Model Manipulation for End-User Modelers

Acretoaie, Vlad

Publication date: 2016

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Model Manipulation for End-User Modelers

Vlad Acrețoaie

DTU

Kongens Lyngby 2016
PhD-2016-405
Summary

End-user modelers are domain experts who create and use models as part of their work. They are typically not Software Engineers, and have little or no programming and meta-modeling experience. However, using model manipulation languages developed in the context of Model-Driven Engineering often requires such experience. These languages are therefore only used by a small subset of the modelers that could, in theory, benefit from them.

The goals of this thesis are to substantiate this observation, introduce the concepts and tools required to overcome it, and provide empirical evidence in support of these proposals. To achieve its first goal, the thesis presents the findings of a Systematic Mapping Study showing that human factors topics are scarcely and relatively poorly addressed in model transformation research. Motivated by these findings, the thesis explores the requirements of end-user modelers, and proposes the VM* family of model manipulation languages addressing them. This family consists of the Visual Model Query Language (VMQL), the Visual Model Constraint Language (VMCL), and the Visual Model Transformation Language (VMTL). They allow modelers to specify and execute queries, constraints, and transformations using their modeling notation and editor of choice.

The VM* languages are implemented via a single execution engine, the VM* Runtime, built on top of the Henshin graph-based transformation engine. This approach combines the benefits of flexibility, maturity, and formality. To simplify model editor integration, the VM* Runtime is deployed as a collection of lightweight Web Services. The claim that VM* languages offer end-user modelers superior learnability compared to existing model manipulation languages is verified empirically via user experiments complemented by qualitative evidence.


This thesis was submitted to the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for the degree of PhD in Computer Science. The PhD program was funded through a full scholarship from the Department of Applied Mathematics and Computer Science.

The thesis deals with model manipulation languages and tools for end-user modelers. Motivated by the outcomes of a literature review on human factors in model transformation, a new family of usability-oriented model manipulation languages is proposed. The family includes languages for model querying, constraint specification, and transformation. The usability of the proposed languages is evaluated empirically, and the tools supporting them are described.

The work presented in this thesis has been conducted under the supervision of Associate Professor Harald Störrle.

Kongens Lyngby, 14 February 2016

Vlad Acretoaie
I would like to thank my supervisor, Harald Störrle, for his guidance and always insightful feedback on my work, as well as numerous thought-provoking discussions. His door has always been open to me, while at the same time I have enjoyed significant freedom to pursue my own ideas. Throughout the past three years he has been my role model regarding the ethics and practice of being a scientist, and he will more than likely continue to be so in the future.

I would also like to thank Gabriele Taentzer for welcoming me as a visitor in her research group at Philipps-Universität Marburg. The time spent there under her guidance has been an invaluable contribution to both my PhD work and my development as a researcher. I am also grateful to Thorsten Arendt, Daniel Strüber, Kristopher Born, and Nebras Nassar, all of whom have helped make my stay in Marburg in equal measure productive and enjoyable.

My work has benefited from the feedback and expertise of other members of the Software Engineering section at DTU Compute. I would like to express my gratitude to Ekkart Kindler and Hubert Baumeister for sharing their thoughts on my work, and allowing me to conduct empirical studies in their classes. I also thank Anne Haxthausen for helping me translate the summary of this thesis to Danish. Finally, I thank Bahram Zarrin, Linh Vu Hong, and Jóhan Davidsen for having been the best office mates one could ask for.

The fact that I look back on my time as a PhD student as an extraordinary experience is also due to having great friends to turn to for advice and encouragement, both here in Denmark and at home in Romania.

Last but certainly not least, I would like to thank my mother. I owe all of my achievements to her love and support.


# Contents

Summary

Resumé

Preface

Acknowledgements

Publications

Contents

Abbreviations

1 Introduction

1.1 Problem Statement

1.2 A Vision for End-User Modelers

1.3 Contributions

1.4 Thesis Outline

2 Models and End-User Modelers

2.1 Models in Software Engineering and Beyond

2.1.1 What Is a Model?

2.1.2 Models in Software Engineering

2.1.3 Models in Neighboring Disciplines

2.2 End-User Modelers

2.2.1 Definition

2.2.2 High-Level Requirements
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Related Work</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Model Querying</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Software Model Querying</td>
<td>24</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Business Process Model Querying</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Model Constraints</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Software Model Constraints</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Business Process Model Constraints</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Model Transformation</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Textual MTLs</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Abstract Syntax Visual MTLs</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Concrete Syntax Visual MTLs</td>
<td>39</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Model Transformation By-Example</td>
<td>41</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Executing Model Transformations</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>Human Factors in Model Transformation</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Motivation and Research Questions</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Research Method</td>
<td>48</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Study Search and Selection</td>
<td>48</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Data Extraction and Quality Assessment</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Results</td>
<td>51</td>
</tr>
<tr>
<td>4.3.1</td>
<td>RQ 1 – Human Factors in MT Literature</td>
<td>52</td>
</tr>
<tr>
<td>4.3.2</td>
<td>RQ 2 – Languages and Tools</td>
<td>56</td>
</tr>
<tr>
<td>4.3.3</td>
<td>RQ 3 – Evaluation Methods and Quality Assessment</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Threats to Validity</td>
<td>62</td>
</tr>
<tr>
<td>4.5</td>
<td>Discussion</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>The VM* Family of Model Manipulation Languages</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>Transparent Model Manipulation</td>
<td>68</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Syntax Transparency</td>
<td>68</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Environment Transparency</td>
<td>69</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Execution Transparency</td>
<td>70</td>
</tr>
<tr>
<td>5.2</td>
<td>An Example Modeling Scenario</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>The Visual Model Query Language</td>
<td>73</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Introductory Examples</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2</td>
<td>The Structure of a VMQL Query</td>
<td>77</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Annotation Syntax</td>
<td>81</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Operational Semantics</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>The Visual Model Constraint Language</td>
<td>87</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Introductory Examples</td>
<td>87</td>
</tr>
<tr>
<td>5.4.2</td>
<td>The Structure of a VMCL Constraint</td>
<td>89</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Annotation Syntax</td>
<td>90</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Operational Semantics</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>The Visual Model Transformation Language</td>
<td>92</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Introductory Examples</td>
<td>92</td>
</tr>
</tbody>
</table>
## CONTENTS

5.5.2 The Structure of a VMTL Transformation ..................... 98  
5.5.3 Annotation Syntax ........................................... 101  
5.5.4 Operational Semantics ...................................... 102  
5.6 Practical Considerations ........................................ 104  
5.6.1 Host Language Requirements ................................. 105  
5.6.2 Execution Engine Requirements .............................. 106  

6 Engineering Evaluation ........................................... 107  
6.1 Executing VM* Specifications ................................... 107  
6.1.1 Executing VMTL Transformations ............................ 108  
6.1.2 Executing VMQL queries and VMCL constraints ............ 111  
6.2 Model Manipulation as a Service ............................... 115  
6.3 Tool Support Limitations ....................................... 119  

7 Human Factors Evaluation ......................................... 123  
7.1 Evaluation Plan .................................................. 123  
7.2 Experiment on the Learnability of VMQL ....................... 125  
7.2.1 Methods and Materials ...................................... 125  
7.2.2 Observations ................................................. 127  
7.2.3 Interpretation ................................................ 129  
7.2.4 Threats to Validity ......................................... 130  
7.3 Experiments on the Learnability of VMTL ...................... 131  
7.3.1 Methods and Materials ...................................... 131  
7.3.2 Observations ................................................. 135  
7.3.3 Interpretation ................................................ 140  
7.3.4 Threats to Validity ......................................... 141  
7.4 Think-Aloud Protocol Analysis of the Learnability of VMTL 143  
7.4.1 Methods and Materials ...................................... 143  
7.4.2 Results and Interpretation ................................... 145  
7.4.3 Threats to Validity ......................................... 146  

8 Conclusion ......................................................... 149  
8.1 Summary of Results .............................................. 149  
8.2 Lessons Learned ................................................ 152  
8.3 Future Work ..................................................... 154  

A List of Publications Included in the SMS ....................... 159  

B Experiment Questionnaires ......................................... 181  
B.1 VMQL Learnability User Experiment ............................ 181  
B.2 VMTL Learnability User Experiment 1 .......................... 192  
B.3 VMTL Learnability User Experiment 2 .......................... 203  

Bibliography .......................................................... 235
Abbreviations

AD – UML Activity Diagram
ANOVA – Analysis of Variance
AOM – Aspect Oriented Modeling
API – application programming interface
ATL – Atlas Transformation Language
BPEL – Business Process Execution Language
BPM – Business Process Management
BPMN – Business Process Model and Notation
CS – Computer Science
CTL – computation tree logic
DSL – domain-specific language
DSML – domain-specific modeling language
EA – Enterprise Architecture
EMF – Eclipse Modeling Framework
EOL – Epsilon Object Language
EPC – Event-driven Process Chain
ETL – Epsilon Transformation Language
GPL – general-purpose programming language
HCI – Human-Computer Interaction
HOT – Higher-Order Transformation
IDE – Integrated Development Environment
JSON – JavaScript Object Notation
JVM – Java Virtual Machine
LHS – Left-Hand Side
LTL – linear temporal logic
MBSE – Model-Based Software Engineering
MDA – Model-Driven Architecture
MDE – Model-Driven Engineering
MCL – Model Constraint Language
MOF – Meta Object Facility
MQL – Model Query Language
MTBD – Model Transformation By-Demonstration
MTBE – Model Transformation By-Example
MTL – Model Transformation Language
NAC – Negative Application Condition
OCL – Object Constraint Language
PAC – Positive Application Condition
QM – Query Models
RHS – Right-Hand Side
OMG – Object Management Group
QVT – Query/View/Transformation
QVT-O – QVT-Operations
QVT-R – QVT-Relations
REST – representational state transfer
SD – Story Diagrams
SE – Software Engineering
SLR – Systematic Literature Review
SMS – Systematic Mapping Study
SUS – system under study
SQL – Structured Query Language
TMM – Transparent Model Manipulation
UML – Unified Modeling Language
VMCL – Visual Model Constraint Language
VMQL – Visual Model Query Language
VMTL – Visual Model Transformation Language
XSLT – Extensible Stylesheet Language Transformations
1.1 Problem Statement

In Model-Driven Engineering (MDE), models replace code as the main software development artifact [164]. In this context, models afford software engineers the opportunity to "mechanize those facets of software development that do not depend on human ingenuity" [32]. To achieve the benefits of automation, many task-specific languages and tools have been proposed. These technologies enable users to analyze models, extract the information they contain, and automatically generate implementation code, testing code, and documentation.

Languages designed for performing these operations can be classified as model query languages (MQLs), model constraint languages (MCLs), and model transformation languages (MTLs). They are collectively referred to as model manipulation languages. Today, the landscape of query and constraint languages is dominated by the Object Constraint Language (OCL [139]), a standardized declarative textual language. At the same time, a plethora of transformation languages are available, including the widely adopted Atlas Transformation Language (ATL [89]), Epsilon Transformation Language (ETL [100]), and Henshin [14], to name just a few.

All of these model manipulation languages expose users to metamodel details
rarely encountered in modeling practice. Furthermore, many of them resemble general-purpose programming languages in terms of both syntax and semantics. These aspects are understandable when considering the target audience of such languages: technically-savvy users, many of whom are software developers.

MDE practitioners, however, only account for a relatively small fraction of the total population of modelers. The primary and most general utility of models lies in their ability to describe and enable communication about various topics of interest. In SE, models are commonly used to represent a system’s architecture, capture its requirements, and document its implementation. In Business Process Management (BPM), models are used to represent and reason about the processes driving an organization’s activity. In Enterprise Architecture (EA), models are used to describe the structure, processes, technology, and information channels that together define an organization. These are only some examples of domains in which modelers create and use models with the help of modeling languages very similar to those employed by MDE practitioners.

We refer to these “non-MDE” modelers as end-user modelers. Apart from domain expertise, their distinguishing feature is a lack of metamodeling and programming expertise\(^1\). End-user modelers create, modify, and consult models using an editor supporting their modeling language of choice. However, in doing so, they routinely perform the same model manipulation operations as MDE practitioners: querying the model in search of a particular model fragment, ensuring that the model respects domain-specific constraints, or refactoring the model to improve its quality.

Today, end-user modelers manipulate models without the help of the kind of dedicated model manipulation languages that have become commonplace in MDE. This state of affairs, illustrated in Figure 1.1, is primarily due to the fact that these manipulation languages are not designed to accommodate the requirements of end-user modelers. Reaping the productivity benefits of MDE-style model manipulation languages demands that end-user modelers become familiar with metamodel intricacies, abstract syntax model representation, and, more often than not, basic programming concepts. These demands come on top of the learning curve imposed by adopting a new tool that is separate from the model editor. Finally, MDE-style model manipulation languages are task-specific, implying that a new language must be learned for each task.

Thus, the problem addressed in this thesis is the lack of adequate support for end-user modelers performing common model manipulation operations: querying, constraint specification, and transformation.

\(^1\)While it can be argued that a software architect or requirements engineer may possess programming skills, this is not a prerequisite, nor is it always part of the position’s description.
1.2 A Vision for End-User Modelers

As an answer to the problem identified in Section 1.1, this thesis proposes the vision illustrated in Figure 1.2. Its goal is to enable end-user modelers to master advanced model manipulation languages and tools at little or no learning cost. This goal is arguably achievable under one important condition: these technologies must be based on concepts and notations already familiar to end-user modelers. With this constraint in mind, this thesis adopts the approach of repurposing the end-user’s modeling language of choice, as well as its associated editor, to function as a model manipulation language and tool.

The realization of this vision, and the main result of this thesis, is the VM* family of model manipulation languages. The VM* family consists of the Visual Model Query Language (VMQL), the Visual Model Constraint Language (VMCL), and the Visual Model Transformation Language (VMTL). While they play different roles, the VM* languages are closely related, and are all guided by the common aim of allowing end-user modelers to carry out advanced model manipulation operations. To illustrate their utility, consider the following practical application scenarios:

- **Querying models with VMQL:** Imagine a software architect dealing with several models collectively describing a software system, including a domain model, a requirements model, and an architecture model. The models may be expressed using a single modeling language, such as the Unified Modeling Language (UML [143]), or may each utilize a different modeling language. In either case, traceability between the models must be ensured. To this end, the software architect can verify that key do-
Figure 1.2: Thesis vision: end-user modelers benefit from advanced model manipulation support via already familiar languages and tools

main concepts and their interrelations are reflected in all models. She can achieve this by specifying relevant VMQL structural queries for each model. As VMQL is a by-example query language, these queries take the form of valid model fragments in each modeling language, and can be specified using the same editors originally used to create the models. Due to VMQL’s service-based deployment, the queries can be executed via lightweight model editor extensions. The software architect is therefore neither required to learn a different query language for each modeling language she encounters, nor to use any tool other than the conventional model editors she already has access to.

- **Detecting model constraint violations with VMCL**: Consider now a scenario from the financial domain. In recent years, substantial new legislation has been issued to regulate financial markets. It is now widely accepted that “a comprehensive understanding of business processes is crucial for an in-depth audit of a company’s financial reporting and regulatory compliance” [128]. Making this possible involves audits of the involved business processes, or, to be more precise, audits of business processes models expressed using notations such as UML Activity Diagrams or Business Process Model and Notation (BPMN [137]). Auditors with a legal or financial professional background are therefore required to manually identify violations of legal constraints in such models. Using VMCL, they can precisely express these constraints using the business process modeling notations themselves, as well as automatically detect their violations. Massive amounts of manual error-prone work can thus be avoided.

- **Transforming models with VMTL**: The two scenarios discussed above have only addressed one half of their respective potential to improve end-user modelers’ work. What does the software architect do when faced with a missing traceability link, or the financial auditor when uncovering a constraint violation? Instead of manually adding the missing model elements, or replacing non-compliant process patterns with compliant ones, they can
both automate these actions using VMTL. In MDE terms, they would be specifying and executing model transformations. However, unlike existing MDE solutions, VMTL enables them to do so within the confines of the modeling notations and editors they are already working with.

Taken individually, model querying, constraint specification, and transformation are topics addressed extensively in MDE literature. It may be argued that the hypothetical software architect and financial auditor whose modeling-related conundrums are briefly discussed above would be better served by simply adopting one of the many existing solutions in these areas. There is, however, a case to be made against this proposal. First of all, as will be shown in this thesis, existing model manipulation technologies are lacking when it comes to addressing human factors considerations. Furthermore, as already argued in Section 1.1, these technologies fail to consider the specific requirements of end-user modelers. And finally, very few of the existing solutions offer a coherent and uniform approach to more than one model manipulation task.

1.3 Contributions

This thesis provides concepts and tools that enable end-user modelers to benefit from powerful model manipulation tools similar to those accessible to MDE practitioners. Its research contributions follow three main directions: (1) demonstrating that existing languages and tools do not appropriately address end-user modelers' requirements, (2) proposing languages and tools that appropriately address these requirements, and (3) showing that the proposed languages and tools are superior to the existing state of the art along dimensions relevant to end-user modelers. The following contributions are included:

- A Systematic Mapping Study (SMS) of human factors research in the area of model transformation has been conducted following widely accepted guidelines for this type of empirical Software Engineering (SE) research [96]. The study addresses the high-level topics of maintainability and usability as related to model transformation languages and tools. The SMS uncovers several significant findings. The first is that some human factors topics, especially those in the sphere of usability, are relatively scarcely addressed in the model transformation literature. The second is that, at least among those languages and tools that do address human factors, there exists a software monoculture [63] in terms of implementation and modeling technologies. The third is that the quality of empirical
evidence available in this area is unambiguously low. Together, these findings support the scientific relevance of the remaining contributions of this thesis.

- End-user modelers are defined and characterized as a distinct category of modelers. Namely, end-user modelers are users of a modeling language familiar with its syntax and semantics, but unfamiliar with its metamodel, abstract syntax, and applicable model manipulation languages. This characterization contrasts with that of MDE practitioners and language engineers. As a result, end-user modelers pose a distinct set of requirements to any model manipulation technology addressing them as a target audience. These requirements – utility, learnability, understandability, maintainability, genericity, and model-to-model manipulation support – are justified and discussed.

- Based on end-user modelers’ requirements, a set of general guidelines for the creation of model manipulation languages for end-user modelers are proposed. They are referred to as the principles of Transparent Model Manipulation (TMM): syntax transparency, environment transparency, and execution transparency. The syntax transparency principle ensures that modelers can reuse familiar modeling notations for specifying model manipulations. The environment transparency principle ensures that they can reuse familiar tools, in particular conventional model editors, for specifying these manipulations. Finally, the execution transparency principle ensures that end-users have control over how the specifications they create are executed, regardless of which tools are used to create them.

- Following the TMM principles, the VM* family of model manipulation languages is proposed. It consists of a model query language (VMQL), a model constraint language (VMCL), and a model transformation language (VMTL). VMQL supports the by-example specification of model queries as model fragments in the host modeling language. To ensure query expressiveness, VMQL defines a small textual annotation language. VMCL is then specified as an extension to VMQL, seeing as model constraints can be interpreted as the dual of model queries. VMTL is also built on top of VMQL, but instead of only allowing the identification of model patterns in a source model, it also supports modifying the identified pattern instances. VMTL is therefore an endogenous model-to-model transformation language that can express both in-place and out-place transformations. As TMM languages, the members of the VM* family share the feature that any query, constraint, or transformation specification created using them is also a valid model in the host modeling language. It should be noted that the versions of VMQL and VMCL presented in this thesis significantly extend earlier existing versions [177, 176], while VMTL was developed entirely as a contributions of this thesis.
• The VM* Runtime is introduced as a unified execution environment for VM* specifications. Following the execution transparency principle, the VM* Runtime delegates query, constraint, and transformation execution to an interchangeable underlying engine. The current implementation employs the Henshin graph-based model transformation engine for this purpose, but any sufficiently expressive model transformation engine could be used instead. The Henshin engine has been adopted due to its maturity and its formal graph transformation semantics closely resembling the operational semantics of VMTL. In effect, the VM* Runtime is a compiler between VM* and Henshin specifications. The deployment of the VM* Runtime is guided by the environment transparency principle. More specifically, a lightweight Web Service API referred to as the VM* API acts as a single access point to the VM* Runtime. This approach simplifies the development of VM* plugins for conventional model editors, while also enabling alternative user interfaces for manipulation execution, such as Web or mobile applications. However, the current implementations of the VM* Runtime and API are incomplete. In particular, the VM* Runtime only supports a subset of the full VM* language specifications.

• The learnability of the VM* languages has been validated empirically via user experiments complemented by qualitative evidence. Namely, this thesis presents the results of three user experiments and one think-aloud protocol analysis. The first experiment compares VMQL with OCL from a learnability standpoint in the context of querying business process models, indicating a clear advantage for VMQL. The second and third experiments compare VMTL with the Epsilon textual MTL and the Henshin abstract syntax visual MTL in the context of model quality assurance. The results of these experiments show that VMTL is superior to Henshin and comparable to Epsilon in terms of learnability. Finally, the results of the think-aloud protocol analysis have confirmed many of the design decisions behind VMTL, while also suggesting improvements that have been included in the version of the language presented in this thesis.

1.4 Thesis Outline

The remainder of this thesis is structured into the following chapters:

Chapter 2 discusses the two concepts that form the background of the thesis: models (Section 2.1) and end-user modelers (Section 2.2). A broad definition of models as they are understood in Software Engineering is put forward. It is argued that other disciplines, such as Business Process Modeling and Enterprise
Architecture, rely on models fundamentally similar to those employed in Software Engineering. Across these disciplines, end-user modelers are a distinct and substantial category of model users. They are defined and characterized here.

Chapter 3 presents related work in the area of model manipulation languages. It covers model querying languages (Section 3.1), model constraint languages (Section 3.2), and model transformation languages (Section 3.3). A special emphasis is placed on language syntax and usability.

Chapter 4 reports the findings of a Systematic Mapping Study of human factors research in the field of model transformation. The study’s motivation and research questions are presented in Section 4.1, followed by a description of the review protocol in Section 4.2. The findings of the review are described in Section 4.3 and interpreted in Section 4.5. Potential threats to the validity of the study are addressed in Section 4.4. The outcomes of this SMS indicate that, due their lack of focus on human factors, current model transformation solutions are not suitable for adoption by end-user modelers.

Chapter 5 introduces the VM* family of model manipulation languages. The principles of Transparent Model Transformation (TMM) guiding the design of these languages are described in Section 5.1. An example modeling scenario used to illustrate the VM* languages is presented in Section 5.2. The members of the VM* family – the Visual Model Query Language (VMQL), the Visual Model Constraint Language (VMCL), and the Visual Model Transformation Language (VMTL) – are described in Section 5.3, Section 5.4, and Section 5.5. Section 5.6 discusses practical considerations regarding the VM* languages.

Chapter 6 describes tool support for the VM* languages. The VM* Runtime responsible for executing VMQL, VMCL, and VMTL specifications is presented in Section 6.1, while its service-oriented deployment in the shape of the VM* API is detailed in Section 6.2. Tool limitations are discussed in Section 6.3.

Chapter 7 presents a comprehensive human factors empirical evaluation of the VM* languages. An evaluation plan is provided in Section 7.1. A user experiment comparing the learnability of VMQL with that of the standardized and widely used OCL model query and constraint language is then described in Section 7.2. In two similar experiments discussed in Section 7.3, VMTL is evaluated against the popular Epsilon and Henshin model transformation languages. Section 7.4 complements these user experiments by presenting qualitative data resulting from a think-aloud protocol analysis of the learnability of VMTL.

Chapter 8 concludes the thesis by summarizing its findings (Section 8.1), discussing the lessons learned in the process of arriving at these findings (Section 8.2), and proposing future research directions (Section 8.3).
2.1 Models in Software Engineering and Beyond

2.1.1 What Is a Model?

A large number of human endeavors rely on the creation and continued use of models, where “model” can be quite an open and diverse concept:

- To a civil engineer or an architect, a model may be a scaled-down physical replica of a building, or a three-dimensional representation of the building in a Computer-Aided Design (CAD) system.

- To a physicist, a model is a collection of laws accurately describing an observable physical phenomenon, such as Maxwell’s equations describing electromagnetism.

- To a software engineer, a model is “a set of statements about some system under study”[163], where the system under study is a software application.
As illustrated in Figure 2.1, a model is necessarily understood in relation to an original – what is being modeled. Based on this relationship, the model is said to be either descriptive or prescriptive [153] (also referred to as a specification model [28]). Descriptive models encapsulate information about already existing originals, while prescriptive models act as instructions for creating as-of-yet nonexistent features of an original. A model and its original belong to different domains of discourse, referred to as the modeling domain and subject domain. Note that the relationship between a model and its original is entirely relative: the model may itself act as an original for another model.

Regardless of application area or model type, modeling can generally be seen as “the cost-effective use of something in place of something else for some cognitive purpose” [153]. The key term in this definition is “cost-effective”: models allow reasoning about their original at a lower financial, time, and effort expense than directly using or creating the original would. In addition, the cognitive purpose of a model may actually be impossible to achieve by directly using the original. For example, SE models created using formal specification languages such as Alloy [85] enable the proof of desirable properties of software systems. Proving the same properties for the implemented software systems, i.e. the originals, is often unfeasible. The cost-effectiveness of models is due to an abstraction process, whereby any features of the original which are unnecessary for the model’s purpose are removed. This is followed by a projection onto a different representation, such as a modeling language [101].

Figure 2.1 also shows that a model can be used by several modelers, each with their own characteristics and intent. These modelers rely on potentially differing mental representations of the actual model. Furthermore, an object can form the basis for multiple models fulfilling different purposes and addressing different categories of modelers. For instance, the blueprint models of a building address an audience consisting of architects and civil engineers. At the same time, prospective tenants will likely find a scale model of the building created for sales purposes more appealing. The distinction between different modeler categories is elaborated in Section 2.2, where end-user modelers are introduced.

### 2.1.2 Models in Software Engineering

As already mentioned, in Software Engineering (SE) “a model is a set of statements about some system under study” [163], where the system under study (SUS) is a software application. The statements comprising a software model are logic expressions with truth values that can be evaluated on the SUS. For practical purposes, these statements are expressed using a modeling language, the most widely used of which is the Unified Modeling Language (UML [143]).
A distinguishing characteristic of SE models is their view of modeling languages as models, referred to as metamodels. In this context, a model is said to conform to its metamodel, and model elements are said to be instances of metamodel elements. The use of metamodels allows a precise specifications of modeling language constructs and of the relationships between these constructs and end-user models. Since metamodels are themselves models, they must also be specified using a modeling language: the meta-metamodel. As an example, the UML metamodel conforms to a meta-metamodel called the Meta Object Facility (MOF [141]). This process of defining evermore abstract metamodels could, in theory, continue ad infinitum. However, most meta-metamodels used in practice, such as MOF and Ecore (the meta-metamodel of the Eclipse Modeling Framework, EMF [173]), are reflexive: they simply conform to themselves.

While not an actual standard, the four-layer metamodel hierarchy endorsed by the Object Management Group (OMG) plays the role of a framework for specifying modeling languages (see [135], Sections 7.9–7.12). This hierarchy is illustrated in Figure 2.2 using a fragment of a UML Activity Diagram – the process notation provided by UML. The end-user model, i.e. the model created by a UML user supported by a model editor, occupies layer M1 of this hierarchy. It consists of two Actions (“Gather data” and “Check eligibility”) connected by a Control Flow. Run-time instances of the process modeled on layer M1 are represented as models on layer M0. Each model element instantiates exactly one element of the model on the layer immediately above (its metamodel), as indicated by “instanceOf” relations. Layer M2 defines the elements of the modeling language accessible to end-users, as well as the allowed relations between them. In this case, layer M2 consists of a small fragment of the UML metamodel. Finally, layer M3 provides reusable constructs supporting the specification of
Figure 2.2: The four-layer metamodel hierarchy illustrated on a UML Activity Diagram fragment.

modeling languages. In this example, layer $M3$ is occupied by a fragment of the MOF meta-metamodel. MOF could be replaced by a different meta-metamodel without implying changes to UML.

A notable aspect of the four-layer metamodel hierarchy is the use of abstract syntax and concrete syntax. Models at the $M2$ and $M3$ layers are most commonly expressed using a boxes-and-arrows notation roughly corresponding to a subset of the UML Class Diagram notation. Here, the focus lies on generality and expressiveness, as the notation must unambiguously describe any metamodel. Its users are primarily language designers. Meanwhile, models at lower layers are represented using concrete syntax: a visual or textual notation combining expressiveness with understandability, sometimes hiding cumbersome metamodel details. UML Activity Diagrams, for instance, adopt a visual flowchart-like notation. Although UML and most other modeling languages used in SE adopt
a visual syntax, empirical evidence suggests that for some tasks, such as architecture documentation, textual syntax is more usable [75]. Ultimately, concrete syntax can be seen as syntactic sugar, and can, from a strictly technical point of view, always be replaced by an equivalent abstract syntax representation. The equivalence between concrete and abstract syntax representations is potentially problematic for end-user modelers, as the vast majority of model editors exclusively support concrete syntax.

Meta-metamodels are designed to be sufficiently general to allow the definition of a wide range of metamodels. They are thus a suitable foundation for the development of domain-specific modeling languages (DSMLs) aimed at specific sub-disciplines of Software Engineering such as real-time or embedded systems. However, since many of the constructs required in these areas overlap with constructs already included in general-purpose modeling languages such as UML, defining a new DSML from scratch may not always be cost-effective. Therefore some modeling languages, including UML, provide a lightweight extension mechanism in the form of profiles. Examples of DSMLs defined as UML profiles include the Systems Modeling Language (SysML [142]), aimed at systems engineering applications, and Modeling and Analysis of Real Time and Embedded systems (MARTE [136]), aimed at real-time and embedded applications.

Profiles extend UML using stereotypes. A stereotype is a definition of an extension to the concrete syntax and/or semantics of a UML metamodel element, such as a Class, an Action, or an Actor. Stereotypes do not alter metamodel elements’ existing semantics and constraints, simply augmenting them instead. As an example, consider the UML Class in Figure 2.3 (left), representing the concept of insurance in a banking domain model. The `<java_class>` stereotype is applied in Figure 2.3 (center), indicating that the Class now also models a corresponding class in the Java programming language, perhaps for the purpose of code generation. A second stereotype, `<sql_table>`, is applied in Figure 2.3 (right). This indicated that, in addition to representing a Java class, the “Insurance” UML Class also models a database table. Each of the two applied stereotypes adds an additional interpretation to the original domain class.

SE models serve many different purposes, including requirements specification, software architecture description, and code documentation. However, a particular sub-field of SE, Model-Driven Engineering (MDE), places models at the center of the software development process. Just as object-oriented programming has adopted “everything is an object” as its mantra, MDE is governed by the principle that “everything is a model” [28]. In MDE, models replace code as the principal software development artifact. The intention is that all arti-

\[\text{1The stereotypes in this example were created for illustrative purposes only. They are not part of a standardized profile.}\]
facts typically created in a software development project – source code, test cases, documentation – should be *generated* from sufficiently accurate models.

One particular incarnation of MDE is the Model-Driven Architecture [138], an OMG standard specifying an MDE process based on UML.

The need for technologies supporting the MDE vision has motivated the development of a large variety of model manipulation languages and tools. The most prominent of these are model transformation languages (MTLs), sometimes referred to as “the heart and soul of MDE” [165]. MTLs allow software developers to precisely specify how other artifacts, such as code and related models, can be generated from existing models. Section 3.3 provides an overview of the state of the art in MTLs. Other categories of model manipulation languages developed in the context of MDE are model query languages (see Section 3.1) and model constraint languages (see Section 3.2). The vast majority of model manipulation languages originating in MDE share the assumption that their users are software developers possessing advanced modeling skills and familiarity with abstract syntax representations.

### 2.1.3 Models in Neighboring Disciplines

Several disciplines adopt a working definition of models similar to that presented in the previous subsection for SE models. Instead of using models to express statements about “systems under study”, these disciplines employ models to describe business processes, industrial processes, or organizational structures, to name just a few use cases. Despite their different application domains, the modeling languages and meta-modeling technologies supporting such models are often closely related to those found in SE and MDE.

One of the areas emphasizing the use of SE-like models is Business Process Management (BPM), defined as “a discipline involving any combination of modeling,
automation, execution, control, measurement and optimization of business activity flows, in support of enterprise goals, spanning systems, employees, customers and partners within and beyond the enterprise boundaries." [206]. The role of business process models is to capture, organize, and support the analysis of the knowledge underlying an organization’s processes. Therefore, business process models are considered “essential knowledge assets” to be “stored and maintained in process model repositories for reference” [104].

Since they are capable of modeling complex processes in any application domain, it is no surprise that UML Activity Diagrams (ADs) have been adopted by BPM practitioners. ADs offer comprehensive support for the control and data flow aspects of business process modeling, but have been noted to fall short of dedicated BPM notations in terms of resource-related and organizational aspects [157]. To address these shortcomings and provide better support for users with a business background, the OMG has standardized a dedicated business process modeling language: the Business Process Model and Notation (BPMN [137]). Just like UML, BPMN is based on the MOF metamodel. The similarities between BPMN and UML ADs are substantial, to the point where they can be used interchangeably to model the vast majority of business process models. As an example, Figure 2.4 presents equivalent BPMN and UML AD representations of a process from the library management domain. The most noticeable syntactic differences include swimlane orientation (vertical for ADs and horizontal for BPMN) and the fact that BPMN explicitly separates control flow from data flow.

Interestingly, empirical evidence fails to support BPMN’s claim of offering superior usability to business users compared to ADs. Birkmeier et al. [31] report on a user experiment indicating that there is no significant difference between BPMN and ADs in terms effectiveness, efficiency, or user satisfaction. Recker et al. [151] compare the complexity of the two languages’ metamodels and concrete syntax representations, concluding that BPMN is more complex on both accounts. In light of these findings, the choice between the two notations primarily comes down to individual modelers’ preferences.

BPMN and UML ADs are not the only business process modeling languages in widespread use. Event-driven Process Chains (EPCs [92]), the precursor of modern process modeling languages, are popular due to their well understood semantics [95] and relative simplicity. Another notable modeling language in this area is the Business Process Execution Language (BPEL [144]), which supports the creation of business process models executable via Web Service invocations. BPEL lacks a standard concrete syntax, motivating the proposal of a mapping between BPMN and BPEL to allow the use of BPMN’s concrete syntax with BPEL execution engines [137]. The combined use of process models defined using these languages is facilitated by the XPDL interchange format [192].
Figure 2.4: Equivalent UML Activity Diagram (top) and BPMN (bottom) process models describing the return of a loaned item to a library
Petri nets [148] are a general-purpose process modeling language that has found a fertile application ground in business process modeling [196]. Originally proposed as a method for modeling chemical processes, Petri nets benefit from several decades of practical use and research. This has resulted in a host of mathematically sound methods of reasoning about Petri nets’ formal properties, e.g. whether or not the execution of a process is guaranteed to eventually terminate. As a pure workflow language, Petri nets are highly expressive, being capable of representing choice points, concurrency, and iteration. Their expressiveness has been further augmented by extensions such as timed Petri nets [150] and Colored Petri Nets (CPN [86]). However, as they are a mathematical modeling language, they lack some of the business process-specific concepts included in dedicated business process modeling languages. For this reason, their use as a means of formalizing the operational semantics of business process modeling languages such as BPMN [54], EPCs [199], YAWL [197], and IPM-PDL [41] is arguably more common than their direct use by end-user modelers.

Enterprise Architecture (EA) is another area where visual models play a central role. As opposed to BPM, which limits itself to studying organizations’ processes, EA provides a holistic view of the factors that define an organization and allow it to achieve its business goals. Apart from business processes, EA deals with organizational structure and “data, computing systems for mission-related and business support, networks and other technology infrastructure, for both the baseline, or current, and target architectures” [74]. The enterprise architecture of an organization is thus a repository of knowledge on a broad range of topics. This knowledge is structured according to the specifications laid out by an enterprise architecture framework. Examples of such frameworks include the TOGAF [190] standard, maintained by The Open Group, and DoD AF [195], maintained by the United States Department of Defense.

EA frameworks rely on dedicated modeling languages. The models produced using such languages are the central artifacts resulting from applying an EA framework in the context of a given organization. An example of an EA modeling language is ArchiMate [191], designed to support the TOGAF framework. ArchiMate models are structured following a hierarchical layer design, with lower layers providing services to upper layers. The foundation of this hierarchy is the Technology layer, modeling hardware infrastructure and system software. Next comes the Application layer, modeling application software components. The Business layer resides at the top of this hierarchy, and models the organization’s structure and processes. The example ArchiMate diagram in Figure 2.5 illustrates these layers, as well as ArchiMate’s concrete visual syntax. ArchiMate’s concrete syntax is inspired by UML, adopting similar notations for concepts like Composition, Aggregation, Business Interfaces, and Business Actors. The use of colors to visually identify ArchiMate layers is common, although, like UML, ArchiMate does not explicitly assign semantics to colors.
Several engineering fields, such as electrical engineering and mechanical engineering, employ visual models to design and implement systems. In particular, *graphical block diagramming languages* are in widespread use in areas at the intersection of electrical engineering, electronics, and SE. Application areas include automatic control, digital signal processing (DSP), embedded systems development, data acquisition, and industrial automation. Some examples of modeling languages in these domains are Simulink [120], LabView [106], and VisSim [7], each supported by proprietary model editors and tools. Unlike modeling languages in SE, these languages are developed commercially by individual vendors, and are not subject to standardization by a standards body.

