auto-Complete for names | 100% |  | 0% |
| 20. | One-Click-Extensions | 100% |  | 0% |
| 21. | Prediction of new Relations | 100% |  | 0% |
| 22. | Quick Fixes | 100% |  | 0% |
| 23. | Common Property Mappings | 100% |  | 0% |
| 24. | Transformation Overview Diagram | 100% |  | 0% |
| 25. | Cycle and Deadlock Detection | 0% | was not used | 0% |
| 26. | Suggestion of Top Relations | 0% | was not used | 0% |
| 27. | Detection of unused Relations | 100% |  | 0% |
| 28. | Visualization of source and target model | 100% |  | 0% |
| 29. | Green and Yellow markings | 100% |  | 0% |
| 30. | The usage of QVTr's graphical syntax | 100% |  | 0% |
| | <i>Did the offered suggestions...</i> | <i>Yes</i> | | <i>No</i> |
| 31. | help to better understand the used metamodels? | 100% |  | 0% |
| 32. | help to better understand the QVTr language? | 100% |  | 0% |
| 33. | help to prevent errors? | 100% |  | 0% |
| 34. | increase productivity / reduce effort? | 100% |  | 0% |
| | <i>Did the green and yellow markings...</i> | <i>Yes</i> | | <i>No</i> |
| 35. | increase the traceability of objects? | 100% |  | 0% |
| 36. | help to better understand the QVTr language? | 100% |  | 0% |
| 37. | help to identify and locate errors? | 100% |  | 0% |
| 38. | increase productivity / reduce effort? | 100% |  | 0% |
| | <i>Did the graphical syntax...</i> | <i>Yes</i> | | <i>No</i> |
| 39. | make QVTr more readable? | 100% |  | 0% |
| 40. | make QVTr easier to understand? | 100% |  | 0% |
| 41. | ease navigation? | 100% |  | 0% |
| 42. | increase productivity / reduce effort? | 100% |  | 0% |
| | <i>When would you favor QViT over medQVT?</i> | <i>Yes</i> | | <i>No</i> |
| 43. | Novice user in MDE | 50% |  | 50% |
| 44. | Experienced user in MDE | 80% |  | 20% |
| 45. | Novice user in QVTr | 50% |  | 50% |
| 46. | Experienced user in QVTr | 80% |  | 20% |
| 47. | Small-scale project | 70% |  | 30% |
| 48. | Large-scale project | 100% |  | 0% |

Table 5.2: Questionnaire Results

5.4 Observations

The following list summarizes the observations that we have made by (i) observing the participants during the experiment's execution, and (ii) by analyzing the screen recordings.

- Participants did not perceive the text view to be read-only, but expected that changes made to it would be applied to the diagram.
- After generating the target model by clicking on the *generate* button, participants sometimes had trouble to make the connection between the modeling actions that they applied, and the generated target model.
- As expected, participants used different ways of achieving the same goal. This was especially true for navigation, as some participants heavily used the project explorer for opening detail dialogs of relations and transformations, whereas others used the *Edit* menu in the toolbar, and yet others used the context menu of the diagram canvas for it.
- Participants heavily let themselves be guided by the offered quick fixes, read the problem and suggested solution descriptions, and also applied the suggested quick fix.
- Participants heavily benefited from the color-encoded trace model visualization, as they said they better understood why the elements in the target model have been generated.
- An interesting behavior that we have observed is that, as intended, the participants unconsciously adapted themselves to using the naming convention as connoted by the name suggestions. This supports our theory that it is possible for editors to subtly enforce certain conventions on the user. As described in Chapter 4, an editor may support multiple conventions, but choose to select one as the default one. Then, the users either adapt themselves to using that convention, or start using another one. And in the latter case, an editor may ask the user by means of a non-committal, subtle pop-over to use that convention as the new default.
- Some participants assumed that the creation of new object templates is done in the detail dialog of the respective relation, instead of using the OCEs directly on the diagram canvas.
- Some participants were confused about the chains of confirmation alerts, as they did not expect to be prompted again after confirming the respective previous one.
- One participant mentioned that it was pleasing to be notified about when specific names have already been defined by other elements, such as relations or object templates. This would have improved the awareness of what was already defined in the transformation, and it was easier to keep the overview.

5.5 Interpretation

In this section, we interpret our gained data from the user experiments in order to draw conclusions for our last two research questions.

How does our concept perform in practice, compared to a state of the art tool?

How is the Effectiveness influenced? Except for a single case, all participants were able to successfully achieve the task that they were given. We consider this completion rate of 84% of our experiments to be a promising result, taking into account the mixed level of knowledge and experience of the participants with QVTr. In this context, multiple participants argued during the interviews, that they would not have been able to achieve the task using medQVT, because of missing user guidance. In particular, they were referring to all techniques that were about offering suggestions (OCEs, CPMs, Default Names, Auto-Complete, Relation Prediction, Quick Fixes), as well as those about enhanced traceability (trace model visualization, Occurrence Highlighting).

How is the Efficiency influenced? Participants said that our concept significantly reduces the required modeling effort, due to our techniques of automation.

Figure 5.5 interestingly shows that the task of *navigation* nearly took participants the same amount of time as modeling itself. We assume that the reason for that is the high degree of separation in the GUI. There are 3 separate views and 6 different dialogs (some of them also having a tab view) implying continuous focus-switching from view-to-view, view-to-dialog, dialog-to-view, dialog-to-dialog and tab-to-tab. The heatmap of the main view in Figure 5.12 also shows, that the project explorer has been the dominant choice of users to switch between relations, as opposed to clicking on the tab headers. We assume that the reason for that is the directory/folder analogy implemented by the project explorer, as it visualizes the relations in a hierarchical order. By this, it is clearer to the user to which transformation a specific relation belongs to, and this appears more accessible. We also notice that in all heatmaps of dialogs (Figures 5.11, 5.13, 5.14, 5.15), the users have clicked on the window's title bar to re-position it. We assume that this was necessary, because the dialogs hid relevant information that the user needed underneath. In QViT, we have implemented an algorithm, that positions all dialogs in the center of the respective screen, thus mostly covering the diagram canvas of the main view.

On the other side, we notice a comparably small amount of 6% that users spent with the execution of the transformation. This reduced effort is also represented by means of a comparably small user-input effort in Figures 5.7, 5.8 and 5.9. Furthermore, we can also see it in Figure 5.16, as the usage function of the execution view is not as steep as that of the main and dialog views. We believe that the reason for this is our technique of automatic execution profile creation, as well as the option to automatically generate the transformation anew after every change made by the user. By this, fast feedback cycles are provided to the user. We also notice that although the actual *execution* of the

transformation required a relatively low amount of effort, the manual *analyzation* by the user did not, as can be seen in Figure 5.3. There we can see, that users closely spent half the time (19%) viewing the source and target models, in comparison to the time of actual modeling in the main (38%) and dialog (39%) views. However, according to the questionnaire and the interviews, the participants perceived the source, target and trace model visualizations to be a convenient, time-sparing way of understanding the effects of the modeled transformation. In this context, one participant also mentioned that the predictability of certain changes to the (*i.e.* CPMs and editing of primitive properties in object templates) transformation was positively influenced by these visualizations.

