

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

# Towards Automated Support for the Co-Evolution of Meta-Models and Grammars

WEIXING ZHANG



Division of Interaction Design & Software Engineering  
Department of Computer Science & Engineering  
Chalmers University of Technology and University of Gothenburg  
Gothenburg, Sweden, 2023

# **Towards Automated Support for the Co-Evolution of Meta-Models and Grammars**

WEIXING ZHANG

Copyright ©2023 Weixing Zhang  
except where otherwise stated.  
All rights reserved.

Department of Computer Science & Engineering  
Division of Interaction Design & Software Engineering  
Chalmers University of Technology and University of Gothenburg  
Gothenburg, Sweden

This thesis has been prepared using L<sup>A</sup>T<sub>E</sub>X.  
Printed by Chalmers Reproservice,  
Gothenburg, Sweden 2023.

*“What is life? Life is a process of continuous self-improvement  
through perception.”*  
- Kazuo Inamori (Renowned Japanese Entrepreneur)



# Abstract

Blended modeling is an emerging paradigm involving seamless interaction between multiple notations for the same underlying modeling language. We focus on a model-driven engineering (MDE) approach based on meta-models to develop textual languages to improve the blended modeling capabilities of modeling tools. In this thesis, we propose an approach that can support the co-evolution of meta-models and grammars as language engineers develop textual languages in a meta-model-based MDE setting. Firstly, we comprehensively report on the challenges and limitations of modeling tools that support blended modeling, as well as opportunities to improve them. Second, we demonstrate how language engineers can extend Xtext's generator capabilities according to their needs. Third, we propose a semi-automatic method to transform a language with a generated grammar into a Python-style language. Finally, we provide a solution (i.e., GrammarOptimizer) that can support rapid prototyping of languages in different styles and the co-evolution of meta-models and grammars of evolving languages.

## Keywords

Blended Modeling, Systematic Literature Review, Xtext, Grammar Optimization, Co-Evolution



# Acknowledgment

I want to express my sincere gratitude to my three supervisors: Dr. Jan-Philipp Steghöfer, Dr. Regina Hebig, and Dr. Daniel Strüber (in the order by time). They provided me with ample assistance, support, enthusiasm, patience, and tolerance. They guided me with their extensive expertise and rigorous scholarly attitude, helping me acquire knowledge, skills, and methodology in my research field, and successfully transitioning my mindset from an almost rigid engineer to a researcher. I have also learned an important thing from them: Do things in the right way. All of those have laid a solid foundation for my future academic career. I would also like to thank my former colleague, independent researcher Dr. Jörg Holtmann. During his time in the CSE department, he offered me a lot of help, both in research and engineering.

Furthermore, I want to express my gratitude to my examiner, Prof. Johan Karlsson, and other members of the doctoral school for their constructive feedback and administrative assistance. Special thanks to the post-doc Dr. Shiliang Zhang at the University of Oslo, senior researcher Dr. Yemao Man at ABB company, and my colleague Wardah Mahmood. Shiliang and Yemao have answered many of my questions and provided valuable and helpful advice. In the second year of my Ph.D., I felt a lot of pressure and lacked confidence during a certain period, and Wardah provided many effective suggestions on how to adjust my mentality, and that helped me get through the hard times. Additionally, I would like to thank all my colleagues in the IDSE division and the wonderful atmosphere they collectively created. Working and studying in the IDSE division is the luckiest thing. In addition, I would also like to thank Dawen Liang, a senior software engineer from the United States, and Jiawen Wu, a doctoral student at the University of Jyväskylä, who have always supported me with their best wishes and encouragement.

Finally, I am deeply thankful to my wife Jinying Li, and my whole family. Without their support and blessings, I would not have been able to study abroad, let alone live abroad for my doctoral studies. In particular, I would like to express my gratitude to my wife for her incredible and endless love, support, and patience. Without her sacrifices and positive energy, I would not have been able to come this far.





# List of Publications

## Appended publications

This thesis is based on the following publications:

- [A] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J. Steghöfer, R. Hebig  
“Blended Modeling in Commercial and Open-source Model-Driven Software Engineering Tools: A Systematic Study”  
*Software and Systems Modeling (SoSyM)*, 2023, 22(1), pp. 415-447.
- [B] J. Holtmann, J. Steghöfer, W. Zhang  
“Exploiting Meta-Model Structures in the Generation of Xtext Editors”  
*11th International Conference on Model-Based Software and Systems Engineering, SciTePress*, 2023, pp. 218-225.
- [C] W. Zhang, R. Hebig, J. Steghöfer, J. Holtmann  
“Creating Python-style Domain Specific Languages: A Semi-automated Approach and Intermediate Results”  
*11th International Conference on Model-Based Software and Systems Engineering, SciTePress*, 2023, pp. 210-217.
- [D] W. Zhang, J. Holtmann, D. Strüber, R. Hebig, J. Steghöfer  
“Supporting Meta-model-based Language Evolution and Rapid Prototyping with Automated Grammar Optimization”  
*Revised and Re-submitted to Journal of Systems and Software*, 2023

## Other publications

The following publications were published during my PhD studies. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to this licentiate thesis.

- [a] Wenli Zhang, Weixing Zhang, D. Strüber, R. Hebig  
“Manual Abstraction in the Wild: A Multiple-Case Study on OSS Systems’  
Class Diagrams and Implementations”  
*Accepted in 26th International Conference on Model Driven Engineering  
Languages and Systems (MODELS). ACM. 2023.*
  
- [b] W. Zhang, R. Hebig, D. Strüber, J. Steghöfer  
“Automated Extraction of Grammar Optimization Rule Configurations  
for Metamodel-Grammar Co-evolution”  
*16th ACM SIGPLAN International Conference on Software Language  
Engineering (SLE). ACM. 2023, pp. 84–96.*

## Research Contribution

I was the main driver and contributor of papers C and D. Also, I have significant contributions to papers A and B. My contributions in these papers are classified according to the Contributor Roles Taxonomy (CRediT).

In Paper A, I did an entire gray literature review with Dr. Steghöfer and Dr. Hebig. I participated in developing the methodological design of the gray literature review and jointly performed the investigation of about 1,500 web pages. I jointly identified modeling tools in the gray literature and participated in data collection based on the identified tools. This data collection requires downloading and trying out the tool and completing the data collection by verifying the functionality and features of the tool. I reported the results of the gray literature review to the paper team and participated in writing the content of the gray literature.

In paper B, I worked with Dr. Steghöfer and Dr. Holtmann on the implementation of the project involved in the paper. To address the four technical challenges in this paper, I investigated technical manuals and open-source web pages. I provided technical solutions for two of the four challenges, i.e. “content-assist for new model elements with unique names” and “scoping for cross-references”. I also investigated materials for the writing in this paper, i.e., finding related work. I also participated in the validation work of the paper, i.e., validating whether the engineering implementation we have completed conforms to the design of the technical solutions for solving the four technical challenges. I once provided a presentation on the academic work of the EATXT editor at the SE division seminar, which included the research content of paper B.

In paper C, I completed the conceptualization of engineering. During the engineering, I developed a script to automate grammar modification work and applied this script to multiple languages as a validation. I wrote the initial full draft of this paper and then combined it with the review comments of Dr. Hebig and Dr. Steghöfer to make it perfect. Dr. Hebig suggested I add the validation part which is one of the key steps to make the paper perfect. Before this paper, Dr. Holtmann, Dr. Steghöfer, and I jointly implemented the development of the EATXT editor which the content is involved in paper B. The concept of paper C originally originated from the EATXT technical discussion, but it was not implemented in EATXT in the end. Therefore, I extracted the concept separately and searched for the language background of the case to complete paper C.

The concept of paper D was first proposed by my co-author Dr. Hebig, while the concept originated from a script I developed, so I further refined and determined the concept. I, together with Dr. Holtmann and Dr. Hebig, performed a large number of systematic analyses in this paper, most of which were performed by me, resulting in many documents and data. The development of the paper also involved a significant amount of Java programming, most of which was performed by me. The model-driven architecture idea of the software developed in this paper came from Dr. Holtmann. Dr. Steghöfer implemented the initial steps, and I fully developed the architecture. Early in the paper, I made contributions to the investigation by checking whether initial sample languages identified by Dr. Steghöfer were appropriate for our

research context and purpose. The entire process of the paper was managed as a project, and the initial administration plan was developed by Dr. Steghöfer, later, the plan was maintained by me, in particular, during the revision in the journal revision process. Supported by Dr. Holtmann and Dr. Hebig, I performed the validation of the research results of the paper (i.e., the software we developed), taking on the majority of the effort. I made major contributions to the manuscript in both its initial and the revised version. For the revised version, I addressed the majority of the reviewer comments, supported by Dr. Strüder, who joined the team of authors for the revision. Additionally, I gave presentations on the paper on several occasions.

<b>Role</b>	<b>Paper A</b>	<b>Paper B</b>	<b>Paper C</b>	<b>Paper D</b>
Conceptualization			X	X
Data curation	X	X	X	X
Formal analysis				
Funding acquisition				
Investigation	X	X	X	X
Methodology	X		X	
Project administration			X	X
Resources				
Software		X	X	X
Supervision				
Validation		X	X	X
Visualization		X	X	X
Writing – original draft	X	X	X	X
Writing – Review and Editing			X	X

# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgement</b>	<b>vii</b>
<b>List of Publications</b>	<b>ix</b>
<b>Personal Contribution</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Related Work . . . . .	4
1.1.1 Blended Modeling . . . . .	4
1.1.2 EAST-ADL . . . . .	4
1.1.3 Xtext and Meta-Model-Based DSL Engineering . . . . .	4
1.1.4 Co-Evolution in MDE Contexts . . . . .	5
1.2 Methodology . . . . .	6
1.2.1 Stage 1: Multi-Vocal Literature Review . . . . .	6
1.2.2 Stage 2: Extending Xtext’s Generator Capabilities . . . . .	8
1.2.3 Stage 3: Python-Style Prototyping . . . . .	9
1.2.4 Stage 4: Generalize Grammar Optimization Approach . . . . .	10
1.3 Results and Evaluation . . . . .	11
1.3.1 Results and Evaluation in Stage 1 . . . . .	12
1.3.2 Results and Evaluation in Stage 2 . . . . .	13
1.3.3 Results and Evaluation in Stage 3 . . . . .	13
1.3.4 Results and Evaluation in Stage 4 . . . . .	14
1.4 Answers to the RQs . . . . .	15
1.5 Threats to Validity . . . . .	16
1.5.1 External Validity . . . . .	16
1.5.2 Internal Validity . . . . .	17
1.6 Summary of Contributions . . . . .	17
1.7 Conclusion and Future Work . . . . .	19
<b>2 Paper A</b>	<b>21</b>
2.1 Introduction . . . . .	22
2.1.1 What is blended modeling? . . . . .	22
2.1.2 What is <i>not</i> blended modeling? . . . . .	23
2.1.3 Motivation and aim . . . . .	23
2.1.4 Structure . . . . .	24
2.2 Background . . . . .	25