As an example, consider the Simulink model of an Anti-Lock Braking System controller shown in Figure 2.6. The model is effectively a visual representation of the mathematical equations governing the behavior of this system. Each computation is represented by a model element, while directed edges between model elements indicate the transmission of computation results between adjacent elements. The Simulink metamodel is considerably smaller and less complex than that of UML. Nevertheless, Simulink is widely adopted in industry – especially in the embedded systems domain [114] – for software development using an MDE process. Simulink can thus be seen as successfully bridging the gap between engineering and software models.
2.2 End-User Modelers

2.2.1 Definition

The above-mentioned modeling languages have several distinct categories of users, the differences between which are rarely acknowledged explicitly in the modeling literature. These differences are crucial for the purpose of this thesis, and are highlighted in what follows.

To begin with, all general-purpose or domain-specific modeling languages are created by language engines, who can be seen as expert users of metamodeling frameworks such as MOF or EMF. Their level of metamodeling expertise is arguably matched by MDE practitioners. Modelers falling under this description are expected to master the underpinnings of the modeling languages they employ, an assumption commonly made by model manipulation tools put forward in the context of MDE. However, the majority of modeling language users can be described as *end-user modelers*.

**DEFINITION 2.1** End-user modelers are users of a modeling language familiar with its concrete syntax and semantics, but unfamiliar with its metamodel, abstract syntax, and applicable model manipulation languages.

End-user modelers are highly trained domain experts, but do not have the level of Software Engineering expertise demanded to master today’s model manipulation tools. Their skills typically do not include metamodeling or model manipulation languages such as OCL. They may have some understanding of
2.2.2 High-Level Requirements

Since they have different skills and needs compared to other categories of modelers, end-user modelers impose a unique set of high-level requirements on model manipulation tools. Some of these requirements (e.g. R6) could be more accurately described as goals, but they are listed together for convenience. The following six high-level requirements are arguably critical to ensuring the adoption of a model manipulation language or tool by end-user modelers.

**R1 Utility.** End-user modelers’ primary motivation is achieving meaningful domain goals with the help of models. Thus, the cost-benefit ratio of using model manipulation languages is a key concern for them.

**R2 Learnability.** The learning curve imposed by a model manipulation language must be gentle. In particular, end-user modelers’ initial contact with a new manipulation language must clearly indicate its accessibility and usefulness.

**R3 Understandability.** End-user modelers must be able to unambiguously understand manipulation specifications and trust the outcome of their execution.

**R4 Maintainability.** Maintenance is an important stage in the life-cycle of any model manipulation specification. Therefore, end-users must be able to maintain manipulation specifications at an abstraction level similar to that used to create them. This concerns tasks such as modifying, testing, and debugging specifications. The ability to reuse specifications is also important, as it influences productivity.

---

Table 2.1: Skill sets of end-user modelers, language engineers, and MDE practitioners in terms of domain knowledge, modeling, metamodeling (MM), model manipulation languages (MLs), and programming

<table>
<thead>
<tr>
<th></th>
<th>Domain</th>
<th>Modeling</th>
<th>MM</th>
<th>MLs</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-user modeler</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Language engineer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MDE practitioner</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

programming, but are not professional software developers. As illustrated in Table 2.1, this skill set differs significantly from that of a language engineer or MDE practitioner.
R5 Genericity. In order to maximize the benefit of learning a manipulation language, it must be sufficiently generic to be applicable in the context of different and diverse modeling languages.

R6 Model-to-model manipulation. Code generation based on models, one of the main concerns for MDE practitioners, is less relevant from the end-user modeler perspective, since most end-user modelers do not model for the purpose of producing software. Instead, various use cases involving consistent global model updates take center stage. From a model transformation perspective, end-user modelers require a greater emphasis on model-to-model transformations compared to model-to-text transformations (see Section 3.3 for an overview of the state of the art in model transformation).

With the exception of R6, which is modeling-specific, these requirements have their origins in the area of human factors in Software Engineering. Many state-of-the-art model manipulation tools address MDE practitioners as their target users, and, as Table 2.1 shows, this user group has a different profile. Due to their programming competences, MDE practitioners are prepared to work at a lower abstraction level and learn model manipulation languages that resemble general-purpose programming languages (GPLs). This has led to a decreased emphasis on human factors in the MDE area, and in particular in what concerns model transformation languages. The precise extent to which human factors are considered in the development of model transformation languages is investigated in Chapter 4.
This chapter presents the current state of the art in the area of model manipulation languages. Such languages are used, for instance, for retrieving information from models, ensuring model quality and regulatory conformance, performing consistency-preserving model refactorings, and migrating models to new modeling language versions. All major model manipulation paradigms are covered in this chapter, with a special focus on how different approaches address the topic of language usability. Section 3.1 gives an overview of model querying languages, Section 3.2 addresses model constraint languages, while Section 3.3 discusses model transformation languages.

3.1 Model Querying

Model querying is the operation which “retrieves a set of model elements that satisfy a given condition from the persistent storage and transfers them to the client” [146]. It is one of the activities performed repeatedly throughout the process of creating, maintaining, and using models, regardless of application domain. Therefore, most model editors support at least one of the querying solutions listed in Table 3.1.

Model visualization tools (e.g. tree editors) and full-text search are supported
by most modeling environments. However, these solutions lack expressiveness – they cannot express model structure – and, in the case of full-text search, can produce large numbers of false-positive results. Predefined queries alleviate some of these shortcomings, but ultimately also lack sufficient expressiveness, since not all conceivable queries can be predefined by tool vendors. To remove any limits on query expressiveness, modeling environments such as MagicDraw [118] and Enterprise Architect [60] allow modelers to specify queries using general-purpose programming languages (GPLs) aided by tool-specific APIs. The drawbacks of this approach are that users must possess programming skills and must learn a new API for each tool. Increased portability is afforded by low-level query facilities providing direct access to the model persistency mechanism, such as database or XML query languages. Just like tool-specific APIs, low-level query facilities assume technically proficient users.

Dedicated MQLs have been proposed as expressive querying solutions whose use does not entail learning a full-fledged GPL or low-level query language. Several MQLs have been developed for models used in Software Engineering (see Section 3.1.1) and Business Process Management (see Section 3.1.2). Modelers in most other disciplines only have access to a subset of the querying approaches listed in Table 3.1, mainly model visualization tools and full-text search.

### 3.1.1 Software Model Querying

Model querying plays a dual role in SE. In the context of models as architecture or documentation artifacts, querying plays an information retrieval role. In an MDE context, model querying is more often seen as the first step in applying other model manipulation operations such as model transformation: they determine the model elements to which the transformation should be applied. As the majority of SE model querying approaches have been developed by and for MDE practitioners, they share a tendency to expose their users to the meta-
3.1 Model Querying

model and/or abstract syntax of the queried model.

OCL is the widely adopted standard MQL for querying MOF-based models, including UML. Originally proposed as a declarative language for unambiguously specifying model constraints, OCL has been standardized by the OMG to also play the role of a low-level query language. Despite its precise semantics [39] and adequate tool support, OCL is demonstrably lacking in terms of usability. Empirical studies show that using OCL to express all but the most trivial queries can be challenging even for experienced modelers [177]. An alternative model query language standardized by the OMG is Query/View/Transformation (QVT [140]), which, as its name suggests, also functions as a model transformation language. The QVT standard defines the QVT-Relations (QVT-R) and QVT-Operations (QVT-O) languages, suitable for the roles of declarative and imperative MQL, respectively. However, QVT is plagued by an imprecise operational semantics, which has led to fragmented and incomplete tool support [175].

Apart from the OMG’s standard languages, several textual MQLs have been proposed by the MDE research community. GMQL [51] is a declarative query language primarily aimed at applicability across modeling languages in the SE and BPM domains. MOCQL [178] has a similar focus, with an added emphasis on usability – as a textual language, it has been demonstrated to be more usable than OCL. A host of textual MQLs emphasize performance for querying large models or model repositories. These include IncQuery [27], Hawk [22], and MorisaQL [146]. IncQuery in particular stands out as an incremental MQL: it keeps track of model changes and updates the results of previously executed queries to reflect these changes. IncQuery-D [84] provides a distributed query execution engine as a back-end to IncQuery’s original textual syntax. The distributed approach has the benefit of enabling the in-memory storage of very large models across several machines, which makes incremental queries on models of up to 50 million elements feasible. Apart from MOCQL, which queries models expressed as Prolog fact databases, these MQLs operate in the EMF technology space.

Despite the predominance of textual MQLs in SE-related modeling practice, they expose users to a medium gap between the typically visual modeling languages they are accustomed to and the textual syntax of queries. For this reason, a subset of the textual MQLs mentioned above (GMQL [174] and QVT-R) support alternative visual syntaxes. Even the widely used OCL has a visual counterpart in the form of Visual OCL [33], which, however, has not been standardized or implemented. DMQL [50] is another model querying approach which proposes its own visual concrete syntax, different from the syntax of the visual languages it queries. The main argument put forward by DMQL is portability, as its concrete syntax is generic enough to represent queries on any graph-like visual modeling language.
All of the MQLs mentioned so far define their own concrete syntax, be it textual or visual. In contrast, several query by-example approaches are based on the observation that model queries can be expressed as model fragments—or, more commonly, model patterns—using the concrete syntax of the queried model. This is illustrated by the diagram in Figure 3.1, which queries a UML model containing Sequence Diagrams using the Query Models (QM [171, 172]) by-example MQL. Note that the UML Sequence Diagram concrete syntax is augmented to express pattern constructs, a feature shared by all by-example MQLs. For instance, a double dash across a relationship arrow indicates a path of arbitrary length. Executing this query amounts to finding all matches of the pattern in the queried model.

The advantage of by-example MQLs is that users do not have to learn the syntax of a dedicated MQL, but can instead use an already familiar modeling language syntax. This also promotes accessibility for end-user modelers. The expressiveness of a by-example querying approach is only limited by the expressiveness of its pattern language. However, considering that UML and other modeling languages do not have a one-to-one mapping between their concrete and abstract syntax, expressing patterns at the concrete syntax level can sometimes be problematic. In the absence of mechanisms enabling access to meta-level constructs, some patterns cannot be expressed using concrete syntax.

The already mentioned Query Models are a query by-example approach which supports UML Class, Object, and Sequence Diagrams, and benefits from an EMF-based implementation. QMs were originally defined in the context of Aspect Oriented Modeling (AOM), where crosscut specifications are direct applications of model querying. Another query by-example solution originating in the AOM area is MATA [209], which uses a graph matching algorithm to execute by-example queries on UML Class, Sequence and State Machine Diagrams.
3.1 Model Querying

Notably, all existing query by-example approaches are severely limited in terms of applicability, as they restrict themselves to small subsets of UML.

3.1.2 Business Process Model Querying

Of the disciplines mentioned in Section 2.1.3, BPM places the strongest emphasis on model querying. This is due to the fact that large organizations commonly maintain repositories containing thousands of business process models [53]. The processes documented by these models are queried by domain experts for purposes such as regulatory compliance verification and discovery of process templates for re-use. Therefore, MQLs in the BPM area must satisfy two equally important requirements: usability and performance.

Two broad categories of business process MQLs can be distinguished: structural MQLs and behavioral MQLs. Structural MQLs support querying the static structure of a process model: retrieving model fragments that match a given structure, without considering how instances of the process are executed. Behavioral MQLs also take into account the execution semantics of the queried process model. Thus, whereas structural MQLs answer queries of the type "Does the model contain activities A and B?", behavioral MQLs answer queries of the type "Is activity A always executed before activity B?".

Several structural MQLs in the BPM domain support querying BPMN models\(^1\). Some follow a by-example approach, allowing the specification of queries as concrete syntax model fragments. These are BPMN-Q [16], PPML [52], BPMN-VQL [126], as well as the approaches by Markovic et al. [119] and Belhajjame et al. [26]. Other business process modeling languages also benefit from dedicated structural MQLs, including BPEL (supporting the BP-QL [25] query language), IPM-PDL (supporting the IPM-PQL [40] query language), XPDL (supporting the BPQL [124] query language), and YAWL (supporting the YNet [87] query language). Finally, some structural business process MQLs (e.g. FNet [216]) operate on abstract process representations and are, at least conceptually, modeling language-agnostic. Structural MQLs such as YNet [87] and the solution by Belhajjame et al. [26] support querying semantically annotated process models.

Since business process models can be interpreted as directed labeled graphs, executing structural queries reduces to solving the subgraph isomorphism problem. Although this problem is NP-complete [44], practical algorithms for solving it are available (e.g. Ullman [194]). In contrast, behavioral querying cannot be

---

\(^1\)Limiting the use of MQLs to a specific modeling language is a characteristic of the BPM area. MQLs developed in other disciplines, such as SE, are typically applicable to all modeling languages sharing the same meta-metamodel.
Figure 3.2: A SeaFlows behavioral query on a BPMN model. The query is represented as a compliance rule graph [117].

reduced to a basic graph-theoretical problem, as it involves causal and temporal relationships between the activities executed in a process. Instead, behavioral queries are commonly solved by mapping to a model checking formalism such as linear temporal logic (LTL [149]) or computation tree logic (CTL [59]). MQLs applying this approach include BPMN-Q [18], APQL [189], and BeehiveZ [169]. Alternative formalisms include compliance rule graphs as employed by the SeaFlows MQL [117], and abstract trace inclusion [104]. With the exception of BPMN-Q, behavioral MQLs do not follow a by-example approach, instead defining their own textual or visual syntax. Consider, for instance, the SeaFlows query in Figure 3.2, represented as a compliance rule graph. Annotations on the graph nodes and edges define the query’s semantics (e.g. the A ⊗ B annotation specifies that exactly one of activities A and B must be executed).

A number of process model querying approaches optimize query execution performance (e.g. BPMN-Q, FNet, YNet). Separate experiments evaluating the performance of these approaches [17, 216, 87] indicate query execution times for realistic queries on the SAP R/3 Reference Model\(^2\) in the order of milliseconds. However, since the reported experiments were conducted independently under different experimental conditions, a direct comparison of the results is not possible. One notable aspect is that all three approaches rely on offline indexing, a pre-processing step performed on the queried model prior to query execution. This indexing process is time consuming, reported at 405 seconds for the SAP R/3 Reference Model in the case of BPMN-Q [17].

\(^2\)The SAP R/3 Reference Model [47] is a repository containing 600 business process models represented as EPCs, each including between 1 and 50 tasks.
3.2 Model Constraints

Enforcing constraints and executing queries on a model are closely related operations, to the extent that they have been referred to as “two sides of the same coin” [176]. Whereas a model query returns all model fragments satisfying a given query specification, applying a constraint with an equivalent specification will return all model fragments that violate this specification.

Model constraint languages (MCLs) are employed in the SE and BPM areas, where they serve different purposes. For SE models, the primary role of constraints is to ensure model integrity and quality (see Section 3.2.1). Constraints are extensively used in the specification of metamodels such as UML to precisely define the characteristics of valid metamodel instances. At the model level, constraints are used to verify the application of quality guidelines and design patterns. In BPM, on the other hand, model constraints are mainly used to check regulatory compliance of the modeled business process (see Section 3.2.2).

3.2.1 Software Model Constraints

Due to the fact that UML is the “lingua franca” of SE modeling, OCL, the standardized constraint language used in UML’s specification, is by far the most widely known MCL. Although adequate tool support for OCL has historically been problematic, several viable OCL interpreters are currently available (e.g. the Dresden OCL Toolkit [56], Eclipse MDT/OCL [57]), and OCL support is included in many commercial UML editors. Listing 3.1 shows an OCL constraint included in the UML specification (see [143], page 150). The constraint states that a redefined Property must be inherited from a more general Classifier.

Listing 3.1: Example of an OCL constraint included in the UML specification

```plaintext
inv: (redefinedProperty ->notEmpty()) implies
  (redefinitionContext ->notEmpty()) and
  redefinedProperty ->forAll(rp)
  ((redefinitionContext ->collect(fc)
    fc . allParents()|->asSet()|->collect(c|)
    c . allFeatures()|->asSet()|->includes(rp |))
```

Although OCL is theoretically [39] and practically sufficiently expressive, empirical evidence indicates that it is not particularly usable [176]. In fact, OCL offers no usability advantage over the Java general-purpose programming language for the task of specifying constraints on UML models [217]. This shortcoming of OCL has been a driving factor behind the proposal of alternative constraint language for UML and other modeling languages in the SE area.
Constraint Diagrams (CD [93]) are the earliest visual constraint language for UML. CDs follow a by-example approach, but are only applicable to a now obsolete version of UML Class Diagrams. In addition, Constraint Diagrams were never implemented in a tool. Spider Diagrams [79] are a further development of CDs, adopting a formal semantics based on Venn diagrams and Euler circles, but ultimately suffering from the same applicability and practicality limitations. A similar visual approach is adopted by Join Point Designation Diagrams (JPDD [180]), a successor of the Query Models MQL. As opposed to Constraint Diagrams, JPDDs support the entire UML and benefit from an implementation in the form of an Eclipse plug-in. However, they are also limited to the now outdated UML 1.5 standard [134].

An SE-specific motivation for the development of MCLs is the assessment of model quality with respect to structural design patterns. In particular, the conformance of UML Class Diagrams to object oriented design patterns originally devised in the context of programming languages [66] has attracted some interest. In this direction, RBML [94] is a visual/textual hybrid constraint language for expressing structural patterns in UML 1.5 Class Diagrams. As opposed to CDs and JPDDs, RBML constraints are specified using the abstract syntax of Class Diagrams, while deferring the specification of more complex constructs such as relationship chains to OCL.

Liu et al. [115] propose RIDE, a textual production system language\(^3\) as a means of detecting design pattern violations on UML models. Instead of specifying model patterns, users of this system define a set of rules (or productions) of the form “IF conditions THEN actions”. Each rule effectively specifies a model constraint, and can be evaluated against a given model by a rule engine – in this case Jess [65], an off-the-shelf Java rule engine.

Structural constraints on UML models can also be expressed using a generic constraint and behavior specification language such as Alloy [85]. By supporting the verification of structural constraints on object models, Alloy provides a constraint execution framework for any modeling language, as long as instances of this language can be transformed into equivalent Alloy models. Such a transformation has been defined for UML [10], although inconsistencies between the UML and Alloy metamodels require the use of an Alloy profile for UML. As an added benefit, the OCL constraints accompanying a UML model can also be translated to Alloy for execution, obviating the need for an OCL interpreter.

\(^3\)A production system is a logic reasoning system based on forward-chaining derivation techniques. It represents knowledge in the form of productions, and can execute actions based on these productions and on knowledge derived from them [34].
3.2 Model Constraints

One of the main motivations for organizations to engage in business process modeling is the fact that models enable reasoning about the extent to which the business processes they are based on comply to relevant laws and regulations. One example of such a law is the Sarbanes-Oxley Act [159], which sets requirements on the business practices of public and private companies operating in the United States. Its promulgation, along with that of similar laws in other countries, has led to an increased research interest in business process model constraint languages and tools.

Two of the process model querying approaches mentioned in Section 3.1.2, BPMN-Q [18] and SeaFlows [117] (together with its extension for run-time compliance monitoring [98]), also address regulatory compliance constraints. These are usually behavioral constraints, enforcing restrictions on how different activities are executed in relation to each other. An example is the BPMN-Q constraint shown in Figure 3.3. This constraint indicates, via BPMN-Q’s `<precedes>` annotation, that any execution of activity B must be preceded by an execution of activity A. The interrupted sequence flow indicates that any number of other activities may be executed in-between. Just as in the case of behavioral queries, several mathematical formalisms can be used to check behavioral constraints. For instance, the BPSL constraint language [116], which operates on BPEL models, transforms the constrained models into pi-calculus formulas and finite state machines before applying constraints expressed using LTL. Hoffman et al. [77] investigate the verification of compliance rules on BPMN models via model checking techniques, and provide a prototype implementation based on the SPIN [78] model checker.

Several MCLs in the BPM area support extensions to the types of behavioral constraints that can be specified, such as taking into account contextual information or activity durations. Van der Werf et al. [198] propose a solution for evaluating context-aware constraints, where the context of a process model is understood as the state of the information systems surrounding it. The ap-
proach involves constructing a unified ontology of the context and the process model, upon which constraints can be evaluated using the Semantic Web Rule Language (SWRL [214]). Kumar et al. [103] take into account concrete activity durations when evaluating temporal constraints. Their approach allows controlled violations of temporal constraints and provides a run-time solution for minimizing the total penalty incurred from these violations. The catalog of time patterns collected by Lanz et al. [108] contains ten common time-related constraints grouped into four categories: duration and time lags, restricting execution times, variability, and recurrent process elements.

Not all business process model constraint languages were designed for verifying regulatory compliance. PPSL [64], for instance, is motivated by quality assurance – much like MCLs in the SE area. Only constraints on UML Activity Diagrams are supported. For constraint evaluation, the Activity Diagrams are translated into equivalent labeled transition systems, on which LTL formulas derived from PPSL diagrams are evaluated.

### 3.3 Model Transformation

Model transformation (MT) is an operation that “deals with producing different models, viewpoints, or artifacts from a model based on a transformation pattern” [138]. The concepts involved in a model transformation are illustrated in Figure 3.4. The core of every MT is a transformation specification expressed using a model transformation language (MTL)\(^4\). According to the transformation rules included in this specification, a transformation engine takes as input a source model and produces as output a target model. Some transformations operate on multiple source/target models at a time. In most MTLs, the transformation specification makes direct reference to the source and target metamodels.

In Figure 3.4, the source and target models conform to different metamodels, in which case the transformation is called exogenous. They may, however, also conform to the same metamodel, leading to an endogenous transformation [48, 122]. Furthermore, as the source and target model differ, this is an out-place MT (conversely, transformations where the source and target model coincide are referred to as in-place). In fact, the source or target of a MT are often other types of software artifacts, such as source code\(^5\) or documentation. There is a

\(^4\)In the case of Model Transformation By-Example (MTBE), the transformation specification is deduced from concrete transformation examples, and is not visible to end-users. Nevertheless, a specification must always exist.

\(^5\)Source code can be considered a particular kind of textual model conforming to its own metamodel – the programming language grammar.
considerable number of other dimensions on which MTs and MTLs may vary, as documented by model transformation taxonomies [122] and feature models [48].

As a generative software development approach, MDE heavily relies on MT to bridge the gap between models on different abstraction levels, as well as between models and other artifacts [165]. Due to the importance of MT to MDE theory and practice, nearly all MTLs and their adjacent tooling have been developed by MDE researchers and practitioners. These technologies are occasionally applied in other modeling fields, such as BPM (e.g. [181, 49]). However, most end-user modelers, regardless of their domain of activity, still rely on primitive tools – mostly manual editing – to accomplish what amounts to model transformation.

Since the focus of this thesis is on end-user modelers, the following subsections give an overview of existing MTLs based on an important aspect to end-users: language syntax. Following this criteria, MTLs can be classified as textual (see Section 3.3.1) or visual, with a further sub-classification into abstract syntax and concrete syntax visual languages (see Section 3.3.2 and Section 3.3.3, respectively). Model Transformation By-Example (MTBE), on the other hand, largely circumvents the use of MTLs by end-user modelers (see Section 3.3.4).

An MTLs execution environment can influence its maintainability, and thus its appeal to end-users. Therefore, the main approaches to executing model transformations are presented in Section 3.3.5. Both usability and maintain-
ability can be regarded as human factors. While the following subsections give an overview of the state of the art in MTLs, Chapter 4 investigates the extent to which human factors are addressed in the MT literature by means of a Systematic Mapping Study.

### 3.3.1 Textual MTLs

Modeling frameworks such as EMF offer programmatic model access via general-purpose programming language APIs – in the case of EMF, a Java API is provided. Such an API is a natural option for specifying model transformations, but this option also suffers from a notable drawback: the level of abstraction provided by a GPL is low, resulting in overly verbose specifications. Directly accessing a model’s XML representation in order to transform it using an XML transformation language (e.g., XSLT [215]) suffers from the same drawback, also due to an abstraction level mismatch.

Dedicated textual MTLs, on the other hand, bridge the abstraction gap by offering MT-specific language constructs. Several textual MTLs adopt a purely imperative semantics not entirely different from that of a mainstream imperative or object oriented GPL such as Java or C++. Examples include the QVT-O standard [140], Kermeta [127], and the Epsilon Object Language (EOL [99]). Apart from imperative constructs found in GPLs (operators, variable assignment, conditional and iteration constructs), these MTLs define transformation-specific constructs at the language level. EOL, for instance, has built-in support for model element types (e.g., the ! operator identifying the metamodel containing a given type), and standard collection operators resembling those found in OCL (e.g., includes, excludes, flatten, concat). Imperative languages have the benefit of flexibility and give the developer full control over transformation execution, most importantly with respect to rule scheduling. They are suitable for most transformation scenarios, including in-place or out-place, and endogenous or exogenous transformations.

Other textual MTLs adopt a declarative style. Instead of explicitly specifying the steps and control flow defining how a transformation should be implemented, they allow developers to specify what changes should be applied to the source model in order to obtain the target model. A subset of textual declarative MTLs adopt the relational paradigm, in that they define mathematical relations between source and target model elements. These relations become executable by synthesizing source code in an imperative or logic programming language. MTLs in this category include the textual version of QVT-R [140], KTML [4], Tefkat [110], AutoFOCUS [160], and PTL [6]. The first three are implemented using Java, while the last two rely on Prolog as an execution engine. List-
3.3 Model Transformation

Listing 3.2: Textual QVT-R transformation example: creating a “Schema” element with the same name as a “Package” element [140]

Since software models can be viewed as graphs, with model elements corresponding to graph nodes and relationships between elements corresponding to graph edges, graph transformation theory [154] and tools have been adopted as the foundation of a large number of MTLs. This includes textual MTLs like GrGen.NET [68], GREAT [42], and VIATRA2 [202]. Just like their underlying graph transformation engines, these MTLs are declarative. However, in order to express the control flow of complex transformations, some MTLs in this category include imperative execution and control flow structures, thus becoming hybrid languages. For example, VIATRA2 offers an imperative textual language based on abstract state machines (ASM). As illustrated in Listing 3.3, the ASM language of VIATRA2 has the ability to invoke declarative graph transformation rules within imperative constructs (e.g. on line 2, the “liftAttrsR” rule is invoked from a forall loop).

Listing 3.3: VIATRA2 transformation example: a declarative graph transformation rule (Lines 4–24) invoked within an imperative forall loop construct (Line 28) [202]

```java
import UML;

// declarative transformation rule
gtule liftAttrsR ( inout CP , inout CS , inout A ) =
{
    precondition pattern lhs (CP , CS , A , Par , Attr ) =
    {
        Class (CP );
        Class . parent (Par , CS , CP );
        Class (CS );
        Class . attrs (Attr , CS , A );
        Attribute (A );
    }
    postcondition pattern rhs (CP , CS , A , Par , Attr , Attr2 ) =
    {
        Class (CP );
        Class . parent (Par , CS , CP );
    }
```
Hybrid MTLs combining a declarative pattern language with imperative model manipulation constructs have seen a wide adoption among MDE practitioners. The Atlas Transformation Language (ATL [89]) and the Epsilon Transformation Language (ETL [100]) are two representative approaches. ATL encourages a predominantly declarative specification style, as it is “closer to the way the developers intuitively perceive a transformation” [89], and supports usability-related features such as step-wise debugging. An example ATL rule is presented in Listing 3.4, showing the generation of a relational database model from an existing class model. The from and to keywords indicate the declarative specification of the source and target model patterns, respectively. Two target model elements, of type “Table” and “Column”, are created based on a source model element of type “Class”. ETL, on the other hand, is a task-specific language adding declarative transformation constructs to the imperative Epsilon Object Language. Unlike ATL, which can also perform other model manipulation tasks (e.g. model validation [37] and merging), ETL focuses strictly on model-to-model transformations. In most other respects, the syntax and operational semantics of ATL and ETL are markedly similar.

**Listing 3.4:** ATL transformation example: generating a relational database model from an existing class model [89]
A number of approaches attempt to combine the benefits of a GPL with those of a transformation domain-specific language (DSL). This can be achieved by implementing a textual model transformation language as an \textit{internal DSL} in an existing general-purpose programming language. Such a solution gives developers access to the tool support and libraries of the GPL without lowering the abstraction level of transformation specifications. RubyTL \cite{46}, for instance, is implemented as an internal DSL in the Ruby programming language, while SIGMA \cite{105} and the MTL by George et al. \cite{70} are implemented as internal DSLs in the Scala programming language. Both Ruby and Scala are appropriate choices as host GPLs, since they provide extensive support for the creation of internal DSLs. Furthermore, since Scala programs are executed by the Java Virtual Machine (JVM), Scala internal DSLs can access the Java-based EMF. Tratt’s MT language \cite{193} follows a similar approach using the experimental Converge programming language as a host.

### 3.3.2 Abstract Syntax Visual MTLs

The majority of visual MTLs propose a dedicated syntax that differs from the concrete syntax of the transformed models. Most commonly, the proposed notation resembles that of UML Class or Object Diagrams, and is used to specify model patterns at the abstract syntax level — similarly to how the UML standard adopts Class Diagrams to specify the UML metamodel. This approach has two main advantages: (1) Class and Object Diagrams are relatively simple notations familiar to modelers with an MDE background, and (2) the same visual notation can be used for specifying transformations on any modeling language, regardless of its concrete syntax. However, abstract syntax reveals metamodel details that are not visible in concrete syntax, possibly leading to verbose specifications. And, more importantly, end-user modelers are unfamiliar with abstract syntax.

Abstract syntax visual MTLs can be seen as the counterparts of declarative textual MTLs, relying on similar formalisms and execution engines. For instance, the QVT-R \cite{140} standard provides an equivalent abstract syntax visual notation to its declarative textual notation. Consider the QVT-R specification in Figure 3.5, representing a direct counterpart to the textual specification in Listing 3.2. In this case, the transformation developer may freely choose between the visual and textual versions of a specification.

Several abstract syntax visual MTLs are based on the graph rewriting formalism, again manifesting important similarities with some declarative textual MTLs. Examples include VIATRA \cite{45}, VMTS \cite{112}, and AToM³ \cite{109}. Algebraic graph transformation \cite{58}, another prominent graph transformation paradigm, is adopted by AGG \cite{187}. Due to its underlying algebraic graph transformation
Figure 3.5: Visual QVT-R transformation example [140]. This specification is equivalent to the textual specification in Listing 3.2

Figure 3.6: Henshin transformation example: an Independent Unit (top) and one of its five invoked transformation rules (bottom) [29]
engine, AGG permits the static analysis of transformation properties such as termination or confluence. DSLTrans \[23\] goes a step further and guarantees the termination and confluence of transformations, at the expense of expressiveness—the language is Turing incomplete. The algebraic approach is enhanced by Henshin \[14\] and GReAT \[20\] with expressive rule execution control mechanisms. For instance, Henshin relies on Units: arbitrarily nested control structures providing iterative, priority-based, or conditional execution of transformation rules. Figure 3.6 (top) shows an Independent Unit supporting the non-deterministic execution of exactly one of five given transformation rules. One of these rules is shown in Figure 3.6 (bottom), illustrating Henshin’s abstract syntax notation: a subset of the UML Class Diagram syntax.

Story Diagrams (SDs \[62\]) are another graph-based model transformation approach, this time relying on Triple-Graph Grammars (TGGs \[162\]) as an underlying formalism. SDs support end-user modelers by re-purposing several UML diagram types (Activity, Class, and Collaboration Diagrams) as components of a transformation specification language. Despite their strong reliance on the concrete syntax of UML, SDs are ultimately an abstract syntax approach, since the concrete syntax of the transformed models, which may differ from UML, is not reflected in transformation specifications. SDs have evolved from their initial implementation in the Fujaba tool \[130\] into MOFLON \[8\], a MOF 2.0-compliant implementation, and finally into eMoon \[11\], a re-implementation of MOFLON based on EMF.

### 3.3.3 Concrete Syntax Visual MTLs

The verbosity of abstract syntax visual specifications, as well as end-user modeler’s unfamiliarity with the notation, has led to the proposal of several concrete syntax visual MTLs. However, the transition from abstract syntax to concrete syntax for MT specifications hides some intrinsic difficulties which have so far prevented concrete syntax MTLs from achieving widespread user adoption. First, the fact that concrete syntax hides certain metamodel details, while productive for specifying models, is a setback for specifying MTs. Because MTs rely on model matching, the values of model attributes that are not represented in concrete syntax may alter the outcome of the matching process. Perhaps even more importantly, the concrete syntax of a modeling language is likely inappropriate for use as a model pattern language without some augmentations. For instance, path constructs and attributes with range values cannot be expressed in the concrete syntax of most modeling languages.

\[6\]This issue is not as pressing for abstract syntax MTLs, which do not claim to be a fully accurate representation of a model’s abstract syntax representation and are supported by custom editors, as opposed to the off-the-shelf editors used to create concrete syntax models.
One approach to utilizing the concrete syntax of a modeling language to specify MTs has been mentioned in Section 3.3.2 in the context of SDs. In this case, the concrete syntax of a subset of UML is used, while the diagrams are assigned a transformation-specific semantics. SDs can only be considered a concrete syntax MTL when transforming UML models—the concrete syntax of any other models will not be reflected in SD specifications. The strategy of adopting a subset of UML as a transformation specification language is also embraced by MOLA [90], UMLX [212], and VMT [166]. Despite their constrained use of concrete syntax, these MTLs are mentioned in this section due to their motivation of providing support for end-user modelers, a motivation shared with the more comprehensive concrete syntax approaches.

A generic approach to concrete syntax MT specifications must be able to leverage the concrete syntax of any modeling language. Kühne et al. [102] propose systematically modifying a modeling language’s metamodel in order to support concrete syntax patterns. The proposed modifications take place in three stages: relaxation (i.e. relaxing metamodel constraints, such as those related to element multiplicity), augmentation (i.e. augmenting the metamodel with MT-specific features), and modification (i.e. modifying the concrete syntax of language elements not suited for specifying patterns). Arguably, end-user modeler support requires the modification phase to be minimal. The Patterns in Concrete Syntax (PICS [19]) proposal by Baar et al. follows the same principles, but does not systematize the steps of the metamodel modification process. An example PICS specification is shown in Figure 3.7, showcasing the use of OCL as an application condition specification language, as well as borrowed elements from the QVT-R visual syntax (e.g. the hexagonal symbol linking the source and target patterns). The intent of this transformation is to rename an Attribute of a UML Class. Rumpé et al. [155] propose a similar concrete syntax MT approach tailored to textual modeling languages. AToMPM [186] automatically generates an MTL from an existing metamodel, additionally providing a Web-based tool for both language and transformation engineering.

A less extensive metamodel modification process is sometimes sufficient for allowing concrete syntax MT specification. In support of this observation, Schmidt [161] advocates the use of UML Stereotypes as a lightweight metamodel extension mechanism. The described process is only suitable for transforming UML models, and suffers from expressiveness limitations. Of the three steps defined by Kühne et al., stereotypes are only capable of metamodel augmentation and concrete syntax modification. Constraint relaxation is not possible when relying exclusively on stereotypes, as the UML standard states that “it is not possible to remove any of the Constraints that apply to UML using a Profile” (see Section 12.3.1.1 of [143]).

The idea of adapting UML to act as a transformation language via an MT profile
3.3 Model Transformation

Figure 3.7: PICS transformation example: renaming an Attribute of a UML Class [19]

is also investigated by van Gorp et al. [200], albeit with a different purpose: providing a unified front-end for existing graph transformation languages. This proposal is in fact not concerned with concrete syntax model transformation. In contrast, the MATA [209] approach, also mentioned as a query language in Section 3.1.1, is intended as a concrete syntax MTL to be executed on top of the AGG graph transformation engine. MATA is only usable in an Aspect-Oriented Modeling context, and its scope is limited: only UML Class, Sequence, and State Machine Diagrams can be transformed. AGG is also used as an execution engine in the solution proposed by Grønmo et al. [72], and exemplified by transforming UML Sequence Diagrams to equivalent State Machine Diagrams. However, the pattern language proposed in this approach is relatively inexpressive, and the topics of metamodel relaxation and augmentation are not addressed.

3.3.4 Model Transformation By-Example

Although it is beneficial for end-user modelers, specifying transformations at the concrete syntax level raises important theoretical and practical problems. This has led to the proposal of a separate MT paradigm under the name of Model Transformation By-Example (MTBE). The core principle of MTBE is that transformation specifications do not necessarily need to be created by end-users. Under certain circumstances, the specifications can be generated automatically based on examples of source and target model pairs complying with the said specifications. The effort of learning an MTL, regardless of its syntax, can thus be entirely circumvented. To avoid imposing any additional requirements on end-user modelers, the example source and target models are expressed using the concrete syntax(es) of the source and target modeling languages. Nevertheless, MTBE approaches should not be confused with the concrete syntax MTLs mentioned in Section 3.3.3, which still rely on transformation developers
to manually create a specification, as opposed to just providing application examples. This subsection gives an overview of the main MTBE proposals. An in-depth review of this area is presented by Kappel et al. in [91].

The first MTBE proposal has been put forward by Varró [201], and features the automatic deduction of graph transformation rules based on a user-provided mapping model. The mapping model consists of example source and target model elements between which explicit correspondences are provided (represented visually as dotted lines). As is the case in any MTBE approach, the transformation rules generated from a single example are often too general to capture the end-user’s actual intention. To improve the quality of the rules, the user can choose to provide additional examples (in this case, to augment the mapping model) or to manually edit the generated rules. Sometimes the second option, although less desirable as it exposes end-user modelers to transformation rules, is the only viable alternative. The main differentiating factor between the various available MTBE approaches is the mechanism used to automate transformation rule generation, with the goals of maximizing accuracy and minimizing the time required to generate rules.