The actual modeling of the transformation is done in the main and dialog views, in which participants spent 77% of their time on average. A similar picture is drawn in Figure 5.5, which shows that 67% of the time, participants were busy with actual modeling including error resolving. It is interesting to see, that modeling and error resolving roughly took participants the same amount of time. This shows that, even in spite of preventive error techniques, a significant amount of the total modeling effort is taken by incorporating reported errors. We see this as significant evidence, that it is vital for editors to implement a usability-focused approach that aids the user to (*i*) identify, (*ii*) locate, (*iii*) understand and (*iv*) resolve errors. The fact that participants especially pointed out the usefulness of the quick fixes also supports this claim. In the context of dialogs, the major part was taken by the dialog that allows for the editing of relation details. When we take a look on the heatmap 5.13 for this dialog, we see that what users commonly did was naming the relation, setting its top-property and editing domains. In particular, the name and the type of the domains have been changed quite often. We are tempted to say *unfortunately*, since the naming of domains underlies our techniques of default names and automatic name collision resolving. Yet still, users spent time to edit the domain names. Consequently, we are thinking to take element naming even a step further and fully automate it in future work, wherever applicable. Possible use-cases of this form are the generic names of (primitive) domains and object templates. We also notice an agglomeration in the lower right quarter of the dialog, which we identify as the *Apply* button of modal confirmation prompts, that appear when clicking on the *New Domain* and *New Primitive Domain* buttons. A similar agglomeration of this kind can be found on the heatmap 5.15 of the dialog for editing object template details. Also here were users quite busy with naming the respective object template, which could have been avoided as explained before. It is also interesting to see in Figure 5.16, that modeling was mainly done in dialogs, as users mostly spent their time in dialog views, closely followed by the main view. We see this evolution critically, as our concept intends to concentrate the user's modeling tasks on to the diagram canvas in the main view, and leaving detailed editing in dialogs to be the exception. In reality however, we noticed that it was not obvious to users that they could achieve the same tasks on the main view, without opening a detail dialog. Yet still, the dialogs have been used as the main tool to modify the transformation. We assume that this is because *outward modeling* is an unusual approach to diagram modeling, as users were surprised to discover that the objects on the diagram canvas could be selected with a mouse-click, and thus being

presented with OCEs. It was also not obvious to users that multiple elements could be selected at the same time and thus being presented CPMs. It would be interesting to evaluate techniques and styles in future work, that try to increase the “click-ability” of elements on the canvas.

Comparing mouse and keyboard effort of the actual modeling of a transformation, we see in Figure 5.9 that our concept has successfully outsourced keyboard input into the detail dialogs, while minimizing it in all other views. This separation of editing granularity has also projected a form of consistency on the participants, as they quickly recognized that if keyboard input is required (*e.g.* for naming elements), it has to be done in a detail dialog. This allowed them to quickly accommodate with the GUI of QViT, which was completely new to them. We also notice in the heatmap of the main view in Figure 5.12, that users hardly spent time with the manual arrangement of vertices on the diagram canvas. This is positive evidence that our automatic vertices positioning algorithm has successfully reduced the modeling effort of users due to automation. However, one participant argued that it was confusing that in spite of the types of some object templates being *higher* in the hierarchy according to the metamodel, they still have been automatically positioned *below* their parenting object template.

Finally, we would like to mention that the participants considered OCEs in particular as a significant decrease of the modeling effort, since with them it is not longer required to manually inspect the metamodel documentation, as it is required in medQVT. However, the participants did not perceive the OCEs to increase the learnability of a yet unknown metamodel, in contrast to our expectation.

Since the resolving of reported errors took participants 37% of their time on average (Figure 5.5), we conclude that a usability-focused, corrective error reporting and resolving system like the quick fixes in our concept are vital for editors to have. The participants mentioned that they let themselves be guided by the quick fixes and that this has significantly contributed to working towards the correct end result. In spite of our proposed *preventive* error techniques, users still made many errors, some of them preventing the QVTr execution engine to produce any output model at all. This emphasizes the importance of not only indicating users with the mere existence of errors, but also to offer appropriate suggested solutions, that can be applied in a one-click-fashion. Multiple participants also said that the fact that only 2 clicks are necessary to inspect and apply a quick fix was the main motivation for them to perceive QViT to be *easier* than medQVT.

How is the user Satisfaction influenced? In the beginning of our user experiments, the participants where overwhelmed with the complexity of QVTr, as they were presented an example model transformation using medQVT. They said that during this presentation, they instantly expected QViT to have a much higher degree of user-friendliness. And as it turned out in the questionnaire and the interviews, these expectations were broadly met, as the single major concern they equally had was the still high level of understanding in QVTr needed for effective usage of QViT. One participant pointed out that the high level of automation was a pleasing experience, especially the pre-selection of appropriate relation call parameters. Another participant felt that the tidiness and clearness of the

GUI in QViT also made QVTr appear to be easier to understand, because it was easy to keep the overview. In contrast, another participant mentioned that the learning curve of both editors for novice users would be roughly the same. Taking these observations into account, we conclude that QViT left the participants with a positive feeling about the feasibility of our implemented approach behind QVTr.

Which role does QVTr's graphical concrete syntax play concerning readability and traceability?

The usage of QVTr's GN had a positive influence on the usability in general, according to the questionnaires and interviews. We assume that this is mainly due to our techniques that target to increase the readability, such as the loosened condensity of the presented information on the diagram canvas, the trace model visualization through element-coloring in the diagram canvas, or visual dependency analysis in the execution view. In addition, due to automatic variable management, the participants in the experiment did not have to deal with the declaration and management of variables and their types at all. This has been successfully automated for them in QViT. On the flipside, it came apparent to us during the experiments that the participants were confused about effects of outward modeling. That is, that object templates are being defined through OCEs, for which an already existing object template on the diagram canvas has to be clicked on *before* being presented with the various different OCEs. Similarly, it took some time for the participants to find out about the possibility of multi-selection on the diagram canvas, and thus the advantages of CPMs.

As an implication of these observations, we identify that the mere existence of various different functional behaviors in an editor stays without effect, as long as these existences are poorly communicated to the user. In the case of OCEs and how we have implemented them in QViT, participants initially perceived the pop-up, in which the OCEs are displayed, to be a configuration window for primitive properties of the respectively selected object template. Figure 5.19 illustrates how OCEs of a domain of type `RDBMS::Table` are being presented to the user in QViT.

The fact that some participants tried to apply changes to the modeled transformation through textually changing the textual representation in the text view makes us see the GN of QVTr *not* to be a replacement for the TN, but rather to be complementary. This is contrary to our expectation in the very beginning of this thesis, as we assumed that a stand-alone graphical approach is key to an increase in usability and productivity. But under the assumption that an editor offers two fully editable views using the GN and TN, then we see multiple advantages. On the one hand, an editor of this form is more tolerant towards personal preferences of the user with the type of modeling – be it textually or graphically. Hence, such an editor implements less assumptions than an editor, which strictly enforces either of the two modeling styles. On the other hand, the choice between GN and TN not only affects the way users actively *model* a transformation, but also how they *perceive* and *analyze* it passively. With techniques such as syntax highlighting for

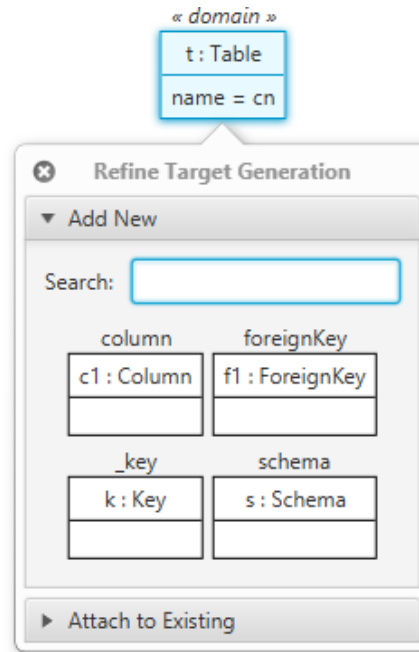


Figure 5.19: OCEs of a RDBMS::Table domain in QViT

the TN, and the loosened information condensity in the GN, we have considered effective techniques to serve the readability needs in both cases in our concept.