2.2.1	Multiple notations . . . . .	25
2.2.1.1	Multi-view modeling . . . . .	25
2.2.1.2	Multi-Paradigm Modeling . . . . .	25
2.2.2	Seamless interaction . . . . .	26
2.2.2.1	Text-based modeling with graphical visualizations . . . . .	26
2.2.2.2	Mixed textual and graphical modeling . . . . .	27
2.2.2.3	Projectional editing . . . . .	28
2.2.3	Inconsistency management . . . . .	28
2.2.4	Related secondary literature . . . . .	29
2.3	Study design . . . . .	33
2.3.1	Process . . . . .	34
2.3.1.1	Planning . . . . .	34
2.3.1.2	Conducting . . . . .	35
2.3.1.3	Documenting . . . . .	36
2.3.2	Research questions . . . . .	36
2.3.3	Search and selection . . . . .	37
2.3.3.1	Systematic Reviews . . . . .	37
2.3.3.2	Tool identification . . . . .	41
2.3.4	Classification framework definition . . . . .	43
2.3.5	Data extraction . . . . .	44
2.3.6	Data validation . . . . .	44
2.3.7	Data analysis . . . . .	44
2.3.7.1	Vertical analysis . . . . .	45
2.3.7.2	Horizontal analysis . . . . .	45
2.4	Results . . . . .	45
2.4.1	Overview . . . . .	48
2.4.2	User-oriented characteristics (RQ1) . . . . .	50
2.4.2.1	Notations . . . . .	50
2.4.2.2	Visualization and navigation . . . . .	52
2.4.2.3	Flexibility . . . . .	54
2.4.3	Realization-oriented characteristics (RQ2) . . . . .	55
2.4.3.1	Mapping and platforms . . . . .	55
2.4.3.2	Change propagation and traceability . . . . .	56
2.4.3.3	Inconsistency management . . . . .	57
2.5	Orthogonal findings . . . . .	58
2.5.1	Number of notation types and Overlap of notations . . . . .	58
2.5.2	Seamless interaction . . . . .	59
2.5.3	Flexibility and inconsistency management . . . . .	60
2.5.4	Technological trends . . . . .	60
2.6	Discussion . . . . .	61
2.6.1	Takeaways . . . . .	61
2.6.2	Challenges and opportunities . . . . .	62
2.7	Threats to validity . . . . .	65
2.7.1	External validity . . . . .	65
2.7.2	Internal validity . . . . .	65
2.7.3	Construct validity . . . . .	66
2.7.4	Conclusion validity . . . . .	66
2.8	Conclusions . . . . .	66

<b>3</b>	<b>Paper B</b>	<b>71</b>
3.1	Introduction . . . . .	72
3.2	Related Work . . . . .	72
3.3	Background . . . . .	73
3.3.1	Xtext . . . . .	73
3.3.2	EAST-ADL and EATXT . . . . .	74
3.4	Challenges and Solutions . . . . .	75
3.4.1	Template Proposals . . . . .	76
3.4.2	Content-assist for new Model Elements with Unique Names	78
3.4.3	Formatters . . . . .	79
3.4.4	Scoping for Cross-references . . . . .	80
3.5	Conclusion and Outlook . . . . .	82
<b>4</b>	<b>Paper C</b>	<b>83</b>
4.1	Introduction . . . . .	84
4.2	Background . . . . .	84
4.3	Methodology . . . . .	85
4.4	Results . . . . .	86
4.4.1	Analysis . . . . .	86
4.4.2	The Semi-automated Approach . . . . .	88
4.4.3	Evaluation . . . . .	89
4.5	Discussion . . . . .	92
4.5.1	Threats to Validity and Limitations . . . . .	92
4.5.2	Future Work . . . . .	93
4.6	Related Work . . . . .	94
4.7	Conclusion . . . . .	95
<b>5</b>	<b>Paper D</b>	<b>97</b>
5.1	Introduction . . . . .	98
5.2	Background: Textual DSL Engineering based on Meta-models	101
5.3	Related Work . . . . .	102
5.4	Methodology . . . . .	105
5.4.1	Selection of Sample DSLs . . . . .	105
5.4.2	Exclusion of Language Parts for Low-level Expressions	107
5.4.3	Meta-model Preparations and Generating an Xtext Grammar	107
5.4.4	Comparing EBNF and Xtext grammars . . . . .	109
5.4.5	Analysis of Grammars . . . . .	110
5.4.5.1	First Iteration: Identify Optimization Rules	111
5.4.5.2	Second iteration: Validate Optimization Rules	113
5.5	Identified Optimization Rules . . . . .	114
5.6	Solution: Design and Implementation . . . . .	116
5.6.1	Grammar Representation . . . . .	116
5.6.2	Optimization Rule Design . . . . .	116
5.6.3	Configuration . . . . .	117
5.6.4	Execution . . . . .	118
5.6.5	Post-Processing vs. Changing Grammar Generation	120
5.6.6	Limitations and Caveats . . . . .	120
5.7	Evaluation . . . . .	121

5.7.1	Grammar Adaptation (RQ1)	121
5.7.1.1	Cases	122
5.7.1.2	Method	122
5.7.1.3	Metrics	122
5.7.1.4	Results	123
5.7.2	Supporting Evolution (RQ2)	126
5.7.2.1	Cases	126
5.7.2.2	Preparation of the QVTo Case	127
5.7.2.3	Method	128
5.7.2.4	Metrics	129
5.7.2.5	Results	129
5.8	Discussion	132
5.8.1	Threats to Validity	132
5.8.1.1	Construct Validity	132
5.8.1.2	Internal Validity	133
5.8.1.3	External Validity	133
5.8.1.4	Reliability	133
5.8.2	The Effort of Creating and Evolving a Language with the GRAMMAROPTIMIZER	134
5.8.3	Implications for Practitioners and Researchers	134
5.8.4	Future Work	135
5.9	Conclusion	137

## Bibliography

**139**



# Chapter 1

## Introduction

Blended modeling is a rapidly emerging modeling technology that involves seamless interactions between multiple notations (i.e., concrete syntax) and a single model (i.e., abstract syntax), allowing for a certain degree of temporary inconsistency [1]. Different modeling notations have their respective advantages. For example, the visual connections between elements representing domain concepts in graphical notations make it easier for users to understand the relationships between concepts, the tree-based notation of the model makes its hierarchical structure clearer for users, and textual notations have unique advantages in fast editing and global replacement. By blending different modeling notations in the same modeling tool and making them adhere to the same abstract syntax, respective advantages of different modeling notations will be available at the same time. Modifications made to the model in one notation can be synchronized to the other notations. Blended modeling increases modeling flexibility and efficiency as well as productivity. Additionally, engineers can choose their preferred notation for modeling based on their preferences.

Adding textual notations to a language through development is a way to improve a language’s blended modeling capability. There are many existing tools for developing textual domain-specific languages (DSLs), and Xtext [2] is one of the popular textual DSL development tools [3]. Xtext relies on the Eclipse Modeling Framework (EMF) [4] and uses its Ecore (meta-)modeling facilities as a basis. Developing a textual DSL in Xtext involves two main artifacts: a grammar, which defines the concrete syntax of the language, and a meta-model, which defines the abstract syntax. Xtext allows either the grammar or the meta-model to be created first, and then automatically generates the one from the other (or alternatively, writing both manually and aligning them). We use the term “model-driven engineering (MDE) approach” to refer to the strategy of first creating a meta-model and then generating a grammar from that meta-model. Applying the MDE approach, language engineers can generate a textual grammar from the meta-model instead of designing the grammar from scratch. Additionally, the MDE approach is most suited for the scenario where multiple concrete syntaxes adhere to a single abstract syntax, which coincides with the goal of blended modeling.

However, in the MDE approach, grammars generated from metamodels

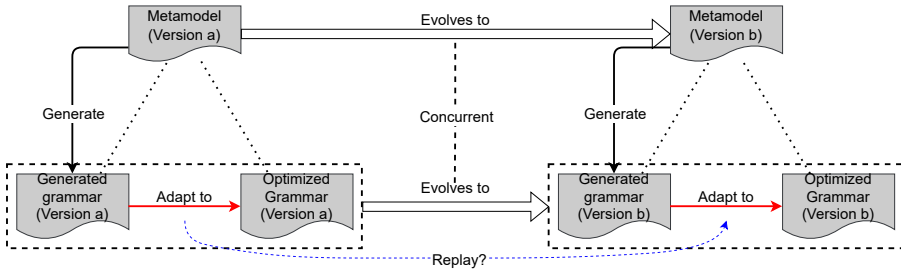


Figure 1.1: In a MDE approach, the grammar is adapted before being put into use, and when the language evolves, the grammar generated from the evolved meta-model needs to be adapted again.

(hereinafter referred to as **generated grammars**) often require adaptation before being put into use. The grammar generated by Xtext from the meta-model is composed of grammar rules. These grammar rules and their contained attributes, keywords, and other elements always follow a fixed format. In a real DSL application, some of these elements may be unnecessary or not expressive enough. For example, to express the semantics of “assign 0 to the variable *value*”, the default output generated by Xtext “*value* 0” does not agree with how variable assignment is typically represented. One solution is for the language engineer to modify the grammar definition, i.e., add an equal sign “=” after the keyword “*value*”. Moreover, the curly braces included by default in the generated grammar often lead to deep nesting in the program, which makes the grammar cumbersome to use.