To achieve these goals, Balogh et al. [21] propose the use of inductive logic programming in conjunction with a Prolog-based model representation. Saada et al. [158] adopt Relational Concept Analysis (RCA [80]) instead, while generating transformation specifications executable by the Jess rule engine. Other approaches generate textual transformation rules expressed using the Atlas Transformation Language (ATL). Strommer et al. [182] propose generating ATL rules via a higher-order transformation, given that ATL has its own metamodel and can thus be seen as a modeling language. This approach has also been applied for transforming business process models represented as EPCs into equivalent UML Activity Diagrams [181], as illustrated in Figure 3.8. Note the red dotted lines denoting model element correspondences, which can be specified using either concrete or abstract syntax. The implementation by García-Magariño et al. [67] generates ATL rules via an ad-hoc algorithm. CONVeR-T [15] is an MTBE framework that supports end-user modelers by providing correspondence recommendations between the example model pairs, while generating XSLT transformation specifications.

The approaches mentioned in the previous paragraph exclusively address exogenous transformations. Endogenous transformations, and in particular in-place model updates, are addressed by a separate group of approaches following the Model Transformation By-Demonstration (MTBD) paradigm. Whereas MTBE relies on users explicitly identifying corresponding source and target model elements, MTBD allows users to manually modify a source model, and automatically derives transformation rules from these modifications. Sun et al. [183] propose the MT-Scribe editor, which incorporates an event listener monitoring
all editor operations. The sequence of recorded operations is then optimized by eliminating redundant changes, and can be replayed on other source models. The Operation Recorder designed by Brosch et al. [35] takes a different implementation approach: instead of recording editor actions, a model difference is computed each time the end-user exemplifies a change on the source model. The difference is used to generate Java transformation specifications. Langer et al. [107] extend this approach to support exogenous transformations via the generation of ATL rules.

3.3.5 Executing Model Transformations

An MTL’s execution environment can influence its accessibility to end-user modelers. Maintainability considerations such as traceability, debugging support, and testability depend on the underlying execution approach. A separate motivation for addressing related work on this topic is the fact that the transparency principles described in Section 5.1 also address MT execution.

The most straightforward MTL implementation approach is to provide a dedicated interpreter for the transformation language. Since the majority of ex-
isting MTLs address the EMF ecosystem, their interpreters are typically Java programs. The syntactic paradigm adopted by an MTL is not necessarily an indication of its execution paradigm. For instance, interpreted transformation languages include imperative textual MTLs (e.g., Kermeta [127]), hybrid textual MTLs (e.g., ETL [100]), and visual MTLs (e.g., Henshin [14]).

Much like general-purpose languages, interpreted MTLs suffer from some drawbacks, the most notable being execution performance. A solution to the performance problem is to instead compile transformation specifications to a better-performing GPL. An additional advantage is that the GPL’s tool infrastructure becomes accessible to transformation developers – although at a lower abstraction level, which makes it unappealing. Existing MTL compilation targets include Java (used by MTLs such as GREAT [42], KTML [4], and Fujaba [208]), C# (used by VMTS [112]), and C++ (used by GReAT [20]).

The significant abstraction gap introduced when compiling an MTL specification into a GPL program can be overcome by compiling to another, less abstract MTL instead. This approach may re-introduce performance concerns, but has the advantage that the properties of the target MTL, such as formal execution guarantees, are translated to the compiled MTL. Although not usually adopted by traditional MTLs, this approach is common among by-example solutions. The MTBE proposal by Strommer et al. [182] generates ATL rules, while the CONVERt [15] framework generates XSLT rules. Similarly, the MTBD proposal by Langer et al. [107] generates ATL rules.

Finally, instead of compiling to another MTL, some transformation languages are compiled to MT-specific virtual machines. Similarly to GPL virtual machines like the Java Virtual Machine or .Net Common Language Runtime, model transformation virtual machines execute low-level bytecode. However, apart from general-purpose flow control instructions, MT bytecode instructions perform simple model manipulations. For instance, the EMFTVM [207] virtual machine supports 47 instructions such as ADD, REMOVE, and SET for managing model element properties. EMFTVM can be used as a runtime for ATL transformations, as well as simple graph transformations. The original ATL runtime, the ATL VM [89], offers similar features. T-Core [185] is designed as a library of model transformation primitives that can be used to implement higher-level MTLs. It offers an abstraction level between that of a VM and that of an MTL, and is employed in the implementation of AToMPM [186].
Chapter 4

Human Factors in Model Transformation

4.1 Motivation and Research Questions

Engineering has as an eventual goal the production of artifacts and services used by humans. In this context, optimizing the interaction between humans and the engineered artifacts is a long-standing research area. Wickens et al. [210] identify the purpose of human factors engineering (also referred to as ergonomics) as understanding “how knowledge of human strengths and limitations, both mental and physical, can lead to better system design, more effective training of the user, and better assessment of the usability of a system”. As a tool set for studying the actions and attitudes of users towards a system, human factors engineering relies to a great extent on empirical research methods.

In Computer Science, human factors play a central role in the area of Human-Computer Interaction (HCI [38]), which researches the interfaces between computers and their users. As software engineers are a special category of computer users interacting with tools such as programming languages, Integrated Development Environments (IDEs), and modeling languages, they are the subjects of many human factors studies (e.g. [97, 121, 152, 170]). In particular, the usabil-
ity of programming languages is an area raising significant interest\(^1\). Research on human factors in SE is facilitated by the adaptation of empirical research methods such as experiments [213] and case studies [156] to the particularities of SE.

This chapter investigates the extent to which human factors are taken into consideration in the model transformation literature. The investigation is performed as a Systematic Mapping Study (SMS), following the relevant guidelines laid out by Kitchenham et al. [96]. The scope of this SMS is limited to model transformation, as MT languages constitute the core of MDE, while also greatly exceeding other model manipulation languages in terms of the number and variety of proposed solutions. The mapping study addresses the following research questions:

- **RQ 1** To which extent are human factors addressed in MT research?
  - **RQ 1.1** What are the bibliometric trends regarding research on human factors in MT?
  - **RQ 1.2** Which human factors-related topics are addressed in the MT literature, and which are not?
  - **RQ 1.3** Which study types are prevalent in the human factors in MT literature?

- **RQ 2** Which MTLs address concerns related to human factors?
  - **RQ 2.1** Which existing MTLs have been evaluated from a human factors perspective?
  - **RQ 2.2** Which modeling languages are targeted by MTLs addressing human factors?
  - **RQ 2.3** What technologies are used to implement MTLs addressing human factors?

- **RQ 3** What is the quality of the evaluations carried out in human factors studies in the MT area?
  - **RQ 3.1** What evaluation methods are adopted by human factors studies in the MT area?
  - **RQ 3.2** What is the quality of quantitative human factors studies in the MT area?
  - **RQ 3.3** What is the quality of qualitative human factors studies in the MT area?

\(^1\)See [170, 97, 121, 152] for some examples of human factors studies on general-purpose programming languages.
4.1 Motivation and Research Questions

Table 4.1: Usability and maintainability attributes addressed by the SMS

<table>
<thead>
<tr>
<th>Usability</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>learnability [83]</td>
<td>modularity [83, 129]</td>
</tr>
<tr>
<td>readability [9]</td>
<td>composition [184]</td>
</tr>
<tr>
<td>understandability [129]</td>
<td>reusability [83, 129]</td>
</tr>
<tr>
<td>comprehensibility [123]</td>
<td>modifiability [83, 129, 123]</td>
</tr>
<tr>
<td>aesthetics [83, 123]</td>
<td>testability [83]</td>
</tr>
<tr>
<td>user satisfaction [83]</td>
<td>debugging [9]</td>
</tr>
<tr>
<td>user comfort [83]</td>
<td>traceability [9]</td>
</tr>
</tbody>
</table>

RQ 1 aims to uncover if and how human factors are addressed in MT research. The precise human factors aspects that are addressed, as well as the prevailing study types are of interest here. RQ 2 investigates which MTLs have been studied from a human factors perspective, and also catalogs their technology space. This will allow us to identify metamodelling frameworks or implementation technologies that are over- or under-represented. Finally, RQ 3 explores the quality of the empirical studies carried out so far in the MT area. Taken together, the answers to these questions will show a complete picture of the state of human factors in model transformation research. None of the listed questions have previously been addressed in a systematic manner in the literature.

Our definition of human factors encompasses all the attributes of an MT approach that influence how transformations are created, maintained, and used by humans. The considered attributes are based on the ISO/IEC standard 25010 on Systems and software Quality Requirements and Evaluation (SQuaRE [83]), as well as literature on non-functional attributes of MT approaches [129, 184, 9, 123]. The high-level attributes under study are usability and maintainability. For each of these attributes, the sub-attributes listed in Table 4.1 are considered.

The scope of this mapping study is limited to model transformation languages as commonly understood in the MDE context. This roughly coincides with the set of MTLs introduced following the QVT Request For Proposals (QVT RFP [133]). Specifically, this SMS does not address graph and program transformation languages, as these are separate and well defined research fields. Including these fields would considerably widen the scope and likely alter the conclusions of our study.
4.2 Research Method

4.2.1 Study Search and Selection

The employed search strategy aims to maximize the number of retrieved peer-reviewed studies published in journals, conference proceedings, and workshop proceedings. Studies pertaining to grey literature (e.g., technical reports, work in progress, newsletters) are not considered, and neither are studies published before the year 2002 (the publication year of the QVT RFP).

The literature search is automated by using the DTU Findit\(^2\) digital library, which indexes journals, conference proceedings, and workshop proceedings published by ACM, IEEE, Springer, Elsevier, Wiley, and CEUR-WS. DTU Findit also indexes literature databases such as Scopus, Compendex, Inspect, and Thomson Reuters Web of Science. These resources collectively cover all journals and conference proceedings in the field of SE, as well as the majority of workshops.

Studies identified by our search must contain the phrase “model transformation”, as well as one of the attributes listed in Table 4.1, within their title or abstract. Based on these considerations, the following conjunctive normal form search term was constructed:

\[
\text{"model transformation"} \quad \text{AND} \\
\text{("usability" OR "learnability" OR "learning" OR "readability" OR} \\
\text{"understandability" OR "understanding" OR "comprehensibility" OR} \\
\text{"comprehension" OR "aesthetics" OR "satisfaction" OR "comfort" OR} \\
\text{"maintainability" OR "modularity" OR "module" OR "composition" OR} \\
\text{"reusability" OR "reuse" OR "modifiability" OR "testability" OR "testing" OR} \\
\text{"test" OR "debugging" OR "tracing" OR "trace")} \\
\text{AND} \\
\text{(year \geq 2002)}
\]

The DTU Findit search engine retrieves studies with titles or abstracts matching this search term. Different variations of the attributes of interest are included in the search term (e.g., both “learnability” and “learning” are included). Additionally, the search engine performs automatic stemming of the search terms, implying that each term is reduced to its elemental root and all known forms of this root are searched for. For instance, searching for the term “testing” would

\(^2\)http://findit.dtu.dk
also retrieve studies containing the terms “test” and “tested”. One limitation of the DTU Findit search engine is that at most 500 search results can be exported to a BibTeX file at one time, leading us to split the search term into several less wide-ranging terms, one for each year between 2002 and 2015. It should be noted that the search was performed in June 2015, implying that any articles published after this date are out of scope.

Selection criteria were applied to the search results with the aim of selecting only those studies which are relevant to our research questions. Study selection took place in two stages, corresponding to the application of exclusion criteria and inclusion criteria. Splitting the study selection process into these two sequential stages aims to eliminate studies that are not of interest as accurately and efficiently as possible.

**Exclusion criteria.** The application of exclusion criteria eliminates studies which are clearly out of scope with a minimum amount of effort. Applying these criteria only requires consulting the title and bibliographical data of the study under consideration. According to our exclusion criteria, the following types of studies were excluded from the mapping study:

- duplicate search results;
- studies whose titles clearly indicate they do not address the area of Model-Based Software Engineering (MBSE), which encompasses MDE;
- studies whose titles clearly indicate they do not address any of the topics of interest;
- studies published in languages other than English;
- studies whose full text is inaccessible to us (e.g. unavailable online or locked behind a paywall); although our institution provides full-text access to the vast majority of venues, exceptions are still possible;
- grey literature: technical reports, conference and workshop summaries, work in progress, newsletters, non peer-reviewed magazine articles, and PhD theses.

**Inclusion criteria.** Only studies addressing one or more of the human factors-related topics listed in Table 4.1 as a main topic were included. We only consider as main topics those topics that make up at least one full section of the study. Implicitly, establishing whether a study satisfies the inclusion criteria requires reviewing its full text.
4.2.2 Data Extraction and Quality Assessment

Once the list of included studies is finalized, data of interest to the mapping study can be collected. First, to address \textit{RQ 1.1}, bibliometric information is collected for each study. This includes publication year, publication venue, and publication type. A three-tier system is used for recording publication type:

- **Tier A**: journal publications;
- **Tier B**: publications appearing in conference proceedings as part of the main technical track;
- **Tier C**: publications appearing in workshop proceedings; publications appearing in conference proceedings outside of the main technical track;

The main benefits of this tiered system are that it differentiates between different types of papers published in conference proceedings, while also acting as a preliminary indication of paper quality.

In relation to \textit{RQ 1.2}, the appropriate entry in Table 4.1 is recorded as the primary topic for each study. Where applicable, additional table entries representing secondary and tertiary topics are also recorded.

To address \textit{RQ 1.3}, studies are classified according to an adaptation of the criteria introduced by Wieringa et al. [211] for classifying publications in the field of Requirements Engineering. The following study types are identified:

- **Solution proposal.** A study presenting an MTL, MT approach, or MT tool addressing human factors considerations;
- **Conceptual proposal.** A study addressing the theoretical foundations of human factors in MT. Possible approaches include formal methods, mathematical theories, cognitive theories, and classification frameworks;
- **Evaluation research.** A study investigating human factors in one or more MT solutions from a practical perspective. Possible approaches include experiments, surveys, interviews, and case studies;
- **Experience report.** A study reporting on the author’s personal experience regarding human factors in MT;
- **Opinion paper.** A study presenting the author’s personal opinion on human factors in MT or in a particular MT solution. Vision papers are included here;
4.3 Results

- **Secondary study.** A literature review of primary studies dealing with human factors in MT.

The name of the MTL or MT solution discussed in each study is recorded in order to answer *RQ 2.1*. The technological space of each solution is also documented in order answer *RQ 2.2*. To this end, the types of models supported by the approach are recorded. We adopt a loose definition of model type, since some approaches define their applicability area from a technological perspective (e.g. any MOF-based modeling language), while others define it from an application perspective (e.g. a specific modeling language). Addressing *RQ 2.3*, the programming language used to implement the approach (if any) is also recorded.

Quality assessment was intentionally not included in the study selection process in order to obtain a complete picture of the human factors in MT research area. Nevertheless, evaluating the quality of the included studies is important for the purpose of understanding the maturity and strength of evidence available in this field. When addressing human factors, the quality of available empirical evidence is of particular interest, both for computer-based and user evaluations. To this end, our quality assessment procedure is based on a subset of the quality checklists proposed in [96]. Three quality checklists were adopted: one for all studies containing an empirical evaluation, one for studies containing a qualitative empirical evaluation, and one for studies containing a quantitative empirical evaluation. The contents of these checklists are listed in Table 4.2. Following the classification in [213], we consider experiments as quantitative studies, interviews as qualitative studies, and case studies and surveys as potentially providing both quantitative and qualitative evidence.

4.3 Results

Our initial digital library search retrieved a total of 3549 studies. The high number of search results is partially due to the fact that the DTU Findit library is an index of other primary sources. Since many studies are found in several primary sources, they appear as duplicate results in a DTU Findit search. Following the removal of duplicates and other studies matching the exclusion criteria, we were left with a corpus of 325 potentially in-scope studies. Out of these, the full text of 9 studies was unavailable. The remaining studies were read in their entirety, and an inclusion decision based on the inclusion criteria described in Section 4.2.1 was made for each one. This process resulted in the final set of 188 studies listed in Appendix A and discussed in the following subsections.
Table 4.2: Quality checklists for studies featuring an empirical evaluation

<table>
<thead>
<tr>
<th>Checklist</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Is the study rationale stated?</td>
</tr>
<tr>
<td></td>
<td>Is the context in which the study was carried out described?</td>
</tr>
<tr>
<td></td>
<td>Are data collection methods appropriately described?</td>
</tr>
<tr>
<td></td>
<td>Are the study outcomes presented?</td>
</tr>
<tr>
<td></td>
<td>Are potential threats to validity discussed?</td>
</tr>
<tr>
<td></td>
<td>Does the study feature human participants? If so, how many?</td>
</tr>
<tr>
<td></td>
<td>What is the background of participants?</td>
</tr>
<tr>
<td></td>
<td>Are study participants appropriately described?</td>
</tr>
<tr>
<td>Quantitative</td>
<td>What is the experiment’s sample size?</td>
</tr>
<tr>
<td>studies</td>
<td>Is the sample size justified?</td>
</tr>
<tr>
<td></td>
<td>Does the study design include a control group?</td>
</tr>
<tr>
<td>Qualitative</td>
<td>Is the selection of statistical analysis methods justified?</td>
</tr>
<tr>
<td>studies</td>
<td>Does the study address its original aims?</td>
</tr>
<tr>
<td></td>
<td>Are the study’s conclusions generalizable?</td>
</tr>
</tbody>
</table>

4.3.1 RQ 1 – Human Factors in MT Literature

Addressing RQ 1.1, the bibliometric trends characterizing the human factors in MT literature are showcased in Figure 4.1. As the bar chart at the top of this figure shows, interest in this topic has developed relatively late in the timeline of MT research. Although the QVT RFP was published in 2002, studies taking into account human factors started appearing in significant numbers only in 2006, with a marked increase starting in 2009 – a full 7 years after the QVT RFP. This indicates that, at least at the time of their initial development, the first generation of MTLs have placed little emphasis on usability and maintainability.

The chart in Figure 4.1 (center) visualizes the most popular venues for publishing human factors in MT research. Only venues with at least 3 publications are shown. Unsurprisingly, the International Conference on Model Transformation (ICMT) is the top venue, with 25 published studies. With its broader focus on modeling topics, the International Conference on Model Driven Languages and Systems (MODELS) is also a popular destination for human factors in MT studies. The International Journal on Software and Systems Modeling (SoSyM), closely associated to the MODELS conference, is in turn a significant venue, with 15 published studies. In fact, most journals, conferences, and workshops included in this chart are SE modeling venues, suggesting that human factors
Figure 4.1: Number of retrieved studies by publication year (top) and publication venue (center); percentage of retrieved studies by publication tier (bottom)
in MT research does not benefit from a community sufficiently large and well defined to warrant a dedicated publication venue.

The doughnut chart at the bottom of Figure 4.1 illustrates the distribution of human factors in MT studies across publication venue tiers. Most studies (59%) were published in Tier B venues: proceedings of conference main technical tracks. Tier A and Tier C venues hold roughly equal shares of the remaining publications. Journals (i.e. Tier A venues) account for only 21% of included studies, with the SoSyM journal publishing the vast majority of them. This finding is interesting and somewhat unexpected in light of the space restrictions imposed by conference and workshop proceedings, which may hinder the appropriate presentation of the type of empirical results required by human factors studies. Nevertheless, the reported percentages are in line with the emphasis historically placed by SE research on conference and workshop publications.

RQ 1.2 is answered by Figure 4.2 (top). The topic most addressed by human factors in MT research is testability, featured in 61 (i.e. close to one third) of the 188 included studies3. The related topic of traceability is addressed by a further 41 studies, while reusability is discussed in 49 studies. In fact, the 5 most commonly addressed topics fall in the sphere of maintainability. Understandability, the most discussed usability-related topic, ranks sixth with 26 studies. Learnability and readability are covered by 11 and 7 studies, respectively, while topics such as aesthetics, user satisfaction, and user comfort are entirely unaddressed in the literature. Overall, maintainability topics (especially testing and debugging) are relatively widely discussed, unlike usability topics.

Figure 4.2 (center) reveals that solution proposals are by far the best represented study type in this research area. As this category of studies covers the introduction of new MTLs and MT tools, its significant share of publications is to be expected. However, the fact that solution proposals account for roughly twice the number of publications as all the other study types combined indicates an imbalance. Among the remaining study types, the relatively low number of conceptual proposals (18) suggests a strong focus on applied research. Finally, with 6 and 7 representatives each, opinion papers and secondary studies are the least common study types. The lack of secondary studies in particular may indicate a low level of consolidation.

It should be noted that a single publication can include sections conforming to different study types. We have counted these sections separately for the purpose of Figure 4.2. For instance, a solution or conceptual proposal may be accompanied by evaluation research. Figure 4.2 (bottom) shows that only 24% of the solution proposals and 11% of the conceptual proposals included in our

---

3Note that any single study may address several research topics.
Figure 4.2: Number of included studies by research topic (top) and study type (bottom); percentages of solution and conceptual proposals that also include evaluations (bottom). A single study may address several research topics and fit several study types.
review are accompanied by evaluation research within the same publication. These percentages are unexpectedly low, and reveal that the vast majority of proposals dealing with human factors in MT are put forward in the absence of evidence to support their claims.

A breakdown of the research topics featured in included publications by study type is presented in Figure 4.3 as a bubble chart. The numbers shown in the chart indicate the number of studies belonging to a particular combination of research topic and study type. Again, note that a single publication may address several research topics and include instances of different study types. The following observations can be extracted from this figure:

- Solution proposals dominate other study types regardless of research topic. A particularly interesting case is that of learnability, where the 10 solution proposals are not accompanied by any instance of evaluation research. Similar discrepancies stand out for composition, modularity, reusability, and testability.

- Some research topics feature a high number of solution proposals in the near complete absence of conceptual proposals. This is the case for debugging, modularity, testability, and traceability, and is indicative of a focus on engineering at the expense of theory building.

- Secondary studies are lacking even for research topics boasting a relatively high number of solution proposals, such as debugging. Conducting such secondary studies could therefore be a fruitful future endeavor.

- Testability stands out as a particularly active research topic, with 55 solution proposals. It also constitutes the subject of 16 evaluations, the highest number among all research topics.

- The only topic in the sphere of usability that is addressed in a consistent and somewhat well rounded manner is understandability, which features 17 solution proposals and 8 instances of evaluation research, with at least one study belonging to each of the other study types.

4.3.2 RQ 2 – Languages and Tools

Only a subset of existing MTLs emphasize human factors in their design, or have been evaluated from a human factors perspective. Figure 4.4 (top) shows the MTLs addressed by at least 3 of the studies included in our review. The Atlas Transformation Language (ATL [89]) is discussed in 43 studies, over three
Figure 4.3: Mapping of study types to research topics. Abbreviations: CP – Conceptual proposal, ER – Evaluation research, OP – Opinion paper, SS – Secondary study, SP – Solution proposal.
times as many as those addressing the QVT-R and QVT-O languages [140] standardized by the OMG. Kermeta [127] and ETL [100] are addressed in 10 and 9 studies, respectively. Interestingly, Java is also discussed as a transformation language in 9 studies. TGGs [162] and GReAT [20] are the graph-based MTLs most often addressed from a human factors perspective, while RubyTL [46], mentioned in 5 studies, is the most commonly addressed embedded DSL. It should be pointed out that the MTLs included in this chart are only a small subset of all MTLs proposed over time by the MDE community – most do not consider human factors at all. Another remarkable aspect is that 5 of the most often addressed 6 MTLs are textual languages, suggesting that visual MTLs suffer from a lack of human factors evaluations.

As indicated by Figure 4.4 (center) in answer to RQ 2.2, the overwhelming majority of included studies present or evaluate MT solutions based on the EMF technology stack. That is, these solutions are implemented using the EMF Java API and operate on models conforming to modeling languages defined as instances of the Ecore meta-metamodel. A much smaller subset of the MTLs described in the included studies provide their own metamodeling infrastructure. In order to transform models using these MTLs, users must first (re-)define any modeling languages of interest using this infrastructure. Transforming existing models is therefore impossible. 10 studies discuss MT solutions for modeling languages defined using OMG’s MOF meta-metamodel, while 9 are restricted to transforming UML models. The predominance of human factors studies addressing EMF technologies over those addressing the standardized MOF-based stack may come as a surprise. Finally, only 6 studies discuss transformations on models conforming to domain-specific modeling languages, while even fewer studies discuss transforming Alloy [85], GEMS [69], or GME models [111].

Some of the included studies state the programming language used to implement the described MTL or MT tool, allowing us to answer RQ 2.3. As indicated by Figure 4.4 (bottom), virtually all implementations are Java-based – 49 out of the 62 studies mentioning an implementation language feature Java in this role. This finding correlates with the predominance of the Java-based EMF as a modeling framework, but the extent of Java’s popularity is nonetheless notable. Ruby and Scala come in as the distant second and third most popular implementation languages, likely due to their support for internal DSLs. Prolog is remarkable as the only mentioned implementation language that does not support the object-oriented programming paradigm.

The overarching answer to RQ 2 is that ATL, EMF, Java are the predominant technologies discussed in human factors in MT studies. In particular, the combination of EMF and Java forms the foundation of most MT approaches taking into account human factors.
4.3 Results

Figure 4.4: Number of retrieved studies by addressed MTL (top), addressed modeling language (center), and implementation language (bottom). A single study may address several MTLs.
4.3.3 RQ 3 – Evaluation Methods and Quality Assessment

Over two thirds of the studies included in our review fail to present any form of empirical evidence in support of their claims, an aspect illustrated in Figure 4.5. Given the context of human factors research, this is perhaps our most counter-intuitive finding, to the extent that it warrants further clarification. In deciding whether a study features an empirical evaluation, our approach has been to simply classify it according to its own claims. That is, studies claiming to present a certain type of evaluation were categorized as such, regardless of the quality of this evaluation or any misuse of terminology. For example, if a publication claims to present a case study, we have classified it as a case study even if the presented evaluation could more accurately be described as an experiment – a situation we have encountered in several instances.

Among studies that include an empirical evaluation, experiments and case studies are almost equally represented, with 26 and 23 respective instances. Two studies include both an experiment and a case study. Meanwhile, interviews, surveys, and think-aloud protocol analyses are not adopted as evaluation methods in any of the included studies.

The results of applying the empirical research quality checklists described in Section 4.2.2 are presented in Table 4.3. The first checklist was applied to each of the 47 studies that feature an empirical evaluation. Out of these, only 36 studies (i.e. 77%) include a clear statement of the study rationale, while 31 (i.e. 66%) present the context in which the study was conducted. Although 42 of the studies benefit from appropriately presented outcomes, only 12 (i.e.
Table 4.3: Number and percentage of affirmative answers to each yes-no question included in the quality assessment checklists

<table>
<thead>
<tr>
<th>Checklist</th>
<th>Item</th>
<th>Affirmative answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Rationale statement</td>
<td>36 of 47 (77%)</td>
</tr>
<tr>
<td></td>
<td>Context described</td>
<td>31 of 47 (66%)</td>
</tr>
<tr>
<td></td>
<td>Data collection methods described</td>
<td>12 of 47 (26%)</td>
</tr>
<tr>
<td></td>
<td>Outcomes presented</td>
<td>42 of 47 (89%)</td>
</tr>
<tr>
<td></td>
<td>Threats to validity discussed</td>
<td>15 of 47 (32%)</td>
</tr>
<tr>
<td></td>
<td>Featuring human participants</td>
<td>9 of 47 (19%)</td>
</tr>
<tr>
<td></td>
<td>Participants appropriately described</td>
<td>6 of 9 (67%)</td>
</tr>
<tr>
<td>Quantitative</td>
<td>Sample size justified</td>
<td>0 of 26 (0%)</td>
</tr>
<tr>
<td></td>
<td>Using a control group</td>
<td>7 of 26 (27%)</td>
</tr>
<tr>
<td></td>
<td>Analysis methods justified</td>
<td>4 of 26 (15%)</td>
</tr>
<tr>
<td>Qualitative</td>
<td>Addressing original aims</td>
<td>16 of 23 (70%)</td>
</tr>
<tr>
<td></td>
<td>Generalizing conclusions</td>
<td>4 of 23 (17%)</td>
</tr>
</tbody>
</table>

26%) describe the data collection methods employed to record these outcomes. A similarly low number of studies – 15, i.e. 32% – address threats to validity.

Despite addressing topics in the area of human factors, only 9 of the 47 studies feature human participants, while the rest of the studies are based on computer experiments. Of these 9 studies, 6 provide descriptions of the participants' backgrounds and other relevant characteristics. Only 2 studies include participants with an industry background, while 2 others include students as participants. Participants with an academic background are featured in 5 studies (one of these studies also features students), and one study omits mentioning its participants' background. The numbers of study participants range between 1 and 22, with the additional observation that all case studies are limited to 1 participant. The mean number of participants across all studies (μ) is 7.5, the median (M) is 4.5, and the standard deviation (σ) is 7.4.

None of the 26 included studies that feature a quantitative evaluation provide a justification or discussion of their respective sample sizes. The sample sizes adopted in the design of these studies vary widely, with values ranging between 1 and 360 (μ=40.5, M=5.5, σ=85). The lack of a sample size justification is indicative of a low quality evaluation, especially for low sample size studies. Furthermore, only 7 of the studies (i.e. 27%) use a control group, casting further doubt over the remaining studies' results. Finally, the suitability of the adopted statistical analysis methods is only justified in 4 of the 26 studies (i.e. 15%).
As far as qualitative evaluations are concerned, 16 of the 23 such evaluations included in our review (i.e. 70%) successfully address their original aims. While this is a positive quality indicator, only 4 of the studies (i.e. 17%) explicitly address the generalizability of their conclusions beyond the sample population.

Our quality assessment findings are summarized as histograms in Figure 4.6. To create these histograms, each study was assigned a quality score by adding one point for each successfully addressed checklist item in Table 4.3. The scores obtained by quantitative and qualitative studies are illustrated separately, as the maximum scores for the two categories differ. The maximum score attainable by quantitative studies is 10, while the maximum score for qualitative studies is 9. Nevertheless, the two plots in Figure 4.6 show similar results: the vast majority of studies are in the low and average quality ranges, with an approximately equal number of low quality and average quality studies. Notably, the most common score for quantitative studies is 2, while no such study has attained the maximum score of 10. At the same time, only a single qualitative study is in the high-quality range, attaining the maximum score of 9.

The presented quality assessment results suggest that the empirical evaluations featured in the included studies suffer from notable shortcomings. The extent of these shortcomings may be high enough to question the validity of some of the existing empirical results in the field of human factors in MT.

### 4.4 Threats to Validity

As any literature review, the Systematic Mapping Study presented in this section faces two categories of threats to validity: threats related to study selection and threats related to the data extraction process. These are also the threats to validity commonly addressed in Systematic Literature Reviews (SLRs) conducted in the area of SE (see, for example, the SLRs by Siegmund et al. [167] and Budgen et al. [36]).

To avoid any bias related to study selection, we have based the selection process on a clearly defined protocol following the guidelines proposed in [96]. Searching for studies using a meta-search tool which indexes publications by all major publishers in the area of SE allows us to assert with a reasonably high degree of confidence that the vast majority of in-scope studies have been retrieved. Nevertheless, the risk that some studies published in low profile venues, especially workshops, have not been retrieved by our search remains. This risk cannot be entirely mitigated by an automated search. The employed search keyword has been developed following a systematic iterative process, and benefits from
4.4 Threats to Validity

The main difficulty we have encountered in the data extraction process was related to study classification. Namely, we have observed that a significant number of studies misuse empirical research terminology, making it difficult to properly classify them in this respect. The terms “case study” and “experiment” are used interchangeably by some authors, and the “case study” label is commonly applied to toy examples. To mitigate this and other threats to the quality of extracted data, a second researcher has carried out the data extraction process on a randomly selected subset of the included studies. Roughly half of the included studies were read by both researchers. Disagreements were solved by
verbal discussion, and have lead to minor adjustments to the data extraction process applied to the remaining studies.

4.5 Discussion

Our findings paint a contrasting picture of the study of human factors in model transformation. In certain respects, this is an established and relatively widely addressed research area within the wider field of MDE. In other respects, it is afflicted by insufficient scientific maturity and lack of consolidation.

The topic of human factors in MT came under the spotlight of the MDE community as recently as 2009, seven years after the QVT RFP triggered a flurry of new and diverse MTL proposals authored by this community. This lengthy delay has arguably been unfortunate, since the fundamental value proposition of MDE is that of improving developer output in terms of productivity and quality, as well as facilitating the involvement of non-developers in the software development process. These objectives are by definition tied to human factors considerations, and their achievement may have benefited from a much earlier explicit and systematic focus on human factors.

The observed publication trends show that there has never been a major (or at least consistent) publication venue dedicated exclusively to human factors in MDE research. The Workshop on Analysis of Model Transformation (AMT) may be the closest approximation of such a venue, although it has a strong focus on formal verification and validation. The lack of a dedicated venue that is also consistent over time could have hindered the formation of a human factors in MT community. At the same time, we notice a focus on conference proceedings as a preferred dissemination method. While in the author’s experience this is in line with SE and CS research in general, it should be noted that the space restrictions imposed by conference proceedings formats can be incompatible with the level of detail commonly present in a reproducible empirical study.

Topic-wise, we have observed a significant imbalance between maintainability-related and usability-related topics, with the former being represented in a much higher number of publications. This is unambiguous evidence showing a research gap that must be addressed in the future. A second research gap has emerged from our findings regarding the predominant study types in the area. Solution proposals are abundant, accounting for over three times as many studies as the second best represented study type – evaluation research. Furthermore, the majority of solution and conceptual proposals are not accompanied by any kind of evaluation, while secondary studies are scarce even on topics relatively rich
in solution proposals. We submit that this tendency to propose brand new languages and tools at the expense of sound scientific evaluation practices is detrimental to the progress of this research area.

Among the evaluations that do appear, very few feature human participants. Notably, interviews, surveys, and think-aloud protocol analyses are completely absent as evaluation methods. This finding may be correlated with the predominance of maintainability-related topics, many of which are suitable for quantitative computer-based evaluations. Nevertheless, the fact that very little is known about how human users employ and interact with MT languages and tools should be a cause for concern, especially in the context of the wider industrial adoption of MDE.

When it comes to technologies, there are some clear “winners” in terms of number of appearances: ATL, EMF, and Java. The focus on textual MTLs in a human factors context is surprising given the common (but often unverified) assumption that such languages are less usable than their visual counterparts. Also, the quasi-universal adoption of EMF and Java as implementation technologies is indicative of a software monoculture, a term originally introduced by computer security researchers [63]. A software monoculture emerges when all applications in a certain domain are underpinned by the same technologies or even the same code base. This situation has the significant drawback that any fault in the common foundation — be it related to security, usability, or maintainability — is propagated to the entire application ecosystem. The prevalence of EMF and Java has led to MDE tools being developed almost exclusively as plug-ins for the Eclipse IDE, a solution singled out for imposing a steep learning curve on novices [24]. Based on our findings, we advocate an increase in the diversity of MDE frameworks and implementation technologies.

The quality assessment of included studies, primarily focused on the quality of empirical evaluations, has also revealed problematic aspects. Namely, discussion points that should appear in any text reporting on empirical research (e.g. data collection methods, threats to validity) are too often overlooked. The majority of reviewed experiments fail to follow existing guidelines for conducting SE experiments, such as those proposed by Wohlin et al. [213], as well as more general experimental design and analysis practices, such as those presented by Montgomery [125]. As a negative highlight, we have only encountered four studies putting forward a justification for the statistical analysis methods employed. Meanwhile, qualitative studies do not address the possible generalization of their results. All in all, the quality of included empirical evaluations leaves room for improvement. Along with the scarcity of evaluations, this observation supports our hypothesis that the field of human factors in MT is insufficiently developed from an empirical research perspective.
The findings of this mapping study can be summarized in one word as “paradoxical”. On the one hand, we see strong evidence of consolidation around certain languages, technologies, and frameworks. On the other hand, evidence that this consolidation is justified from a human factors perspective is insufficient in terms of both quantity and quality. The same term, “paradoxical”, is appropriate for describing the very low number of studies dealing with usability in a field whose very existence is to a large extent predicated on this quality of its exponents. These conclusions offer a strong motivation for the usability-oriented nature of the concepts, languages, and tools presented in the following chapters, as well as the extensive focus on their empirical validation.
We have so far described end-user modelers, as well as their requirements from a model manipulation language and its implementation (see Section 2.2). We have also established that, at least in what concerns the development of model transformation languages, some of these requirements are rarely taken into account (see Chapter 4). Therefore, in this chapter we introduce the VM* family of model manipulation languages aimed at addressing the needs of end-user modelers. We start by identifying the general principles that should guide the creation of model manipulation languages for end-user modelers in Section 5.1. Then, in Section 5.2, we present the modeling scenario employed throughout the remainder of this chapter to exemplify the use of the VM* languages. The members of the VM* language family are subsequently introduced: the Visual Model Query Language (VMQL) is described in Section 5.3, the Visual Model Constraint Language (VMCL) is described in Section 5.4, and the Visual Model Transformation Language (VMTL) is described in Section 5.5.
5.1 Transparent Model Manipulation

End-user modelers come from different and often non-technical professional backgrounds. The design of any model manipulation language adopting them as its target audience must take this observation into account, and keep assumptions about its users’ existing skills to a minimum. The only assumption likely to hold true for end-user modelers is that they are reasonably proficient users of the modeling language and editor that they rely on in their work. It is therefore useful to make explicit some general design principles which, if followed, offer a good indication that a model manipulation language for end-user modelers does not rely on exceedingly ambitious assumptions about its audience. These are the principles of Transparent Model Manipulation (TMM), named so due to their objective of ensuring that a language is “transparent” to its users, allowing them to leverage existing knowledge and minimizing learning effort.