5.6 Conclusion

We now summarize our key findings that we gained by the user experiments.

In QViT, mouse-clicks dominate over key-strokes. As Figure 5.6 illustrates, mouse-clicks are the dominant type of user-input in our concept. On the one hand, it is therefore vital for editors to use a clear style for elements that are supposed to be clicked by the user. On the other hand, our heatmaps show the locations in the GUI of QViT, where mouse-click user-input is concentrated the most, and thus the widgets that should require the smallest amount of accommodation time. In the case of QViT, these widgets were mainly textboxes, having an auto-complete dropdown attached to them.

QViT does not take the burden off of knowing QVTr. During the experiments, it came apparent to us that one of the users had quite the hard time to achieve the exercise, due to a lack of understanding of QVTr and no software engineering background. While the concept implemented in QViT theoretically enables the usage of QVTr also for novice users, it is not its key strength. The concepts of QVTr that also other participants struggled with the most were when/where clauses, and the correct usage of variables. In the latter case, it may be confusing what the expression `name = pn` really does, without

considering its surrounding context. If located in an object template that is within in the hierarchy of a *checkonly* domain, then the direction of assignment is left-to-right. This means that the “name” value of the respective object template is stored into variable for later use. However, if located within a hierarchy of an *enforce* domain, the direction of assignment is right-to-left, meaning that the value the variable holds is assigned to the “name” property of the respective object template. From that, we derive that our concept generates the most value for MDE practitioners, that have already modeled QVTr transformations in another editor. In addition, developers of future editors for QVTr and MTLs benefit from our proposed concept as it gives insights into how a high level of usability and productivity can be implemented. In this case, the users benefit from a significant increase in usability and productivity. However, for users that are new to MDE, MTLs and QVTr in particular, our concept is limited in the sense that it does not explain the semantics behind these concepts. One of the participants in our experiments pointed out, that a solution to that might be a guided tour or tutorial within the editor, where the user is guided via explanatory audio-commentary through an example transformation.

QViT is a promising alternative to the medQVT state of the art tool. Our experiments at the LieberLieber Software GmbH company shed light on the potential of our concept. We were granted to get valuable hands-on feedback from industrial MDE practitioners, for which our proposed concept is of professional interest. The promising completion rate of roughly 84% and moderate session duration of 38 minutes on average of our experiments have demonstrated the feasibility of our concept, taking into account the comprehensive task to be achieved, which covered the lion’s share of QVTr concepts. Furthermore, participants have argued in the questionnaire and during the interviews that they would favor QViT over the state of the art tool medQVT for productivity reasons. However, as illustrated in the last questions of the second part of the questionnaire 5.2, participants gave this recommendation with having worked with QVTr before as prerequisite. Therefore we identify such users to be the group, that our concept would have the most impact on in terms of productivity improvement. For completion’s sake we point out that QViT has been implemented in a way such that it is decoupled from an underlying execution engine. Hence, we invite future developers to target their own QVTr execution engines with QViT.

Conclusion

6.1 Summary

In the beginning of this thesis, the main question was to find out about the reasons behind QVTr's poor level of adoption, taking into account its claim to be the de-facto standard for model transformations. During our assessment of respective state of the art tools, we quickly discovered 4 critical shortcomings. That is, although QVTr's initial public release was 10 years ago, the availability of appropriate, *matured* tools is still sparse. Furthermore, the few tools that can be considered mature are focused on enabling the technology rather than being user-centric and usability-focused. In particular, modeling is difficult and tedious in these tools, because feedback cycles are comparably slow. Finally, we discovered unused potential for approachability, readability and traceability with the neglect of QVTr's GN in these editors.

After the tools analysis, we have conducted a comprehensive, theoretical analysis of the modeling process and workflow when using QVTr. Thereby, we have assigned all needed modeling tasks into 4 separate categories, namely modeling itself, the incorporation of reported errors, navigation and the execution of the transformation. We also revealed that the usage of the GN opens doors to novel approaches to the modeling process itself. In particular, we have defined new methods to variable management, layouting, reading directions, condensity, and well-formedness. We then packaged our gained insights from both the tools and modeling process analyses into a theoretical concept, and have evaluated it in an industrial context. By this we have shown, that there is significant potential for an increase in the usability of modeling tools for QVTr, and that a formula of *automation*, *interactivity* and *understandability* can push these boundaries.

We have also elaborated on the question of finding the context, in which our concept generates the most value. Regarding this question, we came to the conclusion, that our concept does not take off the burden of understanding the concepts of QVTr from users,

but provides a significant productivity increase for those that have already modeled transformations in other state of the art tools.

Nonetheless, QViT has pushed the usability boundaries by a margin such that the editor is also feasible for novice users of QVTr, since our proposed concept of quick fixes offer them a certain amount of user guidance, which can be even further improved in future work.

Tables 6.1, 6.2 list all described techniques of our concept along with a concise description of which methods are used to implement them.

| Technique | Achieved By |
|--|---------------------------------------|
| <i>for an increase in readability</i> | |
| Definition and Outlaying of Views | Design Decisions |
| Syntax Highlighting | Type Analysis, Substring Search |
| Domain Numbering | Domain Counting |
| Top Property Indicator | Dynamic Relation Analysis |
| Icons | GUI Design |
| Layouting | GUI Design |
| View Granularity | GUI Design |
| Automatic Vertices Positioning | BFS Heuristics |
| <i>for an increase in traceability</i> | |
| Trace Model Visualization | Color-Encoding & Trace Model Analysis |
| Occurrence Highlighting | Dynamic Relation Analysis |
| Navigable Relation Calls | GUI Design |
| Dependency Graph Visualization | Dependency Graph Analysis |