Because of that generated grammars require manual adaptation before they can be used. However, a further problem faced when adapting grammar is related to the workload of adaptation. When manually adapting a generated grammar, language engineers are faced with repeating the same operation across many grammar rules. For example, moving an attribute to the outside of the container braces in many grammar rules would be time-consuming. Another problem of grammar adaptation comes from the evolution of language. There is a practical scenario showing the evolution of a language (as shown in Figure 1.1). When the meta-model evolves, the original grammar would be out of date, so then the grammar needs to be regenerated from the evolved meta-model. However, the grammar generated from the evolved version of the meta-model does not contain the manual improvements in the previous version. In this case, language engineers have to manually adapt the newly generated grammar once again and thus leading to repetitive work. Note that this is an issue of the MDE approach to language development. We will later discuss an alternative, specifically, a grammar-driven approach, and the respective advantages and benefits.

Finally, after the grammar is ready, a complete infrastructure including a parser, compiler, etc. can be obtained based on the grammar using Xtext. However, some comment features of modern editors, such as template proposals, are still not supported in the editor composed of this infrastructure which is based on Xtext out-of-the-box. We identified here another issue that needed to be addressed, i.e., the default Xtext’s generator capabilities are limited.

In this thesis, we propose an approach that can support the co-evolution of meta-models and grammars as language engineers develop textual languages in a meta-model-based MDE setting. To this end, first, we studied the state-of-the-art and practice of modeling tools that support blended modeling by conducting a systematic literature review. After studying the limitations and opportunities of existing modeling tools that support blended modeling, we decided to explore and improve blended modeling technologies on a specific case language. As a start, we developed a textual language (i.e., EATXT) for EAST-ADL based on the MDE approach. In this process, to address the limitations of the default Xtext generators’ capability and its implementation, we proposed ways in which language engineers can extend the Xtext generators’ capability and its implementation according to their own needs. Meanwhile, to solve the inherent problems of grammars generated from meta-models, we first proposed a semi-automatic method that can change the language with the generated grammar to a Python-style language. To solve the problem that manual improvements to the generated grammar cannot be replayed in evolved versions, we proposed a more general grammar adaptation method, i.e., GrammarOptimizer (we name grammar adaptation **grammar optimization**), which can optimize the generated grammar of languages in different styles. Moreover, its optimization on the generated grammar of the previous version is saved in the form of configurations. These configurations can be reused in the generated grammar of the evolved version, thereby supporting the co-evolution of meta-model and grammar.

To guide our research, we address the following research questions in this thesis:

**RQ1:** *What are the user-oriented characteristics of modeling tools most suitable for supporting blended modeling?*

By answering this research question, we aim to identify the external characteristics of modeling tools that are relevant to their adoption and use, e.g., the notations (types) they support, etc.

**RQ2:** *What are the realization-oriented characteristics of modeling tools most suitable for supporting blended modeling?*

By answering this research question, we aim to identify the internal characteristics of modeling tools, as well as the technologies used to implement these characteristics, such as the implementation platforms they use, etc. Answering the above two questions is crucial because practitioners can learn from the answers about the limitations of current blended modeling tools and how they can improve the tools, and researchers, including us, can learn from the answers the state of the practice in blended modeling tools, including the gaps that need to be filled.

**RQ3:** *How can we build a solution to adapt generated grammars to produce the same language as available expert-created grammars?*

We developed a textual language for EAST-ADL to explore and improve blended modeling technologies after studying the state-of-the-art and practice of blended modeling tools and we proposed a method that optimizes the generated grammar and we propose a generalized grammar optimization method. By answering this research question, we aimed to evaluate the generalizability of this optimization method, i.e. whether it can adapt the generated grammar to languages to produce the same language as available expert-created grammars.

**RQ4:** *Can our solution support the co-evolution of generated grammars when the meta-model evolves?*

By answering this research question, we aim to evaluate whether the proposed grammar optimization method supports the co-evolution of meta-models and grammars as the language evolves.

## 1.1 Background and Related Work

In this section, I will first introduce the background and related work such as blended modeling, Xtext, and metamodel-based DSL engineering, co-evolution in MDE contexts, and then other concepts such as EAST-ADL will be introduced in subsequent chapters.

### 1.1.1 Blended Modeling

The integration of graphical modeling and textual modeling was not a new thing [5, 6, 7, 8, 9, 10], however, Ciccozzi et al. formally conceptualized **blended modeling** for the first time in [11], defining it as follows:

*“Blended modeling is the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes), allowing a certain degree of temporary inconsistencies.”*

It distinguishes itself from Multi-View Modeling (MVM) [12] and Multi-Paradigm Modeling (MPM) [13] by possessing the following three characteristics: Multiple notations, Seamless interaction, and Flexible consistency management.

There are existing many modeling tools, e.g., SequenceDiagramOrg [14] which blends both graphical and textual notations. However, which tools support blended modeling and how they support it remains largely unknown.

### 1.1.2 EAST-ADL

EAST-ADL [15] is an architectural description language used in the domain of automotive embedded systems and is based on a metamodel with approximately 300 meta-classes and a hierarchy in which nested elements describe different aspects of the electronic vehicle system. As mentioned before, we use EAST-ADL as a case language to explore and improve blended modeling technology. EAST-ADL can be edited by EATOP [16] which is an eclipse-based modeling tool specifically for EAST-ADL, which provides tree-based and table-based editing capabilities but does not provide text symbol-based editing capabilities.

### 1.1.3 Xtext and Meta-Model-Based DSL Engineering

Eclipse Xtext is a framework used for the development of programming languages and domain-specific languages (DSLs) [17]. It is one of the many popular DSL development tools [3]. Xtext supports both grammar-based and meta-model-based DSL development approaches. In the grammar-based approach, users directly define the grammar of the DSL, and the meta-model can be generated from this grammar. Xtext artifacts are then generated from the grammar, providing the foundation for textual editor infrastructure. Xtext also supports the meta-model-based approach, where users first represent domain

concepts and their relationships by creating a meta-model. From this meta-model, grammar is generated, and subsequently, a textual editor is generated from this grammar. The generation process is controlled by the Modeling Workflow Engine (MWE2). Running MWE2 allows for the generation of a full infrastructure, including a parser, linker, type checker, compiler, and editing support for Eclipse, any editor that supports the Language Server Protocol, and web browsers [2].

At the beginning of the language creation process, it does not need to be clear whether the new language is text-based, graphical, or both [18]. In fact, following the philosophy of blended modeling [11], there are good reasons to support multiple syntaxes, as different developers are likely to benefit from different syntax paradigms. The MDE approach is most suited for this philosophy. Additionally, generating grammar directly from the meta-model avoids designing grammar from scratch. Therefore, in this thesis, we adopted the MDE approach for developing the DSL.

The full infrastructure used for generating text editors, known as the language generator (in the form of Java code), is available out of the box and can be extended or replaced as needed. For large languages such as EAST-ADL, the default content assistant features may be insufficient, allowing users to modify and extend the existing generator. Furthermore, when Xtext generates grammar from the meta-model, it generates a grammar rule for each meta-class and also generates an attribute in the grammar rule for each attribute in a meta-class, resulting in grammar closely adhering to the meta-model. The generated grammar has a default format, where each grammar rule includes a keyword with the same name as the rule and is enclosed in curly braces, containing multiple attributes. Each attribute includes a keyword followed by an attribute string. This inherent format introduces certain challenges, e.g., redundant keywords and curly braces, or the default keywords not effectively conveying semantics, etc. These are inherent characteristics and limitations of grammar generated from the meta-model.

#### 1.1.4 Co-Evolution in MDE Contexts

In model-driven engineering, it is well-known that evolutionary changes to an artifact may affect other artifacts, which leads to several co-evolution scenarios. The most prominent one is *meta-model/model* co-evolution, in which a meta-model is evolved and corresponding instances have to be updated to stay in sync with the meta-model. This scenario has inspired a substantial body of work. Hebig et al. [19] survey 31 relevant approaches, classifying them according to their support for change collection, change identification, and model resolution. Beyond meta-model/model co-evolution, co-evolution between meta-models and other MDE artifacts have received attention as well, including associated OCL constraints [20], model transformations [21, 22], code generators [23], and graphical editor models [24]. Inconsistencies between evolved meta-models and general MDE artifacts have also been addressed in the context of *technical debt management*, with an approach that assists the modeler with the aid of interactive visualization tools [25]. However, except for GrammarOptimizer described in Chapter 5, on which we build and improve with our contribution, we are not aware of previous work on meta-model/grammar co-evolution.

Model federation [26, 27, 28] deals with the challenges of keeping several models synchronized, which is related to our addressed co-evolution scenario. However, to the best of our knowledge, there is no previous work that applies model federation techniques to grammars. Previous work is often focused on establishing links between the different involved artifacts, which, in our scenario, is a non-issue. However, the actual modification for keeping several artifacts synchronized is often simpler if only models are involved, than in our case deals with concrete textual syntaxes. For example, the order of attributes in the grammar does not have to be consistent with the corresponding meta-model attributes but can be changed freely according to the developer’s design intention. In fact, the approach enabled by our contribution could be used to augment available model federation frameworks to make them applicable to grammars as well.

## 1.2 Methodology

Figure 1.2 depicts all the studies included in this thesis and the problems and methodologies involved. We went through four stages to complete these studies. In **Stage 1**, to understand the potential of current commercial and open-source modeling tools to support blended modeling, we have designed and carried out a systematic literature review. Through this study, we discovered the shortcomings of existing blended modeling technology and the opportunities of improving blended modeling technologies, which motivated us to explore and improve blended modeling technology. In **Stage 2**, we developed a textual language for EAST-ADL as a starting point for exploring and improving blended modeling technologies. We introduced ways to extend the Xtext’s generator capabilities. In **Stage 3**, we proposed a semi-automated method for turning a language with generated grammar into a Python-style language, thereby making the language more concise and user-friendly. In **Stage 4**, we extracted grammar optimization rules from the analysis on seven case languages, and developed a grammar optimization tool GrammarOptimizer to include these general optimization rules, and finally evaluated our approach through multiple exemplar languages. The above methodologies aimed to address the identified problems described in Section 1, and will be introduced in detail in the following subsections.

### 1.2.1 Stage 1: Multi-Vocal Literature Review

To understand the state-of-the-art and practice of modeling tools that support blended modeling, we conducted a Multi-vocal Literature Review (MLR) in this stage. An MLR is a form of Systematic Literature Review (SLR) encompassing formally published literature (e.g., journal and conference papers) and Gray Literature (GL), such as blog posts and whitepapers [29]. The specific focus of the MLR in this thesis is blended modeling, making the SLR an academic synthesis of evidence of blended modeling. As a supplement, we also conduct a Grey Literature Review (GLR).