5.1.1 Syntax Transparency

The first TMM principle is that of syntax transparency, which states that model manipulations should be specified using a syntax already familiar to modelers. One candidate for this role is the concrete syntax of the modeling language in which the manipulated model is expressed, as it is implicitly known to modelers. The precise definition of syntax transparency varies slightly for model query, constraint, and transformation languages. Definition 5.1 is applicable to model query and constraint languages, while Definition 5.2 is applicable to model transformation languages. In these definitions, $M$ is referred to as the host metamodel or host language. Note that Definition 5.2 is not easily extendable to transformations operating on two or more metamodels, as this would require the transformation specification to conform to all of them.

\textbf{Definition 5.1} A language capable of expressing specifications for model queries or constraints operating on models conforming to a metamodel $M$ is said to be \textit{syntax transparent} with respect to $M$ iff all such specifications conform to $M$.

\textbf{Definition 5.2} A language capable of expressing specifications for model transformations operating on source models conforming to a metamodel $M$ and producing target models also conforming to metamodel $M$ is said to be \textit{syntax transparent} with respect to $M$ iff all such specifications conform to $M$.

Syntax transparency poses an important challenge: the existing syntax of a modeling language is designed to specify models, as opposed to model ma-
5.1 Transparent Model Manipulation

Manipulations. However, this challenge is not insurmountable. Some modeling languages, such as those based on the MOF infrastructure, provide lightweight extension mechanisms such as profiles and stereotypes. These mechanisms do not break conformance with the host metamodel, and therefore preserve syntax transparency. In their absence, model element naming conventions can be used to identify elements that have additional meaning in the context of a specification. Additionally, most modeling languages support textual annotations or comments, which can act as vehicles for model manipulation instructions.

By implementing the principle of syntax transparency, a model manipulation language addresses some of the requirements imposed by end-user modelers listed in Section 2.2.2. The learnability and understandability requirements ($R_2$ and $R_3$, respectively) are addressed, considering that the number of new syntax elements to be learned is greatly reduced. Maintainability ($R_4$) is also facilitated, under the assumption that testing and debugging of specifications can be carried out using the host modeling language syntax. Finally, the lightweight syntax extension mechanisms mentioned above are applicable to many host modeling languages, thus satisfying the genericity requirement ($R_5$).

5.1.2 Environment Transparency

The learning curve imposed on users by a new model manipulation language has two contributing factors: learning the language itself, and learning how to use the tools supporting it. While the syntax transparency principle mitigates the impact of the first factor, the second factor is addressed by the principle of environment transparency. Just as modelers are implicitly familiar with the host modeling language syntax, they are also familiar with at least one model editor supporting the host language. Using the same editor to specify model manipulations circumvents the costs of learning how to use a dedicated model manipulation editor. The principle of environment transparency as formulated in Definition 5.3 takes this one step further, and specifies that modelers should be able to use their editor of choice for creating model manipulation specifications.

**Definition 5.3** A model manipulation language is *environment transparent* if it allows users to adopt their preferred editor for creating specifications that conform to this language, as well as for viewing the results of applying these specifications to a source model.

Environment transparency is facilitated by syntax transparency, but can also exist independently. For instance, most textual model manipulation languages are supported by dedicated editors, while also allowing the use of general-purpose
text editors as specification tools. They therefore exhibit environment transparency. However, because specifications created using these languages do not conform to the host metamodel, textual MTLs do not exhibit syntax transparency. Due to this counter-example, environment transparency does not generally imply syntax transparency. That being said, the converse implication remains true: syntax transparency always implies environment transparency, since it implies that manipulation specifications conform to the host metamodel.

Many model query, constraint, and transformation languages are experimental, and few are supported by mature, production-ready editors. The ability to specify model manipulations using existing model editors thus benefits end-user modelers in two respects: (1) avoiding the learning curve imposed by a new editor, and (2) leveraging a tested, mature tool. By promoting the loose coupling between manipulation editors and execution engines, environment transparency also facilitates alternative deployment avenues such as remote execution, an approach likely to be beneficial in the case of time-consuming manipulations.

In terms of the end-user modeler requirements listed in Section 2.2.2, environment transparency significantly contributes towards increased learnability (R2) by circumventing the need for users to learn a new tool.

5.1.3 Execution Transparency

The fact that environment transparency allows modelers to select their preferred model manipulation editor can uncover technical limitations, as different editors support different modeling technologies and file formats. For instance, Microsoft Visio [205] is based on a proprietary file format, while other editors such as MagicDraw [118] or Papyrus [147] support incompatible variations of the XMI standard format. If a transparent model manipulation approach is to support all of these editors, it must be able to handle several modeling technologies. To answer this requirement, execution transparency dictates that the execution of manipulation specifications must not be tied to a particular execution engine. Instead, users should have the freedom to select an execution engine appropriate for the modeling technology at hand.

While not as critical as the ability to use a familiar syntax or tool, this liberty can arguably increase the extent to which end-user modelers' expectations are met. For example, in the scenario of a model transformation operating on safety-critical models, users might prefer a transformation engine that supports model checking and state-space exploration over one that aims at highly efficient rule execution. Such scenarios are captured under the principle of execution transparency formulated in Definition 5.4.
5.2 An Example Modeling Scenario

**Definition 5.4** A model manipulation language is *execution transparent* if manipulations expressed using it can be executed by compilation to specifications for one of several execution engines typically (but not necessarily) operating at a lower abstraction level.

In many ways, executing a specification expressed using an execution transparent model manipulation language can be likened to executing a program in a compiled programming language. However, our definition of execution transparency does not impose any restrictions on the level of abstraction offered by the "compilation target". This target may offer low-level model manipulation primitives, or it may be another self-standing model manipulation language with its own execution engine. However, a pre-condition that must be imposed is that the compilation target must support a level of expressiveness at least as high as that of the execution transparent language.

The main contribution of execution transparency towards addressing the end-user modeler requirements listed in Section 2.2.2 concerns utility (*R*1). By having control over how specifications are executed, users can more easily achieve their domain goals. In terms of maintainability (*R*4), the option of tracing and debugging specifications written in an execution-transparent model manipulation language depends on whether the underlying execution engine supports these features. Execution transparency is also a prerequisite for genericity (*R*5), as different execution engines are needed for different modeling technologies. Last but not least, the ability to perform model-to-model manipulations (*R*6) is dependent on the capabilities of the underlying execution engine.

5.2 An Example Modeling Scenario

To exemplify the utility of the VM* languages from an end-user modeler perspective, we consider a modeling scenario representative for the languages’ intended use. Motivated by compliance with legislation regulating the management of public companies (e.g. the Sarbanes-Oxley Act [159]), many such companies create and actively maintain domain models describing their activity. These models allow the company and external auditors to effectively identify violations of the applicable legislation. Furthermore, once the domain model is created, employees can consult it as an authoritative reference. Figure 5.1 shows a hypothetical fragment of such a domain model. The fragment, expressed using UML, describes a part of the activity of a loan-granting financial institution.

The domain model fragment in Figure 5.1 consists of an information model
Figure 5.1: An example domain model representing a financial institution’s loan operations. It consists of an information model expressed as a UML Class Diagram (top-left), a use case model expressed as a UML Use Case Diagram (bottom-left), and a process model expressed as a UML Activity Diagram (right). (top-left), a use case model (bottom-left), and a process model (right). These are respectively expressed using a UML Class Diagram, a UML Use Case Diagram, and a UML Activity Diagram. The information model describes banking products and customers as the main domain entities. As banking products, this institution offers two loan types: installment loans and revolving loans\(^1\). The abstract “Loan” class indicates that a generic loan cannot be offered to customers as a product. The model also states that customers may optionally purchase credit insurance for their loans. The use case model complements the information model by listing the interactions that a customer may have with

\[^1\text{An installment loan is repaid over time as a fixed number of payments, whereas a revolving loan does not have fixed payments. A credit card is an example of a revolving loan.}\]
the institution. Namely, customers may apply for one of the two loan types offered by the institution, specify the details required for a loan application, and purchase credit insurance. Finally, the process model describes the steps followed by the institution upon receipt of a new loan application. According to this process model, the applicant’s eligibility for a loan and existing account with the institution are verified. An eligibility report is subsequently sent to the applicant, optionally accompanied by a credit insurance offer if the requested loan has been granted.

Although they employ a modeling language that has its origins in SE, models of this kind are created and used by domain experts or business analysts, rather than software engineers. The introductory VMQL, VMCL, and VMTL examples in the following subsections will therefore be discussed from the perspective of these user categories. First, VMQL will be introduced from the perspective of a new employee of the financial institution interested in learning about its operations. VMCL will then be introduced from the perspective of an external regulatory compliance auditor interested in determining whether the loan application evaluation process meets industry standards. Finally, VMTL will be introduced from the perspective of a business analyst aiming to improve the quality and regulatory compliance of the model.

5.3 The Visual Model Query Language

5.3.1 Introductory Examples

All VMQL queries are valid model fragments in the host modeling language, and executing a query equates to identifying similar fragments in the source model. As a first example, consider the scenario in which an employee of the financial institution described in Section 5.2 wishes to find all of the institution’s services that are directly accessible to customers. She expresses this inquiry in the form of Query 1 shown in Figure 5.2 (top-left). Query 1 consists of a single VMQL Find Pattern, i.e. a pattern whose matches in the source model are returned as the query result. Two features distinguish this pattern from a regular UML Use Case diagram. The first VMQL-specific feature is the icon in the top-right corner of the diagram, which serves the purpose of visually identifying this diagram as a Find Pattern. The presence of this icon is optional, and it has no bearing on the semantics of the pattern. The second VMQL-specific feature is the name of the Use Case included in the pattern. VMQL variable names are prefixed by the $ character, and $UC is an example of such a variable. As this query includes no VMQL annotations, it is referred to as a base query.
Query 1 can be expressed in plain English as “Find all Use Cases associated to the Actor named «Customer», and store their names in a variable named «$UC».”. Returning to the source model in Figure 5.1, we can see that this query will return four matches, one for each of the “Specify loan details”, “Apply for installment loan”, “Purchase credit insurance”, and “Apply for revolving loan” Use Cases. After each successful match, the $UC variable will hold the name of the matched Use Case as its value. All source model elements not included in the query, such as the Extend relations between some of these Use Cases, will be ignored at query execution time.

Query 2, shown in Figure 5.2 (top-right), is another example of a base query. It identifies all domain Classes that own a Property of type “Product Type”, storing, for each match, the name of the Class in the $C variable and the name of the Attribute in the $P variable. This query has only one match in the source model, namely the “Banking Product” Class owning the “type” Property.

Consider now a case in which the employee wishes to identify all entities in the domain model which do not correspond to a real-life artifact. She is, in other words, interested in finding all abstract domain model Classes. She can express this inquiry using VMQL as shown in Query 3 (see Figure 5.2, center-left). Query 3 features the first example of a textual annotation, a key feature of VMQL. The annotation is encapsulated in a UML Comment distinguished
from other user-defined Comments by the <VM* Annotation> stereotype. It consists of a single assignment, in which the true boolean value is assigned to the “isAbstract” meta-attribute of the annotated Class. Note the use of the self keyword, which acts as an access point to the properties of the annotated model element. This is a special variable, as opposed to $C$, which is a user-defined variable. Special variables are typed value containers with a predefined meaning in VMQL. Also note the navigation operator (".") used to access the meta-attributes, association ends, and operations of a model element. Upon execution, Query 3 will match the “Loan” Class in the domain model.

Because it is formulated as a Class Diagram fragment, the Find Pattern employed in Query 3 fails to match other abstract elements included in the domain model, such as the “Apply for loan” Use Case. VMQL offers a workaround for this in the form of the type special variable exemplified in Query 4 (see Figure 5.2, center-right). This variable allows altering the meta-type of a pattern element, a feature required for including host metamodel elements that cannot be instantiated or lack a concrete syntax in VMQL patterns. In Query 4 it specifies that, even though the pattern consists of a Class, it should in fact match all instances of the Classifier UML metaclass. Since in the UML metamodel the Class and Actor metaclasses are generalized by the Classifier metaclass, Query 4 returns both the “Loan” Class and the “Apply for loan” Use Case. Note how the two variable assignments in Query 4 are separated by a comma. VMQL uses commas to separate any atomic annotations, generally referred to as clauses.

By default, VMQL patterns have an injective match semantics, meaning that any two pattern elements must match different source model elements. However, in some cases a non-injective match is desirable. For example, consider a scenario in which the employee wishes to find all domain Classes associated to the “Customer” Class. She can accomplish this using Query 5 shown in Figure 5.2 (bottom). By assigning the false boolean value to the injective special variable she instructs the VMQL execution engine to also return non-injective matches, thus accounting for the fact that the “Customer” Class may be associated to itself. This is, in fact, the case, and Query 5 returns two successful matches: one including the “Banking Product” Class and one including the “Customer” Class. Were the textual annotation to be omitted from this query, the second match would have not been returned, as it is not injective (both Classes in the pattern match the “Customer” source model Class).

One feature of VMQL that is especially useful for querying process models is the ability to express path queries. Suppose that the considered employee is interested in learning the steps involved in sending an eligibility report in response to a new loan application. She knows only that the process starts with receiving the application and ends with sending the eligibility report, but does not know anything about the Actions taking place in between. Her first attempt
at answering this inquiry is Query 6a shown in Figure 5.3 (top-left). Here, the steps special variable is used to denote a path of defined length consisting of Actions connected by Control Flows. In the resulting matches, the names of the intermediate Actions will be bound to the $A user-defined variable. However, this query returns no results due to the fact that the specified path length of at most two Control Flow edges is too low. The shortest path between the “Receive loan application” Accept Event Action and the “Send eligibility report” Action in the domain model consists of four Control Flow edges. Query 6b shown in Figure 5.3 (top-right) resolves this issue by assigning the “*” placeholder value to the steps special variable. This indicates that paths of any length can be matched. Nevertheless, Query 6b also fails to return any matches, this time due to the fact that the Fork Node and Join Node included in the source model cannot be matched by the employed pattern. Only intermediary model elements of type Action can be matched. Query 6c shown in Figure 5.3 (bottom) makes use of the type special variable to rectify this shortcoming. This query also replaces the steps := * clause with the semantically equivalent indirect clause offered by VMQL as syntactic sugar.

VMQL allows users to control which elements of a pattern should be matched in the source model, and which should not. As an example of this feature’s utility, consider Query 7a in Figure 5.4 (top). The intention of this query is to find all Use Cases in the source model that are not extensions of another Use Case, regardless if an Extension Point is specified or not. In conjunction, the omit and optional clauses allow this intention to be expressed. First, the omit clause specifies that the annotated Use Case and Extension must not appear in any matching source model fragment. Then, the optional clause specifies that the annotated Extension Point may or may not appear in matching source model fragments. In case it does appear, its name is to be stored in the $E user-defined variable. When applied to the model in Figure 5.1, Query 7 matches all Use Cases with the exception of the “Purchase credit insurance” Use Case.

Up to this point, only Find Patterns have been exemplified. However, VMQL also defines Forbid Patterns, which support the specification of negative match conditions. Forbid Patterns are always applied in conjunction with a Find Pattern. Whenever the Find Pattern is matched in the source model, all of its associated Forbid Patterns must fail to match in order for a query result to be returned. Forbid Patterns are a convenience feature, since they can always be replaced by omit annotations in the Find Pattern. The main benefit of their use is the reduction of clutter in case many omit annotations are required. Query 7b in Figure 5.4 (bottom) illustrates a Find Pattern and Forbid Pattern pair. The Find Pattern matches all Use Cases included in the source model, while the Forbid Pattern eliminates all matches in which the matched Use Case is extended by another Use Case. This query is thus equivalent to Query 7a. The id special variable is used to identify corresponding elements in
the two patterns, but pattern element names could have alternatively been used for this purpose. Note that the **Forbid Pattern** has a different visual icon.

### 5.3.2 The Structure of a VMQL Query

As illustrated by the introductory examples, VMQL queries consist of one or more model fragments and, optionally, a set of textual annotations. The VMQL metamodel shown in Figure 5.5 formalizes this observation and provides the precise structure of VMQL queries. A query is composed of one or more patterns, where each pattern is included in exactly one query. Both queries and patterns have user-defined names. VMQL specifies two pattern types: **Find Patterns**
and Forbid Patterns. Each fragment of the source model that matches a query’s Find Pattern is returned as a query result if and only if the same fragment fails to match any of the query’s Forbid Patterns. Queries must consist of exactly one Find Pattern and zero or more Forbid Patterns.

VMQL patterns consist of host language model elements. For instance, as the host language of the patterns featured in Section 5.3.1 is UML, all example patterns consist exclusively of UML model elements. Queries and patterns can include any number of textual annotations, which are also expressed as host language model elements. Such annotations can play one of two roles, depending on which type of model elements they are anchored to. When anchored to
a host language element included in a VMQL pattern, annotations offer additional information or specify constraints related to that specific element. When anchored to the query itself, annotations specify query execution options. See Section 5.3.3 for a description of the available annotations.

Following the principle of syntax transparency, all non-abstract elements of the VMQL metamodel shown in Figure 5.5 are mapped to host metamodel elements. The mapping can be implemented using the host metamodel's extension mechanism, or, if such a mechanism is not available, using model element naming conventions. To illustrate these two scenarios, Figure 5.6 shows how VMQL constructs are mapped to UML and BPMN as host languages. The UML implementation relies on Stereotypes, such as the $\text{VM* Query}$ Stereotype applied to Packages containing VMQL queries, and the $\text{VM* Annotation}$ Stereotype applied to Comments containing VMQL annotations. Meanwhile, the BPMN implementation relies on naming conventions, such as the $[\text{VM* Query}]$ prefix for Package names and the $[\text{VM* Annotation}]$ prefix for Text Annotation IDs.

Metamodel extension mechanisms are the more systematic, and therefore preferred solution for mapping VMQL to a host modeling language. As an example of such a mapping, the VMQL Profile for UML, defined in Table 5.1, includes a Stereotype for each VMQL metamodel element. The $\text{VM* Annotation}$ Stereotype is applicable to Comments, and indicates that a Comment contains one or more VMQL annotations. The $\text{VM* Query}$ stereotype is applicable to Packages, and indicates that a Package contains a VMQL query. Finally, the $\text{VM* Find}$ and $\text{VM* Forbid}$ Stereotypes are applicable to both Packages and Com-
Table 5.1: The VMQL Profile for UML. This Profile is applied to UML Packages containing VMQL query specifications.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies To</th>
<th>Description</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>«VM* Annotation»</td>
<td>Comment</td>
<td>Stereotype applicable to Comments containing VMQL annotations</td>
<td>–</td>
</tr>
<tr>
<td>«VM* Query»</td>
<td>Package</td>
<td>Stereotype applicable to Packages containing a VMQL query specification</td>
<td>–</td>
</tr>
<tr>
<td>«VM* Find»</td>
<td>Package, Comment</td>
<td>Stereotype applicable to Packages containing a Find Pattern or to Comments included in such Packages, in which case the Find icon replaces the UML Comment notation.</td>
<td>![Magnifying Glass Icon]</td>
</tr>
<tr>
<td>«VM* Forbid»</td>
<td>Package, Comment</td>
<td>Stereotype applicable to Packages containing a Forbid Pattern or to Comments included in such Packages, in which case the Forbid icon replaces the UML Comment notation.</td>
<td>![Access Forbidden Icon]</td>
</tr>
</tbody>
</table>

Figure 5.6: Mapping the VMQL metamodel to UML (left) and BPMN (right)

ments. When applied to a Package, these Stereotypes indicate that the Package contains a VMQL Find Pattern or Forbid Pattern, respectively. When applied to a Comment, these Stereotypes replace the standard UML Comment notation with the corresponding icon shown in Table 5.1. The icons are intended as optional visual indicators of a pattern’s type: a magnifying glass indicates the “search” functionality of a Find Pattern, while an “access forbidden” sign indicates that the Forbid Pattern imposes restrictions. Applying these stereotypes to Packages is sufficient for defining a query’s internal structure.
5.3.3 Annotation Syntax

The fact that VMQL patterns are valid model fragments in the host modeling language can potentially limit their expressiveness. Since most modeling languages do not provide pattern specification support, expressing VMQL patterns often requires the relaxation of well-formedness constraints included in the host metamodel. Furthermore, elements of the VMQL metamodel must sometimes be referenced explicitly, and query execution options must be specified. In VMQL, all of these objectives are achieved with the help of textual annotations. Several annotation examples are included in Figure 5.2, Figure 5.3, and Figure 5.4. As described in Table 5.1, these annotations are encapsulated in UML Comments annotated by the VM* Annotation stereotype.

We start our overview of VMQL’s annotation syntax by describing user-defined variables. They can be declared and manipulated within VMQL annotations, and also used as meta-attribute values in pattern specifications. The names of user-defined variables are prefixed by the $ character. Their scope extends across all patterns included in a query, and they are therefore employed for identifying corresponding model elements across different patterns. For example, Query 7a in Figure 5.4 includes a user-defined variable named $A to identify a Use Case that must be matched in the source model by both of its patterns. In the absence of this variable, the two patterns could match different source model elements, thus altering the intended meaning of the query.

The type of a user-defined variable is inferred at query execution time. VMQL supports the Boolean, Integer, Real, and String data types, in addition to the Element data type used for storing instances of host language metaclasses. Regardless of their type, user-defined variables also accept the undefined value ("*"). A variable with this value is interpreted as possibly storing any accepted value of its respective data type.

For variable manipulation, VMQL supports the arithmetic, comparison, and logic operators listed in Table 5.2. Logic operators can be expressed using shorthand notations ("","","","!","->") or full textual notations (and, or, not, if/then). The implication ("->") and disjunction (";") operators can be combined to form a conditional if/then/else construct. The navigation operator ("." ) accesses model element meta-attributes, operations, and association ends.

Apart from user-defined variables, VMQL relies on special variables as a means of controlling query execution (the injective and steps special variables) and accessing the contents of the source model (the id, self, and type special variables). Table 5.3 lists the special variables supported by VMQL, together with their types, descriptions, and usage examples. All operators applicable to
Table 5.2: Operators supported in VMQL annotations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, /</td>
<td>Arithmetic operators applicable to Integer and Real values</td>
<td>1+3, 1.1-0.1</td>
</tr>
<tr>
<td>=, &lt;=, &lt;, &gt;=</td>
<td>Comparison operators applicable to any values</td>
<td>$X$=&quot;name&quot;, $X$&gt;1</td>
</tr>
<tr>
<td>, and</td>
<td>Logic conjunction</td>
<td>$X$=1 , $Y$=2 $X$=1 and $Y$=2</td>
</tr>
<tr>
<td>; or</td>
<td>Logic disjunction</td>
<td>$X$=1 ; $X$=2 $X$=1 or $X$=2</td>
</tr>
<tr>
<td>!</td>
<td>Logic negation</td>
<td>!$X$ not $X$=1</td>
</tr>
<tr>
<td>if &lt;e&gt; then &lt;c&gt;</td>
<td>Logic implication</td>
<td>$X$=true -&gt; $Y$=1 if $X$=true then $Y$=1</td>
</tr>
<tr>
<td>if &lt;e&gt; then &lt;c1&gt; else &lt;c2&gt;</td>
<td>Executes clause &lt;c1&gt; if expression &lt;e&gt; evaluates to true, otherwise executes clause &lt;c2&gt;</td>
<td>if $X$=1 then $Y$=1 else $Y$=2</td>
</tr>
<tr>
<td>.</td>
<td>Accesses a model element’s attributes, association ends, and operations</td>
<td>$X$.visibility=&quot;public&quot;</td>
</tr>
</tbody>
</table>

user-defined variables are also applicable to special variables.

Special variables have a pre-defined scope, identifying the specification fragment to which they are applicable. With the exception of the injective variable, the scope of all special variables listed in Table 5.3 is limited to the annotated model element. The injective variable has a global scope: its value determines how all patterns of a query are matched in the source model.

Clauses are the main building blocks of VMQL annotations: each annotation consists of one or more clauses connected by logic operators. The use of clauses is inspired by logic programming languages, and benefits annotation conciseness. A clause is an assertion about the pattern model elements to which it is anchored, about its containing pattern as a whole, or about user-defined or special variables. The main role of clauses is to act as additional constraints on pattern matching. The clauses included in VMQL’s annotation language are listed in Table 5.4. Note that variable assignment (":=") is treated as a clause.

The either clause can only be included in annotations anchored to several pattern model elements. All other clauses listed in Table 5.4 can be included in annotations anchored to one or more pattern elements. In general, anchoring a clause to several pattern elements instead of creating several annotations...
Table 5.3: Special variables supported in VMQL annotations

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Integer</td>
<td>Stores an optional user-defined pattern element identifier in order to facilitate the identification of corresponding elements across patterns</td>
<td>id:=5</td>
</tr>
<tr>
<td>injective</td>
<td>Boolean</td>
<td>If set to true (the default value), each pattern element can be matched to at most one source model element. Otherwise, each pattern element can be matched to several source model elements.</td>
<td>injective:=true</td>
</tr>
<tr>
<td>steps</td>
<td>Integer</td>
<td>States that the annotated model element, which must represent a relation, can be matched to a chain of relations of the same type in the source model. The length of that chain is determined by the value of this special variable.</td>
<td>steps:=3, steps&gt;3, steps:=*</td>
</tr>
<tr>
<td>self</td>
<td>Element</td>
<td>Represents the annotated model element.</td>
<td>self.visibility := “public”</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Represents the name of the annotated model element’s metaclass. Assigning a new value to this special variable modifies the annotated model element’s metaclass.</td>
<td>type:=&quot;Actor&quot;</td>
</tr>
</tbody>
</table>

Table 5.4: Clauses supported in VMQL annotations

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>Assigns a value to a user-defined variable, special variable, or model element meta-attribute.</td>
</tr>
<tr>
<td>either</td>
<td>Specifies that exactly one of the annotated pattern model elements must be matched in the source model.</td>
</tr>
<tr>
<td>indirect</td>
<td>Specifies that the annotated pattern model element, which must represent a relation, can be matched to a chain of relations of the same type (i.e. the relation’s transitive closure) in the source model.</td>
</tr>
<tr>
<td>omit</td>
<td>Specifies that the annotated pattern model element must not be matched in the source model.</td>
</tr>
<tr>
<td>optional</td>
<td>Specifies that the annotated pattern model element may or may not be matched in the source model.</td>
</tr>
<tr>
<td>unique</td>
<td>Specifies that the annotated pattern model element must be unique within its scope (e.g. its containing Package) in the source model.</td>
</tr>
</tbody>
</table>
containing the same clause leads to more compact specifications. The variable assignment clause (":=") can also appear un-anchored to any pattern elements, as variables always have a query-wide scope.

5.3.4 Operational Semantics

The process of executing a VMQL query is illustrated as a UML Activity Diagram in Figure 5.7. The central operation of this process is the matching of VMQL patterns with corresponding source model fragments. First, matches of the query’s Find Pattern are identified. If no matches are found, query execution terminates and an empty result set is returned. If source model fragments matching the Find Pattern are identified, query execution continues by attempting to match these fragments against all of the query’s Forbid Patterns. If a fragment matches at least one Forbid Pattern, it is discarded from the query’s result set. The remaining fragments, if any, are then returned to the modeler, or, if all matches have been discarded, an empty result set is returned.

As a means of formalizing the identification of matches between VMQL patterns and a source model, it is useful to consider both as typed attributed graphs. This formalization is applicable to both Find Patterns and Forbid Patterns. A similar graph-based interpretation of VMQL match computation has been provided for an earlier version of VMQL in [177]. The interpretation presented here mainly differs in its treatment of annotations. To begin with, a model graph is defined as a typed attributed graph intended for representing models.

**Definition 5.5** A model graph corresponding to a model $M$ is a tuple $(N, E, T, A, V, type, source, target, slot, val)$ where:

- $N$ and $E$ are finite sets of nodes and edges, respectively, with $E \cap N = \emptyset$;
- $T$ is the set of node types corresponding to the metaclasses included in $M$’s metamodel;
- $A$ is the set of node and edge attributes corresponding to the metaattributes included in $M$’s metamodel;
- $V$ is the set of possible attribute values;
- $type : N \rightarrow T$ is a function assigning a type to each node;
- $source : E \rightarrow N$ is a function defining the source node of each edge;
- $target : E \rightarrow N$ is a function defining the target node of each edge;
5.3 The Visual Model Query Language

Figure 5.7: Execution of a VMQL query

- \textit{slot}: \( N \rightarrow 2^A \) is a function assigning a set of attributes to each node; the same name is used for a function assigning a set of attributes to each edge: \( \textit{slot}: E \rightarrow 2^A \);

- \textit{val}: \( N \times A \rightarrow V \) is a partial function associating a value \( v \in V \) to every pair \((n, a)\), where \( n \in N \), \( a \in A \), and \( a \in \text{slot}(n) \).

An additional constraint on the above definition is that edges are uniquely defined by their source, target, and slots, i.e. \( \forall e, e' \in E : (\text{source}(e) = \text{source}(e')) \land (\text{target}(e) = \text{target}(e')) \land (\text{slot}(e) = \text{slot}(e')) \Rightarrow e = e' \). Also, the "undefined" element \((\bot)\) is not a member of any set. Finally, a subscript notation is used in what follows to denote elements of a particular model graph. For example, \( N_g \) and \( E_g \) denote the nodes and edges of model graph \( g \).

We can now proceed to defining \textit{bindings} and \textit{matches} between a VMQL pattern
and a source model, both represented as model graphs. We start by considering base patterns, i.e. patterns that do not include VMQL annotations.

**Definition 5.6** Given two model graphs \( q \) and \( m \) representing a VMQL pattern and a source model, a *binding* is an injective function \( \beta : N_q \rightarrow N_m \).

Based on the definition of a binding as an injective function mapping the nodes of a VMQL pattern to those of a source model graph, a match is defined by adding two additional conditions. First, nodes mapped by the binding must have the same type. Second, model nodes must have at least the same slots, values, and interconnecting edges as the query nodes they are bound to.

**Definition 5.7** A binding \( \beta \) is a *match* between a VMQL pattern \( q \) and a source model \( m \) if the following conditions hold:

1. \( \forall n \in N_q : \text{type}(n) = \text{type}(\beta(n)) \land \forall a \in \text{slot}(n) : \text{val}(n, a) = \text{val}(\beta(n), a) \)
2. \( \forall e \in E_q : \exists e' \in E_m : \text{slot}(e) = \text{slot}(e') \land \beta(\text{source}(e)) = \text{source}(e') \land \beta(\text{target}(e)) = \text{target}(e') \)

Most of the VMQL annotations introduced in Section 5.3.3 have no effect on the above definitions. The majority of them simply modify or augment a pattern before it is matched with a source model. The *self*, *type*, and *steps* special variables are such examples. Other annotations, such as the *either* and *optional* clauses, imply that several different versions of a pattern must be matched with the source model. Again, this does not interfere with each individual pattern’s matching process. The *unique* clause imposes a filter on match results after they have been computed, i.e. only one match is allowed. This filter is also non-intrusive to the match computation. The only annotation that has an effect on match computation is the *injective* special variable, which, when assigned the *false* value, removes the injectivity condition in Definition 5.6.

This view of annotations as immaterial to the graph-based matching process differs from the logic programming-based formalization presented in [177]. There, all annotations are viewed as logic constraints to be checked by an inference engine as part of the match computation process. The simplified annotation semantics proposed in this section is motivated by adherence to the execution transparency principle. The decoupling of pattern matching from interpreting annotations allows a much wider array of existing matching engines, in particular those based on graph matching, to be adopted by VMQL’s implementation. Section 6.1.2 discusses how VMQL is implemented using precisely this strategy.
5.4 The Visual Model Constraint Language

5.4.1 Introductory Examples

Intuitively, model constraints can be regarded as the dual of model queries. To illustrate this, assume (as a simplification) that a model is a set of model elements $ME$, and that $\alpha : ME \rightarrow \{true, false\}$ is a boolean property of model elements $me \in ME$. In this setting, identifying model elements that satisfy property $\alpha$ (i.e. all elements of the set $\{me \in ME \mid \alpha(me)\}$) is a model query, whereas identifying model elements that fail to satisfy property $\alpha$ (i.e. all elements of the set $\{me \in ME \mid \neg\alpha(me)\}$) is a model constraint. Motivated by this insight, VMCL shares the majority of its features with VMQL, while adding a small number of constructs aimed specifically at expressing constraints.

The example constraints discussed in what follows are formulated from the perspective of an external regulatory compliance auditor investigating whether the financial institution described in Section 5.2 complies with applicable rules and regulations. One such rule is that the decision of granting a loan to an applicant must be reached via two independent evaluations. This measure is intended to prevent loans from being granted abusively. In terms of the domain model in Figure 5.1, this rule is translated into a model constraint stating that the “Send eligibility report” Action must be preceded in the Activity’s control flow by two separate instances of the “Check eligibility” Action.

This constraint can be expressed using VMCL as illustrated by Constraint 1 in Figure 5.8, which consists of a single Find Pattern. The focal point of any VMCL constraint is the context annotation, which is anchored to the context element(s) of the constraint. In this example, it is anchored to the “Send eligibility report” Action, indicating that any match of this Action in the source model identifies a potential constraint violation. If the full pattern fails to match at that respective source model location, a constraint violation has been identified. If, on the other hand, the full pattern can be matched, the constraint has not been violated. The two indirect clauses in Constraint 1 specify that two sequences of arbitrary length of Control Flows connecting an Action named “Check eligibility” with an Action named “Send eligibility report” must be matched. Since only one such path exists in the Activity Diagram in Figure 5.1, the Find Pattern fails to match. Constraint 1 is therefore violated in this model.

A second rule of interest to the regulatory compliance auditor states that, with the exception of applications for revolving loans, all loan applications must be accompanied by an explicit specification of loan details (e.g. currency, interest
**Constraint 1**

```
activity Independent Eligibility Checks
  Independent Eligibility Checks
  Check eligibility
  «VM* Annotation» indirect
  Send eligibility report
  «VM* Annotation» context
```

**Constraint 2**

```
package Find [Loan Details Must be Specified]
  «VM* Annotation» context
  Apply for loan
  «VM* Annotation» id := 1
  «include» Specify loan details
```

```
package Forbid [Not Applicable to Revolving Loans]
  «VM* Annotation» id := 1
  Apply for revolving loan
```

Figure 5.8: Example VMCL constraints on the model in Figure 5.1
5.4 The Visual Model Constraint Language

rate). Constraint 2 in Figure 5.8 expresses this rule. It consists of a Find Pattern and a Forbid Pattern. In the Find Pattern, the “Apply for loan” Use Case, as well as the unnamed Use Case it generalizes, are identified as context elements. Thus, the “Apply for loan” Use Case in the source model, together with the “Apply for installment loan” and “Apply for revolving loan” Use Cases it generalizes, represent potential constraint violations. The full Find Pattern in Constraint 2 cannot be matched anywhere in the source model, since the “Specify loan details” Use Case is not included in any other Use Case. Ignoring the Forbid Pattern in Constraint 2, both of the mentioned candidates would be identified as constraint violations. However, the Forbid Pattern states that this constraint is not applicable to the “Apply for revolving loan” Use Case, and therefore only the “Apply for installment loan” Use Case violates Constraint 2. Note the use of the id special variable stating that the unnamed Use Case in the Find Pattern of Constraint 2 corresponds to the “Apply for revolving loan” Use Case in the constraint’s Forbid Pattern.

5.4.2 The Structure of a VMCL Constraint

As illustrated in the introductory examples, VMCL constraints are structurally identical to VMQL queries. The VMCL metamodel shown in Figure 5.9 confirms this: the only difference compared to the VMQL metamodel shown in Figure 5.5 is the replacement of the Query metaclass with the Constraint metaclass. Like a VMQL query, a VMCL constraint consists of one or more patterns, and can also contain VMCL annotations. Patterns consist of host language model elements, and the two defined pattern types are Find Patterns and Forbid Patterns.

The mapping between VMCL constructs and host language elements, illustrated in Figure 5.10 for UML and BPMN, resembles the mapping presented for VMQL in Figure 5.6. However, VMCL introduces the <VM* Constraint> Stereotype and “VM* Constraint” naming prefix for Packages containing constraints.

The VMCL profile for UML is presented in Table 5.5. The <VM* Annotation>, <VM* Find>, and <VM* Forbid> stereotypes are adopted from VMQL. They preserve their meaning and, in the case of the last two, their graphical icon. The <VM* Constraint> stereotype replaces the <VM* Query> stereotype for Packages containing constraints. The only method of differentiating Packages annotated with the <VM* Find> and <VM* Forbid> included in a VMCL constraint from those included in a VMQL query is by consulting the stereotype of the outer Package containing them (either <VM* Query> or <VM* Constraints>). VMCL’s reuse of VMQL pattern names wherever appropriate ensures that users of one language face a minimal learning curve when adopting the other.
5.4.3 Annotation Syntax

The syntax of VMCL textual annotations is, for the most part, identical to that of VMQL annotations. User-defined variables have the same notation, and can be manipulated using the operators already listed in Table 5.2. The same can be said of special variables, as the list of special variables defined in Table 5.3 can also be used in VMCL constraints. In the interest of conciseness, these tables are not reproduced here.