Table 6.1: Techniques in our approach

| Technique | Achieved By |
|--|---|
| <i>for an increase in automation</i> | |
| Derivation of Model Declarations | Static Metamodel Analysis, Substring Search |
| Derivation of Domain Patterns | Static Metamodel Analysis, Substring Search |
| Derivation of Data Types | Typed Containers |
| Automatic Parameter Selection | Type Derivation, Dynamic Relation Analysis |
| Typed Expression Containers | GUI Behavior |
| Automatic Execution Profile Creation | Heuristic User Action Monitoring, Temporary File Management |
| Automatic Target Model Creation | Undo/Redo System, User Action Monitoring |
| Automatic Vertex Positioning | Sorting Heuristics, Dynamic Relation Analysis |
| Default Names | Naming Conventions, Name Collision Detection |
| Default Top Property | Design Decision |
| e Automatic Variable Management | Dynamic Relation Analysis |
| <i>for an increase in preventive interactivity</i> | |
| Auto-Completion | Static Metamodel Analysis |
| Naming Conventions | Design Decision, Offered Suggestions |
| Common Property Mappings | Static Metamodel Analysis, Dynamic Transformation Analysis |
| Prediction of new Relations | Static Metamodel Analysis, Dynamic Transformation Analysis |
| Concatenations | Static Metamodel Analysis, Dynamic Relation Analysis |
| Type Derivation for Object Templates | Static Metamodel Analysis, Dynamic Relation Analysis |
| References to other Object Templates | Static Metamodel Analysis, Dynamic Relation Analysis |
| <i>for an increase in corrective interactivity</i> | |
| Sanity Checks | Rule Checking |
| Invalid Relation Calls and Queries | Rule Checking |
| Invalid Relation Call Parameters | Rule Checking |
| Invalid Names | Regular Expressions |
| Unbound Enforce Variables | Dynamic Relation Analysis |
| Unbound Containment Relations | Static Metamodel Analysis, Dynamic Transformation Analysis |
| Abstract Type Instantiations | Static Metamodel Analysis |
| Unused Relations | Dependency Graph Analysis |
| Cycles and Deadlocks | Dependency Graph Analysis |
| Suggestion of Top Relations | Dependency Graph Analysis |
| Metamodel Coverage | Static Metamodel Analysis, Dynamic Transformation Analysis |

Table 6.2: Techniques in our approach

6.2 Future Work

Although our concept provides an entry-point to developers of future editors, the user experiments showed that there are still further usability concerns to elaborate on. The following list provides the interested reader with some possible directions to consider.

- Implement dialogs in a way such that they do not hide the diagram canvas underneath, since users may need to have a look on it, before users are able to carry out certain tasks within the dialogs. This came apparent to us, as we noticed that users often re-positioned the dialogs, in order to reveal what was underneath.
- The generic naming of elements such as OTs and (primitive) domains could be fully automated, such that users don't have to spend any time with manual naming.
- It would be interesting to investigate on novel techniques and styles that increase the “click-ability” of elements on the diagram canvas, such that it becomes obvious to users that (multiple) elements can be selected.
- AVP could take the type hierarchy of the underlying metamodel into account, when positioning new object templates either below or above a parenting object template.
- Novice users in MDE, MTLs and QVTr in particular may benefit from a guided tour within the editor, were they are guided through an example model transformation by the means of explanatory audio-commentary. In the best case, the example transformation that would be demonstrated in this tour would elaborate on QVTr concepts in an ascending order, sorted by their complexity. We would define such an order, beginning with the least complex topic, as follows.
 1. Usage of Metamodels in the context of QVTr
 2. Creation of Transformations
 3. Creation of Model Declarations
 4. Creation of Relations
 5. Top-Property of Relations
 6. Creations of Domain Declarations
 7. Object Templates
 8. Variables and Primitive Data Types
 9. One-To-One Mappings
 10. Nesting of Object Templates
 11. When and Where Clauses
 12. Relation Calls, Queries
 13. Integration of OCL expressions

- A useful improvement to the text view would be a system such that textual changes made in this text view are automatically applied to the diagram canvas. At the time of writing this thesis, changes are unidirectional from the diagram canvas towards the text view in QViT.
- In order to avoid confusion about what the most recent action was that the user performed, a new view is conceivable, in which a history of all executed actions is presented. This should also increase the traceability. This history is then also automatically updated whenever the user undo's or redo's a specific action. Another advantage of this is that the learnability of QVTr is potentially affected, since it is easier for users to understand the effects of certain changes that they have made to the transformation.
- For novice users to QVTr it may potentially be confusing that a “when” clause refers to a *pre*-condition, unlike a “where” clause, that defines a *post*-condition. It is conceivable for editors to use the terms “pre-” and “post-condition” instead for buttons and widgets in the interface, such that this confusion is mitigated.
- In order to avoid unexpected chains of popping up dialogs, a wizard-like dialog could be used instead. We consider this an effective solution since wizard also reflect the information to users, in which step they currently are, and especially *when* the last step of the procedure has been reached. Hence, any confusion about if there are still further steps to handle in modal dialogs is eliminated.
- In order to emphasize that the editing of transformations, relations, and object templates in dialogs is meant to be for *details*, and that instead the editing of these elements on the diagram canvas is the main way of editing, it is conceivable to also label buttons that open the dialogs with the term “Details”, rather than “Edit”. This technique would also support the “click-ability” of the vertices on the diagram, and thus novice users of the respective editor may find out faster about OCEs and CPMs, which appear only after vertices selection.

List of Figures

| | | |
|------|---|----|
| 1.1 | PIM to PSM model transformation | 4 |
| 2.1 | Example of a QVTr relation modeled with the VMTS Plug-In [28] | 10 |
| 2.2 | An editor to define model queries in a visual way using an adapted version of QVTr's graphical syntax [29,30] | 12 |
| 2.3 | Visual comparison of bidirectional MTLs [26] | 13 |
| 3.1 | Transformations of our interest have exactly one source & target model . . . | 16 |
| 3.2 | The execution direction of our interest is from source to target | 17 |
| 3.3 | The Model Transformation Pipeline [11] | 18 |
| 3.4 | Categorization of tasks in the modeling process of QVTr | 19 |
| 3.5 | Example query to convert QVTr data types into SQL format | 22 |
| 3.6 | <i>Just-in-time</i> query definition | 22 |
| 3.7 | <i>Just-in-time</i> parameter definition | 23 |
| 3.8 | Example usage of a primitive domain | 25 |
| 3.9 | Mapping of the primitive name property with OTs | 26 |
| 3.10 | Binding of an unbound containment relation | 26 |
| 3.11 | Binding of an unbound containment relation | 26 |
| 3.12 | Traceability links between the source and target model | 31 |
| 3.13 | Example of a 1:1 mapping with the <code>PackageToSchema</code> relation | 32 |
| 3.14 | Example of the branching modeling pattern | 33 |
| 3.15 | The <code>AttributeToColumn</code> relation in our example | 33 |
| 3.16 | The <code>PrimitiveAttributeToColumn</code> relation in our example | 33 |
| 3.17 | Example of a model enrichment | 34 |
| 3.18 | Example of the merging pattern | 34 |
| 3.19 | Example of the shared parent pattern | 35 |
| 3.20 | The standardized visual representation of relations using the GN | 36 |
| 4.1 | Color encoding representing the generated trace model | 41 |
| 4.2 | The boxmodel of AVP | 49 |
| 4.3 | Example of the general, naive AVP algorithm | 50 |
| 4.4 | Example of the AVP algorithm for OCEs | 51 |
| 4.5 | Example of the AVP algorithm for dependency graph visualizations | 51 |
| 4.6 | Example of a common property mapping | 54 |