Software engineering practitioners typically lack the time, channels, or expertise to review formally peer-reviewed papers. They are more likely to turn to sources like online blogs, technical reports, and other GL. Furthermore,

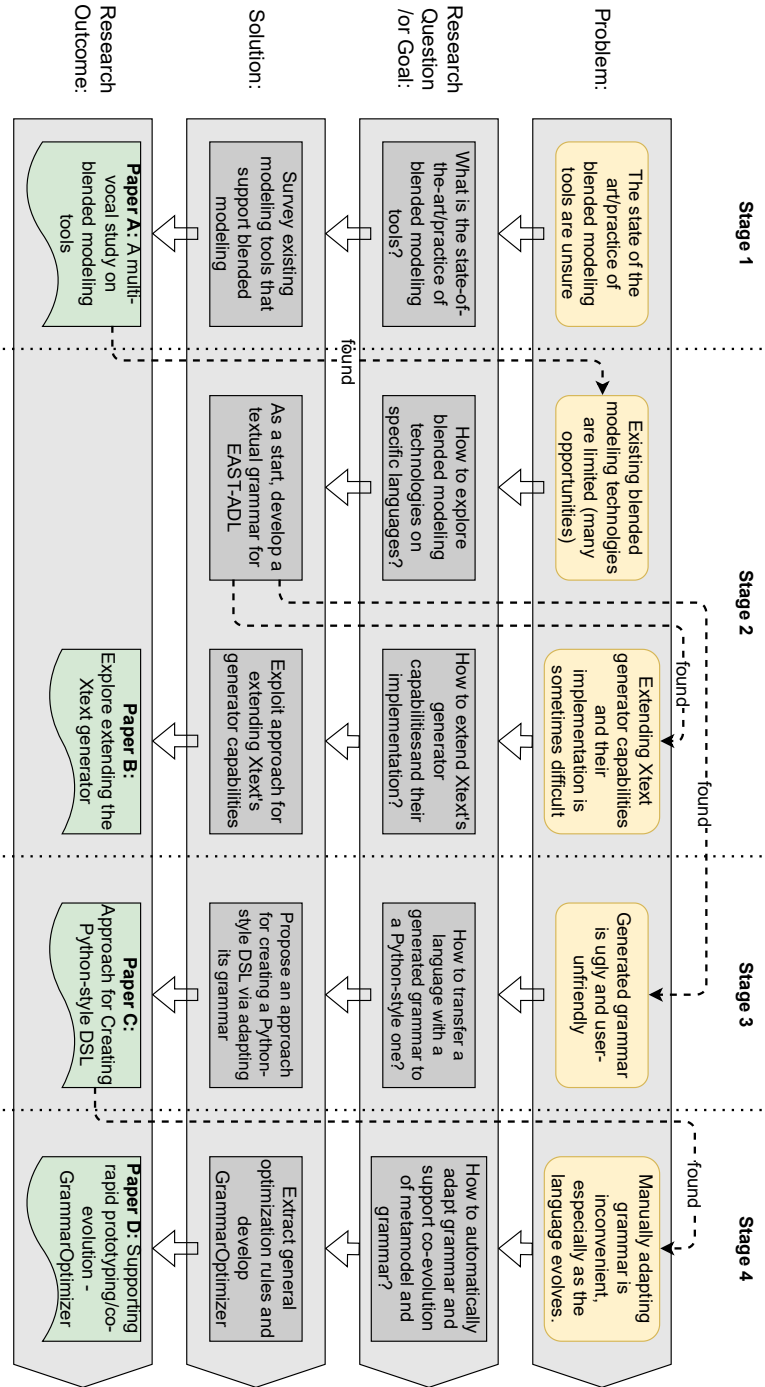


Figure 1.2: The research included in this thesis involves multiple problems. We have implemented different research activities and obtained some research results for these different problems.

they are willing to contribute their thoughts and experiences in the form of GL, which can serve as a valuable source of data for research [30]. Moreover, the research objective of the investigation was modeling tools, which often provide specific information about the tools in their user manuals, such as whether they support textual modeling notations. Based on the above advantages of GL, we conducted GLR as the supplement for this investigation.

At the same time, GL also have their inherent limitations. I.e., GL have not gone through conventional publication channels, which means that they do not undergo rigorous and formal peer review [31], and gray literature can be difficult to search and retrieve for evidence synthesis. Therefore, in this investigation, we combine SLR with GLR instead of adopting one of them alone.

We followed the common process for conducting a GLR, which involves the following steps [32]: 1) Select where to search, such as Google. 2) Set search strings. The search engine will search based on them. 3) Source selection. There may be many search results, so it is necessary to set a selection criterion, such as what sources should be excluded, and perform the selection according to this criterion. 4) Data extraction. Each source may provide a variety of different information, so in this step, it is needed to set the extraction goal, i.e., what data to extract. 5) Data synthesis. This step is for summarizing and analyzing the extracted data to draw meaningful insights and conclusions.

MLR, which combines SLR and GLR, provides several advantages for our investigation in this thesis. SLR ensures rigorous analysis of published academic literature to provide a solid foundation for existing research. On the other hand, GLR expands the scope and captures information lacking in research papers, such as descriptions of product features in user manuals in modeling tools. This combination allows our investigation to access a wider range of data sources, resulting in a more comprehensive investigation of blended modeling tools and reducing publication bias. Practitioners and researchers, including ourselves, can benefit from such a comprehensive investigation to enhance decision-making and facilitate the identification of gaps and future research directions in academic and practical contexts.

### 1.2.2 Stage 2: Extending Xtext’s Generator Capabilities

The results of MLR in Stage 1 showed that textual notation is a popular type of notation. However, there are still modeling tools that do not support textual notations, including EAST-ADL. We used EAST-ADL as a case language and improved its blended modeling capabilities by developing textual notations for it. EAST-ADL is a language based on a large meta-model. The ground fact that Xtext can generate a textual grammar directly from the meta-model allows us to avoid designing a grammar from scratch that involves a large number of domain concepts. However, due to limitations of Xtext’s default generator capabilities, part of the common features of modern editors (e.g., template proposals) are not supported by DSLs based on Xtext out-of-the-box. Therefore in this stage, we take EATXT as an example to describe how language engineers can extend Xtext’s generator capabilities according to their own needs.

Our research methodology was based on *design science* [33] and consisted of four iterations. We extended one capability of the Xtext generator in each of



the first three iterations and improved the scoping feature in the last iteration. We added new features incrementally, i.e., for each iteration, we provide the editor with added features from that iteration to our industrial partners and receive feedback from them.

The work is limited in the scope of using Xtext to create textual languages for DSLs. Xtext generates Xtext artifacts such as parser from the grammar to build the editor. The generation process is controlled by a workflow for the Modeling Workflow Engine (MWE2) [34]. Xtext provides a language generator that can be customized and extended with custom fragments. A fragment generates code based on the generator’s configuration, the grammar, and the corresponding meta-model. Xtext out-of-the-box provides a number of such fragments, which can be extended or replaced. These fragments add a number of features to the generated editors. We use Xtext’s ability to change the standard configuration to add custom fragments that provide better formatting, content-assist, and template proposals. These custom fragments are written in Xtend [35].

For the scoping feature, our approach is to generate a cross-reference lookup map from the plug-in’s activator. This generation traverses the metamodel exactly once with a complexity of  $O(n)$ . This lookup map contains the corresponding type of the cross-reference target for any source context type, and we compute it by iterating over all cross-references in the metamodel. We generate the lookup map during the first activation of the plugin. After that, the `scoping/(language name)ScopeProvider.java` accesses it via an interface but does not need to perform the same computation on every cross-reference content-assist keystroke. The lookup map is implemented as a Java `HashMap` whose `get()` method has a complexity of  $O(1)$  in most cases.

### 1.2.3 Stage 3: Python-Style Prototyping

As mentioned previously, when we explored and improved blended modeling technologies based on the case language EAST-ADL, we identified a problem with the grammar generated from the meta-model, i.e., the generated grammar was not concise and user-friendly. Our industrial partner provided some requirements of appearance for the language. For example, they requested that curly braces and keywords be reduced in the grammar to make the language more concise. Python is a language renowned for conciseness and provides a very clean coding style, and it is considered easy to learn [36]. Therefore, in this stage, we proposed a method that turns a DSL with a generated grammar into a Python-style language.

Our methodology at this stage was based on *constructive research* [37]. First, we took a small architecture description language DemoADL as a case language, and generated a grammar from the meta-model of it and wrote example code that conforms to the generated grammar. We then wrote pseudocode expressing the same content in Python style. We compared the two pieces of source codes and summarized the gaps between languages in those two styles. To address these gaps, we proposed a series of steps for adapting the text of the grammar and developed a script to semi-automate these adaptation steps.

To evaluate the usability and generality of the proposed method, we apply it to two other DSLs, i.e., Xenia and ACME. With the help of the script, we

completed the adaptation of the generated grammars for these two languages. We observe whether the adapted grammar is with a Python-style feature, i.e., using spaces and indents to express hierarchy. For each of the two languages, we successively compared the adapted grammar with the generated grammar, and compared the adapted grammar with the expert-created grammar, to observe the improvement of conciseness, and being as compact as the expert-created grammar.

### 1.2.4 Stage 4: Generalize Grammar Optimization Approach

In the last stage, we learned that some generalization (for the case of Python styles) is possible, which can be re-applied to other languages. Therefore, in this stage, we aimed to develop a more complete system that can adapt the generated grammar, which is rule-based and can be applied to different styles of languages. Also, our work on generating infrastructure in stage 2 inspired our ideas for generating adaptations. Therefore, the proposed rule-based system in this stage will include rule configurations that support the generation of grammar adaptations in the evolved version. In this section, we will introduce the methodology we adopted in **Stage 4**, including the extraction of candidate optimization rules, two iterations, and how to evaluate the proposed methodology.

We selected seven case languages, i.e., ATL<sup>1</sup>, Bibtex<sup>2</sup>, DOT<sup>3</sup>, SML<sup>4</sup>, Spectra<sup>5</sup>, Xcore<sup>6</sup>, and Xenia<sup>7</sup>, and obtained their meta-models and expert-created grammars. To smoothly generate grammar from the meta-model, we slightly process some meta-models, e.g., adding values to the namespace `URI` and `prefix` in Bibtex. We took two iterations to analyze the grammars of these languages and extracted candidate grammar optimization rules from the analysis. To reduce the complexity of the analysis, we excluded OCL expression language parts from the meta-models and grammars of both ATL and SML.