In addition to the clauses supported by VMQL and defined in Table 5.4, VMCL introduces the context clause. Its role is to identify the pattern elements to which it is anchored as context elements of the constraint. A constraint’s context determines the source model locations at which the constraint is applicable. More precisely, the constraint only applies to source model elements
Table 5.5: The VMCL Profile for UML. This Profile is applied to UML Packages containing VMCL constraint specifications.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies to</th>
<th>Description</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;VM* Annotation&gt;</td>
<td>Comment</td>
<td>Stereotype applicable to Comments containing VMCL annotations</td>
<td>–</td>
</tr>
<tr>
<td>&lt;VM* Constraint&gt;</td>
<td>Package</td>
<td>Stereotype applicable to Packages containing a VMCL constraint specification</td>
<td>–</td>
</tr>
<tr>
<td>&lt;VM* Find&gt;</td>
<td>Package, Comment</td>
<td>Stereotype applicable to Packages containing a Find Pattern or to Comments included in such Packages, in which case the Find icon replaces the UML Comment notation.</td>
<td>🕵️‍♀️</td>
</tr>
<tr>
<td>&lt;VM* Forbid&gt;</td>
<td>Package, Comment</td>
<td>Stereotype applicable to Packages containing a Forbid Pattern or to Comments included in such Packages, in which case the Forbid icon replaces the UML Comment notation.</td>
<td>✖️</td>
</tr>
</tbody>
</table>

Table 5.6: Clauses supported in VMCL annotations in addition to those supported in VMQL annotations and listed in Table 5.4

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>Specifies that the annotated pattern model elements represent the context of the constraint. The constraint is only applicable at source model locations matching its context elements. Any VMCL constraint must include at least one instance of this clause.</td>
</tr>
</tbody>
</table>

that match its context elements. Therefore, the absence of context elements from a constraint specification, i.e. the absence of the context annotation from a constraint’s Find Pattern, renders the constraint inapplicable. All VMCL constraints must contain at least one annotation featuring the context clause. Although VMCL introduces only one clause in addition to those already defined for VMQL, it is listed in a separate table (Table 5.6) for ease of reference.

5.4.4 Operational Semantics

The process of executing VMCL constraints, illustrated as a UML Activity Diagram in Figure 5.11, consists of the three steps detailed in what follows.
The VM* Family of Model Manipulation Languages

- Step 1. Matches of the constraint’s context elements (i.e. those elements annotated with the context clause in its Find Pattern) are identified in the source model. If no matches are found, the constraint is deemed inapplicable to the source model, and an empty set of constraint violations is returned. If, on the other hand, the context elements can be matched, the execution process continues with Step 2.

- Step 2. An attempt is made to match the constraint’s full Find Pattern at each successful match location of the context elements. Successful matches represent instances of the constraint being satisfied. Thus, if all matches succeed, an empty set of constraint violations is returned. If at least one match of the full Find Pattern fails, the execution process continues with Step 3.

- Step 3. For each constraint violation identified in Step 2, an attempt is made to match any Forbid Patterns included in the VMCL specification at the violating location in the source model. If at least one Forbid Pattern is matched, the constraint violation is discarded (i.e. not included in the returned set of constraint violations). Violating locations that fail to match any of the Forbid Patterns are returned as results. If no such locations exist, an empty result set is returned.

The pattern matching semantics of VMCL is in line with the graph-based semantics presented for VMQL in Section 5.3.4. The VMCL-specific context annotation does not interfere with the semantics of individual pattern matching, as it simply adds an additional match computation for the annotated elements. The only effect of this additional computation is a potential increase in constraint execution times when compared to query execution times. However, this increase is limited by the fact that, in most cases, a constraint will have relatively few context elements. Matching the model graph they define is therefore less time consuming than matching the graph defined by the full Find Pattern.

5.5 The Visual Model Transformation Language

5.5.1 Introductory Examples

VMQL and VMCL allow identifying patterns and constraint violations within a source model. However, they do not provide any means of modifying the model as a result of identifying a pattern or constraint violation. This capability is provided by the Visual Model Transformation Language (VMTL), which extends
Figure 5.11: Execution of a VMCL constraint
VMQL and VMCL with support for model manipulation. While the extensions brought by VMTL are significant, they are also targeted at end-user modelers. We therefore introduce VMTL via a series of example transformations on the domain model in Figure 5.1. The transformations are discussed from the perspective of a business analyst aiming to improve the model’s quality.

As a first example, consider the “Customer” Actor in the Use Case diagram included in Figure 5.1. This Actor is associated to the “Purchase credit insurance”, “Apply for installment loan”, and “Apply for revolving loan” Use Cases, the last two of which are extended by the first. However, a UML Use Case extending another Use Case “typically defines behavior that may not necessarily be meaningful by itself” (see [143], page 671). This anti-pattern can be removed by deleting the Associations between the Actor and the extending Use Cases. Transformation 1, shown in Figure 5.12 (top), expresses this specification in VMTL. It consists of a single Update Pattern named “Delete Association”. Following a successful match in the source model, an Update Pattern specifies modifications to this model via textual annotations. In this case, the delete annotation states that any Associations between an Actor and an extending Use Case must be removed from the source model. This pattern is applied twice, once for the “Apply for installment loan” Use Case and once for the “Apply for revolving loan” Use Case. The two anti-pattern instances are thus removed from the domain model.

Update Patterns can also be used to create new model elements, as illustrated by Transformation 2 in Figure 5.12 (center). The transformation addresses violations of Constraint 2 in Figure 5.8 by ensuring that all loan requests, with the exception of requests for revolving loans, are accompanied by loan specification details. This is accomplished using the create annotation to add an Include relationship between each Use Case inheriting from the “Apply for loan” Use Case and the “Specify loan details” Use Case. Also note the use of the “.” (navigation) operator to access a Use Case’s name meta-attribute.

Financial institutions must ensure that critical background checks are performed at least twice. The domain model in Figure 5.1 violates this requirement, as the “Check eligibility” Action is performed only once. Transformation 3, shown in Figure 5.12 (bottom), illustrates how VMTL can be used to duplicate this Action so that it is performed twice in parallel. As opposed to the previous examples, this transformation is specified using two patterns: a Find Pattern and a Produce Pattern, which intuitively correspond to the “before” and “after” states of the transformation. The Find Pattern is matched in the source model, determining where the Produce Pattern is applied. In Transformation 3, the Find Pattern will match any sequence of three Actions where the middle Action is named “Check eligibility”. It will do so regardless of the outer Actions’ types: all instances of Activity Node, a UML metaclass generalizing all Action types, are considered via an assignment to VMTL’s type special variable.
Figure 5.12: Example single-rule VMTL transformations on the model in Figure 5.1: Transformation 1 (top), Transformation 2 (center), Transformation 3 (bottom)
When the Find Pattern is matched, the $A$ and $B$ variables are instantiated with the matched Actions' names ("Gather background data" and "Prepare eligibility report", respectively). Once instantiated, the $A$ and $B$ variables retain their values in the Produce Pattern. The differences between the Produce Pattern and the Find Pattern determine which elements will be created or deleted in the source model. Therefore, the outcome of this transformation is that a second Action named "Check eligibility" is created in the source model, together with a Fork/Join Node pair stating that this Action is executed in parallel with the existing Action of the same name. Finally, note that a functionally equivalent transformation can be specified using a single Update Pattern.

The final example discussed here, the "Pull-Up Attribute" refactoring, addresses a widespread UML Class Diagram design anti-pattern [13]. Its specification states that common Attributes of Classes sharing the same abstract superclass must be deleted, and an Attribute with the same name, type, and visibility (i.e. the same signature) must be created in the superclass. In Figure 5.1, this refactoring is applicable to the "amount" and "confidential notes" Attributes shared by the "Installment Loan" and "Revolving Loan" Classes. The VMTL specification for the refactoring relies on two rules. Rule 1, shown in Figure 5.13 (top) addresses the basic case with only two sub classes, while Rule 2, shown in Figure 5.13 (bottom), handles additional subclasses.

Rule 1 contains an Update Pattern and two Forbid Patterns. The Update Pattern will match any Class that has at least two subclasses sharing an Attribute with the same signature. The name of the Class is stored in the $Class$ variable, while the name, type, and visibility of the Attribute are stored in the $Attribute$, $V$, and $T$ variables. The delete annotation removes the Attribute from the subclasses, while the create annotation creates a new Attribute in the superclass. The name, type, and visibility of the new Attribute are set to the values stored in the $Attribute$, $V$, and $T$ variables. Using VMTL's if operator, the visibility of the new Attribute is set to "protected" if the deleted Attribute's visibility was "private", so that it is visible to subclasses.

The two Forbid Patterns of Rule 1 act as application conditions. If any one of them is matched, the rule will no longer be applied to that specific source model fragment, regardless if the Update Pattern is matched. The first Forbid Pattern, named "Attribute in Superclass", ensures that the rule is not applied if an Attribute with the same name as the Attribute to be pulled-up already exists in the superclass. The visibility := * annotation allows the pattern to match any Attribute visibility value. Finally, the refactoring should only be applied if all subclasses of the considered Class own the Attribute to be pulled-up. This condition is enforced by the "Subclass without Attribute" Forbid Pattern using the omit annotation.
Figure 5.13: Example multiple-rule VMTL transformation on the model in Figure 5.1: Transformation 4 – Rule 1 (top), Rule 2 (bottom)
Rule 2 addresses the scenario in which there are more than two subclasses owning an Attribute to be pulled-up. Since an identical Attribute has already been created in the superclass, this rule removes all Attributes appearing in both the superclass and its subclasses. To this end, a single Update Pattern with no additional application conditions is required.

5.5.2 The Structure of a VMTL Transformation

VMTL specifications simultaneously conform to their respective host metamodels and to the VMTL metamodel shown in Figure 5.14. According to this metamodel, a transformation consists of one or more rules, each having a positive integer priority. Rules consist of one or more patterns expressed using the host modeling language, typically at the concrete syntax level. Pattern model elements and meta-attributes that do not have a concrete syntax representation are included in the transformation specification. VMTL patterns correspond to the notions of Left-Hand Side (LHS), Right-Hand Side (RHS), Negative Application Condition (NAC), and Positive Application Condition (PAC) from graph transformation theory [58]. The following pattern types are defined:

- **Find Pattern.** Represents the left-hand side (LHS) of a transformation rule, specifying the source model locations at which the transformation is to be applied. A Find Pattern can be seen as a model query, and a rule may contain at most one such pattern. If the rule does not contain a Find Pattern, it must contain an Update Pattern.
5.5 The Visual Model Transformation Language

**Figure 5.15:** Mapping the VMTL metamodel to UML (left) and BPMN (right)

- **Produce Pattern.** Represents the right-hand side (RHS) of a transformation rule, specifying how the target model is to be obtained from the source model. A rule may contain at most one Produce Pattern, and its presence is conditioned by the presence of a Find Pattern.

- **Update Pattern.** A concise representation specifying both the source model locations at which a transformation is to be applied and how the target model is to be obtained from the source model. A rule may contain at most one Update Pattern, under the condition that it does not contain a Find Pattern.

- **Require Pattern.** Represents a positive application condition (PAC) for a transformation rule. A rule can contain any number of Require Patterns, and will be executed only if all of these patterns are matched in the source model.

- **Forbid Pattern.** Represents a negative application condition (NAC) for a transformation rule. A rule can contain any number of Forbid Patterns, and will be executed only if none of these patterns are matched in the source model.

Just as in the case of VMQL and VMCL, the components of the VMTL metamodel must be mapped to existing elements of the host metamodel. This mapping can be achieved either via lightweight extension mechanisms (e.g., profiles and stereotypes) or via naming conventions. These two scenarios are illustrated in Figure 5.15, showing how VMTL rules, patterns, and annotations are mapped via Stereotypes to Packages and Comments in the case of UML, and via naming conventions to Packages and Text Annotations in the case of BPMN.

To exemplify the full mapping of VMTL constructs to a host language, the VM* Profile for UML is defined in Table 5.7. The profile includes a Stereotype for each non-abstract metaclass of the VMTL metamodel.
Table 5.7: The VMTL Profile for UML. This Profile is applied to UML Packages containing VMTL transformation specifications.

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Applies to</th>
<th>Description</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>«VM* Annotation»</td>
<td>Comment</td>
<td>Stereotype applicable to Comments containing VMTL annotations</td>
<td>–</td>
</tr>
<tr>
<td>«VM* Transformation»</td>
<td>Package</td>
<td>Stereotype applicable to Packages containing a VMTL transformation</td>
<td>–</td>
</tr>
<tr>
<td>«VM* Rule»</td>
<td>Package</td>
<td>Stereotype applicable to Packages containing a VMTL rule</td>
<td>–</td>
</tr>
</tbody>
</table>
| «VM* Find»          | Package, Comment | Stereotype applicable to Packages containing a **Find Pattern** or to Comments included in such Packages, in which case the Find icon replaces the UML Comment notation. | 🕵️
| «VM* Produce»       | Package, Comment | Stereotype applicable to Packages containing a **Produce Pattern** or to Comments included in such Packages, in which case the Produce icon replaces the UML Comment notation. | 🔨
| «VM* Update»        | Package, Comment | Stereotype applicable to Package containing a **Update Pattern** or to Comments included in such Packages, in which case the Update icon replaces the UML Comment notation. | 🕵️
| «VM* Forbid»        | Package, Comment | Stereotype applicable to Packages containing a **Forbid Pattern** or to Comments included in such Packages, in which case the Forbid icon replaces the UML Comment notation. | ⚠️
| «VM* Require»       | Package, Comment | Stereotype applicable to Packages containing a **Require Pattern** or to Comments included in such Packages, in which case the Require icon replaces the UML Comment notation. | ✅
5.5 The Visual Model Transformation Language

Table 5.8: Special variables supported in VMTL annotations in addition to those supported in VMQL annotations and listed in Table 5.3

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>Integer</td>
<td>Determines the application priority of a rule. Only positive values are allowed, with lower values implying a higher execution priority.</td>
<td>priority := 1</td>
</tr>
</tbody>
</table>

5.5.3 Annotation Syntax

The textual syntax of VMTL annotations is based on that of VMQL, with some extensions required for expressing transformations. VMTL user-defined variables, as well as their supported data types, are identical to those introduced for VMQL in Section 5.3.3. The notation and semantics of the operators listed in Table 5.2 remain unchanged in VMTL annotations. However, the scope of user-defined variables is modified. Whereas in VMQL user-defined variables have a query-wide scope, in VMTL their scope is reduced to a single transformation rule. Different transformation rules may contain identically named variables holding different values. Note that this does not impede the use of user-defined variables as pattern element names in view of identifying corresponding elements across patterns, since this identification is based purely on variable names.

All special variables defined in Table 5.3 are also available in VMTL annotations. In addition, VMTL introduces the \texttt{priority} special variable (see Table 5.8) used for specifying rule priorities. Assigning a value to this special variable can occur in any VMTL annotation within a given rule. Typically, this assignment is encapsulated in an annotation anchored to the rule container (e.g. a Package, in the case of UML). The scope of this special variable is limited to one rule.

VMTL also adds several transformation-specific clauses to the list of VMQL clauses defined in Table 5.4. The new clauses, detailed in Table 5.9, enable the creation and deletion of model elements. The \texttt{create} clause specifies that the annotated pattern model elements must be added to the target model, while the \texttt{create if not exists} clause adds the additional condition that identical elements must not already exist in the source model. The \texttt{delete} clause accomplishes the complementary task of deleting the annotated pattern model elements from the source model. These clauses may only be used in Produce Patterns and Update Patterns, as these are the only pattern types that specify model modifications. Furthermore, they must be anchored to at least one pattern model element.
Table 5.9: Clauses supported in VMTL annotations in addition to those supported in VMQL annotations and listed in Table 5.4

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Creates the annotated pattern model element in the target model. If a model element not included in the Find Pattern of a rule is included in the rule's Produce Pattern, the element is implicitly created in the target model. In such cases, the create clause is optional.</td>
</tr>
<tr>
<td>create if not exists</td>
<td>Creates the annotated pattern model element in the target model only if it does not exist in the source model.</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the annotated pattern model element from the source model. If a model element included in the Find Pattern of a rule is not included in the rule's Produce Pattern, the element is implicitly deleted in the target model. In such cases, the delete clause is optional.</td>
</tr>
</tbody>
</table>

5.5.4 Operational Semantics

VMTL is a model-to-model transformation language. It supports endogenous transformations, that is, transformations in which the source and target models conform to the same metamodel [48, 122]. VMTL transformations can be executed in-place to modify an existing model, as well as out-place to produce a new model.

Based on the considerations introduced so far, the full VMTL transformation execution process is illustrated as a UML Activity Diagram in Figure 5.16. The core of this process is a rule priority queue, and the steps of the process are described in what follows.

- **Step 1.** The priority queue is initialized based on the value of each rule’s priority attribute. Rules with lower priority values precede those with higher values in the queue. Rules are ordered randomly within groups of rules with equal priority values.

- **Step 2.** If the priority queue is empty, the transformation execution terminates. Otherwise, the rule at the front of the queue is dequeued.

- **Step 3.** All matches of the dequeued rule’s Find Pattern or Update Pattern in the source model are identified. If no matches are found, Step 2 is executed once more.
Figure 5.16: Execution of a VMTL transformation
- **Step 4.** The application conditions of the rule are verified at each match location by attempting to match the rule's *Forbid Patterns* and *Require Patterns* at these source model locations. If the application conditions are not satisfied at any of the match locations, Step 2 is executed again.

- **Step 5.** The rule is applied to the source model at all matching locations where the application conditions are satisfied. That is, the modifications specified either by the differences between the rule's *Find Pattern* and *Produce Pattern*, or directly by its *Update Pattern*, are applied to the source model. Step 1 is then executed on the updated source model.

The process described above implies that transformations are executed in-place. However, VMTL transformations can be executed following either an in-place or an out-place semantics. To achieve an in-place semantics, the updated source model is persisted to its original location. To achieve an out-place semantics, the updated source model is persisted to a new location (i.e. the target model).

Only *Find Patterns* and *Update Patterns* may trigger the application of a Rule. While all *Update Patterns* can be expressed as semantically equivalent *Find/Produce Pattern* pairs, the reverse statement is not generally true. This is due to the fact that some transformations cannot be specified using a single pattern without violating host metamodel constraints. Consider, for instance, a variation of the 'Pull-Up Attribute' refactoring shown in Figure 5.13. The variation requires existing Attributes in the abstract superclass to be deleted if they have the same name as the Attribute to be pulled-up. The *Update Pattern* in Rule 1 of this transformation would then need to contain two Attributes with the same name, one annotated with the *create* annotation, and one with the *delete* annotation. However, UML does not allow Classes to own multiple Attributes with identical names, so the pattern could not be expressed in a UML-compliant editor. A possible solution is to relax the host metamodel constraints. Nevertheless, because this solution would violate the principles of syntax and environment transparency, it is not adopted by VMTL.

### 5.6 Practical Considerations

The VM* languages illustrate the feasibility of the transparency principles introduced in Section 5.1. That said, their design is equally driven by practical considerations, such as applicability to different modeling languages and ease of implementation. It is therefore worth pointing out precisely what makes VMQL, VMCL, and VMTL *practical* as model manipulation languages.
5.6.1 Host Language Requirements

The VM* languages can manipulate models expressed using host languages that meet the following prerequisites:

- **Prerequisite 1.** The host metamodel must define a *container element*, such as Packages in the case of UML and BPMN. Container elements are used to delimit VM* specifications from other model elements, as well as to specify the internal structure of specifications. Each specification is contained by a separate container instance, as is each pattern that the specification consists of.

- **Prerequisite 2.** The host metamodel must define an *annotation element*, such as Comments in the case of UML and Text Annotations in the case of BPMN. All host metamodel elements must support annotations, which act as a vehicle for VM* clauses.

- **Prerequisite 3 (optional).** The host metamodel should support a lightweight extension mechanism, such as UML Stereotypes, allowing the identification of model elements as VM* constructs. Such a mechanism is optional, and can be substituted by element naming conventions.

These prerequisites are intentionally minimal, and cover a very large number of known useful modeling languages. They ensure that the basic syntactic constructs of VM* can be mapped to host language constructs, effectively allowing syntax transparency. However, one could in principle map the constructs of any existing model manipulation language to the constructs of almost any host modeling language. The only strict limitation is the number of individual constructs provided by the host language. Of course, such a mapping would be highly impractical in almost all cases. Nonetheless, it does help to consider achieving syntax transparency as a problem of optimizing the alignment between the syntax transparent model manipulation language and its targeted host modeling languages.

Another aspect that must be pointed out is that the VM* languages do not require that manipulations are specified using the concrete syntax of the host modeling language. Any syntax supported by the host model editor can be used, including abstract syntax, containment trees, or an alternative visual or textual concrete syntax. Categorizing the VM* languages as concrete syntax query, constraint, and manipulation languages is thus inaccurate.
5.6.2 Execution Engine Requirements

The execution transparency of the VM* languages is also subject to some practical considerations. Namely, an engine capable of executing VM* specifications must meet the following minimal criteria:

- **Pattern matching.** The engine must support a mechanism for expressing model patterns and matching them on models.
- **Variables.** Either statically or dynamically typed variables with user-defined names and values must be supported.
- **Rules.** Self-contained execution units that can be mapped to VMTL rules must be supported.

Additional features such as a rule scheduling mechanism and rule application conditions can be implemented on top of the execution engine. Ultimately, VMTL could be implemented using a general-purpose programming language. However, execution transparency explicitly encourages the reuse of existing model manipulation engines, as this minimizes implementation effort.

Translating VM* specifications into a format compatible with the underlying execution engine requires a one-time software development effort. This is obviously not a task for the end-user modeler, but one for a tool provider. The development effort and skills required from this provider are a function of the selected execution engine: the closer its constructs and semantics to those of VM*, the less effort is required. This was one of the main reasons behind the adoption of the semantically similar Henshin graph transformation engine for the implementation of VM* described in Chapter 6.
Chapter 6

Engineering Evaluation

6.1 Executing VM* Specifications

The three members of the VM* language family are designed to facilitate a common implementation. Because the task of executing model queries and constraints is subsumed by the more complex task of executing model transformations, the common runtime for executing VMQL, VMCL, and VMTL specifications is based on model transformation technologies. We therefore start by describing the execution support for VMTL in Section 6.1.1, and show how support for VMQL and VMCL can be achieved via relatively minor adaptations to the existing model transformation runtime in Section 6.1.2. At the time of writing, only a subset of the architecture described in the following subsections has been implemented in practice. Namely, only base manipulations that do not include textual annotations can currently be executed, and the Web Service API proposed in Section 6.2 is only partially implemented. Nevertheless, this chapter provides a blueprint for a full implementation.

It should be noted that tool support for an earlier version of VMQL has been developed by the author as part of a master’s thesis project. A brief description of this implementation based on the Prolog programming language is available in [2]. However, the mentioned tool is outside the scope of this thesis, and will not be discussed any further.
6.1.1 Executing VMTL Transformations

The implementation of VMTL is based on the Eclipse Modeling Framework and the Henshin model transformation engine. We have adopted Henshin due to its graph transformation-based operational semantics, which aligns well with the semantics of VMTL. As a stand-alone Java API, the Henshin engine also supports VMTL’s environment transparency. However, following the principle of execution transparency, any sufficiently expressive transformation engine could be used to execute VMTL specifications (see Section 5.6.2).

The high-level architecture of VMTL’s implementation, shown in Figure 6.1, consists of three components. First, a general purpose model editor is used to create the source model and transformation specification. The same editor is used to view the produced target model. The Henshin engine applies the transformation specification to the source model, thus producing the target model. Finally, the VM* Runtime – the only strictly necessary component of this architecture created specifically for the purpose of supporting VMTL – compiles VMTL specifications into equivalent Henshin specifications. The compilation performed by the VM* Runtime can be seen as a Higher-Order Transformation (HOT), the four steps of which are shown in Figure 6.1 and described in what follows. An optional VMTL-specific component is a host model editor extension for sending and receiving models and transformation specifications, as well as displaying the target model.

In step 1 model fragments representing transformation rules are identified in the VMTL specification. The Left-Hand Side (LHS), Right-Hand Side (RHS), Negative Application Conditions (NAC), and Positive Application Conditions (PAC) of each rule are respectively identified as VMTL Find, Produce, Require, and Forbid Patterns. A pre-processing step converts any VMTL Update Patterns into equivalent pairs of Find and Produce Patterns. All elements of an Update Pattern annotated with a create or delete clause are omitted from the generated Find Pattern or Produce Pattern, respectively. The identification of model fragments as rules and patterns is informed by VMTL stereotypes (or, alternatively, naming conventions).

In step 2 the extracted model fragments are translated into structurally equivalent Henshin graphs intended to play the same role (LHS, RHS, NAC, or PAC) in the generated Henshin transformation. To avoid binding the implementation to a particular modeling language, the fragments are processed in terms of the Ecore meta-metamodel using the EMF Reflection API. Thus, the task at hand is to perform an exogenous transformation between an Ecore model instance and an instance of the Henshin graph metamodel shown in Figure 6.2. This transformation is facilitated by the fact that Henshin metamodel elements (e.g.
Figure 6.1: The architecture of VMTL’s implementation. Numbers encircled in black show the sequence of steps in the VMTL to Henshin HOT.

Node, Edge, and Attribute) maintain explicit references to corresponding Ecore metamodel elements (e.g. EClass, EReference, and EAttribute).

In step 3 a set of atomic Henshin rules are created by constructing mappings between the nodes of each LHS graph and the corresponding nodes in every other graph belonging to the same rule. As a mapping is a connection between two matching nodes, obtaining the set of mappings between two graphs is equivalent to computing a match between the graphs. The EMF Compare model comparison framework [132] is used for match computation. In order for the generated Henshin rules to have the expected behavior, the computed match must be exact. The use of VMTL’s id special variable to uniquely identify unnamed pattern elements across patterns guarantees an exact match, as the id value can be stored as an attribute of the corresponding Henshin graph node.

In step 4 the generated rules are nested in Units, Henshin’s control flow formalism. Each Henshin rule is assigned the priority of the VMTL rule from which it was derived. First, all rules with the same priority are nested inside a single Independent Unit, allowing non-deterministic rule selection. Next, all Independent Units are assigned as sub-units to a Priority Unit, ensuring that the highest-priority Independent Unit is executed. Finally, the Priority Unit is encapsulated in a Loop Unit, so that it is executed as often as it is appli-
cable. The resulting control structure implements the operational semantics of VMTL: the highest-priority applicable rule is executed until no applicable rules exist, at which point the transformation terminates. This approach does not enforce that all transformation rules must be executed. It is possible that only some rules (i.e. those with higher priorities) will execute. The transformation process is neither stopped nor rolled back if some of the rules are never applied.

VMTL user-defined variables are mapped to Henshin rule parameters, as there is a direct correspondence between the semantics and use cases of these two features: specifying corresponding elements across patterns (or, in Henshin’s case, across graphs), providing user-defined inputs, and functioning as a container for values resulting from transformation execution. User-defined variables are instantiated before rule execution in order to account for user input, and after rule execution in order to provide output values generated by the Henshin engine in the corresponding rule parameter. Henshin rule parameters accept values of any Java primitive type, as well as any EMF model element. This is, effectively, a superset of VMTL user-defined variable types. The only limitation of this approach is that any operations on user-defined variables must be interpreted by the VM* Runtime, as Henshin does not support operations on rule parameters. This slightly increases the VM* Runtime’s complexity.

Nearly all VMTL special variables have a direct correspondent in the Henshin
6.1 Executing VM* Specifications

Executing VM* Specifications

engine, a fact which considerably simplifies the HOT described above. The injective special variable is mapped to the injectiveMatching flag of the Henshin engine, while the priority special variable determines the settings of the Priority Unit generated in step 4 of the HOT discussed above. The self special variable provides access to instances of the Attribute Henshin meta-class, and the type special variable provides access to the type meta-attribute of a Henshin graph node. Finally, the steps special variable is addressed in the graph generation process in step 2 by creating the required number of intermediate nodes and edges to form a path of length equal to the value of this variable. One problematic case, however, is that of edge paths of indefinite length, specified by the steps := * assignment. The Henshin engine currently lacks support for matching the transitive closure of a relation between graph nodes, rendering the execution of this assignment unfeasible.

The interpretation of VMTL clauses in terms of features of the Henshin engine is also relatively straight-forward. The create and delete clauses are handled in step 1 of the HOT, when pairs of Find Patterns and Produce Patterns are generated from semantically equivalent Update Patterns. The either and optional clauses are both handled by generating several Henshin rules for each VMTL rule containing these annotations, with the annotated elements either present or missing from the generated rules. The omit clause is implemented by simply setting the value of the action of the Henshin node generated from the annotated VMTL element to "forbid". However, the indirect clause equivalent to the steps := * special variable assignment mentioned above also lacks a suitable implementation in the Henshin engine.

6.1.2 Executing VMQL queries and VMCL constraints

VMQL queries and VMCL constraints are also executed via the VM* Runtime and Henshin transformation engine, following an approach similar to that adopted for the execution of VMTL transformations. This unified execution approach is based on two observations. First, identifying the locations at which a transformation is to be applied is semantically equivalent to executing a model query. Furthermore, as noted in [176] and discussed in Section 5.4, model constraints and model queries are effectively "two sides of the same coin", and executing a VMCL constraint is reducible to executing two derived VMQL queries.

As shown in Figure 6.3, the high-level architecture of VMQL's implementation resembles that presented in Figure 6.1 for VMTL's implementation. The three main components remain a general purpose model editor, the VM* Runtime, and the Henshin engine. However, some adjustments are required to the roles played by these components, the most significant of which affect the VM*
Figure 6.3: The architecture of VMQL’s implementation. Numbers encircled in black show the sequence of steps in the VMQL to Henshin HOT.

Runtime. It receives as input a VMQL query specification, and produces as output an equivalent query specification suitable for execution by the Henshin engine. It should be noted that, from the point of view of the Henshin engine, the query specification produced by the VM* Runtime is simply the LHS of a transformation specification. The translation performed by the VM* Runtime can therefore again be referred to as a HOT.

Steps 1 and 2 of this HOT are identical to the similarly numbered steps described in Section 6.1.1 for VMTL transformation. Their outcome is a set of Henshin graphs equivalent to the Find Pattern and optional Forbid Patterns contained by the VMQL query. Step 3, however, is significantly simplified. First, the graph produced in Step 2 from the query’s Find Pattern is encapsulated in an atomic Henshin transformation rule in which it plays the role of LHS graph. The graphs resulting from the VMQL query’s Forbid Patterns act as NAC graphs for the same rule. This rule is then encapsulated in a Loop Unit to ensure that all successful matches (as opposed to just the first one) are retrieved. Thus, the match semantics of VMQL as described in Section 5.4.4 is implemented: all source model fragments matching the query’s Find Pattern without also matching its Forbid Patterns are returned.

Note that the Henshin engine receives what is effectively a transformation specification consisting of a rule missing a RHS graph. Our implementation takes advantage of an execution option exposed through the Henshin API which in-
6.1 Executing VM* Specifications

Figure 6.4: The MagicDraw model editor showing the results of VMQL Query 5 in Figure 5.2 by highlighting them in the source model

structs the engine to only return rule matches and omit actually applying the rules at their respective match locations. This allows us to use the Henshin transformation engine as a query execution engine. The output it produces under this configuration is a nested set of source model elements, with each subset containing the source model elements identified in a successful match. These match sets, shown in Figure 6.3 in orange, can simply be displayed to the end-user either visually or textually. A more visually appealing method of displaying the results of a query is to automatically highlight them in the source model using the host model editor. This is illustrated in Figure 6.4, where the results of Query 5 in Figure 5.2 are highlighted in the source model using the host editor – in this case MagicDraw. However, this approach inevitably requires an extension of the model editor to be developed specifically for this purpose, thus sacrificing environment transparency.

The high-level architecture of VMCL’s implementation, shown in Figure 6.5, differs somewhat from that of VMQL. Once provided with a model fragment representing a VMCL constraint specification, the VM* Runtime generates (via Step 1 and Step 2) a pair of Henshin graphs from the Find Pattern, as well as one graph from each Forbid Pattern. The first graph generated from the Find Pattern (referred to as the context graph) is produced from only those pattern elements which are annotated with the context clause (shown in red in Figure 6.5). The second graph (referred to as the full graph) is produced from
the full Find Pattern, including both the red and grey nodes in Figure 6.5.

Based on these two graphs, Step 3 produces two Henshin rules. Rule 1 has the context graph as its LHS, and all the graphs produced from Forbid Patterns as NACs. Rule 2 also has the context graph as its LHS, while the full graph acts as the rule’s NAC. Note how Rule 2 effectively behaves as a model constraint: it identifies the context elements as matches of the LHS graph, but only succeeds as a whole if the NAC graph (i.e. the full constraint specification) fails to match. These atomic rules are encapsulated in a Henshin Sequential Unit, which ensures that Rule 1 is executed first, followed by Rule 2 only if Rule 1 has succeeded. This way, the full execution semantics of VMCL constraints described in Section 5.4.4 is implemented: matches of the context elements that fail to match the full Find Pattern are returned, as long as they do not match any Forbid Patterns. Finally, the Sequential Unit is encapsulated in a Loop Unit to ensure that all constraint violations are found.

Note that only matches of the context elements (i.e. those elements annotated in the Find Pattern with the context clause) are included in the produced set of match descriptions. These elements can then be highlighted in the source model assuming, just like in the case of VMQL, that an appropriate extension exists for the host editor. In absence of such an extension, constraint violation descriptions can be listed textually.
6.2 Model Manipulation as a Service

The architectures presented in Figure 6.1, Figure 6.3, and Figure 6.5 are suitable to several deployment options. In a monolithic plugin-based deployment, illustrated in Figure 6.6 (top-left), a VM* Plugin for a conventional model editor encapsulates both the VM* Runtime and the MT engine. This is arguably the most widespread deployment approach adopted by model manipulation tools today, many of which, including Henshin, are developed as full-featured plugins for the Eclipse IDE. However, this approach offers limited portability, since a separate plugin must be developed for every model editor that is not based on the Eclipse platform.

To improve portability without sacrificing editor integration, the VM* Runtime and the transformation engine can be deployed remotely as the VM* Back-End and accessed via a RESTful Web Service API\(^1\), as shown in Figure 6.6 (top-right). This way, business logic is removed from the editor plugin, facilitating its re-implementation. The plugin is, however, still required, as it must act as a Web service client. In addition, any distributed deployment brings a number of inherent drawbacks. First, an additional artifact warehouse must be included as part of the VM* Back-End in order to avoid transferring potentially large models and manipulation specifications over the network with every HTTP request. Even so, such transfers are inevitable when first uploading a source model or when retrieving a target model, and may become a performance bottleneck. Furthermore, remote model processing requires additional access control provisions that are non necessary in a monolithic deployment.

A third option, illustrated in Figure 6.6 (bottom), is to forego editor integration altogether and develop a separate Web application as a user interface for the VM* languages. This solution allows specifying queries, constraints, and transformations using any editor that supports the host modeling language, without requiring a custom plugin. The drawback of this approach is that users must leave the model editor in order to execute a model manipulation, significantly hindering the implementation of facilities that benefit from editor integration, such as interactive execution and stepwise debugging. The already mentioned issues related to remote model processing also apply.

Each of these deployment options has its own advantages and drawbacks. However, only a service-based deployment can realistically support the transparent model manipulation principles underlying the VM* languages. This type of deployment implicitly facilitates environment transparency by decoupling the model manipulation tool from the model editor. Furthermore, remote execu-

---

\(^1\) Any other remote code execution technology may be used.
Figure 6.6: Deployment options available for VM* tool support: a self-contained model editor plugin (top-left), a thin client model editor plugin (top-right), and a Web application (bottom)
tion is an important step towards execution transparency, as several execution engines can be made available to users without modifying their local modeling environments. Additional execution engines can be deployed as they become available. With these considerations in mind, we propose the VM* API – a unified RESTful Web Service API for the remote execution of VM* specifications.

Following the REST architectural style [61], the VM* API described in Table 6.1 exposes resources for clients to interact with via HTTP requests. Collection resources (models and specs) act as access points to an artifact warehouse storing models and VM* specifications, whereas the corresponding atomic resources (model and spec) represent a unique model or VM* specification stored in this warehouse. The query, constraint, and transformation resources are used to apply a particular VM* specification to a particular model. All resources are manipulated via HTTP requests, where the HTTP method determines the operation to be performed.

All operations invoked via POST or PUT requests carry Ecore files as request payloads. Status messages indicating the success or failure of the request are encapsulated in JSON [82] documents and returned as response payloads. The

<table>
<thead>
<tr>
<th>Resource</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/models</td>
<td>GET</td>
<td>List all uploaded models.</td>
</tr>
<tr>
<td>/models</td>
<td>POST</td>
<td>Upload a model.</td>
</tr>
<tr>
<td>/model/&lt;id&gt;</td>
<td>GET</td>
<td>Download a model.</td>
</tr>
<tr>
<td>/model/&lt;id&gt;</td>
<td>PUT</td>
<td>Replace a model.</td>
</tr>
<tr>
<td>/model/&lt;id&gt;</td>
<td>DELETE</td>
<td>Delete a model.</td>
</tr>
<tr>
<td>/specs</td>
<td>GET</td>
<td>List all uploaded VM* specifications.</td>
</tr>
<tr>
<td>/specs</td>
<td>POST</td>
<td>Upload a VM* specification.</td>
</tr>
<tr>
<td>/spec/&lt;id&gt;</td>
<td>GET</td>
<td>Download a VM* specification.</td>
</tr>
<tr>
<td>/spec/&lt;id&gt;</td>
<td>PUT</td>
<td>Replace a VM* specification.</td>
</tr>
<tr>
<td>/spec/&lt;id&gt;</td>
<td>DELETE</td>
<td>Delete a VM* specification.</td>
</tr>
<tr>
<td>/query/model/&lt;mid&gt;/spec/&lt;sid&gt;</td>
<td>GET</td>
<td>Execute a VMQL query on a model.</td>
</tr>
<tr>
<td>/constraint/model/&lt;mid&gt;/spec/&lt;sid&gt;</td>
<td>GET</td>
<td>Execute a VMCL constraint on a model.</td>
</tr>
<tr>
<td>/transformation/model/&lt;mid&gt;/spec/&lt;sid&gt;</td>
<td>GET</td>
<td>Execute a VMTL transformation a model.</td>
</tr>
</tbody>
</table>

Table 6.1: Operations supported by the VM* API
response JSON document will also include an automatically generated identifier for the newly created resource. Similar status messages are returned in response to DELETE requests. Meanwhile, the type of response payload for GET requests differs as a function of the accessed resource. GET requests to the `models` and `specs` resources return JSON documents listing the uploaded models and VM* specifications, respectively. GET requests to the `model` and `spec` resources return the specified resource in its original Ecore-based format. GET requests to the `query` and `constraint` resources return JSON documents representing lists of query matches or constraint violations. Finally, GET requests to the `transformation` resource return Ecore-based target models.