| | | |
|------|---|-----|
| 4.7 | The workflow of how One-Click-Extensions (OCE) are used | 60 |
| 4.8 | The workflow of referencing OTs to other OTs | 61 |
| 4.9 | Visual indicator that points to OTs, which are out-of-sight in the viewport . . | 61 |
| 4.10 | A clear and focused layout for the required main view | 69 |
| 4.11 | Syntax highlighting for QVTr's GN | 70 |
| 4.12 | Syntax highlighting for QVTr's TN | 71 |
| 4.13 | Visualizing the order of domains in a relation with numbering | 71 |
| 4.14 | An example of Domain Kind Sensitive Captions (DKSC) | 71 |
| 4.15 | An indicator for the top property beside the name of a relation | 72 |
| 4.16 | No indicator for the top property beside the name of a relation | 72 |
| 4.17 | A landing page communicates necessary actions to the user | 72 |
| 4.18 | A relation that maps classes with attributes to columns in tables. | 74 |
| 4.19 | Objects <i>selected</i> by the AttributeToColumn relation in a <i>source</i> model | 74 |
| 4.20 | Objects <i>generated</i> by the AttributeToColumn relation in a <i>target</i> model | 74 |
| 4.21 | Occurrences of the <i>p</i> object template are highlighted on mouse-hover | 75 |
| 4.22 | Occurrences of the <i>cn</i> variable are highlighted on mouse-hover | 75 |
| 4.23 | Hyperlink-behavior for relation calls | 75 |
| 4.24 | Overview of the call hierarchy among multiple relations | 76 |
| 5.1 | The source model to be transformed | 80 |
| 5.2 | The target model to generate from the source model | 80 |
| 5.3 | Time spent in Views | 84 |
| 5.4 | Time spent in Dialogs | 84 |
| 5.5 | Time spent with Modeling Tasks | 84 |
| 5.6 | User Input Distribution | 84 |
| 5.7 | Mouse Input Distribution | 84 |
| 5.8 | Keyboard Input Distribution | 84 |
| 5.9 | Mouse/Keyboard Distribution | 84 |
| 5.10 | Heatmap over the Execution View | 85 |
| 5.11 | Heatmap over the Text View | 86 |
| 5.12 | Heatmap over the Main View | 86 |
| 5.13 | Heatmap over the Relation Details Dialog | 87 |
| 5.14 | Heatmap over the Transformation Details Dialog | 87 |
| 5.15 | Heatmap over the Object Template Details Dialog | 88 |
| 5.16 | View-usage over time of different participants | 89 |
| 5.17 | Cumulative user-input over time of different participants | 90 |
| 5.18 | Word cloud representing the results of our questionnaire and interviews | 91 |
| 5.19 | OCEs of a RDBMS::Table domain in QViT | 98 |
| A.1 | The SimpleRDBMS metamodel as defined in [41] | 115 |
| A.2 | The SimpleUML metamodel as defined in [41] | 116 |
| B.1 | The call hierarchy of relations in the transformation | 117 |
| B.2 | Input: SimpleUML Model | 118 |

| | | |
|-----|-------------------------------------|-----|
| B.3 | Output: SimpleRDBMS Model | 118 |
|-----|-------------------------------------|-----|

List of Tables

| | | |
|-----|---|-----|
| 4.1 | Required views in our concept | 70 |
| 5.1 | Questionnaire Results | 91 |
| 5.2 | Questionnaire Results | 92 |
| 6.1 | Techniques in our approach | 102 |
| 6.2 | Techniques in our approach | 103 |
| C.1 | Questionnaire | 119 |
| C.2 | Questionnaire | 120 |

Bibliography

- [1] LieberLieber Software GmbH. <https://www.lieberlieber.com/en/>, 2016. [Online; Accessed: 13.10.2016].
- [2] Tata Consultancy Services Ltd. ModelMorf. https://web.archive.org/web/20120323171429/http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm, Online; Accessed: 10.02.2018.
- [3] Abrahão, Silvia and Bordeleau, Francis and Cheng, Betty and Kokaly, Sahar and Paige, Richard F and Störrle, Harald and Whittle, Jon. User Experience for Model-Driven Engineering: Challenges and Future Directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 229–236. IEEE, 2017.
- [4] Arendt, Thorsten and Biermann, Enrico and Jurack, Stefan and Krause, Christian and Taentzer, Gabriele. Henshin: advanced concepts and tools for in-place EMF model transformations. In *2010 ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 121–135. Springer, 2010.
- [5] Babicki, Sasha and Arndt, David and Marcu, Ana and Liang, Yongjie and Grant, Jason R and Maciejewski, Adam and Wishart, David S. Heatmapper: web-enabled heat mapping for all. *Nucleic acids research*, 44(W1):W147–W153, 2016.
- [6] Balogh, Zoltán and Varró, Dániel. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2009.
- [7] C. Kotronis and A. Tsadimas and G. D. Kapos and V. Dalakas and M. Nikolaidou and D. Anagnostopoulos. Simulating SysML transportation models. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 001674–001679, 2016.
- [8] Caldiera, VRBG and Rombach, H Dieter. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.
- [9] Cunha, Alcino and Garis, Ana and Riesco, Daniel. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, 14(1):5–25, 2015.

- [10] Czarnecki, Krzysztof and Helsen, Simon. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.
- [11] Czarnecki, Krzysztof and Helsen, Simon. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
- [12] Dan, Li and Danning, Li. Towards a Formal Behavioral Semantics for UML Interactions. In *Proceedings of the 2010 Third International Symposium on Information Science and Engineering, ISISE '10*, pages 213–218, Washington, DC, USA, 2010. IEEE Computer Society.
- [13] Eclipse QVTd (QVT Declarative). The Eclipse Foundation. <https://projects.eclipse.org/projects/modeling.mmt.qvtd>, 2010. [Online; Accessed: 27.03.2018].
- [14] EcoreTools - Graphical Modeling for Ecore. The Eclipse Foundation. <http://www.eclipse.org/ecoretools/>, 2016. [Online; Accessed: 13.10.2016].
- [15] Greenyer, Joel and Kindler, Ekkart. Reconciling TGGs with QVT. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MODELS'07*, pages 16–30, Berlin, Heidelberg, 2007. Springer-Verlag.
- [16] Hirschberg, Daniel S. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.
- [17] Holzinger, Andreas. Usability engineering methods for software developers. *Communications of the ACM*, 48(1):71–74, 2005.
- [18] Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts, 2017.
- [19] Jouault, Frédéric and Allilaire, Freddy and Bézivin, Jean and Kurtev, Ivan and Valduriez, Patrick. ATL: A QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06*, pages 719–720, New York, NY, USA, 2006. ACM.
- [20] Kappel, Gerti and Langer, Philip and Retschitzegger, Werner and Schwinger, Wieland and Wimmer, Manuel. Model transformation by-example: a survey of the first wave. In *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215. Springer, 2012.
- [21] Kessentini, Marouane and Sahraoui, Houari and Boukadoum, Mounir and Omar, Omar Ben. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

-
- [22] Kiegeland, Jörg and Eichler, Hajo. Enabling comprehensive tool support for QVT. *Eclipse Summit Europe*, 2007.
 - [23] Kleppe, Anneke G and Warmer, Jos B and Bast, Wim. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.
 - [24] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. *The Epsilon Transformation Language*, pages 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
 - [25] Kusel, Angelika and Schwinger, Wieland and Wimmer, Manuel and Retschitzegger, Werner. Common Pitfalls of Using QVT Relations - Graphical Debugging As Remedy. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '09, pages 329–334, Washington, DC, USA, 2009. IEEE Computer Society.
 - [26] L. Samimi-Dehkordi and B. Zamani and S. Kolahdouz-Rahimi. Bidirectional model transformation approaches a comparative study. In *2016 6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 314–320, Oct 2016.
 - [27] Langer, Philip. *Adaptable model versioning based on model transformation by demonstration*. PhD thesis.
 - [28] Lengyel, László and Madari, István and Asztalos, Márk and Levendovszky, Tihamér. Validating Query/View/Transformation Relations. In *Proceedings of the 2010 Workshop on Model-Driven Engineering, Verification, and Validation*, MODEVVA '10, pages 7–12, Washington, DC, USA, 2010. IEEE Computer Society.
 - [29] Li, Dan and Li, Xiaoshan and Stolz, Volker. QVT-based Model Transformation Using XSLT. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, Jan. 2011.
 - [30] Li, Dan and Li, Xiaoshan and Stolz, Volker. Model Querying with Graphical Notation of QVT Relations. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012.
 - [31] Lieberman, Henry. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.
 - [32] Macedo, Nuno and Cunha, Alcino. Implementing QVT-R bidirectional model transformations using Alloy. In *International Conference on Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.
 - [33] Macedo, Nuno and Guimaraes, Tiago and Cunha, Alcino. Model repair and transformation with Echo. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 694–697. IEEE, 2013.
 - [34] mediniQVT. ikv++ Technologies. <http://projects.ikv.de/qvt/>, 2016. [Online; Accessed: 13.10.2016].