Regarding grammar analysis, we drew on ideas from *design science*. We analyzed different languages in different iterations and incrementally added extracted candidate optimization rules. We analyzed four of the seven languages in the first iteration. This analysis identifies differences between the generated grammar and the expert-created grammar, and we extracted candidate optimization rules from this analysis. These candidate rules can optimize the generated grammar, and the optimized grammar produces the same language as the expert-created grammar. We obtained a set of optimization rules by excluding the duplicate candidate rules. In the second iteration, we applied the optimization rules extracted from the first iteration to optimize the generated grammars for the other three languages and analyzed any remaining differences between the optimized grammars and the expert-created grammars. If there are differences, we derived more optimization rules. We developed the grammar

---

<sup>1</sup><https://eclipse.dev/atl/>

<sup>2</sup><https://www.bibtex.com/>

<sup>3</sup><https://graphviz.org/doc/info/lang.html>

<sup>4</sup><http://scenariotools.org/scenario-modeling-language/>

<sup>5</sup><http://smlab.cs.tau.ac.il/syntech/spectra/index.html>

<sup>6</sup><https://wiki.eclipse.org/Xcore>

<sup>7</sup><https://github.com/rodchenk/xenia>

optimization tool GrammarOptimizer. In this development, we implemented the optimization rules.

We evaluated the method proposed in this stage (i.e., GrammarOptimizer) from two aspects. On the one hand, we applied it to the seven case languages to evaluate its usability and generalization. For each language, we configured the GrammarOptimizer to optimize the grammar generated by that language, aiming to adapt it to a grammar that produces the same language as the expert-created grammar. On the other hand, we applied GrammarOptimizer to multiple versions of two languages (i.e., EAST-ADL and QVTo<sup>8</sup>) to evaluate its support for language evolution. For the first version of each language (e.g., QVTo's V1.0), we configured GrammarOptimizer to optimize the generated grammar, and the optimized grammar can produce the same language as the expert-created grammar. We reused these optimization rule configurations so that they optimize the generated grammar of evolved versions (e.g., QVTo's V1.1). We then compared this optimized grammar and the expert-created grammar and modified the configurations based on the difference(s). The goal was that GrammarOptimizer could completely optimize the grammar driven by the modified configurations, i.e., the optimized grammar could produce the same language as the expert-created grammar. We evaluated GrammarOptimizer's support for language evolution by counting the number of modifications to optimization rule configurations in the evolved version.

## 1.3 Results and Evaluation

As mentioned before, the work of this thesis consists of four stages, and the four stages of work are highly connected. Our research results in the first stage comprehensively reported the state-of-the-art and practices of blended modeling tools, including the limitations of existing tools. This motivates us to explore and improve existing blended modeling techniques. As a start, we developed a textual language EATXT for the case language EAST-ADL in Stage 2. During this development, we used EATXT as a case to demonstrate how language engineers can extend the Xtext generator capabilities and its implementation according to their own needs. While designing EATXT's grammar, we identified another problem, i.e., the grammar generated from the meta-model was cumbersome and non-user-friendly. To this end, in stage 3 we proposed a semi-automated method for converting a language with a generated grammar into a Python-style language, thus improving its conciseness and user-friendliness. We learned from the results of Stage 3 that the generalization of some rules about grammar adaptation is possible, and we aimed to support the co-evolution of meta-models and grammars, which motivated us to propose a new solution in Stage 4, the solution is a rule-based system consisting of general optimization rules. It can support the optimization of different languages and support the co-evolution of meta-models and grammars. In this section, we will elaborate on the results of the four stages respectively.

---

<sup>8</sup><https://wiki.eclipse.org/QVTo>

### 1.3.1 Results and Evaluation in Stage 1

To understand the potential of existing modeling tools to support blended modeling, we conducted a GLR in **Stage 1** in which we reviewed nearly 5,000 academic papers and nearly 1,500 gray literature. Based on these, we identified 133 candidate modeling tools that involve blended modeling technologies and finally identified 26 of them as the most advanced and practical modeling tools that represent the current range of modeling tools. We investigated these 26 modeling tools for their support of various blended aspects, such as inconsistency tolerance, and then obtained many results.

The obtained results were divided into results on user-oriented characteristics and results on realization-oriented characteristics. From the perspective of user-oriented characteristics, the results showed that more than half of the 26 tools supported two modeling notations, and about one-third of the tools supported three modeling notations. Among them, Boston Professional [38] and TopBraid Composer [39] supported four modeling notations. Graphical notation and textual notations were the most available notations, i.e., all of the 26 tools supported graphic notations, and most of them supported textual notations. Usability aspects are generally difficult to measure, so we evaluated a tool's usability by evaluating whether it supports multi-notations visualization and provides seamless navigation between notations. The results showed that all 26 tools supported the visualization of two or more notations. Our evaluation of the navigation across notations was mainly divided into two points, i.e., whether the navigation between multiple notations is synchronized, and the speed of navigation between different notations. The results showed that more than half of the tools provided simultaneous navigation facilities, while the majority of the tools provided immediate navigation from one notation to another. Flexibility is the tolerance of inconsistent user-related embodiment at various levels of abstraction across the modeling stack and various modeling facilities. We considered three categories of flexibility, i.e., model, language, and persistence. The results showed that most of the 26 tools did not support deviations between different notations describing the same model, while the other six supported model-level flexibility. Second, most tools did not allow inconsistencies between the model and the language, and four tools allowed such inconsistencies. The tool Umple [40] supported both model-level and language-level flexibility. Third, most tools did not support persisting inconsistency models.

We elaborate on our findings related to the realization characteristics of sampled tools from the following three aspects, i.e., mapping and platforms, change propagation and traceability, and inconsistency management. The mapping between abstract syntax and notations is usually implemented in a parser- or projection-based fashion. In the parser-based approach, the user modifies the model through different notations and the parser generates an abstract syntax tree. However, in the projection method, the abstract syntax tree is modified directly. The results showed that 22 of the 26 tools implement parser-based editors, while four tools have projection tools. The platforms used by these 26 sampled tools are almost all Eclipse, and only mbeddr [41] is the only one of these tools based on MPS [42]. In addition, MagicDraw [43] supports multiple platforms. During the data extraction phase, we failed to obtain any

useful information in the categories “change propagation” and “traceability”. In the context of inconsistency management for modeling tools, a majority of the tools (58%) lack visualization for inconsistencies (Table 19). In terms of inconsistency management types, there are two fundamental approaches: prevention and allow-and-resolve. Half of the tools (50%) focus on prevention, while others either manage inconsistencies on the fly (42%) or on-demand (8%) (Table 20). When it comes to inconsistency management automation, 50% of the tools follow a preventive approach and do not offer inconsistency resolution, while the remaining 13 tools provide varying degrees of automation for resolving inconsistencies, with only two relying on manual resolution (Table 21).

### 1.3.2 Results and Evaluation in Stage 2

As we mentioned in Section 1.2.2, we developed a textual language (i.e., EATXT) for EAST-ADL as a starting point for exploring and improving blended modeling technology, and during this period demonstrated how language engineers could extend Xtext’s generator capabilities according to their own needs. In this section, we present the research results of Stage 2.

Firstly, in the case of EATXT, the generated template file contains 194 code templates with a total of more than 1,000 XML lines and covers all meta-classes of the metamodel and their mandatory sub-elements. Secondly, for each meta-class with the mandatory attribute, our approach generates a proposal provider that includes a corresponding overriding content-assist method which proposes a unique name. Overall, in the case of EATXT, the generated `EatxtProposalProvider` encompasses 188 such methods. Thirdly, for the formatter feature, we implemented the generator fragment `formatting2/EatxtFormatter2Fragment.xtend` as part of the plugin. This fragment is executed by the MWE2 workflow and automatically generates the formatter class `formatting2/EatxtFormatter.xtend` as part of the same plugin. The generated formatter class in the EATXT case encompasses 51 dispatch methods [44] and 141 calls of the formatting methods for the nested sub-elements. Fourthly, we generated a cross-reference lookup map in the activator of the plugin (i.e., `org.bumble.eatxt` in the case of EATXT). In the EATXT case, the map encompasses the source context meta-classes and corresponding target metaclasses for 261 cross-references of the EAST-ADL metamodel.

### 1.3.3 Results and Evaluation in Stage 3

In Stage 2, we developed a textual language (i.e., EATXT) for EAST-ADL. When designing the grammar for EATXT, we identified the problem that the grammar generated from the metamodel was usually cumbersome and not user-friendly. To improve the conciseness and user-friendliness of the language with a generated grammar, in this stage, we proposed a method that can transform the language into a Python-style language. In this section, we present the results and our evaluation in Stage 3.

Firstly, we compared the program conforming to the generated grammar and the program in Python style, and observed that the program conforming to the generated grammar has issues in the following aspects: 1) inappropriate positioning of identifiers, 2) heavy separation of code blocks; 3) duplicate

keywords, and 4) nested curly braces. In this regard, we proposed a method to address these problems, which consists of steps that directly modify the text of grammar definition. The steps are: 1) introduce the white-space-awareness feature and remove curly braces, 2) reposition the identifier, 3) remove commas, and 4) refine keywords (especially remove duplicate keywords). As stated in Section 1.2.3, we semi-automated these adaptation steps by developing a script.

We applied the proposed method along with the script to two other DSLs, i.e., Xenia and ACME. For each of them, we compared the adapted grammar to the generated grammar and compared the adapted grammar to the expert-created grammar. The comparison showed that the adapted grammar was more concise and user-friendly than the generated grammar, and like Python, the program used whitespace and indents to express hierarchy. Moreover, the adapted grammar was closer to the expert-created grammar in terms of compactness. Our case languages Xenia and ACME were different languages, which showed that the proposed method and its script could be adapted to different DSLs. Language engineers could use it to quickly reach a Python-like grammar, which could then be used as a basis for further refinement of the grammar.