To exemplify the use of the VM* API, Figure 6.7 shows an HTTP message exchange between a client and the API. The purpose of the exchange is to upload a model and a VMTL specification to the artifact warehouse, and to apply the transformation to the model. First, the model is added to the artifact warehouse via a POST request to the `models` resource. Upon this request’s successful handling, a new `model` resource representing the model is available to the client. A unique identifier is generated for the new resource. This identifier is returned as part of the JSON document attached as payload to the HTTP reply message sent in response to the POST request. The client can parse this document and extract the model identifier. A similar exchange is used to upload the VMTL specification via the `specs` resource, giving the client access to the transformation’s unique identifier. The client can subsequently use the model identifier and the transformation identifier to construct the appropriate URL for a GET request to the `transformation` resource, such that the transformation is applied to the model by the VM* Back-End. The resulting target model is returned to the client as an HTTP response payload.

Since using the VM* API implies uploading models and VM* specifications to a remote server, the security of these artifacts must be ensured. With this in mind, the OAuth [81] authentication framework is a widely deployed solution that can provide some important security guarantees. The most important such guarantee is that a user cannot gain access to the models and specifications uploaded by other users. When combined with a role-based authentication policy, a sound authentication framework such as OAuth is an effective method of managing model access rights. At a technical level, implementing OAuth requires all VM* API clients to obtain an access token prior to using the API. This process can be carried out through a separate channel, such as a dedicated API management Web application.
6.3 Tool Support Limitations

The benefits afforded by the VM* Runtime and its service-based deployment come together with some limitations. Some are specific to the tools described in this chapter, while others are intrinsic to any environment or execution transparent implementation. For instance, as a general consequence of execution transparency, possible incompatibilities between the operational semantics of the VM* languages and the capabilities of the selected execution engine must be taken into account. One such example is the indirect clause, allowing VM* patterns to express a relation’s transitive closure, i.e. a chain of undefined length of instances of this relation. Transitive closure computation is problematic for most graph-based transformation engines, including Henshin.

In the context of environment transparency, model editors are employed for a task they were not designed for – specifying model manipulations. Consequently, the well-formedness and syntactical correctness of manipulation specifications cannot be verified inside the editor in the absence of a dedicated plugin. Most model editors will, however, enforce the conformance of VM* patterns to the host metamodel. The expressiveness limitation resulting from this is mitigated.

Figure 6.7: Example HTTP session between a client and the VM* API. A model and a transformation are uploaded to the artifact warehouse, and the transformation is applied to the model.
by the textual annotations supported by VM* languages. At execution time, tracing and debugging must be performed through an editor extension or outside the model editor, such as through the Web application described as a deployment option in Section 6.2. Finally, in the case of VMTL, displaying target models in the host editor is complicated by the fact that EMF separates diagram layout from the host metamodel. Therefore, the target model of a transformation cannot preserve a layout resembling that of the source model. This inconsistency is emphasized in an environment transparent context, where end-users likely expect the layout to be preserved if the two models are viewed using the same editor. A possible solution for layout preservation in EMF involves extending the host modeling language and using a diagram reconciliation tool [5], but at the cost of breaking both syntax and environment transparency.

To preserve environment transparency, VMTL does not support explicit mappings between the elements of different patterns included in a transformation rule. Instead, the VM* Runtime infers the mappings as described in Section 6.1.1. In contrast, most declarative MTLs assume that these mappings are specified by the transformation developer, since inferring them programmatically requires model elements to have unique identifiers corresponding across patterns. An element's name and type can be used to construct such identifiers, but with no guarantee of uniqueness. Furthermore, some host language elements might not have a name meta-attribute. This may lead to ambiguities when a transformation is executed. VMTL addresses the issue at the language level, by providing the \textit{id} special variable to attach an optional identifier to each pattern element. It is the developer's responsibility to ensure that corresponding elements have the same identifier in all patterns.

Due to its declarative nature, VMTL is susceptible to rule application conflicts. Two rules are said to be in conflict if one of them modifies the source model in a manner that affects the applicability of the other. Furthermore, some VMTL transformations might fail to terminate. For example, any transformation adding elements to a model without imposing application conditions falls in this category. In such cases, the underlying MT engine can support the VMTL transformation developer by formally analyzing specifications. The Henshin engine supports \textit{critical pair analysis}, a technique originating in graph transformation theory [30]. However, this technique has its limitations: the termination of a graph transformation system is undecidable in the general case.

In the current implementation of VMTL, it is the responsibility of the end-user to ensure that transformations eventually terminate. This task is, however, facilitated by the fact that the only execution control mechanisms supported by VMTL are rule priorities and application conditions. Some basic heuristics can therefore be applied to determine if a risk of non-termination exists. The most important such heuristic is the presence of a \textit{create} clause within a given rule.
In the absence of a Forbid Pattern, a Require Pattern, or an omit clause limiting its applicability, the creation of new model elements can continue without termination. On the other hand, VMTL rules including the delete clause in the absence of any create clauses are guaranteed to terminate. As a good practice, we encourage the use of Forbid Patterns and Require Patterns to make rule application conditions explicit, even in cases where they could be embedded as clauses in a Find Pattern or an Update Pattern.

Last but not least, execution performance is a potentially problematic aspect. As already mentioned, transferring large models over a network can become a performance bottleneck. The compilation of VM* specifications to equivalent Henshin specifications also brings a performance penalty. The execution performance of VMQL and VMCL specifications is of particular concern, as the Henshin engine is not optimized for these operations. Most likely, dedicated query and constraint execution engines outperform our implementation. The performance of VMQL has so far only been investigated using a now deprecated Prolog-based implementation, yielding satisfactory results [3]. This at least indicates that the language itself does not rely on performance-hindering constructs. However, a performance evaluation of the current implementation is required (and proposed as future work in Section 8.3).
Chapter 7

Human Factors Evaluation

7.1 Evaluation Plan

The claim that the VM* languages are more usable to end-user modelers than their existing counterparts must be validated empirically by studying human subjects that, ideally, are representative of end-user modelers. To this end, the evaluation plan shown in Figure 7.1 consists of four studies, each assessing the learnability of a member of the VM* language family. We focus on learnability as it is one of the key end-user modeler requirements identified in Section 2.2.2. Furthermore, the Systematic Mapping Study in Chapter 4 reveals that learnability evaluations are altogether missing in MT research (see Figure 4.3).

We first present a user experiment comparing VMQL with OCL in the context of querying business process models expressed using BPMN. The outcomes of this experiment are reported in Section 7.2. Since VMQL and VMCL are by design very similar, a separate experiment comparing VMCL with OCL as business process model constraint languages was not conducted – although such a comparison may be of interest as future work. Section 7.3 presents two closely related user experiments comparing the initial learnability of VMTL with that of a textual MTL (Epsilon) and that of a visual abstract syntax MTL (Henshin). Finally, Section 7.4 adds qualitative data to our evaluation of VMTL's learnability by presenting the results of a think-aloud protocol analysis.
The concept of learnability is a common thread running through these different evaluations. Since several definitions of learnability have been proposed by Human-Computer Interaction researchers (see [55, 131] for examples), we must first specify which aspects of learnability are addressed in our evaluations. We do so using the learnability taxonomy proposed in [73], according to which learnability scope can be separated into initial and extended learnability [73].

The first is concerned with users’ initial performance when first faced with a new language, while the second deals with changes in performance over time. Our evaluations address initial learnability, as it is a necessary prerequisite for the adoption of any new language by end-users.

Once a learnability scope has been established, a user definition describing the population of interest must be adopted. Our evaluations were conducted on novice MTL users with varying degrees of modeling and software development knowledge, ranging from bachelor level CS students to doctoral level students in CS and other research fields. Participants’ backgrounds and their impact on the interpretation of results will be discussed for each study. Finally, in the context of the experiments presented in Section 7.2 and Section 7.3, appropriate evaluation metrics must be selected. We have considered both task metrics measuring the extent to which participants were able to complete a task optimally and the time they required to attempt doing so, and subjective metrics measuring the perceived cognitive load imposed on participants by the use of each language.
7.2 Experiment on the Learnability of VMQL

7.2.1 Methods and Materials

We have conducted a questionnaire-based experiment comparing the initial learnability of VMQL and OCL as query languages for BPMN models. The design of the experiment is summarized in Figure 7.2. The experiment is a crossover study, a variant of within-subject design [88]: all participants were sequentially exposed to each query language. The crossover design is statistically efficient, as it minimizes the number of participants required to correctly identify statistically significant differences between treatments. A replication package containing the questionnaire and collected data is available online (see [1]), and the full questionnaire is shown in Appendix B, Section B.1.

A total of 24 undergraduate CS students took part in the experiment. Based on their subjective self-assessments, nearly all participants had limited modeling experience and little to no experience using BPMN or either of the query languages. Participation was strictly voluntary, and the questionnaires were anonymous except for those participants that chose to identify themselves.

Our experiment consisted of three phases. First, participants were provided a set of written instructions on how to fill the questionnaire, including descriptions of the two query languages. They were asked to read these instructions and allowed to consult them at any time during the experiment.

In the second phase, participants were challenged with two tasks: a comprehension task and a production task. In the comprehension task, participants were presented with eight queries, each given three possible interpretations in plain English. Exactly one of these interpretations was correct in every case, and participants were asked to identify it. Figure 7.3 presents the VMQL version of
Figure 7.3: VMQL specification included in the comprehension task of the experiment. Listing 7.1 shows the equivalent OCL specification.

Figure 7.3: VMQL specification included in the comprehension task of the experiment. Listing 7.1 shows the equivalent OCL specification.

a question included in the comprehension task. The OCL version of the same question is shown in Listing 7.1. The queries are equivalent, and their purpose is to identify all Activities executed immediately before the End Event of a BPMN Process. To answer the question, participants had to select one of the options listed in Table 7.1. In the production task, participants were presented with four queries expressed using plain English, and were asked to write down the corresponding query in one of the two query languages. The following plain English query descriptions were used:

- **Query 1.** Which Tasks are not connected to the End Event?
- **Query 2.** Which Parallel Gateways have branches that start with the same task?
- **Query 3.** What precedes Task A?
- **Query 4.** What are the Choreography Tasks on which Participant A and Participant B collaborate?

Finally, in the third phase, participants were asked to subjectively assess the cognitive load imposed on them by the tasks. Namely, they were asked to rate the difficulty and effort required to solve each task for each of the query languages. They did so using five-point Likert scales.

**Listing 7.1:** OCL specification included in the comprehension task of the experiment. Figure 7.3 shows the equivalent VMQL specification.

```ocl
class Process
  def: query() : Set(Activity) =
    self.flowElements->select(e |
      eoclIsKindOf(Activity) and
      e.outgoing->collect(targetRefoclIsTypeOf(EndEvent))->size > 0)
endclass
```

Two versions of the questionnaire were handed out to study participants in
Table 7.1: Natural language description options provided for the query specifications in Figure 7.3 and Listing 7.1. The correct option is (c).

<table>
<thead>
<tr>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Which Activities are executed more than once?</td>
</tr>
<tr>
<td>(b) Which Task precedes the first Exclusive Gateway?</td>
</tr>
<tr>
<td>(c) Which Activities are executed immediately before the End Event?</td>
</tr>
<tr>
<td>(d) I don’t know.</td>
</tr>
</tbody>
</table>

order to control learning effects. One version begins each task with VMQL questions, while the other begins each task with OCL questions. To mitigate any participant bias about the query languages, VMQL and OCL were referred to in the questionnaires as Language X and Language Y, respectively.

Separate task metrics were computed for the comprehension and production tasks. For the comprehension task, the score obtained by a participant is equal to the number of correct answers provided, whereas for the production task a score between 0 and 10 was subjectively assigned by the researchers for each of the four questions, with a higher score indicating a higher degree of answer correctness. For display and analysis purposes, all scores and cognitive load metrics were normalized to the $[0..10]$ interval.

7.2.2 Observations

We first describe the scores obtained by participants in the comprehension task. Overall means and standard deviations are reported in Table 7.2 and visualized in Figure 7.4 (left). We can observe that the average comprehension score is considerably higher for the VMQL treatment than for the OCL treatment. Looking at individual scores, we find that 47% of the participants performed better with VMQL than with OCL, and only 32% performed better with OCL. The remaining 21% of the participants achieved a perfect score. We see a matching picture when considering standard deviations: the standard deviation is larger for the OCL treatment than for the VMQL treatment. The score differences are statistically significant (two-tailed t-test, $p = 0.015$), while the differences in variance are not statistically significant (F-test, $p = 0.54$).

For the production task, observed scores are considerably lower than those observed for the comprehension task. Again, scores under the VMQL treatment are higher than those under the OCL treatment. Despite the low number of data points (the completion rate for this task was 27%, as opposed to 84% for
Table 7.2: Mean ($\mu$) and standard deviation ($\sigma$) of observed scores for the comprehension and production tasks under the VMQL and OCL treatments. Scores are normalized to the $[0..10]$ interval.

<table>
<thead>
<tr>
<th></th>
<th>Comprehension</th>
<th></th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$</td>
<td>$\sigma$</td>
<td>$\mu$</td>
</tr>
<tr>
<td>VMQL</td>
<td>8.00</td>
<td>4.00</td>
<td>2.34</td>
</tr>
<tr>
<td>OCL</td>
<td>6.82</td>
<td>4.65</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Figure 7.4: Box plots illustrating observed task metrics (left) and cognitive load ratings (right) under the VMQL and OCL treatments.

The findings are corroborated by the subjective assessments (see Table 7.3 and Figure 7.4 (right)), where participants assess OCL as more difficult than VMQL. Testing the comprehension data with a two-tailed t-test, we find that the results are statistically significant, both for the assessment as such ($p = 0.047$), and even more so for their variance ($p = 0.026$). We see similar results for the production task ($p = 0.073$ and $p = 0.018$, respectively). Results for the assessment of effort point in the same direction, but are not statistically significant.
7.2 Experiment on the Learnability of VMQL

Table 7.3: Mean ($\mu$) and standard deviation ($\sigma$) of observed cognitive load ratings under the VMQL and OCL treatments. Ratings are normalized to the $[0..10]$ interval.

<table>
<thead>
<tr>
<th></th>
<th>Comprehension</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Difficulty</td>
<td>Effort</td>
</tr>
<tr>
<td></td>
<td>$\mu$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>VMQL</td>
<td>5.47</td>
<td>1.64</td>
</tr>
<tr>
<td>OCL</td>
<td>6.95</td>
<td>2.79</td>
</tr>
</tbody>
</table>

7.2.3 Interpretation

Participants clearly perform better on average when using VMQL than when using OCL. This holds true for both the overall average and the individual performance, and both with regards to absolute scores and their variance. These findings are corroborated by the subjective assessments, where participants consistently report lower cognitive load on the VMQL tasks than on the OCL tasks. The results of this experiment therefore allow us to conclude with a high degree of certainty that the initial learnability of VMQL is higher than that of OCL.

Arguably, the most indicative result is seen in the observed variances. Overall, they are relatively high, which we attribute to the somewhat low qualification level of participants. However, variances are consistently lower for the VMQL treatment than for the OCL treatment. This may indicate that the difficulty of using VMQL is lower than that of using OCL, so that individual variations in general cognitive ability are more visible in the more difficult task. Put in other words: capable participants can cope with OCL's difficulty, while less capable ones cannot. For VMQL, there is less need to cope, so we see less variation.

Another interesting outcome of this experiment is the very large difference between the scores obtained by participants in the comprehension and production tasks, respectively. This difference can also be partially attributed to participants' backgrounds, as none of them had extensive prior contact with any of the query languages, or with model querying in general. Their ability to produce syntactically and semantically correct queries without the benefit of tool support can therefore be expected to be low. Nevertheless, the fact that virtually none of the participants were able to formulate correct OCL queries, even when provided with a written description of the language, indicates notable initial learnability deficiencies for this language.
Concerning the cognitive load assessments, Figure 7.4 (right) shows that none of the participants provided ratings of "low" or "very low" for perceived difficulty and effort. This suggests that, despite the comparatively good ratings obtained by VMQL, there is still room for improvement. Also, the fact that participants provided higher cognitive load ratings in the context of the production task comes as a confirmation of the lower scores obtained for this task. Participants were very likely aware of their poor performance, suggesting that the query languages at least succeed in avoiding to offer users a false sense of confidence.

7.2.4 Threats to Validity

There are several potential threats to the validity of our results. We enumerate them and discuss their mitigation in what follows. First, we eliminated bias through the experimenter by assigning the tasks randomly, providing written instructions, and asking participants to fill in the questionnaires anonymously. We eliminated bias through learning effects by the randomized blocking design of our study, with all treatments occurring with approximately the same frequency. This way, any learning effects are canceled out.

Bias through unrepresentative population sample was controlled by considering a relatively large sample size \(n = 24\), but it must be said that the sample population is relatively homogeneous. Also, despite being novices in model querying, participants had a background in CS and limited modeling skills, very likely making them less representative for end-user modelers. Another potential source of bias is the measurement procedure, in particular with respect to cognitive load measures. We collected two different measurements (difficulty and effort), both of which are recognized as aspects of cognitive load (see [145]). These measurements show the same effect, though to varying degrees. Using subjective assessments rather than objective measurements such as skin conductivity or pupillary dilatation is justified by the high correlation between subjective and objective measurements of cognitive load (see [71]).

It should be mentioned that the reported experiment was carried out in the spring of 2013, using a previous version of VMQL's syntax. There is a risk that changes brought to the syntax since that time — including changes made as a result of this experiment’s outcomes — mean that the results are no longer representative for the version of VMQL presented in this thesis. Nevertheless, these changes have not altered VMQL's core query by-example semantics. They mostly concern the textual annotation syntax, which has been updated to match that of the other members of the VM* language family. We therefore argue that VMQL's superiority relative to OCL in terms of learnability has been preserved.
Finally, we note that only some of the reported observations are statistically significant. All of the reported observations are, however, highly consistent. They support and corroborate each other, allowing valid overall conclusions.

As a quantitative empirical study, our experiment can be measured against the quality checklists listed in Table 4.2 and used in the Systematic Mapping Study presented in Chapter 4. Applying these quality criteria, we note that the rationale, context, and data collection methods used in our experiment are appropriately described. Its outcomes and threats to validity are also discussed. The experiment features a sufficiently large number of human participants, whose backgrounds are known and described. However, participant's background is not entirely suited to the research questions. There is no control group, but the employed statistical analysis methods are discussed and justified. Our experiment and its description in this section thus warrant a score of 9 out of a maximum of 10 points in this quality assessment, placing the study in the "high quality" bracket according to Figure 4.6 (top).

7.3 Experiments on the Learnability of VMTL

7.3.1 Methods and Materials

We have carried out two questionnaire-based experiments comparing the initial learnability of VMTL with that of a textual MTL (Epsilon) and a visual abstract syntax MTL (Henshin). The designs of the two experiments, referred to in what follows as Experiment 1 and Experiment 2, are summarized in Figure 7.5. Similarly to the learnability evaluation of VMQL described in Section 7.2, both experiments presented here are crossover studies. A replication package containing the questionnaires, statistical analysis scripts, and the collected data is available online (see [1]). The questionnaires used in Experiment 1 and Experiment 2 are respectively shown in Appendix B, Section B.2 and Section B.3.

We selected Epsilon and Henshin as comparison points for VMTL due to their widespread adoption.\(^1\). Taken together, the characteristics of Epsilon, Henshin, and VMTL span the gamut of state-of-the-art MTL approaches. Each of these languages shares important syntactic and semantic commonalities with other MTLs in its respective category. It is therefore arguable that our findings extend to similar MTLs.

\(^1\)A list of Epsilon’s industrial users is available at https://www.eclipse.org/epsilon/users/. A collection of publications describing Henshin’s use in various projects is available at https://www.eclipse.org/henshin/publications.php.
Experiment 1 took place in Spring 2014 and included 34 bachelor level CS students as participants. Experiment 2 took place in Spring 2015 and included 40 bachelor, master, and doctoral level CS students. In a subjective self-assessment, over 80% of participants rated their own knowledge of UML and programming as good or very good, and their knowledge of OCL and MT as poor or very poor. Despite participants’ CS background, these self-assessment ratings are in line with the skill set of end-user modelers presented in Section 2.2. Immediately before the experiments, participants were offered a written hand-out containing a brief introduction to MT and descriptions of the three MTLs. They were asked to read the hand-out and allow to consult it during the experiment. Participants were then presented with a questionnaire consisting of two sections: a comprehension section and an assessment section. Different questionnaires were used in Experiment 1 and Experiment 2.

The comprehension sections of questionnaires used in both experiments contain a total of nine multiple-choice questions, three for every MTL. To answer a question, participants were required to select the correct natural language description of a given MT specification from a set of three answer options. In Experiment 1, each participant was presented with the same transformation three times, once for every MTL. In Experiment 2, participants were presented with each transformation only once, with a balanced number of participants receiving each transformation specified in each MTL. The MT specifications included in Experiment 1 operate on UML Class, Use Case, and Activity diagrams. In addition to these diagram types, Experiment 2 also includes specifications operating on
Table 7.4: Answer options provided in Experiment 2 for the transformation specifications in Figure 7.6. The correct option is (c).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>If an Action has exactly one outgoing Control Flow to a Decision Node, remove the Decision Node.</td>
</tr>
<tr>
<td>(b)</td>
<td>If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, remove the guards.</td>
</tr>
<tr>
<td>(c)</td>
<td>If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, add a Decision Node between the Action and its outgoing Control Flows.</td>
</tr>
<tr>
<td>(d)</td>
<td>I don’t know.</td>
</tr>
</tbody>
</table>

UML Object diagrams. The comprehension section produces our experiments’ task metrics: the comprehension score, i.e. the number of correct answers provided by a participant for each MTL (ranges between 0 and 3), and the time required by a participant to answer the questions for each MTL.

As an example, the Epsilon, Henshin, and VMTL specifications of a transformation used as a comprehension question in Experiment 2 are shown in Figure 7.6. The answer options for this question are listed in Table 7.4, with (c) being the correct option. This example is representative for the type of transformations included in our experiments, namely concise in-place model updates as commonly used in the context of model quality assurance [13].

The sizes of the transformation specifications included in our experiments are summarized in Table 7.5. The size of the Epsilon specifications is measured by counting lines of code, while the size of the Henshin and VMTL specifications is measured by counting individual shapes, line segments, and textual labels. According to [179], the latter is an appropriate diagram size metric. Questions included in Experiment 2 address slightly more complex transformations and offer at least two plausible answer options per question, leading us to replace the Epsilon Transformation Language (ETL) with the closely related but less constraining Epsilon Object Language (EOL).

The assessment section of the questionnaires addresses participants’ subjective evaluation of the cognitive load imposed by each MTL. Two metrics were collected using Likert scales: difficulty and effort ratings. To facilitate evaluating the effect of the MTLs on participants with different skill and capability levels, data originating from high-performing and low-performing participants is analyzed separately in what follows. The average comprehension score is used as a threshold value for identifying high-performers and low-performers.
Figure 7.6: Equivalent transformation specifications in Epsilon (top), Henshin (center), and VMTL (bottom) used as questions in Experiment 2
7.3 Experiments on the Learnability of VMTL

Table 7.5: Mean ($\mu$), median ($M$), and standard deviation ($\sigma$) of the sizes of the transformation specifications used in comprehension questions

<table>
<thead>
<tr>
<th>Specification Size</th>
<th>$\mu$</th>
<th>$M$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epsilon Experiment 1</td>
<td>18.00</td>
<td>15</td>
<td>4.24</td>
</tr>
<tr>
<td>Henshin Experiment 1</td>
<td>22.67</td>
<td>27</td>
<td>8.34</td>
</tr>
<tr>
<td>VMTL Experiment 1</td>
<td>26.00</td>
<td>28</td>
<td>4.32</td>
</tr>
<tr>
<td>Epsilon Experiment 2</td>
<td>18.44</td>
<td>16</td>
<td>8.37</td>
</tr>
<tr>
<td>Henshin Experiment 2</td>
<td>78.00</td>
<td>68</td>
<td>33.40</td>
</tr>
<tr>
<td>VMTL Experiment 2</td>
<td>21.22</td>
<td>21</td>
<td>5.49</td>
</tr>
</tbody>
</table>

Analysis of Variance (ANOVA [125]) and the Wilcoxon signed rank test were adopted for statistical hypothesis testing. Effect sizes were evaluated using the $\eta^2$ statistic in the case of ANOVA\(^2\), and Spearman’s $\rho$ in the case of the Wilcoxon signed rank test\(^3\). The variance homogeneity and normal distribution of observations required as prerequisites for applying ANOVA were verified, as recommended in the literature [125], using residual plots and Q-Q plots.

7.3.2 Observations

The means and standard deviations of the comprehension scores resulting from Experiment 1 and Experiment 2 are presented in the leftmost data columns of Table 7.6 and Table 7.7, respectively. The scores are visualized as stacked bar graphs in Figure 7.7. Each horizontal bar in the figure is split into sections corresponding to possible comprehension scores. The size of each section is proportional to the number of participants which have obtained that score. Since in both experiments the comprehension section consists of three questions for each transformation language, possible comprehension scores range between 0 and 3. In Figure 7.7, lighter colors correspond to higher scores: white sections represent the proportion of participants that have obtained the maximum score (3), while dark grey sections represent the proportion of participants that have obtained the minimum score (0). All plots in the figure are centered by a vertical line drawn between the sections corresponding to scores of 1 and 2.

\(^2\)Guidelines suggest that $\eta^2$ values greater than 0.06 indicate a moderate effect size, and values greater than 0.14 indicate a large effect size [43].

\(^3\)Values for Spearman’s $\rho$ range in the interval $[-1, 1]$. Values closer to 0 indicate a lower correlation, and therefore a larger effect size.
When it comes to comprehension scores, in Experiment 1, language is a significant factor for both high-performing and low-performing participants \((p = 0.01\) and \(p < 0.01\), respectively). However, effect size is larger for low-performers \(\eta^2 = 0.23\) than for high-performers \(\eta^2 = 0.17\). In the case of high-performers, both Epsilon \((p = 0.01, \rho = 0.18)\) and VMTL \((p = 0.03, \rho = 0.16)\) are associated to significantly higher scores compared to Henshin, while the difference between scores obtained under Epsilon and VMTL is not statistically significant \((p = 0.66)\). Similar relative score differences can be observed for low-performing participants: Epsilon \((p < 0.01, \rho = -0.07)\) and VMTL \((p = 0.04, \rho = -0.05)\) are associated to significantly higher scores compared to Henshin, while the difference between scores obtained under Epsilon and VMTL is not statistically significant \((p = 0.16)\). On the other hand, in Experiment 2, language only has

### Table 7.6: Mean \((\mu)\) and standard deviation \((\sigma)\) of comprehension scores, comprehension times, and cognitive load ratings by high-performing (H.P.) and low-performing (L.P.) participants in Experiment 1

<table>
<thead>
<tr>
<th>Score [0..3]</th>
<th>Time (s)</th>
<th>Diff. [1..5]</th>
<th>Eff. [1..5]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\mu)</td>
<td>(\sigma)</td>
<td>(\mu)</td>
</tr>
<tr>
<td><strong>H.P.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epsilon</td>
<td>2.65</td>
<td>0.60</td>
<td>325</td>
</tr>
<tr>
<td>Henshin</td>
<td>2.00</td>
<td>0.71</td>
<td>240</td>
</tr>
<tr>
<td>VMTL</td>
<td>2.53</td>
<td>0.63</td>
<td>275</td>
</tr>
<tr>
<td><strong>L.P.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epsilon</td>
<td>1.82</td>
<td>0.95</td>
<td>268</td>
</tr>
<tr>
<td>Henshin</td>
<td>0.76</td>
<td>0.56</td>
<td>245</td>
</tr>
<tr>
<td>VMTL</td>
<td>1.35</td>
<td>0.79</td>
<td>245</td>
</tr>
</tbody>
</table>

### Table 7.7: Mean \((\mu)\) and standard deviation \((\sigma)\) of comprehension scores, comprehension times, and cognitive load ratings by high-performing (H.P.) and low-performing (L.P.) participants in Experiment 2

<table>
<thead>
<tr>
<th>Score [0..3]</th>
<th>Time (s)</th>
<th>Diff. [1..5]</th>
<th>Eff. [1..5]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\mu)</td>
<td>(\sigma)</td>
<td>(\mu)</td>
</tr>
<tr>
<td><strong>H.P.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epsilon</td>
<td>2.22</td>
<td>0.52</td>
<td>441</td>
</tr>
<tr>
<td>Henshin</td>
<td>2.09</td>
<td>0.79</td>
<td>495</td>
</tr>
<tr>
<td>VMTL</td>
<td>1.96</td>
<td>0.93</td>
<td>326</td>
</tr>
<tr>
<td><strong>L.P.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Epsilon</td>
<td>1.06</td>
<td>0.66</td>
<td>420</td>
</tr>
<tr>
<td>Henshin</td>
<td>0.71</td>
<td>0.59</td>
<td>351</td>
</tr>
<tr>
<td>VMTL</td>
<td>1.47</td>
<td>0.80</td>
<td>402</td>
</tr>
</tbody>
</table>
7.3 Experiments on the Learnability of VMTL

<table>
<thead>
<tr>
<th>Performance</th>
<th>Experiment 1: Score</th>
<th>Experiment 2: Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Epsilon</td>
<td>Epsilon</td>
</tr>
<tr>
<td></td>
<td>Henshin</td>
<td>Henshin</td>
</tr>
<tr>
<td></td>
<td>VMTL</td>
<td>VMTL</td>
</tr>
<tr>
<td>Low</td>
<td>Epsilon</td>
<td>Epsilon</td>
</tr>
<tr>
<td></td>
<td>Henshin</td>
<td>Henshin</td>
</tr>
<tr>
<td></td>
<td>VMTL</td>
<td>VMTL</td>
</tr>
</tbody>
</table>

**Figure 7.7:** Stacked bar graphs illustrating comprehension scores for each MTL. Lighter colors correspond to higher scores: white sections show the proportion of participants obtaining the maximum score (3), dark grey sections show the proportion of participants obtaining the minimum score (0).

A statistically significant effect on comprehension scores for low-performing participants ($p = 0.02, \eta^2 = 0.18$). For this participant group, VMTL is associated to significantly higher scores than Henshin ($p = 0.02, \rho = -0.26$), while other score differences are not statistically significant.

Completion times for the comprehension task are shown in the second group of data columns in Table 7.6 and Table 7.7, respectively. They are illustrated as box plots in Figure 7.8. A first observation is that completion times are longer for Experiment 2, which features slightly more complex transformations. In terms of the effect of language, Experiment 1 participants have required more time to answer questions under the Epsilon language than under the other two MTLs, although the difference is only slightly significant for high-performers ($p = 0.06, \eta^2 = 0.12$), and not significant for low-performers. A similar trend is visible for low-performers in Experiment 2, but again lacking significance. In contrast, the completion times observed for high-performers in Experiment 2 are highly dependent on language ($p < 0.01, \eta^2 = 0.2$). Here, VMTL is possibly associated with shorter completion times than both Epsilon ($p = 0.07$) and Henshin ($p = 0.02, \rho = 0.27$), while Epsilon is possibly associated with shorter completion times than Henshin ($p = 0.06$).
Figure 7.8: Box plots illustrating comprehension question completion times for each MTL, grouped by experiment and participant performance.

Difficulty and effort ratings are summarized in the rightmost data columns of Table 7.6 and Table 7.7, and illustrated in Figure 7.9 as stacked bar graphs. The bars in Figure 7.9 are based on a 5-point scale of possible rating values. Lighter colors correspond to higher difficulty and effort ratings. In the case of Experiment 1, difficulty ratings do not significantly differ as a function of the considered transformation language. The only visually apparent difference, the higher difficulty ratings assigned by high-performing participants to VMTL, is not statistically significant (\( p = 0.23 \) and \( p = 0.33 \) when compared to Epsilon and Henshin, respectively). Similarly low differences in difficulty ratings can be observed in the case of Experiment 2. The only exception is represented by the significantly higher difficulty ratings assigned by high-performing participants to Henshin compared to VMTL (\( p = 0.05, \rho = -0.11 \)).

For Experiment 1, effort ratings generally follow the same pattern as difficulty ratings. However, whereas VMTL was rated as slightly more difficult by high-performing participants, the same participants appear to rate Henshin as requiring higher effort, though the increase is not statistically significant (\( p = 0.75 \) and \( p = 0.44 \) when compared to Epsilon and VMTL, respectively). The only statistically significant impact of an MTL on any of the cognitive load


**Figure 7.9:** Stacked bar graphs illustrating cognitive load ratings for each MTL. Lighter colors correspond to higher ratings: white sections show the proportion of participants assigning the maximum rating (5), dark grey sections show the proportion of participants assigning the minimum rating (1).
measurements is registered for the effort ratings of Experiment 2. Here, high-performing participants rate VMTL as requiring significantly less effort than Henshin ($p < 0.01$, $\rho = 0.41$), and potentially significantly less effort than Epsilon ($p = 0.1$, $\rho = 0.19$). A similar trend, but lacking statistical significance, can be observed for low-performing participants.

7.3.3 Interpretation

None of the evaluated MTLs has emerged as a clear "winner" from our experiments. While Epsilon and VMTL outperform Henshin in terms of comprehension scores, each is superior to the other in some respects: Epsilon is associated to slightly higher comprehension scores, and VMTL is associated to shorter completion times and lower cognitive load ratings. Put in the context of our findings regarding the learnability of model querying languages presented in Section 7.2, where VMQL clearly dominates its textual counterpart (OCL), these outcomes are somewhat unexpected.

The score differences between the visual MTLs (VMTL and Henshin) can be interpreted in at least two ways. First, VMTL may benefit from its adoption of concrete syntax. The very reason why UML and other modeling languages adopt a concrete syntax on top of the abstract one is to increase usability. This is achieved by employing expressive visual notations and hiding unessential metamodel details. A second possible explanation for VMTL's better performance has to do with specification size. As shown in Figure 7.10, the VMTL specifications included in Experiment 2 are considerably more concise than their Henshin counterparts. Completion times observed for Henshin in this experiment are significantly higher, as are cognitive load ratings (although Henshin obtains better comprehension scores for high-performing participants). The hypothesis that diagram size has an important effect on comprehension is supported by previous findings on UML diagram understanding [179].

The high comprehension scores associated to Epsilon are perhaps the most surprising result of the two experiments. Considered together with the high task completion times, Epsilon's high comprehension scores point to a higher level of engagement of participants with this language. We offer two possible explanations for this. The first is participant background, given that participants are Computer Science students with strong programming skills — over 80% rated their own programming skills as good or very good. With its C-style syntax and imperative execution semantics, Epsilon may have appeared familiar to them. This hypothesis suggests that repeating our experiments with participants lacking programming skills may yield a different outcome.
7.3 Experiments on the Learnability of VMTL

Figure 7.10: Number of diagram elements included in the Henshin and VMTL specifications presented to participants

An alternative explanation for the high comprehension scores associated to Epsilon is offered by cognitive fit theory [204], which has primarily been applied in the field of information visualization [188, 203]. According to this theory, the accuracy of a problem solving process increases when the problem solving task matches the problem representation. In our case, the answer options of the comprehension questions are the problem solving task, and the transformation languages are the problem representation. Because the task is represented textually, a textual MTL such as Epsilon represents a better fit for solving it. The cognitive fit hypothesis is supported by a participant’s remark in a follow-up interview: “I couldn’t relate the text to the pictures”. Were this to be true, an experiment providing visual answer options may yield different outcomes.

The low completion times and cognitive load ratings observed for VMTL appear to suggest that its simple syntax has promoted an intuition-based approach to question answering. While fast and not very demanding, relying on intuition is not always accurate, as shown by VMTL’s slightly lower comprehension scores compared to those observed for Epsilon.

7.3.4 Threats to Validity

To evaluate the validity of our experiments, we consider the four categories of threats to the validity of SE experiments described by Wohlin et al. [213].

Construct validity. An experiment manifests construct validity if it measures
the actual phenomena under investigation – in our case, model transformation language learnability. Comprehension correctness and cognitive load are both reasonable measures for learnability. Although we use subjective cognitive load measures, it has been shown that a high correlation exists between subjective and objective cognitive load measures (cf. [71]). To avoid any bias in favor of VMTL, we have presented the MTLs to participants in an impartial manner, replacing their names with pseudonyms. Finally, the construct validity of our experiments could be improved by also presenting participants with a production task, similarly to the VMQL learnability experiment presented in Section 7.2. However, our experience indicates that including both comprehension and production tasks in the same experiment is exceedingly time consuming and leads to high subject mortality (i.e. participants quitting the experiment).

**Internal validity.** An experiment manifests internal validity if a causal conclusion regarding the phenomena under investigation can be drawn from it. The internal validity of our experiments is threatened by learning effects, a typical issue for within-subject designs. Experiment 1 is particularly vulnerable, as it repeats the same questions for every MTL. To counter this threat, we have randomly assigned participants to one of three treatments, each presenting the MTLs in a different order. Learning effects are a much smaller threat for Experiment 2, as it does not reuse questions. However, Experiment 2 is under risk of confounding the effect of the MTLs with that of particular UML diagram types. For this reason, we have also created three versions of the Experiment 2 questionnaire by permuting the questions asked under each MTL. Finally, we avoid selection bias (i.e. the self-selection of only those volunteers that are interested in the topic of the experiment) by offering a small participation prize.