- [35] Mens, Tom and Van Gorp, Pieter. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [36] N. Debnath and C. A. Martinez and F. Zorzan and D. Riesco and G. Montejano. Transformation of business process models BPMN 2.0 into components of the Java business platform. In *IEEE 10th International Conference on Industrial Informatics*, pages 1035–1040, July 2012.
- [37] Nielsen, Jakob. Ten Usability Heuristics. <http://www.nngroup.com/articles/ten-usability-heuristics/>, 2005. [Online; Accessed: 19.12.2013].
- [38] Object Management Group. Meta Object Facility. <http://www.omg.org/spec/MOF>, 2016. [Online; Accessed: 11.02.2018].
- [39] Object Management Group. The Architecture of Choice for a Changing World. <http://www.omg.org/mda/>, 2016. [Online; Accessed: 13.10.2016].
- [40] Object Management Group. Object Constraint Language. <http://www.omg.org/spec/OCL/>, Online; Accessed: 10.02.2018.
- [41] Object Management Group. Query/View/Transformation Specification. <http://www.omg.org/spec/QVT/>, Online; Accessed: 13.10.2016.
- [42] Object Management Group (OMG). XML Metadata Interchange. www.omg.org/spec/XMI, 2015. [Online; Accessed: 12.02.2018].
- [43] Oracle Corporation. JavaFX. <http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html>, 2018. [Online; Accessed: 12.02.2018].
- [44] Rentschler, Andreas and Noorshams, Qais and Happe, Lucia and Reussner, Ralf. Interactive visual analytics for efficient maintenance of model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 141–157. Springer, 2013.
- [45] Roubi, Sarra and Erramdani, Mohammed and Mbarki, Samir. A Model Driven Approach based on Interaction Flow Modeling Language to generate Rich Internet Applications. *International Journal of Electrical and Computer Engineering (IJECE)*, 6(6):3073–3079, 2016.
- [46] Schürr, Andy. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.
- [47] Schütze, Lars and Wilke, Claas and Demuth, Birgit. Tool-Supported Step-By-Step Debugging for the Object Constraint Language. In *OCL@ MoDELS*, pages 73–82, 2013.

-
- [48] Sparx Systems. Enterprise Architect. <http://www.sparxsystems.eu/start/home/>, 2008. [Online; Accessed: 13.10.2016].
 - [49] Strüber, Daniel and Born, Kristopher and Gill, Kanwal Daud and Groner, Raffaela and Kehrer, Timo and Ohrndorf, Manuel and Tichy, Matthias. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. 10th International Conference on Graph Transformation, 2017.
 - [50] Sun, Yu and White, Jules and Gray, Jeff. Model transformation by demonstration. In *International Conference on Model Driven Engineering Languages and Systems*, pages 712–726. Springer, 2009.
 - [51] Terekhov, Andrey and Bryksin, Timofey and Litvinov, Yurii. How to make visual modeling more attractive to software developers. In *Present and Ulterior Software Engineering*, pages 139–152. Springer, 2017.
 - [52] The Eclipse Foundation. Eclipse Modeling Framework (EMF). <http://www.eclipse.org/modeling/emf/>, Online; Accessed: 11.02.2018.
 - [53] The Eclipse Foundation. Graphical Modeling Framework (GMF). <http://www.eclipse.org/modeling/gmf/>, Online; Accessed: 11.02.2018.
 - [54] Topcased UML Editor. PolarSys. <https://www.polarsys.org/topcased>, 2016. [Online; Accessed: 13.10.2016].
 - [55] Varró, Dániel. Model transformation by example. In *International Conference on Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.
 - [56] Willink, Edward D. On Re-use of OCL for QVT Model-checking Editors. Technical report, Technical report, Eclipse UMLX Project, Jun 2007. Available at <http://www.eclipse.org/gmt/umlx/doc/MDDTIF07/MDD-TIF07-QVTEditors.pdf>.
 - [57] Willink, Edward D. Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation. In *EXE@ MoDELS*, pages 26–32, 2016.
 - [58] Willink, Edward D. Optimized Declarative Transformation: First Eclipse QVTc Results. In *BigMDE@ STAF*, pages 47–56, 2016.
 - [59] Willink, Edward D. The Micromapping Model of Computation; The Foundation for Optimized Execution of Eclipse QVTc/QVTr/UMLX. In *International Conference on Theory and Practice of Model Transformations*, pages 51–65. Springer, 2017.
 - [60] Wimmer, Manuel and Kusel, Angelika and Schoenboeck, Johannes and Kappel, Gerti and Retschitzegger, Werner and Schwinger, Wieland. Reviving QVT Relations: Model-Based Debugging Using Colored Petri Nets. In *Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, pages 727–732. Springer Berlin Heidelberg, 2009.

- [61] Wohlin, Claes and Runeson, Per and Höst, Martin and Ohlsson, Magnus C and Regnell, Björn and Wesslén, Anders. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [62] World Wide Web Consortium (W3C). Extensible Markup Language (XML). <https://www.w3.org/TR/2006/REC-xml11-20060816/>, 2006. [Online; Accessed: 12.02.2018].
- [63] Yujian, Li and Bo, Liu. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