### 1.3.4 Results and Evaluation in Stage 4

In Stage 3, we proposed a method that makes the grammar of a language more concise and user-friendly by adapting the text of the generated grammar. We learned that some generalization (for the case of Python style) is possible, which can be re-applied to other languages. Therefore, we provide a solution (i.e. GrammarOptimizer) in stage 4. GrammarOptimizer provides general grammar optimization rules, can support the adaptation of different styles of languages, and supports the co-evolution of meta-models and grammars. We present the results of stage 4 in this section.

Before providing the solution GrammarOptimizer, we completed the following two works. First, through two iterations of comparative analysis of generated grammars and expert-created grammars for seven case languages, we extracted 56 general grammar optimization rules. Secondly, we developed an Eclipse-based grammar optimization tool, i.e., GrammarOptimizer. In this development, we implemented 56 general grammar optimization rules.

To evaluate the usability and generalizability of the proposed method, we applied GrammarOptimizer to the seven case languages, i.e. we used it to optimize the generated grammars of these case languages. The results showed that the generated grammars of all DSLs except Spectra can be fully optimized by GrammarOptimizer until they produce languages as same as the expert-created grammars, i.e., the optimized grammar produces the same language as the expert-created grammar. For Spectra, we were able to use GrammarOptimizer to optimize it to be very close to the expert-created grammar of Spectra, i.e., 96.30% (54/56) of the grammar rules could be optimized to be equivalent to the corresponding grammar rules in the expert-created grammar. This was a limitation of this method, which will explained in detail in Chapter 5.

To evaluate the proposed approach’s support for language evolution, we applied GrammarOptimizer to two versions of EAST-ADL and four versions

of QVTo. The grammar generated by the simplified version of EAST-ADL contains 755 lines of text. We configured 22 optimization rule configurations which completed the optimization of the generated grammar. The grammar generated by the full version 2.2 of EAST-ADL has 2839 lines of text. We only modified the configuration of 10 optimization rules to complete the optimization of the generated grammar of the full version. Similarly, for the 1026 lines of text in the generated grammar of version 1.0 of QVTo, we configured 733 grammar optimization rule configurations to complete the optimization of the generated grammar. However, facing the generated grammars of QVTo versions 1.1, 1.2, and 1.3 with similar amounts of text, we only modified two, zero, and one optimization rule configurations respectively to complete the optimization of the generated grammars respectively.

## 1.4 Answers to the RQs

We will answer all the RQs in this section.

**RQ1:** What are the user-oriented characteristics of modeling tools most suitable for supporting blended modeling?

From a user-oriented perspective, results indicated that over half of the 26 tools supported two modeling notations, while about one-third supported three. Notably, Boston Professional and TopBraid Composer each accommodated four modeling notations. Graphical and textual notations were the most common, with all 26 tools offering graphical notations and the majority providing textual options. Assessing usability, a generally challenging task, involved evaluating whether tools support multi-notations visualization and enable seamless navigation between notations. All 26 tools facilitated the visualization of two or more notations. Evaluating cross-notation navigation mainly considered synchronization and speed; more than half of the tools allowed simultaneous navigation, with most providing immediate transitions between notations. Flexibility, pertaining to inconsistent user-related aspects across modeling levels and facilities, was divided into three categories: model, language, and persistence. Most tools did not support deviations between different notations describing the same model; only six offered model-level flexibility. For language and model inconsistencies, the majority of tools didn't allow them, except for four. Notably, Umple supported both model-level and language-level flexibility. Concerning persisting inconsistency models, most tools did not offer support for this. For more details, please refer to Section 2.4.2 in Chapter 2.

**RQ2:** What are the realization-oriented characteristics of modeling tools most suitable for supporting blended modeling? From a realization-oriented perspective, there are three aspects: mapping and platforms, change propagation and traceability, and inconsistency management. The results showed that most sampled tools (22 out of 26) use parser-based editors while four employ projection tools. Eclipse is the primary platform for these tools, except for mbeddr (based on MPS), and MagicDraw supports multiple platforms. In terms of inconsistency management, the majority of tools (58%) lack visualization. They mainly fall into two categories: prevention-focused (50%) and real-time/on-demand resolution (42% and 8%). Automation-wise, 50% follow a preventive approach, while 13 offer varying levels of automation for

inconsistency resolution, with only 2 relying on manual resolution. For more details, please refer to Section 2.4.3 in Chapter 2.

**RQ3:** How can we build a solution to adapt generated grammars to produce the same language as available expert-created grammars?

The result and evaluations showed that the solution (i.e., GrammarOptimizer) developed in **Stage 4** could adapt the generated grammar to produce the same language as an expert-created grammar. GrammarOptimizer consists of grammar optimization rules, which can be applied to different grammar adaptation needs. We extracted grammar optimization rules from seven different case languages to maximize their usability and generalization. For the grammar comparison, we used a manual comparison to confirm whether the adapted grammar (i.e., the optimized grammar) and the expert-created grammar produce the same language. For example, for the language DOT, in the expert-created EBNF grammar, the grammar rule `node_id` does not contain any curly braces, so in the optimized grammar (in Xtext), the corresponding grammar rule `NodeId` should not contain any curly braces either.

**RQ4:** Can our solution support the co-evolution of generated grammars when the meta-model evolves?

When using GrammarOptimizer to optimize a grammar, language engineers need to configure the grammar optimization rules in the form of configurations. These configurations drive GrammarOptimizer which rules to apply to adapt the grammar. The language engineers reuse these configurations when the language evolves and new grammar is generated from the evolved meta-model. They only need to adjust the configurations accordingly for the changes between meta-models to execute the optimization on the new version of the generated grammar, thereby supporting the co-evolution of the meta-model and grammar.

## 1.5 Threats to Validity

In this section, we will discuss threats to both internal validity and external validity. There will be also additional threats to validity discussed in the subsequent chapters.

### 1.5.1 External Validity

As mentioned in the methodology section, we developed a textual language called EATXT for EAST-ADL. We used EATXT as a case to illustrate how language engineers can extend Xtext generators' capabilities and its implementation according to their own needs, which was the completed work in Paper B. There is a potential threat, i.e. whether the method proposed in Paper B is also applied to other languages. First, we have explicitly stated in Paper B that the discussed approach is for the Xtext editor. Xtext provides a mechanism for extending functionality, and our method demonstrates how to use this extension mechanism with EATXT as an example. Although EATXT as a language has its metamodel distinct from other languages. However, this extension mechanism is generic and is not limited to a specific metamodel.



### 1.5.2 Internal Validity

In the evaluation part of Stage 4, the number of optimization rule configurations required to optimize the grammar generated by different versions of QVTo differs very little, i.e., the maximum difference is two rule configurations. This data serves as one of the evidences that our solution GrammarOptimizer supports language evolution. However, the number of rule configurations required to optimize the generated grammar of any version of QVTo exceeds 700, which threatens the causal because the effort required to configure GrammarOptimizer does not reflect any less effort than manually adapting the grammar. However, we argue that it is more efficient to configure GrammarOptimizer once than to manually rewrite grammar rules every time the language changes – under the assumption that the configuration can be reused for new versions of the grammar. In that case, the effort invested in configuring GrammarOptimizer would quickly pay off when a language is going through changes, e.g., while rapidly prototyping modifications or when the language is evolving. We evaluated this assumption in stage 4. Furthermore, the effort required to configure optimization rules for optimizing the generated grammar can technically be reduced, one solution being to extract the rule configurations by comparing the generated grammar and the target grammar rather than manually writing the configuration from scratch [45].

## 1.6 Summary of Contributions

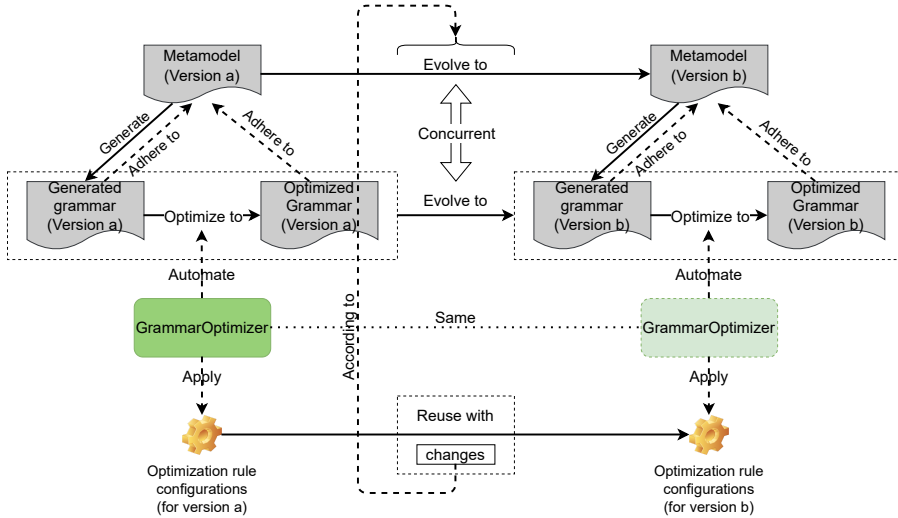


Figure 1.3: Schematic diagram of the proposed solution in **Stage 4** to support the co-evolution of meta-models and grammars.

In this thesis, there is a clear contribution in each of our four stages of work.

**The state-of-the-art of blended modeling tools.** First, prior to our study of Stage 1, a clear overview of the state of the art and practices in modeling

tools that support blended modeling was missing. Our study of Stage 1 filled this gap, including indicating limitations of current tools and opportunities for future research or industrial practice.

**Extension of Xtext’s generator capabilities.** In Stage 1, we comprehensively reported on the state-of-the-art and practices of blended modeling tools. We learned from this report that textual notations are a very popular type of notation. However, not all modeling tools support textual notations, including EAST-ADL. In Stage 2, we developed textual notations for EAST-ADL to explore and improve blended modeling techniques. We identified issues with the limitations of Xtext’s generator capabilities, therefore we used EAST-ADL as a case to explore and show how to extend it and its implementation. Our contribution in Stage 2 is the method of how language engineers can extend Xtext’s generator capabilities and its implementation according to their own needs. Language engineers can get a reference from our specific implementation on the EATXT case.

**Python-style prototyping.** In Stage 2, we developed a textual language (i.e., EATXT) for EAST-ADL to explore and improve blended modeling techniques. When designing the grammar for EATXT, we identified the problem that the generated grammar was cumbersome and user-unfriendly. Our contribution in Stage 3 is that we proposed a general method for converting a language with a generated grammar into a Python-style language, including providing a script that can semi-automate the execution of this method.