**Conclusion validity.** An experiment manifests conclusion validity if the statistical relationship between its factors and outcomes is correctly evaluated. Threats to conclusion validity typically originate in incomplete or incorrect statistical analysis procedures. We avoid such threats by presenting both descriptive and inferential statistics, verifying the assumptions of the employed statistical tests, performing non-parametric hypothesis testing (the Wilcoxon signed rank test), and reporting effect sizes. In particular, the assumptions required by the ANOVA technique were verified by visually inspecting residual plots and Q-Q plots, as suggested by Montgomery et al. [125].

**External validity.** An experiment manifests external validity if its outcomes can be generalized to a wider population. We ensure external validity for the sampled population (undergraduate and graduate CS students) by using sufficiently large sample sizes. Additionally, since participants' modeling skills match the profile of end-user modelers introduced in Section 2.2, the conclusions of our experiments may, at least in what concerns VMTL and Henshin, be extended to this population. Due to most participants' competence in computer
programming, our results concerning Epsilon (which closely resembles a general-purpose programming language) most likely do not generalize to the population of end-user modelers, who typically lack programming skills.

Just like the experiment investigating the learnability of VMQL, the two experiments presented in this section can be evaluated against the quality checklists presented in Table 4.2. The only deficiencies of these experiments with respect to the mentioned quality criteria are related to participants’ backgrounds and the lack of a control group, leading to a quality score of 9 out of a maximum of 10 points. This places our study in the “high quality” bracket according to Figure 4.6 (top).

7.4 Think-Aloud Protocol Analysis of the Learnability of VMTL

7.4.1 Methods and Materials

While the experiments presented in Section 7.3 offer an objective measure of VMTL’s relative learnability compared to that of Epsilon and Henshin, they do not provide any insights into how users approach the task of comprehending a VMTL specification. To gain these insights, we have conducted a think-aloud protocol investigation following the methodology described in [113]. Our study follows the concurrent think-aloud protocol methodology, in which participants verbally describe their thought process as they complete a given task.

Four PhD students with diverse backgrounds took part in the study. They are identified in what follows as Participant 1 through 4. The research domain, as well as the self-reported UML and English language skill levels of each participant are listed in Table 7.8. Notably, they all rate their UML knowledge as poor or very poor. Participants were presented with the three VMTL transformation specifications shown in Figure 7.11. These transformations respectively operate on UML Class Diagrams (Transformation 1), UML Activity Diagrams (Transformation 2), and UML Use Case Diagrams (Transformation 3).

After sitting through an introduction to MT and VMTL, participants were asked to read the three VMTL specifications and explain their intended meaning. While doing so, they were allowed to consult a written specification of VMTL at any time. The experimenter took written notes of participants’ answers and other remarks as they completed the tasks, without providing assistance.
Transformation 1:

If two Classes have a common superclass, and they both have a public Attribute with the same name and type, delete this Attribute from both Classes and add it to the superclass if the superclass does not already own an Attribute with the same name and type.

Transformation 2:

If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, add a Decision Node between the Action and its outgoing Control Flows.

Transformation 3:

If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the extending Use Case.

Figure 7.11: VMTL transformation specifications included in the think-aloud protocol analysis. Transformation 1 is applicable to UML Class Diagram, Transformation 2 is applicable to UML Activity Diagrams, and Transformation 3 is applicable to UML Use Case Diagrams.
Table 7.8: Backgrounds of the think-aloud protocol study participants. UML and English language skills were self-assessed by participants.

<table>
<thead>
<tr>
<th>ID</th>
<th>Occupation</th>
<th>Domain</th>
<th>UML [1..5]</th>
<th>English [1..5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PhD student</td>
<td>Nutrition science</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>PhD student</td>
<td>Theoretical CS</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>PhD student</td>
<td>SE</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>PhD student</td>
<td>SE</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

7.4.2 Results and Interpretation

The think-aloud protocol analysis has yielded several interesting findings. First, it has confirmed that allowing VMTL transformation developers to use either Update Patterns or equivalent Find/Replace Pattern pairs is a positive design decision. One participant found it difficult to relate a Find Pattern with its corresponding Replace Pattern, expressing a preference for the more concise notation of Update Patterns. Meanwhile, two other participants referred to Find/Replace Pattern pairs as “before” and “after” states of the transformation, a concept which they apparently found intuitive. Participants did not place any emphasis on pattern icons, indicating that offering these icons only as optional visual aids is indeed appropriate.

Additional observations of interest emerged regarding VMTL annotations. Two participants expressed confusion as to which pattern elements an annotation refers to in the context of a Class Diagram. Participant 4 could not precisely identify if the delete annotation in Transformation 1 refers to an Attribute or to its containing Class, when the annotation in fact refers to the Class. The same participant expressed surprise that an annotation can be anchored to an Association (as exemplified in Transformation 3), and could not determine which end of the Association it refers to — in fact, the annotation refers to the Association itself. Finally, Participant 2 repeatedly referred to the annotations as “log messages”. These observations suggest that VMTL’s textual annotations may not be as intuitive as hoped, and that ambiguities introduced by the model editor regarding the anchor point of an annotation may play an important role.

The now deprecated create singleton clause included in the version of VMTL used in the study has also caused participants some difficulty. This clause is employed in Transformation 1. All participants needed to consult its written description several times, and Participant 2 described it as “taking an existing model element and turning it into a singleton”. The clause’s intended semantics
is in fact to create a new model element only if an identical one does not already exist in the source model. We suspect that the term “singleton” may have a different or unclear meaning to end-user modelers, and have consequently replaced the create singleton clause with the more explicit formulation of create if not exists.

Each participant’s individual background played a role in the way they approached the tasks. Participants with a CS background attempted to comprehend the specifications by considering similar notions from general-purpose programming languages. For example, Participant 4 correctly remarked that VMTL’s notation for variables resembles the notation adopted by the PHP scripting language. However, Participant 2’s intuition to equate VMTL annotations with log messages was not as helpful. As expected, Participant 4 encountered difficulties in understanding the meaning and purpose of the various UML notations, with Class Diagram elements posing the greatest challenge.

Ultimately, the limited scale of this study prevents us from drawing wide-ranging conclusions. Nevertheless, the study has provided some indications regarding the aspects of VMTL that can be improved, indications which are reflected in the current version of VMTL’s syntax presented in Section 5.5.3. It has also suggested possible contributing factors to the results obtained by VMTL in the learnability experiments presented in Section 7.3.

7.4.3 Threats to Validity

The most important threats to the validity of our study are related to its participants. First, the low number of participants does not allow us to attempt a generalization of the outcomes to any wider population. The diverse backgrounds of participants also hinder any attempts at generalization, while their overall poor knowledge of UML arguably makes participants unrepresentative for the general population of end-user modelers. Under these circumstances, the qualitative data gathered from this study may only be used to complement the interpretation of the experimental results presented in Section 7.3 and inform some relatively minor updates to VMTL.

Participant selection bias is also a possible threat to validity, as all participants volunteered to take part in the study. However, this threat was mitigated by offering a small participation prize.

The instrumentation used in the study also poses some threats to validity, as the three transformations used cover only a small subset of UML. However, three different UML diagram types are addressed, including both structural
and behavioral constructs. A similar observation can be made regarding the relatively small number of VMTL language constructs present in these transformations. While Find, Produce, and Update patterns are all included, pattern types representing application conditions are not included, and neither are most of VMTL’s textual annotation clauses. However, many of the observations presented in Section 7.4.2 are not limited to specific syntax elements, but are more generally applicable to entire categories of VMTL constructs, such as annotations or icons.

Finally, experimenter bias is a possible threat to the validity of any think-aloud protocol analysis, as the qualitative feedback provided by participants may mirror any apparent expectations on the experimenter’s side. Participants in our study were aware of the fact that VMTL was a language developed by the experimenter. They were, however, not informed about the precise purpose of the study or any expectations regarding its outcomes.

With respect to the quality criteria for empirical studies listed in Table 4.2, our study lacks a sufficiently large number of participants, and the background of these participants is not entirely aligned with the study’s aims. These shortcomings prevent the generalization of our results. The study therefore scores a total of 8 points out of a maximum of 9 points in this quality assessment, placing it in the “high quality” bracket according to Figure 4.6 (bottom).
Chapter 8

Conclusion

8.1 Summary of Results

This thesis has achieved three objectives:

- providing evidence for the claim that the requirements of end-user modelers are not fully met by existing model manipulation solutions;
- providing model manipulation languages and tools intended to meet the requirements of end-user modelers;
- demonstrating that the provided languages meet the requirements of end-user modelers to a greater extent than existing languages.

The first step in this direction has been to define the term “end-user modeler”, as this term is currently not consecrated in modeling literature. We have defined end-user modelers as “users of a modeling language familiar with its syntax and semantics, but unfamiliar with its metamodel, abstract syntax, and applicable model manipulation languages” (see Section 2.2.1). Their combination of domain expertise and lack of meta-modeling and programming expertise differentiates end-user modelers from both MDE practitioners and language engineers. Based on this observation, we infer that end-user modelers have distinct
requirements from model manipulation tools. These requirements were investigated in Section 2.2.2. Some of the most important such requirements are *usability*, *learnability*, and *maintainability*, all of which fall in the category of human factors.

After identifying human factors considerations as the main component of end-user modelers' requirements, we have shown that existing research in the area of model transformation (the most complex, extensively studied, and arguably most important model manipulation operation) largely fails to address these considerations (see Chapter 4). We have reached this conclusion by applying an empirical research tool: a Systematic Mapping Study (SMS). Several noteworthy results have emerged from the 188 publications included in this study. First, although interest in human factors has increased in recent years, there are still significantly fewer studies addressing usability-related topics than studies addressing maintainability-related topics. Furthermore, the technology landscape of the implementations described in included studies is strongly indicative of a *software monoculture*: 80% of the implementations are based on Java, and 62% are based on EMF. Finally, empirical evaluations in this area are few and generally of low quality. 74% of the identified studies do not include an empirical evaluation, while only 14% include an experiment, and 12% include a case study. None of the studies presents an interview, survey, or think-aloud protocol analysis, and the vast majority of evaluations that do exist fail to follow empirical SE guidelines. Overall, the findings of this SMS uncover a lack of usability-oriented and empirically validated model transformation languages.

Motivated by the findings of the SMS, we have put forward a series of proposals for giving end-user modelers access to powerful model manipulation tools. As a first step, we have made explicit a set of general principles intended to guide the development of model manipulation languages for end-user modelers (see Section 5.1). Captured under the umbrella-term of Transparent Model Manipulation (TMM), the three principles are termed *syntax transparency*, *environment transparency*, and *execution transparency*. Syntax transparency deals with how model manipulations are expressed. Namely, specifications expressed using a syntax transparent language also represent valid model fragments in the host modeling language. This benefits end-user modelers, as it ensures that the syntax and structure of the manipulation language is already familiar to them. Environment transparency implies that creating and editing model manipulation specifications must not require a dedicated editor. A conventional model editor should be sufficient for this task, meaning that end-user modelers are not burdened with learning how to use an additional tool. Finally, the execution transparency principle states that the execution engine responsible for carrying out a model manipulation should be interchangeable and independent from the manipulation language. This allows end-users to control how specifications are executed, and ensures that several technology spaces can be addressed.
8.1 Summary of Results

We have demonstrated the feasibility of the TMM principles by specifying and implementing the VM* family of model manipulation languages following them. The foundation of the VM* family is the VMQL model query language described in Section 5.3. VMQL adopts a by-example approach to querying, allowing the specification of model queries as textually annotated model fragments. It relies on naming conventions or, where available, lightweight extension mechanisms such as Stereotypes for structuring specifications. The simple textual annotation language defined by VMQL is required for effective pattern specifications, as most modeling languages do not define pattern definition constructs. VMQL is syntax transparent, as all VMQL queries are valid fragments in the host modeling language. It is also environment transparent, as any host modeling language editor – including, for instance, concrete syntax editors and containment tree editors – can be used to create VMQL specifications.

The second member of the VM* family is the VMCL model constraint language described in Section 5.4. It supports the query syntax of VMQL, and extends it to allow constraint specification. The extensions required by VMCL are minor, as queries and constraints are dual operations. Whereas a VMQL query identifies model fragments matching a given pattern, a VMCL constraint identifies fragments that, within a given context, fail to match the pattern. VMQL users can therefore express VMCL constraints with minimal learning effort.

Also as part of the VM* family, we have defined the VMTL model transformation language described in Section 5.5. VMTL is a particularly important contribution of this thesis, since existing model transformation languages are explicitly targeted at MDE practitioners as opposed to end-user modelers. It is an endogenous model-to-model transformation language that extends VMQL to allow the specification of model updates, while preserving its syntax and environment transparency. VMTL adopts an intuitive pattern replacement semantics, while also providing expressive constructs such as rule application conditions and priority-based rule scheduling. All of these constructs are implemented via textual annotations and Stereotypes (or, alternatively, naming conventions).

Our implementation of the VM* languages has also been guided by the TMM principles. As detailed in Section 6.1, we have taken advantage of the similarity between VMTL's operational semantics and that of graph transformation languages by adopting the Henshin graph-based MT engine for executing VM* specifications. Henshin was selected due to its maturity, expressiveness, and amenability to formal analysis. All three VM* languages are executed via this engine. The VM* Runtime is our implementation of a compiler between VM* and Henshin specifications. The deployment of the VM* Runtime, described in Section 6.2, is intended to support environment transparency by separating the specification and execution of model manipulations. A distributed deployment meets this requirement, motivating our proposal of a model manipulation
Web Service API referred to as the VM* API. This API enables the creation of lightweight model editor extensions invoking it, as well as that of clients separate from the model editor, such as Web and mobile applications.

Finally, in Chapter 7 we have presented what is arguably the first comprehensive human factors evaluation in the area of model manipulation languages. The evaluation is focused on learnability, one of the principal end-user modeler requirements, and employs both quantitative methods (user experiments) and qualitative methods (a think-aloud protocol analysis). The first such user experiment has revealed that VMQL surpasses the widely used OCL query and constraint language in terms of learnability. Both the task metrics and the cognitive load ratings collected from this experiment point in this direction.

We have also presented the results of two experiments comparing the learnability of VMTL with that of the Epsilon textual MTL and the Henshin abstract syntax visual MTL (see Section 7.3). Our learnability evaluation was based on two task metrics (comprehension score and task completion time) and two subjective metrics measuring the cognitive load imposed by each language on participants (perceived difficulty and effort). VMTL was associated with the shortest completion times and lowest cognitive load ratings, but also with comprehension scores slightly below Epsilon. We hypothesize that VMTL outperformed Henshin either due to its use of concrete syntax, or due to the known effect of diagram size on comprehension. We also hypothesize that the cognitive fit between Epsilon, a textual language, and the textual questions included in the experiment may have benefited this MTL.

The qualitative data collected from the think-aloud protocol analysis presented in Section 7.4 has facilitated our interpretation of the above experimental results, while also suggesting possible improvements to VMTL, primarily in what concerns its textual annotation syntax. These improvements have been included in the version of VMTL presented in this thesis.

We submit that, taken together, these results constitute significant progress towards achieving the vision for end-user modelers put forward in Section 1.2. End-user modelers now have access to a coherent and powerful set of task-specific model manipulation languages that demonstrably address their requirements.

8.2 Lessons Learned

Usability, expressiveness, and predictability. One of the main challenges in designing the VM* languages has been to strike a balance between their us-
8.2 Lessons Learned

ability, expressiveness, and predictability. First, there is an inherent conflict between learnability (as an aspect of usability) and expressiveness: the more features a language possesses in support of its expressiveness, the higher its learning curve for new users will be. This is an intuitive and well known observation, since learning many or more complex features requires more effort than learning only a few simple features. This situation becomes truly problematic when new advanced features need to be accommodated by increasing the complexity of existing basic features.

However, the trade-off between usability and predictability in model manipulation languages is less often discussed. This trade-off emerges from the attempt to shield end-users from the often significant intricacies of host metamodels. It can be best exemplified by considering the id special variable defined by VMTL for the purpose of unambiguously identifying corresponding elements in different patterns. In the absence of this mechanism, corresponding unnamed elements can only be identified approximately, i.e. several mappings between the elements of different patterns may be possible. Note that in graph-based transformation languages such as Henshin correspondences between nodes in different graphs are always made explicit. In VMTL, attaching textual annotations to unnamed elements for the purpose of specifying their id can become tedious and time consuming, thus negatively impacting usability. However, it is the only way to guarantee the predictability of the transformation’s outcomes.

Pitfalls of MT systematic reviews. The process of conducting the Systematic Mapping Study on human factors in MT has revealed some issues that we believe are characteristic to the wider research area of model transformation. They will likely also affect future Systematic Mapping Studies or Systematic Literature Reviews addressing topics in this field.

The first such issue is the very high number of existing publications. Despite a relatively restrictive search term, our initial automated search retrieved 3549 studies. Although roughly a third of these were duplicate results, the number of remaining unique studies is still surprisingly high. Even after applying exclusion criteria, the 325 remaining studies are indicative of the large size of the MT literature corpus. We thus strongly advise carrying out a pilot search before embarking on an MT-related systematic review, as the large number of relevant studies may dictate addressing a narrower topic or converting a Systematic Literature Review into a less detailed Systematic Mapping Study. We have adopted the first solution after an unsuccessful earlier attempt to carry out a Systematic Mapping Study of the entire field of Model Transformation has yielded over 17000 initial search hits.

We have also learned that model transformation literature is, unfortunately, not particularly suitable for systematic reviews. A significant proportion of the
publications we have seen suffer from poorly chosen non-descriptive titles, while structured abstracts are a rare occurrence. Perhaps even more importantly, the misuse of consecrated terminology makes it difficult to classify studies. For instance, if a study claims to include an experiment when in reality it appears to present a case study or a simple example, what should the study be classified as? We have adopted the solution of taking the claims made in studies at face value and classifying them accordingly, as doing otherwise may have involved subjective judgment on our part. Still, this aspect might call into question the accuracy of a systematic review’s outcomes.

**Experimental evaluation.** Conducting the user experiments evaluating the learnability of the VM* languages has taught us that, mostly due to larger specification sizes, transformation languages raise additional problems compared to query and constraint languages. Answering individual questions regarding MT specifications is more time consuming for participants. Therefore, to avoid participant fatigue and high mortality rates, it might be necessary to decrease the number of experimental questions. This observation is very likely exacerbated for production tasks, i.e. tasks in which participants are asked to produce transformation specifications themselves. These problems are in many ways similar to those encountered by researchers evaluating general-purpose programming languages. A solution adopted in this area is to limit experiments to language features that can be evaluated via short questions, such as the intuitiveness of individual keywords [170]. However, omitting to consider larger examples can only lead to an incomplete picture of a language’s usability.

### 8.3 Future Work

The results presented in this thesis suggest three main directions for future work: extensions to the VM* languages, improvements in tool support, and further empirical evaluations. In terms of language extensions, the following proposals are potentially of interest:

- **Behavioral querying.** VMQL currently only supports structural queries, as it does not take into account the semantics of the source model. This is in one sense an advantage, as it maximizes the range of supported host modeling languages. However, in certain contexts, such as process model querying, support for behavioral queries is desirable. Behavioral queries can take into account process execution semantics, and answer questions such as “Does Activity A always precede Activity B” or “If Activity A is executed, will Activity B also eventually be executed?”. Answering such
queries by simply looking at the graph-like structure of a process is not possible. Instead, a formalism capable of simulating the process is required. Temporal logics such as linear temporal logic (LTL [149]) and computation tree logic (CTL [59]) are promising candidates for this role. CTL, for example, has already been used for this purpose [18].

- **Exogenous transformations.** VMTL has been described in this thesis as an endogenous model-to-model transformation language, since its proposed usage scenarios do not require exogenous transformations, and such transformations are in conflict with the principle of syntax transparency. However, extending it to enable such transformations can very likely be accomplished in practice with little effort. The VM* Runtime's underlying Henshin execution engine already supports exogenous transformations, while VMTL's syntax and environment transparency imply that no editor extensions are required. Model-to-text transformations, on the other hand, may be more problematic, as their semantics is not compatible with VMTL's graph transformation-like operational semantics.

In addition to extending the capabilities of the VM* languages, the following improvements to the tools supporting these languages are desirable:

- **Full support for current specifications.** Current tool support for the VM* languages is not on par with the language specifications. The VM* Runtime is only capable of executing base specifications, i.e. model manipulation specifications that do not include textual annotations. In addition, at the time of writing only a subset of the VM* API is implemented. However, Chapter 6 provides an architectural guide that should facilitate the completion of the VM* Runtime and VM* API implementations.

- **Alternative execution engines.** VM* specifications can currently only be executed using the Henshin transformation engine. Extending the VM* Runtime to accommodate additional execution engines will lend additional credibility to the VM* languages' execution transparency claims.

- **Model editor plugins.** Although VM* specifications can be created using conventional model editors without any extensions, the ability to invoke the VM* Back-End from within the editor can further improve end-user modelers' productivity. The VM* API facilitates this type of lightweight editor extension, which would be feasible for tools such as MagicDraw [118], Enterprise Architect [60], and Microsoft Visio [205].

Finally, the empirical results presented in this thesis can be complemented in a number of directions:
• **Experimental evaluation of VMCL.** Although VMQL and VMTL have been experimentally evaluated from a learnability perspective, no such evaluation has been conducted for the current version of VMCL. A user experiment comparing VMCL's learnability with that of one or more existing model constraint languages (e.g. OCL) would therefore be interesting. However, the fact that VMCL only brings minor additions to VMQL suggests that it should be just as easy to learn.

• **Evaluating other human factors.** The evaluations presented in this thesis all focus on learnability, and in particular on initial learnability. This is partly due to the fact that initial learnability can be evaluated without requiring extensive participant preparations or long-term commitment. However, aspects such as extended learnability, understandability, and user satisfaction should also be part of a complete usability evaluation, making additional studies addressing these topics a necessity.

• **Evaluating execution performance.** The execution performance of the VM* implementation should be evaluated, since poor performance can drastically affect its appeal to end-user modelers. Although it has produced satisfactory results, the existing evaluation of VMQL's performance [3] is based on a now deprecated Prolog-based implementation.

• **Increased participant diversity.** The majority of participants included in the evaluations presented in this thesis are students. Including more experienced modelers with diverse modeling backgrounds as participants will allow us to support the external validity of our results relative to the population of end-user modelers with much higher confidence.

• **Comparing against other model manipulation languages.** Despite the fact that the model manipulation languages included in our experiments are either standardized or widely known, extending our investigation to additional languages may help create a more accurate picture of learnability in model manipulation technologies.

• **Objective cognitive load measurements.** Our experiments rely on participants' subjective assessment of the cognitive load imposed on them by the various model manipulation languages. Although it has been shown that subjective and objective cognitive load measures are correlated [71], including objective cognitive load measurement techniques such as eye tracking and skin conductance tests in future experiments would arguably increase the credibility of our results.

• **Visual questions** In Section 7.3.3 we hypothesize that questions formulated using a textual language (i.e. plain English) lead to a better cognitive fit with textual model manipulation languages. To investigate
this hypothesis, it is necessary to conduct similar experiments in which the questions are formulated visually.

- **Qualitative evidence.** With the exception of a think-aloud protocol analysis, the empirical evidence presented in this thesis is qualitative in nature. An increased emphasis on qualitative studies may lead to additional insights into the needs of end-user modelers, thus informing future developments of the VM* languages. User interviews are a promising research method in this context.
LIST OF PUBLICATIONS INCLUDED IN THE SMS

The following publications were included in the Systematic Mapping Study of human factors in model transformation presented in Chapter 4.


[SMS51] Camillo Fiorentini, Alberto Momigliano, Mario Ornaghi, and Iman


[SMS61] Carlos A. González and Jordi Cabot. ATLTest: A White-Box Test


[SMS79] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, and Abel Heged. A survey and comparison of transformation tools based on the


[SMS126] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona


Appendix B

Experiment Questionnaires

B.1 VMQL Learnability User Experiment

The paper questionnaire used in the VMQL learnability user experiment discussed in Section 7.2 is reproduced in what follows. In this experiment participants were randomly assigned to one of two treatments. The treatments are identical with the exception of the order in which questions concerning the two model query languages are presented. Treatment 1 presents and asks questions about Language X (VMQL) first, while Treatment 2 features Language Y (OCL) first. Only the questionnaire corresponding to Treatment 1 is shown.

The correct answers for Task A of the questionnaire are listed in Table B.1.

Table B.1: Correct answers for Task A of the VMQL learnability questionnaire

<table>
<thead>
<tr>
<th>Question</th>
<th>Correct Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
</tr>
</tbody>
</table>
Instructions

This questionnaire is part of a scientific experiment conducted in April 2013 by Vlad Acretoaie and Prof. Dr. Harald Störrle and, Department of Applied Mathematics and Computer Science, Technical University of Denmark, Matematiktorvet, 2800 Kongens Lyngby, Denmark.

- The goal of this experiment is to compare the relative usability of two different model query specification languages called X and Y in this experiment.
- The data elicited in this experiment are used only for research purposes. All data are stored and published only anonymized. Participation is entirely voluntary.
- Please read these instructions completely and thoroughly before continuing. Then go through the questions one after the other. Never go back to a task after you have moved forward to the next task.
- You are asked to take the exact time when working on the questionnaire, so please make sure you have a watch available before starting.

How to use this sheet

- This sheet contains instructions on the front side, and explanations on the query languages on the back side.
- Read the explanations of the query languages first, and then put them beside the questionnaire for easier reference.

Task A – Comprehension

- You are presented with a set of tasks that contain one query in one of the languages X and Y. Each of the queries comes with three different textual descriptions. Please mark the correct description.
- This task is repeated for different query specification languages. For every language (i.e., every sheet), you are asked to take the time.
- Do not go back after you have completed one page.

Task B – Production

- In this task you are asked to express queries matching the given descriptions.
- Do not go back after you have completed one page.

Task C – Assessment

- In this task you are asked to assess the languages used to express the queries. Just give your personal impression – there is no wrong or right.

Please tick/strikethrough as appropriate

Yes, I would like to be informed about the results of this experiment.

Contact me at _____________@ _________________________________
# Introduction to Business Process Model and Notation (BPMN)

BPMN is a modeling language commonly used for business process modeling. It is in some ways similar to UML Activity Diagrams. The basic elements of a BPMN diagram are presented in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartEvent</td>
<td>Indicates where a Process will start.</td>
<td></td>
</tr>
<tr>
<td>EndEvent</td>
<td>Indicates where a Process will end.</td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td>A task is an Activity that cannot be broken down to a higher level of detail.</td>
<td></td>
</tr>
<tr>
<td>ChoreographyTask</td>
<td>A ChoreographyTask is an Activity representing a set of Message exchanges involving two Participants.</td>
<td></td>
</tr>
<tr>
<td>ExclusiveGateway</td>
<td>Exclusive decision and merging. Only one of the SequenceFlows between two ExclusiveGateways will be executed.</td>
<td></td>
</tr>
<tr>
<td>ParallelGateway</td>
<td>Forking and joining. All SequenceFlows between two ParallelGateways will be executed in parallel.</td>
<td></td>
</tr>
<tr>
<td>SequenceFlow</td>
<td>Shows the order in which Activities are performed in a Process.</td>
<td></td>
</tr>
<tr>
<td>DataOutput</td>
<td>Information about what an Activity produces.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Depicts the contents of a communication between two Participants.</td>
<td></td>
</tr>
<tr>
<td>Association</td>
<td>Links additional information to BPMN graphical elements.</td>
<td></td>
</tr>
<tr>
<td>TextAnnotation</td>
<td>Provides additional information to the reader.</td>
<td></td>
</tr>
<tr>
<td>Pool</td>
<td>The graphical representation of a Participant in a Collaboration</td>
<td></td>
</tr>
</tbody>
</table>
Introduction to the Query Languages

Language X
A query in Language X is expressed as a diagram with annotations in comment boxes. Each comment box is attached to one or more model elements by dashed lines. There may be more than one annotation in any comment box.

- **context** – anchors the expression to an element.
- **indirect** – a relationship may be a path of arbitrary length (including 0 and 1).
- **distinct** – two elements in the query match only with distinct elements in the queried model. Without this constraint, different elements in the query may match with the same model element.
- **optional** – the parts indicated may or may not be present.
- **mclass <: C** – the meta class (type) of the constrained element can be any subclass of C.
- **mclass = *** – the constrained element can have any meta class (type).
- **mattr M = V** – the meta attribute M of the constrained element must have the value V.
- **mattr M = *** – the meta attribute M of the constrained element can have any value.
- **name = N** – the name meta attribute of the constrained element must have the value N.
- **$variable** – all terms starting with the $-sign are variables.

Language Y
A query in Language Y is expressed as a textual expression. Executing a query means finding model fragments in the base model that satisfy the expression.

- **def:** – introduce a new query operation.
- **foo.bar** – select the single element stored in property bar of element foo.
- **foo->bar** – select the set of elements stored in property bar of element foo.
- **context foo: predicate** – for all elements of type foo, evaluate predicate.
- **bar->size()** – count the cardinality of bar.
- **self** – the element.
- **bar->collect(c: predicate)** – return the collection that results from evaluating c for each element of bar.
- **bar->select(c: predicate)** – return the collection containing all elements of bar for which c evaluates to true.
- **bar->append(e)** – append element e to collection bar.
- **bar->exists(c: predicate)** – check if bar has at least one element for which c is true.
- **Set(T)** – a set of elements of type T.
- **Bag(T)** – a bag of elements of type T (can contain duplicate elements).
### Task A – Comprehension (Language X)

The following table contains queries and several candidate descriptions for them. Please mark those descriptions that are correct. If you can’t solve the task, mark “I don’t know”. The first row provides an example. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>No.</th>
<th>Query Expression</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1" alt="Diagram 1" /></td>
<td>Which Tasks are executed immediately after the StartEvent?</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>How many Pools does the process contain?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Tasks are executed before an ExclusiveGateway?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I don’t know.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><img src="image2" alt="Diagram 2" /></td>
<td>What precedes Task A?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>How many Tasks follow Task A?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>What follows Task A?</td>
<td>I don’t know.</td>
</tr>
<tr>
<td>3</td>
<td><img src="image3" alt="Diagram 3" /></td>
<td>Which Activities are executed more than once?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Task precedes the first ExclusiveGateway?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Activities are executed immediately before the EndEvent?</td>
<td>I don’t know.</td>
</tr>
<tr>
<td>4</td>
<td><img src="image4" alt="Diagram 4" /></td>
<td>What are the names of the Pools included in the Process?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Pool contains the start event?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Tasks are executed more than once?</td>
<td>I don’t know.</td>
</tr>
<tr>
<td>5</td>
<td><img src="image5" alt="Diagram 5" /></td>
<td>Which Tasks throw Error events?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which are the CompensationTasks?</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Which Tasks are not CompensationTasks?</td>
<td>I don’t know.</td>
</tr>
</tbody>
</table>

Version 1
5. Which Task is part of at least one Choreography?
Which Tasks are executed in parallel with other Tasks?
Which Task follows the StartEvent?
I don’t know.

6. Which ExclusiveGateways are followed by two different Tasks?
Which ExclusiveGateways are followed by the same Task?
Which Task is executed exactly once?
I don’t know.

7. Which are the DataOutputs of Task A?
Which Pool does Task A belong to?
Which is the first Gateway following Task A?
I don’t know.

8. What are the messages associated to Task A?
What are the participants and optional messages associated to Task A?
What Task follows Task A?
I don’t know.

Please note your precise end time here: ___:___:___
After you have finished this page, do not come back to change it or even just look at it!

Version 1
### Task A – Comprehension (Language Y)

The following table contains queries and several candidate descriptions for them. Please mark those descriptions that are correct. If you can’t solve the task, mark “I don’t know”. The first row provides an example. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>No.</th>
<th>Query Expression</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>context Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>def: query() : Set(Task) =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>self.flowElements-&gt;select(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>e.oclIsTypeOf(Task) and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e.incoming-&gt;collect(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sourceRef.oclIsTypeOf(StartEvent)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)-&gt;size &gt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Tasks are executed immediately after the StartEvent?</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>How many Pools does the process contain?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Tasks are executed before an ExclusiveGateway?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>context Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>def: query() : Bag(FlowElement) =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>self.flowElements-&gt;select(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>e.oclIsTypeOf(Task) and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e.name=&quot;A&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)-&gt;collect(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>outgoing-&gt;collect(targetRef)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>What precedes Task A?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>How many Tasks follow Task A?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>What follows Task A?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>context Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>def: query() : Set(Activity) =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>self.flowElements-&gt;select(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>e.oclIsKindOf(Activity) and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>e.outgoing-&gt;collect(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>targetRef.oclIsTypeOf(EndEvent)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)-&gt;size &gt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Activities are executed more than once?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Task precedes the first ExclusiveGateway?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Activities are executed immediately before the EndEvent?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>context Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>def: query() : Set(Pool) =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>self.flowElements-&gt;select(</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>e.oclIsKindOf(Pool)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>What are the names of the Pools included in the Process?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Pool contains the start event?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Tasks are executed more than once?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Tasks throw Error events?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which are the CompensationTasks?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Which Tasks are not CompensationTasks?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page</td>
<td>Context</td>
<td>Def</td>
<td>Question</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>5</td>
<td>Process</td>
<td>`def: query() : Set(Task) = self.flowElements-&gt;select( e</td>
<td>e.oclIsTypeOf(Task) and e.allPredecessors-&gt;exists( oclIsTypeOf(ParallelGateway) ) and e.allSuccessors-&gt;exists( oclIsTypeOf(ParallelGateway) ) )`</td>
</tr>
<tr>
<td>6</td>
<td>Process</td>
<td>`def: query : Bag(ExclusiveGateway) = self.flowElements-&gt;select( e</td>
<td>e.oclIsTypeOf(ExclusiveGateway) and e.outgoing-&gt;collect(targetRef) -&gt;asSet() -&gt;size() &gt; 1 )`</td>
</tr>
<tr>
<td>7</td>
<td>Process</td>
<td>`def: query() : Bag(DataOutput) = self.flowElements-&gt;select( e</td>
<td>e.oclIsTypeOf(Task) and e.name=&quot;Task A&quot;)-&gt;collect( dataOutputs{ } )`</td>
</tr>
<tr>
<td>8</td>
<td>Process</td>
<td>`def: participants() : Bag(Participant) = self.flowElements-&gt;select( e</td>
<td>e.oclIsTypeOf(ChoreographyTask) and e.name=&quot;A&quot;)-&gt;collect(participantRefs)`</td>
</tr>
</tbody>
</table>

Please note your precise end time here: ___:___:___

After you have finished this page, do not come back to change it or even just look at it!
Task B – Production (Language X)

Using Language X, express the queries stated at the top of each box. Do not consult your earlier answers to solve this task! Do as best as you can, and don’t worry that it might be wrong. However, if you absolutely think you can’t solve one of the tasks, just cross it out. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>Which Tasks are not connected to the End Event?</th>
<th>Which Parallel Gateways have branches that start with the same task?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What precedes Task A?</th>
<th>What are the Choreography Tasks on which Participant A and Participant B collaborate?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please note your precise end time here: ___:___:___

Version 1
# Task B – Production (Language Y)

Using Language Y, express the queries stated at the top of each box. Do not consult your earlier answers to solve this task! Do as best as you can, and don’t worry that it might be wrong. However, if you absolutely think you can’t solve one of the tasks, just cross it out. Please note your precise start time here: __:__:__

<table>
<thead>
<tr>
<th>Which Tasks are not connected to the End Event?</th>
<th>Which Parallel Gateways have branches that start with the same task?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What precedes Task A?</th>
<th>What are the Choreography Tasks on which Participant A and Participant B collaborate?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please note your precise end time here: __:__:__
Task C – Assessment

Please sort the languages from the previous tasks on the relative effort it took you to work with them, your confidence that your results are correct, and the fun that you had using them.

<table>
<thead>
<tr>
<th>Task A</th>
<th>Language X</th>
<th>Language Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task B</th>
<th>Language X</th>
<th>Language Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How good is your knowledge of…
  - OCL?
  - BPMN?
  - first order predicate calculus?
  - English?

If you know UML – how did you learn it? Please tick one or more!

- University courses if yes: how many __
- Books if yes: how many __
- Other courses if yes: how many __
- Other if yes: how ________

Mark appropriate

Sex: male female

Age: up to 30 years 31-60 years 61 years or older

Occupation: student (BSc/BEng) student (MSc.) practitioner scientist/Phd

Thank you for your participation!
Please hand back this questionnaire together with any notes that you may have made.
B.2 VMTL Learnability User Experiment 1

The paper questionnaire used in the first VMTL learnability user experiment (Experiment 1) discussed in Section 7.3 is reproduced in what follows. In this experiment participants were randomly assigned to one of three treatments. The treatments are identical with the exception of the order in which questions concerning the three model transformation languages are presented. The languages were permuted between the three experiments. Only the questionnaire corresponding to Treatment 1 is shown. In the questionnaire, the MTLs are referred to as X (VMTL), Y (Henshin), and Z (Epsilon).

The correct answers for Task A of the questionnaire are listed in Table B.2.

Table B.2: Correct answers for Task A of the questionnaire used in Experiment 1 evaluating the learnability of VMTL

<table>
<thead>
<tr>
<th>Question</th>
<th>Correct answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
</tbody>
</table>
Instructions

This questionnaire is part of a scientific experiment conducted in May 2014 by Vlad Acretoaie and Prof. Dr. Harald Störrle, Dept. of Applied Mathematics and Computer Science, Technical University of Denmark, Matematiktorvet, 2800 Kongens Lyngby, Denmark.