Diagrams of Metamodels

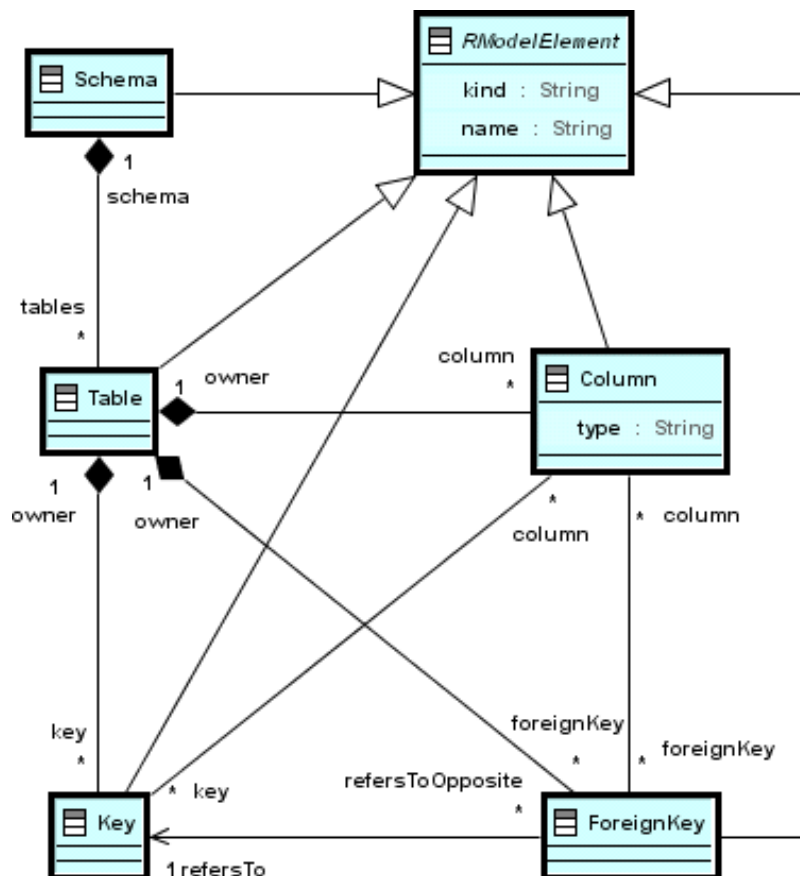
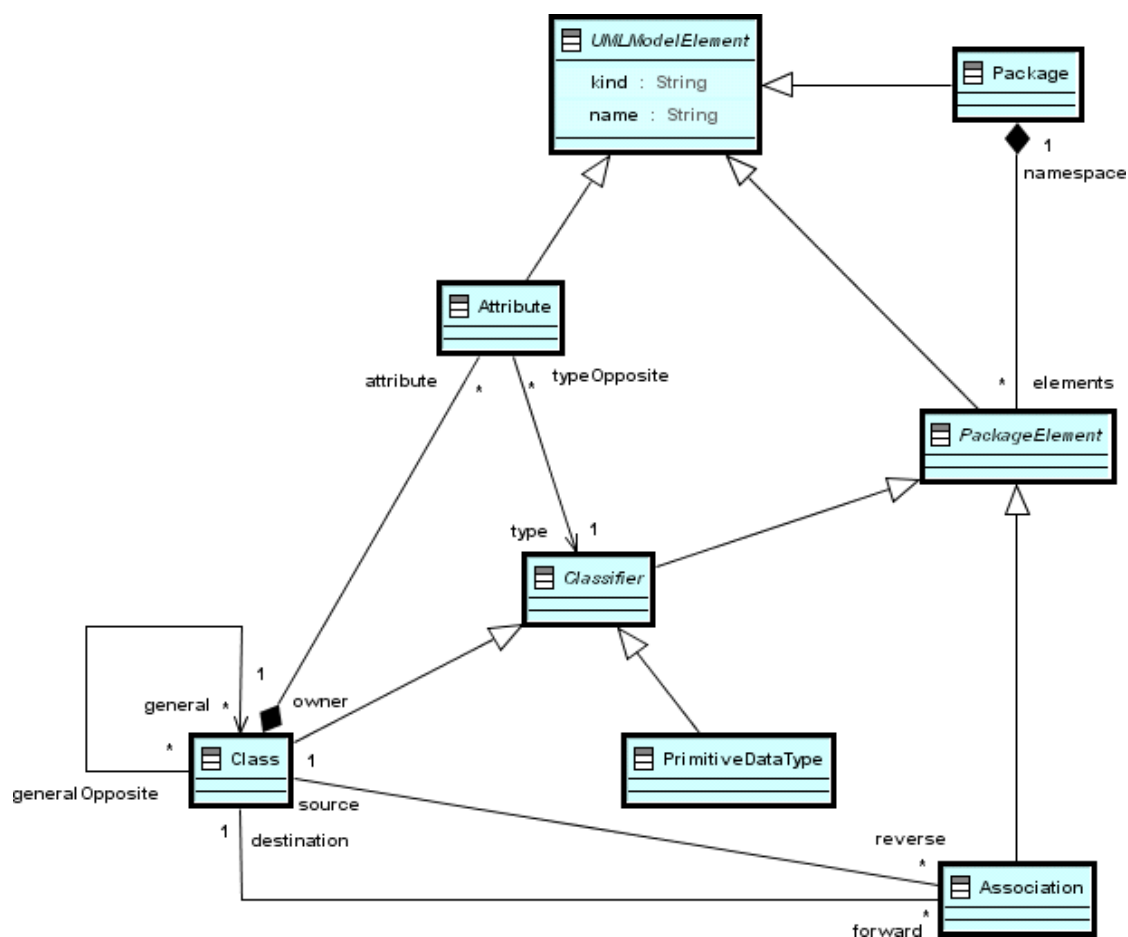


Figure A.1: The SimpleRDBMS metamodel as defined in [41]



User Experiment Exercise

Please use the QViT editor to specify the “UMLToRDBMS” transformation, which comprises the following 3 relations.

1. PackageToSchema
2. ClassToTable
3. AttributeToColumn

The call hierarchy of these relations is defined in Figure B.1.

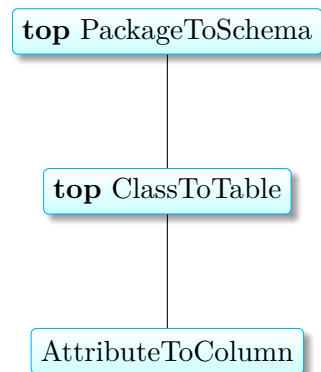


Figure B.1: The call hierarchy of relations in the transformation

The source model, which serves as the input to the transformation, is given as follows.

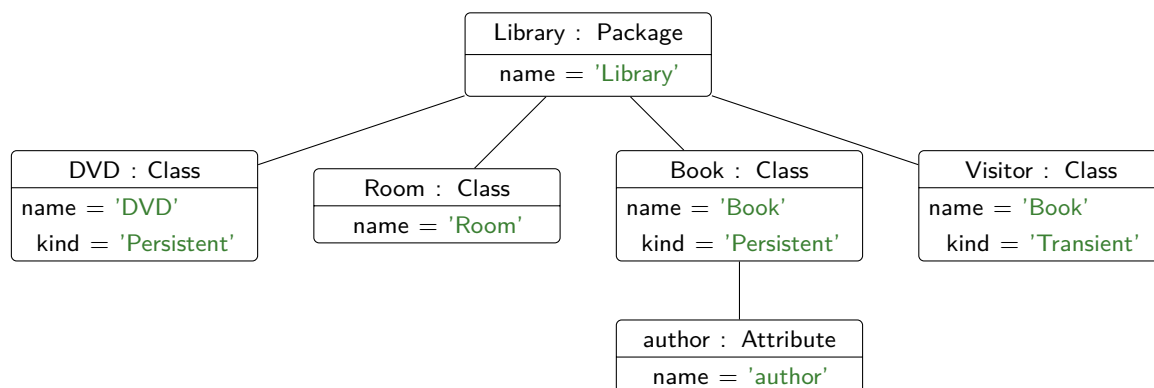


Figure B.2: Input: SimpleUML Model

Your modeled transformation should transform the given source model into the following target model.