**Tool GrammarOptimizer.** Fourth, manually optimizing a grammar generated from a metamodel can be cumbersome and time-consuming, and as the language evolves, manual improvements made in the generated grammar of the previous version of the language can not be replayed in the generated grammar of the evolved version. This causes language engineers to have to perform similar optimizations on the generated grammar once again. Our work in **Stage 4** provided a solution, i.e., the tool GrammarOptimizer, which includes 60 general optimization rules (four added during the evaluation). By configuring these optimization rules, the solution can be applied to different languages for rapid prototyping and co-evolution of meta-models and grammars.

Figure 1.3 depicts how GrammarOptimizer supports the co-evolution of meta-model and grammar from version a to version b. By configuring GrammarOptimizer, language engineers can automate the optimization of the generated grammar of version a. In the same version, generated grammar and optimized grammar adhere to the same meta-model. As the meta-model evolves, language engineers regenerate the grammar from the meta-model (of version b). To optimize the generated grammar (of version b), the language engineers reuse the configurations of version a to replay the previous optimization in the generated grammar (of version b). In addition, language engineers adjust configurations based on differences between versions to optimize the changed parts.

## 1.7 Conclusion and Future Work

In this thesis, we comprehensively report on the challenges and limitations of modeling tools that support blended modeling, and opportunities for improving them. Our report helps tool providers identify the limitations of their tools in supporting blended modeling, and researchers can use this work to better contextualize their research and better position their work in terms of applicability. To contribute to the exploration and improvement of blended modeling technology, in stage 2, this thesis takes EAST-ADL and its textual language EATXT as an example to demonstrate how language engineers can extend the Xtext generator capabilities and its implementation according to their needs. Users and our peers who use Xtext to develop DSLs using the MDE approach will benefit from this. In stage 3, this thesis proposes a semi-automated method that can transform the language with a generated grammar into a Python-style language. Language engineers can apply this method to quickly improve the conciseness and user-friendliness of the language. We provide a systematic rule-based solution in Stage 4 that can generate adaptations in evolved versions' generated grammar of the evolving language, enabling semi-automated co-evolution of metamodels and grammars.

In future work, we plan to expand GrammarOptimizer into a more mature language workbench, supporting advanced features such as automatic extraction of configuration, a “what you see is what you get” view of the optimization of the grammar, optional language style library, and co-evolution of model instances and grammar [46]. We will also explore integration with workflows that generate graphical editors for blended modeling, which will further improve existing blended modeling techniques.



## Chapter 2

# Paper A

Blended Modeling in Commercial and Open-source Model-Driven Software Engineering Tools: A Systematic Study

I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J. Steghöfer, R. Hebig

*Software and Systems Modeling (SoSyM)*, 2023, 22(1), pp. 415-447.



## Abstract

Blended modeling aims to improve the user experience of modeling activities by prioritizing the seamless interaction with models through multiple notations over the consistency of the models. Inconsistency tolerance, thus, becomes an important aspect in such settings. To understand the potential of current commercial and open-source modeling tools to support blended modeling, we have designed and carried out a systematic study. We identify challenges and opportunities in the tooling aspect of blended modeling. Specifically, we investigate the user-facing and implementation-related characteristics of existing modeling tools that already support multiple types of notations and map their support for other blended aspects, such as inconsistency tolerance, and elevated user experience. For the sake of completeness, we have conducted a multivocal study, encompassing an academic review, and grey literature review. We have reviewed nearly 5,000 academic papers and nearly 1,500 entries of grey literature. We have identified 133 candidate tools, and eventually selected 26 of them to represent the current spectrum of modeling tools.

## 2.1 Introduction

Model-driven engineering (MDE) advocates modeling the engineered system at high levels of abstraction before it gets realized. The resulting models serve crucial roles in ensuring the appropriateness (e.g., correctness, safety, optimality) of the system. To keep the cognitive flow of modeling effective and efficient, stakeholders shall be equipped with proper formalisms, notations, and supporting computer-aided mechanisms. This is especially important in the design of modern systems, as their complexity has been increasing exponentially over the past years [47]. Modeling does not remove complexity from the engineering process, but rather, it replaces the accidental complexity of complex systems with essential complexity that is easier to manage [48]. Nonetheless, as a consequence of the increasing complexity of modern systems, modeling itself is becoming more complex.

In this paper, we focus on a specific manifestation of this added complexity stemming from the need for an orchestrated ensemble of modeling notations, aiming to enable seamless interaction with models through any of the notations. Such a need has been reported in multiple academic [49] and industrial domains, e.g., automotive [50], avionics [51], cyber-physical systems [52], and product lines [53]. In such an approach, user experience may also be (temporarily) prioritized over the correctness of the described system, in an effort to enable a smooth process of expressing the stakeholder's cognitive models in terms of the modeling language. This approach is referred to as *blended modeling* [54].

### 2.1.1 What is blended modeling?

Blended modeling was first introduced by Ciccozzi et al. [11] as follows:

*Blended modeling is the activity of interacting seamlessly with a single model (i.e., abstract syntax) through multiple notations (i.e., concrete syntaxes), allowing a certain degree of temporary inconsistencies.*

That is, blended modeling is characterized by the following three features.

**Multiple notations.** This is not to be confused with multiple *languages*. In our terminology, a language is composed of (i) a metamodel (abstract syntax), and (ii) a set of notations (concrete syntax). Blended modeling does not impose different metamodels.

**Seamless interaction.** Different notations have to be carefully integrated and orchestrated to allow for using the most appropriate notation for specific modeling tasks. This requires intuitive navigation between notations, proper change propagation between them, and in many cases, traceability.

**Flexible consistency management.** This aspect entails both vertical inconsistencies [55] (e.g., inconsistencies between the instance model and its metamodel); and horizontal inconsistencies (e.g., inconsistencies between two notations used to manipulate instances of the same metamodel).



### 2.1.2 What is *not* blended modeling?

**Multi-view modeling is not blended modeling.** As shown in Fig. 2.1, Multi-View Modeling (MVM) [12] and blended modeling share the trait of *multi-notation*. The main differences are, that (i) MVM further assumes *multiple languages*, while (ii) blended modeling assumes relaxed consistency rules instead. These differences stem from the different aims of the two approaches. MVM is concerned with constructing the appropriate views for stakeholders with varying backgrounds. Blended modeling focuses on the elevated UX with respect to an ensemble of notations, assuming a single underlying model. Prior work has reported challenges in relaxed consistency in multi-language settings such as MVM [56]. Blended modeling enables relaxed consistency by restricting the number of languages to one.

For example, the SCADE<sup>1</sup> tool suite provides the user with different languages for different purposes within the same model development environment. These languages facilitate multi-view modeling of the overall system and necessitate different abstract syntaxes. Therefore, working with SCADE cannot be considered blended modeling.

**Multi-paradigm modeling is not blended modeling.** In addition to assuming multiple languages, Multi-Paradigm Modeling (MPM) [13] further assumes potentially different semantics behind the languages, giving rise to *multi-formalism* (Fig. 2.1). This added complexity positions MPM even further from blended modeling, and vastly exacerbates consistency management, as reported in prior work [57].

For example, Matlab/Simulink is a typical combination of formalisms for system design, in which the overall system is graphically designed in Simulink<sup>2</sup>, which follows causal block diagrams (CBD) semantics; and the low-level functions in the system are textually described in Matlab<sup>3</sup>, which relies on matrix semantics for complex computations. While some level of navigation is provided between the two formalisms within the Matlab modeling and development environment, relaxed consistency is completely missing. Therefore, working with Matlab/Simulink cannot be considered blended modeling.

### 2.1.3 Motivation and aim

Blended modeling is an emerging new concept, thus, a map of current commercial and open-source tools is needed to properly position it in the research-and-development landscape.

In this article, we report the design, execution, and results of our mapping study on tools that are prime candidates to support blended modeling. Our study shows that these are typically tools with multiple notations for a single underlying abstract syntax, but they lack proper inconsistency tolerance mechanisms or fail to leverage such features for an improved user experience. The aim of our study was to identify, classify, and analyze (i) the user-oriented, and (ii) the realization-oriented characteristics of these tools. To infer this

<sup>1</sup><https://www.ansys.com/products/embedded-software/ansys-scade-suite>

<sup>2</sup><https://se.mathworks.com/products/simulink.html>

<sup>3</sup><https://www.mathworks.com/products/matlab.html>

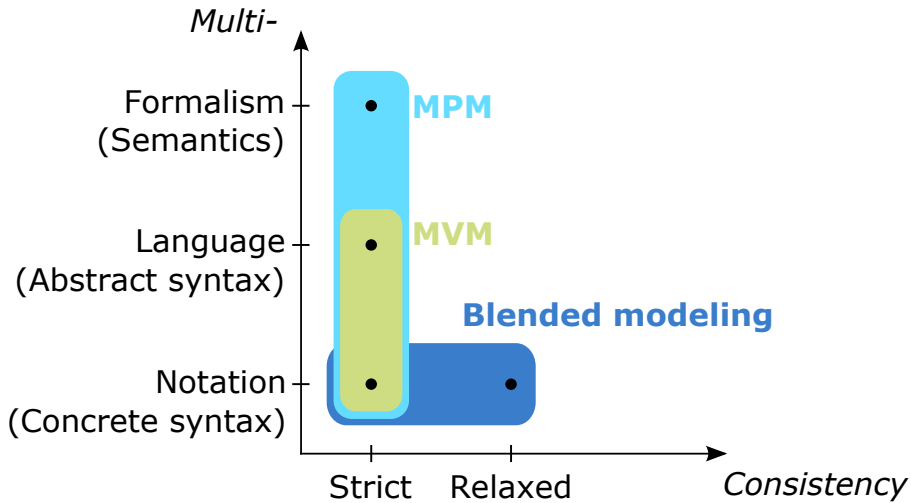


Figure 2.1: Blended modeling in the context of MVM and MPM.

information while ensuring external validity, we surveyed both the academic (peer-reviewed) literature and the grey literature [58], consisting of websites, blogs, and user manuals of engineering tools, following the guidelines for multi-vocal reviews in software engineering [59]. To be able to treat both types of literature uniformly, we made *tools* the primary units of our study, instead of papers. This is motivated by the inherent limitations of grey literature in terms of providing high-fidelity research data. Websites and end-user documentation do not aim to provide such information. We formulated a surveying protocol based on well-established guidelines and we have meticulously followed this protocol in the execution of our study. Eventually, we screened 4,975 academic papers and included 77 of them. Additionally, 1,494 grey literature entries were processed. Out of the academic papers, 68 distinct tools were extracted and complemented by 68 tools extracted from the grey literature. After removing duplicates, the set of 133 tools was reviewed according to the tool selection criteria (see Section 2.3) and we eventually identified 26 tools to be analyzed in detail. Although this list of tools is not exhaustive, we are reasonably confident about its representativeness of the domain of interest.