- The goal of this experiment is to compare the relative usability of three different model transformation languages called X, Y, and Z in this experiment.
- The data elicited in this experiment are used only for research purposes. All data are stored and published only anonymized. Participation is entirely voluntary.
- Please read these instructions completely before continuing. Then go through the questions one after the other. Never go back to a task after you have moved forward to the next task.
- You are asked to take the exact time when working on the questionnaire, so please make sure you have a watch available before starting.

How to use the questionnaire

- This sheet contains instructions on the front side, and explanations on the transformation languages on the back side. The explanations are continued on the following page.
- Read the explanations of the transformation languages first, and then put them beside the questionnaire for easier reference.
- The questionnaire contains two tasks. Each task is repeated for transformation languages X, Y, Z.
- You are asked to write down the current time after completing each task for one of the transformation languages.
- Do not go back after you have completed one task for one of the transformation languages.

Task A – Comprehension

- You are presented with three questions, each containing one transformation specification. Each transformation comes with three textual descriptions. Mark the correct description.

Task B – Assessment

- You are asked to assess the transformation languages. Just give your personal impression – there is no right or wrong.

Please tick/strikethrough as appropriate

☐ Yes, I would like to be informed about the results of this experiment.

Contact me at _____________________@__________________________
Introduction to the Transformation Languages

Language X
A transformation in Language X consists of one or more rules, each expressed as a pair of UML diagrams. A diagram pair consists of a Left-Hand Side (LHS) pattern and a Right-Hand Side (RHS) pattern. A LHS pattern describes possible matches of the rule against a source model. A RHS pattern describes the changes applied to the source model to obtain the target model. Patterns contain annotations in the form of comment boxes. Each comment box is attached to one or more model elements by dashed lines. There may be several statements separated by commas in any comment box.

- **<<VMTL>>** – indicates that a comment box contains transformation statements;
- **<<VMTL_CHECK>>** – indicates that a comment box contains query and constraint statements;
- **<<VMTL_DO>>** – indicates that a comment box contains model manipulation statements;
- **$variable** – terms starting with the $ character are variables names;
- **LHS** – indicates the qualified name of the corresponding LHS pattern of a transformation rule;
- **RHS** – indicates the qualified name of the corresponding RHS pattern of a transformation rule;
- **MA = V** – states that the meta attribute MA of a model element has the value V;
- **MA = * V** – states that the meta attribute MA of a model element may have any value;
- **create** – creates a new model element and adds it to the model;
- **delete** – deletes a model element from the model;
- **exists** – verifies that a model element exists in the model;
- **not** – the negation operator; can be added as a prefix to other statements to negate their effect;
- **sequence = N** – specifies that the containing annotation will be applied after all annotations with sequence number N-1 have been applied;

Language Y
A transformation in language Y consists of one or more rules, each expressed as a class diagram. Rules are formulated at abstract syntax level, i.e. the nodes and edges they contain are elements of the UML meta model. Every model element included in a rule has one of the following stereotypes:

- **<<create>>** – the model element must not exist in the source model and will be created in the target model;
- **<<delete>>** – the model element must exist in the source model and will be deleted in the target model;
- **<<forbid>>** – the model element must not exist in the source model and will not exist in the target model;
- **<<preserve>>** – the model element must exist in the source model and will be preserved in the target model;

If a transformation consists of more than one rule, the rule execution order is specified by a SequentialUnit. SequentialUnits follow the syntax and semantics of simplified UML Activity Diagrams with Actions representing transformation rules. If rule A precedes rule B in a SequentialUnit, rule A is executed before rule B.
Language Z
A transformation in language Z consists of one or more rules respecting the following BNF syntax:

```
rule <name>
    transform <sourceParameterName>:<sourceParameterType>
    to <rightParameterName>:<rightParameterType>
    (, <rightParameterName>:<rightParameterType>)* {
        (guard (:expression)|{statementBlock})
        statement+
    }
```

A statement in language Z can be one of the following:
- `var <name> : <type> = <value>` - declares a variable;
- `<var1> = <var2>` - assigns the value of `var2` to `var1`;
- `<var1> ::= <var2>` - assigns the value of `var2` to `var1` after applying any transformation rules that are applicable to `var2`;
- `if <cond> {<statements1>} else {<statements2>}` - conditional statement;
- `for (<var> : <type> in <collection>) {<statements>}` - `for` statement;

Language Z defines the `String` and `Boolean` primitive data types, as well as the `Bag` (for non-unique elements) and `Set` (for unique elements) collection data types.

The following operations are defined on the `String` data type:
- `endsWith(<str>: String) : Boolean` - checks if a string has `<str>` as a suffix;
- `isSubstringOf(<str>: String) : Boolean` - checks if a string is a substring of `<str>`;

The following operations are defined on collection data types:
- `add(<item> : Any)` - adds an item to a collection;
- `includes(<item> : Any) : Boolean` - checks if an item belongs to a collection;
- `isEmpty() : Boolean` - checks if a collection is empty;
- `selectOne(<iterator> : <type> | <cond>) : Any` - returns the first element of a collection that has type `<type>` and matches `<cond>`;
- `collect(<iterator> : <type> | <expr>) : Collection` - returns the collection obtained by evaluating `<expr>` on every element of type `<type>` of a collection;
- `exists(<iterator> : <type> | <cond>) : Boolean` - checks if a collection contains at least one element of type `<type>` that satisfies `<cond>`;

The following operations are defined on model elements:
- `getAllOfType()` : `Set` - returns all model elements of the same type as the element the operation is called on;

Language Z also supports user-defined operations.
### Task A: Comprehension (Language X)

The following table contains transformation specifications and several candidate textual descriptions for them. Please mark those descriptions that are correct. If you can’t solve the task, mark “I don’t know”. The first row provides an example. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a) Create a public attribute named “overdue” in class “Loan”.</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) Change the visibility of the “overdue” attribute of class “Loan” to “public”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) Delete the attribute named “overdue” from class “Loan”.</td>
<td>I don’t know.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a) Create a public attribute named “id” in all classes whose name starts with the “Record” prefix.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) Create a class named “Record” having a public attribute named “identifier”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) Re-name the public attribute named “id” to “identifier” in all classes whose name ends in the suffix “Record”.</td>
<td>I don’t know.</td>
<td></td>
</tr>
<tr>
<td>No.</td>
<td>Transformation Specification</td>
<td>Description</td>
<td>Correct</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>2</td>
<td>package</td>
<td>&lt;VMTL&gt;</td>
<td>Comprehension Question 2</td>
</tr>
</tbody>
</table>
|     |     | LHS = RHS: Comprehension Question 2 | Log In | a) Create a use case named “Log in and remember” and connect it to the newly created “Remember user” extension point of the “Log in” use case. 
|     |     | Log in and remember |  |

| 3   | activity | Comprehension Question 3 | Comprehension Question 3 |
|     | Log In not exists | |
|     | <datstore> | Credentials | |
|     | <VMTL CHECK> | not exists | |
|     | <VMTL> | create | |

Please note your precise end time here: ___:___:___

After you have finished this task, do not come back to change it or even just look at it!
Task A: Comprehension (Language Y)

The following table contains transformation specifications and several candidate textual descriptions for them. Please mark those descriptions that are correct. If you can’t solve the task, mark “I don’t know”. The first row provides an example. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="example1.png" alt="Rule Example" /></td>
<td>a) Create a public attribute named “overdue” in class “Loan”.</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td><img src="example1.png" alt="Rule Example" /></td>
<td>b) Change the visibility of the “overdue” attribute of class “Loan” to “public”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="example1.png" alt="Rule Example" /></td>
<td>c) Delete the attribute named “overdue” from class “Loan”.</td>
<td>I don’t know.</td>
</tr>
<tr>
<td>1</td>
<td><img src="example2.png" alt="Rule Comprehension Question 1" /></td>
<td>a) Create a public attribute named “id” in all classes whose name starts with the “Record” prefix.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="example2.png" alt="Rule Comprehension Question 1" /></td>
<td>b) Create a class named “Record” having a public attribute named “identifier”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="example2.png" alt="Rule Comprehension Question 1" /></td>
<td>c) Re-name the public attribute named “id” to “identifier” in all classes whose name ends in the suffix “Record”.</td>
<td>I don’t know.</td>
</tr>
</tbody>
</table>

```java
className.indexOf("Record", className.length - "Record".length) !== -1;
```
<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><img src="image" alt="Diagram" /></td>
<td>a) Create a use case named “Log in and remember” and connect it to the newly created “Remember user” extension point of the “Log in” use case.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) Create a use case named “Log in” which includes the use case named “Log in and remember”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) Re-name the extension point “Remember user” of the “Log in” use case to “Log in and remember”.</td>
<td>I don’t know.</td>
</tr>
<tr>
<td>3</td>
<td><img src="image" alt="Diagram" /></td>
<td>a) If it does not already exist, create a data store node named “Credentials” and connect it to the “Log in” action via an object flow.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) Create a new object flow connecting the existing “Credentials” data store node and “Log in” action.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) Remove the object flow connecting the “Credentials” data store node and the “Log in” action.</td>
<td>I don’t know.</td>
</tr>
</tbody>
</table>

Please note your precise end time here: ___:___:___

After you have finished this task, do not come back to change it or even just look at it!
Task A: Comprehension (Language Z)

The following table contains transformation specifications and several candidate textual descriptions for them. Please mark those descriptions that are correct. If you can’t solve the task, mark “I don’t know”. The first row provides an example. Please note your precise start time here: ___:___:___

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
</table>
| Example 1 | `rule` Example_Classes
`transform` source : Source!Class
to target : Target!Class {
  guard : source.name == "Loan"
  var overdue : Source!Property = source.ownedAttribute.selectOne( a | a.name == "overdue");
  overdue.equivalent().visibility = Target!VisibilityKind#public;
}
`rule` Example_Properties
`transform` source : Source!Property
to target : Target!Property {
  target.name = source.name;
  target.class ::= source.class;
} | a) Create a public attribute named “overdue” in class “Loan”. X|
| | b) Change the visibility of the “overdue” attribute of class “Loan” to “public”. | |
| | c) Delete the attribute named “overdue” from class “Loan”. | I don’t know. |
| 1 | `rule` ComprehensionQuestion1
`transform` source : Source!Class
to target : Target!Class {
  guard : source.name.matches("(\w*Record)")
  target.name = source.name;
  for (pl : Source!Property in source.ownedAttribute) {
    var p2 : Target!Property = new Target!Property;
    if (p1.name == "id") {
      p2.name = "identifier";
    } else {
      p2.name = p1.name;
    }
    target.ownedAttribute.add(p2);
  }
} | a) Create a public attribute named “id” in all classes whose name starts with the “Record” prefix. |
| | b) Create a class named “Record” having a public attribute named “identifier”. | |
| | c) Re-name the public attribute named “id” to “identifier” in all classes whose name ends in the suffix “Record”. | |

I don’t know.
<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>rule ComprehensionQuestion2</td>
<td>a) Create a use case named “Log in and remember” and connect it to the newly created “Remember user” extension point of the “Log in” use case.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform source : Source!UseCase to target : Target!UseCase {</td>
<td>b) Create a use case named “Log in” which includes the use case named “Log in and remember”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>guard : source.name = &quot;Log in&quot;</td>
<td>c) Re-name the extension point “Remember user” of the “Log in” use case to “Log in and remember”.</td>
<td>I don’t know.</td>
</tr>
<tr>
<td></td>
<td>target.name = source.name;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var extensionPoint : Target!ExtensionPoint = new Target!ExtensionPoint; extensionPoint.name = &quot;Remember user&quot;; target.extensionPoint.add(extensionPoint);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var extend : Target!Extend = new Target!Extend; extend.extendedCase = target; extend.extensionLocation.add(extensionPoint);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var useCase : Target!UseCase = new Target!UseCase; useCase.name = &quot;Log in and remember user&quot;; useCase.extend.add(extend);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>rule ComprehensionQuestion3</td>
<td>a) If it does not already exist, create a data store node named “Credentials” and connect it to the “Log in” action via an object flow.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transform orig : Orig!OpaqueAction to transf : Transf!OpaqueAction {</td>
<td>b) Create a new object flow connecting the existing “Credentials” data store node and “Log in” action.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>guard : orig.name = &quot;Log in&quot;</td>
<td>c) Remove the object flow connecting the “Credentials” data store node and the “Log in” action.</td>
<td>I don’t know.</td>
</tr>
<tr>
<td></td>
<td>transf.name = orig.name;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>var exists : Boolean = false;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>for (edge : Orig!ActivityEdge in orig.incoming) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (edge.isTypeOf(Orig!ObjectFlow)) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (edge.source.isTypeOf(Orig!ObjectNode) and (edge.source.name = &quot;Credentials&quot;)) {</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>exists = true;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (not exists) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>var credNode = new Transf!DataStoreNode;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>credNode.name = &quot;Credentials&quot;;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>var credFlow = new Transf!ObjectFlow;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>credFlow.source = credNode;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>credFlow.target = transf;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>transf.incoming.add(credFlow);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>credNode.outgoing.add(credFlow);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Please note your precise end time here: ___:___:___

After you have finished this task, do not come back to change it or even just look at it!

Experiment 1 – Treatment 1
Task B: Assessment

Please sort the transformation languages by the relative difficulty and effort it took you to work with them, your confidence that your results are correct, and the fun that you had using them.

<table>
<thead>
<tr>
<th>Language X</th>
<th>Language Y</th>
<th>Language Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty</td>
<td><img src="image1" alt="Difficulty Scale" /></td>
<td><img src="image2" alt="Difficulty Scale" /></td>
</tr>
<tr>
<td>Effort</td>
<td><img src="image4" alt="Effort Scale" /></td>
<td><img src="image5" alt="Effort Scale" /></td>
</tr>
</tbody>
</table>

Participant Background

How good is your knowledge of...?  
don’t know it  a little  medium  good  extensive
...UML?  
...OCL?  
...English?

If you know UML, how did you learn it? Please tick one or more!

- University courses  if yes: how many __________
- Books  if yes: how many __________
- Other courses  if yes: how many __________
- Other  if yes: how __________

Please mark as appropriate:

Sex:  
- male  
- female

Age:  
- up to 30 years  
- 31-60 years  
- 61 years or older

Occupation:  
- student (BSc/BEng)  
- student (MSc.)  
- practitioner  
- scientist/PhD

Thank you for your participation!  
Please hand back this questionnaire together with any notes that you may have made.
B.3 VMTL Learnability User Experiment 2

A printout of the online questionnaire used in the second VMTL learnability user experiment (Experiment 2) discussed in Section 7.3 is reproduced in what follows. In this experiment participants were randomly assigned to one of three treatments. The treatments present participants with the three transformation languages in different orders, and questions are not repeated in more than one language. In order to present all questions expressed in each of the three languages, the questionnaires corresponding to all three treatments are shown in what follows. In the questionnaires, the MTLs are referred to as Vienna (VMTL), Edinburgh (Epsilon), and Hamburg (Henshin).

The correct answers for Task A of the questionnaire are listed in Table B.3.

Table B.3: Correct answers for Task A of the questionnaire used in Experiment 2 evaluating the learnability of VMTL

<table>
<thead>
<tr>
<th>Question</th>
<th>Correct answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
</tr>
</tbody>
</table>
Model Transformation Language Readability and Learnability
A Controlled Experiment

Instructions and Disclaimer
This questionnaire is part of a scientific experiment conducted by researchers at the Technical University of Denmark and Philipps-Universität Marburg. All data is stored and published in anonymized form. Participation in the experiment is entirely voluntary.

The goal of this experiment is to compare the learnability and readability of three model transformation languages called Vienna, Hamburg, and Edinburgh.

Before proceeding with the questionnaire, please read the provided handout document, containing an introduction to model transformation and descriptions of the three transformation languages used in this experiment. You are allowed to consult the handout document at any time during the experiment.

The questionnaire consists of two tasks, and completing it will take around 30 minutes.

Informed Consent
☐ I have read and understood the questionnaire instructions and consent to participating in this experiment.

Handout
☐ I have read the handout document

Feedback
The results of this experiment will be described in a scientific publication. If you want to be notified when this publication becomes available, please provide an e-mail address below.

[ ]
Participant Background

Please provide some information about yourself and your technical background.

Demographics

What is your gender?

☐ Male
☐ Female

How do you prefer to learn?

☐ By looking at pictures and infographics
☐ By listening to lectures, audiobooks, and discussions
☐ By reading textual explanations
☐ By personally exploring and experimenting

What is your occupation?

☐ Student (BSc/BEng)
☐ Student (MSc)
☐ Student (PhD)
☐ Academic/Scientist
☐ Practitioner

Technical Background

How well do you know UML?

A UML expert is able to create and reason about most of the 19 UML 2.5 diagram types.

I don’t know anything about it ☐ ☐ ☐ ☐ I’m an expert

How well do you know OCL?

An OCL expert is able to specify constraints and queries using OCL 2.4 constructs including expressions (e.g. if-then-else, let-in), collection operators (e.g. flatten, including), and iteration operations (e.g. exists, select).

I don’t know anything about it ☐ ☐ ☐ ☐ I’m an expert

How much do you know about model transformation?

A model transformation expert is able to specify transformations in several languages, such as QVT, ATL, Epsilon, or Henshin.

I don’t know anything about it ☐ ☐ ☐ ☐ I’m an expert

How well can you program?

An expert programmer has extensive knowledge about data structures and algorithms, as well as practical experience with most programming paradigms (e.g. imperative, object-oriented, functional, logic programming).

I can’t program at all ☐ ☐ ☐ ☐ I’m an expert
**Task A – Comprehension (Language: Vienna)**

Please select the correct description for each model transformation specification. If you can't solve the task, select “I don't know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a) If a Class has two public attributes of the same type named “id” and “identifier”, create a public Operation named “getId” and add it to the Class. The return type of the new Operation must be the same as the type of the “id” and “identifier” attributes, and the visibility of the “id” and “identifier” attributes must be set to “private”. The new Operation may only be created if the Class does not already own an Operation named “getId”.</td>
<td></td>
<td>a)</td>
</tr>
<tr>
<td></td>
<td>b) If a Class has a public attribute named “id” or “identifier”, create a public Operation named “getId” and add it to the Class. The return type of the new Operation must be the same as the type of the “id” or “identifier” attribute, and the visibility of the “id” or “identifier” attribute must be set to “private”. The new Operation may only be created if the Class does not already own an Operation named “getId”.</td>
<td></td>
<td>b)</td>
</tr>
</tbody>
</table>

![Diagram](attachment:image.png)
Experiment 2

- c) If a Class has a public operation named “getId”, create two public Properties named “id” and “identifier” and add them as attributes to the Class. The type of the new Properties must be the same as the return type of the “getId” Operation, and the visibility of the “getId” Operation must be set to “private”. The new Properties may only be created if the Class does not already own Properties named “id” or “identifier”, respectively.

- a) If two Classes have a public attribute with the same name and type, delete this attribute from both Classes. Also create a new superclass of the two Classes having an attribute with the same name and type as the deleted attribute.

- b) If a Class has at least two subclasses, delete all of the Class’s owned attributes. For every deleted attribute, create a new attribute with the same name and type in each subclass. The new attributes may only be created if the subclasses do not already own an attribute with the same name.
Experiment 2 – Treatment 1

3

c) If two Classes have a common superclass, and they both have a public attribute with the same name and type, delete this attribute from both Classes and add it to the superclass. The attribute may only be added if the superclass does not already own an attribute with the same name.

I don’t know.

a) If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the _extending_ Use Case.

b) If a Use Case extends another Use Case, delete all Actors associated to the extended Use Case.

c) If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the _extended_ Use Case.

I don’t know.
Task A – Comprehension (Language: Hamburg)

Please select the correct description for each model transformation specification. If you can’t solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
</table>
| 4   | Transformation Specification | a) if it does not already exist, create the “Provide credentials” Use Case, as well as an Inclusion relationship between it and all Use Cases more general than the “Authenticate” Use Case.  
     b) if it does not already exist, Create an Include relationship between all Use Cases _more_ general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.  
     c) if it does not already exist, Create an Include relationship between all Use Cases _less_ general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.  
     I don’t know. |
| 5   | Transformation Specification | a) if a course is taught by a high school teacher, remove all primary school students from this course.  
     b) if a course is attended by at least one primary school student, remove all high school teachers from this course.  
     c) if a course is taught by a high school teacher and attended by at least one primary school student, delete the course.  
     I don’t know. |
Experiment 2 – Treatment 1

6

a) If a course has more than three students, mark the course as cancelled.
b) If a course has less than three students, mark the course as cancelled.
c) If a course has less than three students, do not mark the course as cancelled.

I don't know.
Task A – Comprehension (Language: Edinburgh)

Please select the correct description for each model transformation specification. If you can’t solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>a) If an Action has exactly one outgoing Control Flow to a Decision Node, remove the Decision Node.</td>
<td>b) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, remove the guards.</td>
<td>c) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, add a Decision Node between the Action and its outgoing Control Flows.</td>
</tr>
<tr>
<td>8</td>
<td>a) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards a new Activity Final node.</td>
<td>b) If an Action has no incoming Control Flows, create an incoming Control Flow towards this action from a new Initial Node.</td>
<td>c) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards an existing Activity Final node.</td>
</tr>
<tr>
<td>9</td>
<td>a) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>target</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>source</em> of a Control Flow terminating in the deleted Fork Node.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>source</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>source</em> of a Control Flow terminating in the deleted Fork Node.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>target</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>target</em> of a Control Flow originating in the deleted Fork Node.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Task B – Assessment

Please rate the difficulty and effort required to work with each of the three transformation languages.

**Difficulty**

How difficult was it to work with the Vienna language?
Very easy 〇 〇 〇 〇 Very difficult

How difficult was it to work with the Hamburg language?
Very easy 〇 〇 〇 〇 Very difficult

How difficult was it to work with the Edinburgh language?
Very easy 〇 〇 〇 〇 Very difficult

**Effort**

How much effort was required to work with the Vienna language?
Very easy 〇 〇 〇 〇 Very difficult

How much effort was required to work with the Hamburg language?
Very easy 〇 〇 〇 〇 Very difficult

How much effort was required to work with the Edinburgh language?
Very easy 〇 〇 〇 〇 Very difficult

Thank you for your participation!
Please hand back this questionnaire together with any notes that you may have made.
Model Transformation Language Readability and Learnability
A Controlled Experiment

Instructions and Disclaimer
This questionnaire is part of a scientific experiment conducted by researchers at the Technical University of Denmark and Philipps-Universität Marburg. All data is stored and published in anonymized form. Participation in the experiment is entirely voluntary.

The goal of this experiment is to compare the learnability and readability of three model transformation languages called Edinburgh, Vienna, and Hamburg.

Before proceeding with the questionnaire, please read the provided handout document, containing an introduction to model transformation and descriptions of the three transformation languages used in this experiment. You are allowed to consult the handout document at any time during the experiment.

The questionnaire consists of two tasks, and completing it will take around 30 minutes.

Informed Consent
☐ I have read and understood the questionnaire instructions and consent to participating in this experiment.

Handout
☐ I have read the handout document

Feedback
The results of this experiment will be described in a scientific publication. If you want to be notified when this publication becomes available, please provide an e-mail address below.


Participant Background

Please provide some information about yourself and your technical background.

Demographics

What is your gender?
- Male
- Female

How do you prefer to learn?
- By looking at pictures and infographics
- By listening to lectures, audiobooks, and discussions
- By reading textual explanations
- By personally exploring and experimenting

What is your occupation?
- Student (BSc/BEng)
- Student (MSc)
- Student (PhD)
- Academic/Scientist
- Practitioner

Technical Background

How well do you know UML?

A UML expert is able to create and reason about most of the 19 UML 2.5 diagram types.
I don’t know anything about it  ○  ○  ○  ○  I’m an expert

How well do you know OCL?

An OCL expert is able to specify constraints and queries using OCL 2.4 constructs including expressions (e.g. if-then-else, let-in), collection operators (e.g. flatten, including), and iteration operations (e.g. exists, select).
I don’t know anything about it  ○  ○  ○  ○  I’m an expert

How much do you know about model transformation?

A model transformation expert is able to specify transformations in several languages, such as QVT, ATL, Epsilon, or Henshin.
I don’t know anything about it  ○  ○  ○  ○  I’m an expert

How well can you program?

An expert programmer has extensive knowledge about data structures and algorithms, as well as practical experience with most programming paradigms (e.g. imperative, object-oriented, functional, logic programming).
I can’t program at all  ○  ○  ○  ○  I’m an expert
Task A – Comprehension (Language: Edinburgh)

Please select the correct description for each model transformation specification. If you can’t solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a) If a Class has two public attributes of the same type named “id” and “identifier”, create a public Operation named “getId” and add it to the Class. The return type of the new Operation must be the same as the type of the “id” and “identifier” attributes, and the visibility of the “id” and “identifier” attributes must be set to “private”. The new Operation may only be created if the Class does not already own an Operation named “getId”.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>b) If a Class has a public attribute named “id” or “identifier”, create a public Operation named “getId” and add it to the Class. The return type of the new Operation must be the same as the type of the “id” or “identifier” attribute, and the visibility of the “id” or “identifier” attribute must be set to “private”. The new Operation may only be created if the Class does not already own an Operation named “getId”.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
c) If a Class has a public operation named “getId”, create two public Properties named “id” and “identifier” and add them as attributes to the Class. The type of the new Properties must be the same as the return type of the “getId” Operation, and the visibility of the “getId” Operation must be set to “private”. The new Properties may only be created if the Class does not already own Properties named “id” or “identifier”, respectively.

I don’t know.

a) If two Classes have a public attribute with the same name and type, delete this attribute from both Classes. Also create a new superclass of the two Classes having an attribute with the same name and type as the deleted attribute.

b) If a Class has at least two subclasses, delete all of the Class’s owned attributes. For every deleted attribute, create a new attribute with the same name and type in each subclass. The new attributes may only be created if the subclasses do not already own an attribute with the same name.
c) If two Classes have a common superclass, and they both have a public attribute with the same name and type, delete this attribute from both Classes and add it to the superclass. The attribute may only be added if the superclass does not already own an attribute with the same name.

I don’t know.

a) If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the _extending_ Use Case.

b) If a Use Case extends another Use Case, delete all Actors associated to the extended Use Case.

c) If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the _extended_ Use Case.

I don’t know.
**Task A – Comprehension (Language: Vienna)**

Please select the correct description for each model transformation specification. If you can't solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>a) If it does not already exist, create the “Provide credentials” Use Case, as well as an Inclusion relationship between it and all Use Cases more general than the “Authenticate” Use Case.</td>
<td>b) If it does not already exist, Create an Include relationship between all Use Cases <em>more</em> general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.</td>
<td>c) If it does not already exist, Create an Include relationship between all Use Cases <em>less</em> general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.</td>
</tr>
<tr>
<td>5</td>
<td>a) If a course is taught by a high school teacher, remove all primary school students from this course.</td>
<td>b) If a course is attended by at least one primary school student, remove all high school teachers from this course.</td>
<td>c) If a course is taught by a high school teacher and attended by at least one primary school student, delete the course.</td>
</tr>
</tbody>
</table>
Experiment 2 – Treatment 2

6

a) If a course has more than three students, mark the course as cancelled.

b) If a course has less than three students, mark the course as cancelled.

c) If a course has less than three students, do not mark the course as cancelled.

I don't know.
Task A – Comprehension (Language: Hamburg)

Please select the correct description for each model transformation specification. If you can't solve the task, select "I don't know".

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td>a) If an Action has exactly one outgoing Control Flow to a Decision Node, remove the Decision Node.</td>
<td>b) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, remove the guards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, add a Decision Node between the Action and its outgoing Control Flows.</td>
<td>I don't know.</td>
</tr>
<tr>
<td>8</td>
<td><img src="image2.png" alt="Diagram" /></td>
<td>a) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards a new Activity Final node.</td>
<td>b) If an Action has no incoming Control Flows, create an incoming Control Flow towards this action from a new Initial Node.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards an existing Activity Final node.</td>
<td>I don't know.</td>
</tr>
</tbody>
</table>
Experiment 2 – Treatment 2

9

a) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the _target_ of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the _source_ of a Control Flow terminating in the deleted Fork Node.

b) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the _source_ of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the _source_ of a Control Flow terminating in the deleted Fork Node.

c) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the _target_ of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the _target_ of a Control Flow originating in the deleted Fork Node.

I don’t know.
Task B – Assessment

Please rate the difficulty and effort required to work with each of the three transformation languages.

Difficulty

How difficult was it to work with the Edinburgh language?
Very easy ○ ○ ○ ○ Very difficult

How difficult was it to work with the Vienna language?
Very easy ○ ○ ○ ○ Very difficult

How difficult was it to work with the Hamburg language?
Very easy ○ ○ ○ ○ Very difficult

Effort

How much effort was required to work with the Edinburgh language?
Very easy ○ ○ ○ ○ Very difficult

How much effort was required to work with the Vienna language?
Very easy ○ ○ ○ ○ Very difficult

How much effort was required to work with the Hamburg language?
Very easy ○ ○ ○ ○ Very difficult

Thank you for your participation!
Please hand back this questionnaire together with any notes that you may have made.
Model Transformation Language Readability and Learnability
A Controlled Experiment

Instructions and Disclaimer
This questionnaire is part of a scientific experiment conducted by researchers at the Technical University of Denmark and Philipps-Universität Marburg. All data is stored and published in anonymized form. Participation in the experiment is entirely voluntary.

The goal of this experiment is to compare the learnability and readability of three model transformation languages called Hamburg, Edinburgh, and Vienna.

Before proceeding with the questionnaire, please read the provided handout document, containing an introduction to model transformation and descriptions of the three transformation languages used in this experiment. You are allowed to consult the handout document at any time during the experiment.

The questionnaire consists of two tasks, and completing it will take around 30 minutes.

Informed Consent
☐ I have read and understood the questionnaire instructions and consent to participating in this experiment.

Handout
☐ I have read the handout document

Feedback
The results of this experiment will be described in a scientific publication. If you want to be notified when this publication becomes available, please provide an e-mail address below.


Participant Background

Please provide some information about yourself and your technical background.

Demographics

What is your gender?
- Male
- Female

How do you prefer to learn?
- By looking at pictures and infographics
- By listening to lectures, audiobooks, and discussions
- By reading textual explanations
- By personally exploring and experimenting

What is your occupation?
- Student (BSc/BEng)
- Student (MSc)
- Student (PhD)
- Academic/Scientist
- Practitioner

Technical Background

How well do you know UML?
A UML expert is able to create and reason about most of the 19 UML 2.5 diagram types.
I don’t know anything about it
I’m an expert

How well do you know OCL?
An OCL expert is able to specify constraints and queries using OCL 2.4 constructs including expressions (e.g. if-then-else, let-in), collection operators (e.g. flatten, including), and iteration operations (e.g. exists, select).
I don’t know anything about it
I’m an expert

How much do you know about model transformation?
A model transformation expert is able to specify transformations in several languages, such as QVT, ATL, Epsilon, or Henshin.
I don’t know anything about it
I’m an expert

How well can you program?
An expert programmer has extensive knowledge about data structures and algorithms, as well as practical experience with most programming paradigms (e.g. imperative, object-oriented, functional, logic programming).
I can’t program at all
I’m an expert
Task A – Comprehension (Language: Hamburg)

Please select the correct description for each model transformation specification. If you can't solve the task, select "I don't know".

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a) If a Class has two public attributes of the same type named &quot;id&quot; and &quot;identifier&quot;, create a public Operation named &quot;getId&quot; and add it to the Class. The return type of the new Operation must be the same as the type of the &quot;id&quot; and &quot;identifier&quot; attributes, and the visibility of the &quot;id&quot; and &quot;identifier&quot; attributes must be set to &quot;private&quot;. The new Operation may only be created if the Class does not already own an Operation named &quot;getId&quot;.</td>
<td>a) If a Class has two public attributes of the same type named &quot;id&quot; and &quot;identifier&quot;, create a public Operation named &quot;getId&quot; and add it to the Class. The return type of the new Operation must be the same as the type of the &quot;id&quot; and &quot;identifier&quot; attributes, and the visibility of the &quot;id&quot; and &quot;identifier&quot; attributes must be set to &quot;private&quot;. The new Operation may only be created if the Class does not already own an Operation named &quot;getId&quot;.</td>
<td>b) If a Class has a public attribute named &quot;id&quot; or &quot;identifier&quot;, create a public Operation named &quot;getId&quot; and add it to the Class. The return type of the new Operation must be the same as the type of the &quot;id&quot; or &quot;identifier&quot; attribute, and the visibility of the &quot;id&quot; or &quot;identifier&quot; attribute must be set to &quot;private&quot;. The new Operation may only be created if the Class does not already own an Operation named &quot;getId&quot;.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>c)</strong> If a Class has a public operation named “getId”, create two public Properties named “id” and “identifier” and add them as attributes to the Class. The type of the new Properties must be the same as the return type of the “getId” Operation, and the visibility of the “getId” Operation must be set to “private”. The new Properties may only be created if the Class does not already own Properties named “id” or “identifier”, respectively.</td>
<td>I don’t know.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a)</strong> If two Classes have a public attribute with the same name and type, delete the attribute from both Classes and create a superclass of the two Classes having an attribute with this name and type.</td>
<td>b) If two Classes have a common superclass, and they both have a public attribute with the same name and type, create a new attribute with this name and type and add it to the superclass. The attribute may only be added if the superclass does not already own an attribute with the same name.</td>
</tr>
<tr>
<td><strong>Experiment 2 – Treatment 3</strong></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>c)</strong> If two Classes have a common superclass, and they both have a public attribute with the same name and type, remove the attribute from both Classes and add it to the common superclass. The attribute may only be added if the superclass does not already own an attribute with the same name.</td>
<td></td>
</tr>
<tr>
<td>I don’t know.</td>
<td></td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
</tr>
<tr>
<td><strong>a)</strong> If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the <em>extending</em> Use Case.</td>
<td></td>
</tr>
<tr>
<td><strong>b)</strong> If a Use Case extends another Use Case, delete all Actors associated to the extended Use Case.</td>
<td></td>
</tr>
<tr>
<td><strong>c)</strong> If two Use Cases are associated to the same Actor, and one of the Use Cases extends the other, delete the Association between the Actor and the <em>extended</em> Use Case.</td>
<td></td>
</tr>
<tr>
<td>I don’t know.</td>
<td></td>
</tr>
</tbody>
</table>
Task A – Comprehension (Language: Edinburgh)

Please select the correct description for each model transformation specification. If you can’t solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>a) If it does not already exist, create the “Provide credentials” Use Case, as well as an Inclusion relationship between it and all Use Cases more general than the “Authenticate” Use Case.</td>
<td>a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) If it does not already exist, Create an Include relationship between all Use Cases <em>more</em> general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.</td>
<td>b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) If it does not already exist, Create an Include relationship between all Use Cases <em>less</em> general than the “Authenticate” Use Case, and the “Provide credentials” Use Case.</td>
<td>c)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a) If a course is taught by a high school teacher, remove all primary school students from this course.</td>
<td>a)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) If a course is attended by at least one primary school student, remove all high school teachers from this course.</td>
<td>b)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) If a course is taught by a high school teacher and attended by at least one primary school student, delete the course.</td>
<td>c)</td>
<td></td>
</tr>
</tbody>
</table>
Experiment 2 – Treatment 3

| a) If a course has more than three students, mark the course as cancelled. |
| b) If a course has less than three students, mark the course as cancelled. |
| c) If a course has less than three students, do not mark the course as cancelled. |
| I don’t know. |
### Task A – Comprehension (Language: Vienna)

Please select the correct description for each model transformation specification. If you can’t solve the task, select “I don’t know”.

<table>
<thead>
<tr>
<th>No.</th>
<th>Transformation Specification</th>
<th>Description</th>
<th>Correct</th>
</tr>
</thead>
</table>
| 7   | ![Diagram 1](image1.png)    | a) If an Action has exactly one outgoing Control Flow to a Decision Node, remove the Decision Node.  
b) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, remove the guards.  
c) If an Action has two outgoing Control Flows to other Actions guarded by “true” and “false”, add a Decision Node between the Action and its outgoing Control Flows. | ![Diagram 2](image2.png) | I don’t know. |
| 8   | ![Diagram 3](image3.png)    | a) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards a new Activity Final node.  
b) If an Action has no incoming Control Flows, create an incoming Control Flow towards this action from a new Initial Node.  
c) If an Action has no outgoing Control Flows, create an outgoing Control Flow from this Action towards an existing Activity Final node. | ![Diagram 4](image4.png) | I don’t know. |
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>a) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>target</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>source</em> of a Control Flow terminating in the deleted Fork Node.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>source</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>source</em> of a Control Flow terminating in the deleted Fork Node.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c) If two Fork Nodes are connected by a Control Flow, delete the Fork Node acting as the <em>target</em> of this Control Flow. Create a new Control Flow between the remaining Fork Node and any Activity Node previously acting as the <em>target</em> of a Control Flow originating in the deleted Fork Node.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I don’t know.</td>
<td></td>
</tr>
</tbody>
</table>
Task B – Assessment

Please rate the difficulty and effort required to work with each of the three transformation languages.

**Difficulty**

How difficult was it to work with the Hamburg language?
- Very easy
- Very difficult

How difficult was it to work with the Edinburgh language?
- Very easy
- Very difficult

How difficult was it to work with the Vienna language?
- Very easy
- Very difficult

**Effort**

How much effort was required to work with the Hamburg language?
- Very easy
- Very difficult

How much effort was required to work with the Edinburgh language?
- Very easy
- Very difficult

How much effort was required to work with the Vienna language?
- Very easy
- Very difficult

Thank you for your participation!

Please hand back this questionnaire together with any notes that you may have made.
Bibliography