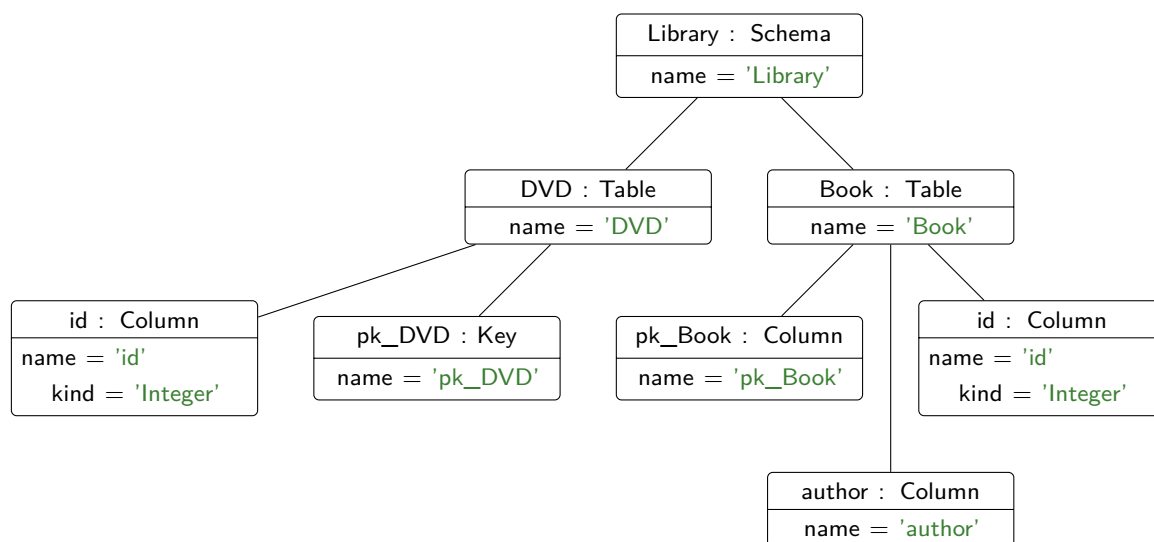


Figure B.3: Output: SimpleRDBMS Model

After the test session you will have the opportunity to compare QViT with medini QVT in the context of an online questionnaire and an interview.

Thank you for your time and interest in the user experiments.

User Experiment Questionnaire

Please fill out the following questionnaire, in order to compare the *textual* modeling approach with medini QVT against *graphical* modeling with QViT.

| | <i>Was using QViT...</i> | <i>Yes</i> | <i>No</i> |
|-----|--|--------------------------|--------------------------|
| 1. | easier | <input type="checkbox"/> | <input type="checkbox"/> |
| 2. | visually better structured and clearer | <input type="checkbox"/> | <input type="checkbox"/> |
| 3. | visually more appealing | <input type="checkbox"/> | <input type="checkbox"/> |
| 4. | more restrictive | <input type="checkbox"/> | <input type="checkbox"/> |
| 5. | more repetitive | <input type="checkbox"/> | <input type="checkbox"/> |
| 6. | faster to use | <input type="checkbox"/> | <input type="checkbox"/> |
| 7. | easier to navigate | <input type="checkbox"/> | <input type="checkbox"/> |
| 8. | easier to edit | <input type="checkbox"/> | <input type="checkbox"/> |
| 9. | faster to learn | <input type="checkbox"/> | <input type="checkbox"/> |
| 10. | more time-consuming | <input type="checkbox"/> | <input type="checkbox"/> |
| 11. | more readable | <input type="checkbox"/> | <input type="checkbox"/> |
| 12. | easier for learning QVTr | <input type="checkbox"/> | <input type="checkbox"/> |
| 13. | easier to understand | <input type="checkbox"/> | <input type="checkbox"/> |
| | <i>What decreased the modeling effort in QViT?</i> | <i>Yes</i> | <i>No</i> |
| 14. | Model Derivation from Transformation names | <input type="checkbox"/> | <input type="checkbox"/> |
| 15. | Domain Derivation from Relation names | <input type="checkbox"/> | <input type="checkbox"/> |
| 16. | Suggestion of Default Names | <input type="checkbox"/> | <input type="checkbox"/> |
| 17. | Automatic parameter-selection in Relation calls | <input type="checkbox"/> | <input type="checkbox"/> |

Table C.1: Questionnaire

| | | | |
|---|--|--------------------------|--------------------------|
| 18. | Automatic configuration of the execution profile | <input type="checkbox"/> | <input type="checkbox"/> |
| 19. | Auto-Complete for names | <input type="checkbox"/> | <input type="checkbox"/> |
| 20. | One-Click-Extensions | <input type="checkbox"/> | <input type="checkbox"/> |
| 21. | Prediction of new Relations | <input type="checkbox"/> | <input type="checkbox"/> |
| 22. | Quick Fixes | <input type="checkbox"/> | <input type="checkbox"/> |
| 23. | Common Property Mappings | <input type="checkbox"/> | <input type="checkbox"/> |
| 24. | Transformation Overview Diagram | <input type="checkbox"/> | <input type="checkbox"/> |
| 25. | Cycle and Deadlock Detection | <input type="checkbox"/> | <input type="checkbox"/> |
| 26. | Suggestion of Top Relations | <input type="checkbox"/> | <input type="checkbox"/> |
| 27. | Detection of unused Relations | <input type="checkbox"/> | <input type="checkbox"/> |
| 28. | Visualization of source and target model | <input type="checkbox"/> | <input type="checkbox"/> |
| 29. | Green and Yellow markings | <input type="checkbox"/> | <input type="checkbox"/> |
| 30. | The usage of QVTr's graphical syntax | <input type="checkbox"/> | <input type="checkbox"/> |
| <i>Did the offered suggestions...</i> | | <i>Yes</i> | <i>No</i> |
| 31. | help to better understand the used metamodels? | <input type="checkbox"/> | <input type="checkbox"/> |
| 32. | help to better understand the QVTr language? | <input type="checkbox"/> | <input type="checkbox"/> |
| 33. | help to prevent errors? | <input type="checkbox"/> | <input type="checkbox"/> |
| 34. | increase productivity / reduce effort? | <input type="checkbox"/> | <input type="checkbox"/> |
| <i>Did the green and yellow markings...</i> | | <i>Yes</i> | <i>No</i> |
| 35. | increase the traceability of objects? | <input type="checkbox"/> | <input type="checkbox"/> |
| 36. | help to better understand the QVTr language? | <input type="checkbox"/> | <input type="checkbox"/> |
| 37. | help to identify and locate errors? | <input type="checkbox"/> | <input type="checkbox"/> |
| 38. | increase productivity / reduce effort? | <input type="checkbox"/> | <input type="checkbox"/> |
| <i>Did the graphical syntax...</i> | | <i>Yes</i> | <i>No</i> |
| 39. | make QVTr more readable? | <input type="checkbox"/> | <input type="checkbox"/> |
| 40. | make QVTr easier to understand? | <input type="checkbox"/> | <input type="checkbox"/> |
| 41. | ease navigation? | <input type="checkbox"/> | <input type="checkbox"/> |
| 42. | increase productivity / reduce effort? | <input type="checkbox"/> | <input type="checkbox"/> |
| <i>When would you favor QViT over medini QVT?</i> | | <i>Yes</i> | <i>No</i> |
| 43. | Novice user in MDE | <input type="checkbox"/> | <input type="checkbox"/> |
| 44. | Experienced user in MDE | <input type="checkbox"/> | <input type="checkbox"/> |
| 45. | Novice user in QVTrs | <input type="checkbox"/> | <input type="checkbox"/> |
| 46. | Experienced user in QVTr | <input type="checkbox"/> | <input type="checkbox"/> |
| 47. | Small-scale project | <input type="checkbox"/> | <input type="checkbox"/> |
| 48. | Large-scale project | <input type="checkbox"/> | <input type="checkbox"/> |

Table C.2: Questionnaire