The results of this study provide a clear overview of the state of the art and practice of the domain of modeling tools closest to blended modeling. The tool characteristics reported in this paper can be particularly useful for tool providers in identifying the limitations of their tools in supporting blended modeling. Researchers of the three main dimensions of blended modeling (multi-notation, seamless integration of languages, inconsistency tolerance) could use this work to better contextualize their research, and position their work better in terms of applicability.

### 2.1.4 Structure

The rest of this paper is structured as follows. First, in Section 5.2, we give an overview of the background concepts of blended modeling and review the

related work. In Section 2.3, we define the methodological framework for carrying out this study. In Section 2.4, we elaborate on the findings of this study, in particular on the results pertaining to the research questions of this study: the user-oriented and realization-oriented characteristics of the tools of interest. In Section 2.5, we provide orthogonal insights on the aggregated data. We discuss the results in Section 5.8, and the threats to validity in Section 5.8.1. Finally, we summarize this paper by drawing the conclusions in Section 2.8.

## 2.2 Background

In this section, we provide the foundational background concepts to contextualize our study. More specifically, we describe the core ingredients of blended modeling: multiple notations (Section 2.2.1), seamless interaction (Section 2.2.2), and flexibility in managing inconsistencies (Section 2.2.3). Additionally, we discuss the secondary literature related to our study (Section 2.2.4).

### 2.2.1 Multiple notations

Interacting with the (abstract) model through multiple notations (concrete syntaxes) is one of the three distinguishing features of blended modeling. A vast body of knowledge on the topic has been produced, especially in relation to multi-view modeling, and multi-paradigm modeling.

#### 2.2.1.1 Multi-view modeling

Multi-view modeling (MVM) tackles the complexity of modeling heterogeneous systems by decomposing the models into multiple views, that are concerned with specific aspects of the system [12]. The ISO/IEC/IEEE 42010:2011 standard [60] defines a view as a set of concerns of specific stakeholders and viewpoints as the specification of conventions utilized to construct a view. The five mutually non-exclusive enabling mechanisms of multi-view modeling are (i) *synthetic*, where views are specified by means of different domain specific modeling languages and synthesized together; (ii) *separate*, a stricter version of synthetic, where synthesis does not take place; (iii) *projective*, where a single metamodel allows for the definition of multiple virtual views; (iv) *orthographic*, where views are orthographic projections of a single underlying model; or (v) *hybrid*, where views represent only a portion of the common metamodel [61]. MVM has been shown to be an effective approach in several complex domains, such as cyber-physical systems [47], and cloud-based software-intensive systems [62]. The principles of MVM are similar to those of blended modeling. However, its goal is different. While MVM is oriented towards the identification of multiple views and the management of consistency between them, blended modeling focuses on enabling an elevated user experience while working with multiple notations at the same time.

#### 2.2.1.2 Multi-Paradigm Modeling

Multi-paradigm modeling (MPM) advocates modeling every aspect of the system explicitly, at the most appropriate level of abstraction, and using the

most appropriate formalism [13, 63]. As such, MPM facilitates the modeling of complex systems that could not be described through a single formalism and at a common level of abstraction due to the heterogeneity of the different components. It combines three research areas: (i) meta-modeling used for the specification of formalisms, (ii) multi-formalism used for the coupling of models specified in different formalisms and their transformations, and (iii) model abstraction used for the relationships among models described in different formalisms [64]. The principles of MPM are similar to those of blended modeling, as both approaches promote employing a variety of notations to model the problem at hand. However, MPM achieves this by employing a variety of separate formalisms, i.e., multiple notations with possibly different semantics. Blended modeling assumes a single abstract syntax, and therefore, single semantics. This simplification allows for greater flexibility in terms of temporarily inconsistent designs.

## 2.2.2 Seamless interaction

Usability in terms of the ability to seamlessly interact with models through multiple different notations is one of the three distinguishing features of blended modeling. In this section, we review how state-of-the-art approaches typically support seamless interaction. We focus on UML tools here since they have received significant attention from research and tool providers of the software engineering domain in the past. We also mention examples for other modeling languages where appropriate.

### 2.2.2.1 Text-based modeling with graphical visualizations

Umple [40] is a modeling tool that supports the creation of UML models using both textual and graphical notations, where the synchronization between the two notations is automated and on the fly. However, the graphical editor does not offer full editing capabilities, and the existing editing capabilities are only available on class diagrams but not on state machines, composite structures, or feature diagrams. FXDiagram<sup>4</sup> is a JavaFX-based framework that can be integrated into Eclipse as well as IntelliJ IDEA. It supports the creation of graph diagrams (nodes and edges) and it is typically used for graphical visualization of textual DSLs but does not provide editing functions. MetaUML<sup>5</sup> is a GNU GPL library for typesetting UML diagrams, using a textual notation. This notation is used for rendering read-only graphical UML diagrams. PlantUML<sup>6</sup> is very similar but supports also non-UML diagrams. ZenUML<sup>7</sup> supports sequence diagrams and flowcharts, again defined using a textual notation that is translated into read-only graphical views. The generation of the sequence diagrams is automatic, as the conversion happens on the browser. Excalibur [65] is a tool that relies on Xtext for textual specification and Sirius for graphical views of the textual specification. The model elements are defined using Messir textual DSL and the generated graphical visualization is read-only. Chart

---

<sup>4</sup><https://jankoehnlein.github.io/FXDiagram>

<sup>5</sup><https://github.com/ogheorghies/MetaUML>

<sup>6</sup><https://plantuml.com>

<sup>7</sup><https://www.zenuml.com>

Mage<sup>8</sup> is a web-based tool that supports automatic and on-the-fly generation of sequence diagrams and flowcharts using a textual notation. DotUML<sup>9</sup> is a javascript application that supports the generation of a subset of UML diagrams (i.e., use-case, sequence, class, state, and deployment) from a textual notation. For all of the aforementioned tools, concrete syntaxes are predefined and not customizable, and the graphical notation is read-only, generated using the textual notation.

### 2.2.2.2 Mixed textual and graphical modeling

Addazi and Ciccozzi [54] present a proof-of-concept implementation for UML and UML profiles modeling using blended textual and graphical notations. The stack of technologies used includes Eclipse Modeling Framework (EMF)<sup>10</sup>, Xtext<sup>11</sup>, and Papyrus [66]. Their solution includes a single underlying abstract syntax, two notations (i.e., graphical and textual), and one single persistent resource that is the UML resource. This architecture enables synchronization by means of serialization/deserialization operations across Xtext and UML models. In addition, the authors conduct an experiment to demonstrate that their solution on blended modeling increases user performance compared to single notation modeling.

Maro et al. [10] introduce a solution that integrates graphical and textual editors for a specific UML profile-based DSL. Being that the graphical editor is already provided, this work focuses on obtaining the textual editor and switching between views (i.e., graphical and textual). To obtain the textual editor, the UML profile-based DSL is first transformed into an Ecore model using an ATL transformation, and then this Ecore model is consumed by the Xtext plugin to generate the textual editor. Switching between views is achieved by employing ATL transformations. Scheidgen [9] provides embedded textual editors for graphical editors as an add-on feature. For each selected model element that needs to be edited, the embedded textual editor creates an initial representation that can be changed by the user and using parsing operations, new edited model elements are created. However, the synchronization is on-demand as the changes in the underlying model are not carried out until they are committed by the user and the textual editor is closed.

Lazăr [67] makes use of the Eclipse modeling environment to integrate the existing UML tree-based editor with the textual editor for Alf language<sup>12</sup> and to create fUML<sup>13</sup> models. However, the synchronization is on-demand as the changes are carried out upon the occurrence of a save action by the user.

Charfi et al. [8] define a hybrid language that integrates textual and graphical notations in one concrete syntax. The contribution consists of a visual notation for the most used UML actions and an editor that supports the proposed notation. The hypothesis that the hybrid notation can perform better than the textual notation is backed by an experiment that takes into consideration the learnability of the hybrid notation, the prevented errors, and the circumstances

<sup>8</sup><http://chartmage.com/index.html>

<sup>9</sup><https://dotuml.com>

<sup>10</sup><https://www.eclipse.org/modeling/emf>

<sup>11</sup><https://www.eclipse.org/Xtext>

<sup>12</sup><https://www.omg.org/spec/ALF>

<sup>13</sup><https://www.omg.org/spec/FUML>

in which the hybrid notation is a better fit than the textual notation. However, this approach is restricted to UML actions only.

Van Rest et al. [68] implement an approach for the robust synchronization of graphical editors generated with the Graphical Modeling Framework (GMF)<sup>14</sup> and textual editors generated with Spoofox<sup>15</sup>. This approach allows error recovery during synchronization and preserves the textual and graphical layout in case of errors. However, layout preservation is not supported at all times, as during cut-paste operations, the elements and their associated layouts are deleted and then recreated, therefore losing the original layout.

### 2.2.2.3 Projectional editing

Projectional editing is an approach where the abstract syntax tree (AST) is modified directly upon every editing action and bypasses the stages of the parser-based approach, where the parser must first check the correctness of the syntactic aspects, and then construct the AST based on the changes in the notation [69]. This course of action allows the definition of multiple notations (e.g., tables, diagrams, formulas) that cannot be supported by parser-based approaches, and supports multiple views of the same program, simultaneously. Moreover, a considerable amount of the ambiguities caused during the parsing process are tackled. Projectional editing is a realization of the intentional programming paradigm [70], and as such, it encourages the combination of a variety of different notations. Some of the state-of-the-art language workbenches that adopted this principle for providing domain-specific tool engineers with efficient tools [71] are JetBrains MPS<sup>16</sup> and MelanEE<sup>17</sup>. However, even though they provide a greater amount of notations, their support for textual notations is limited compared to parser-based approaches, as it is only a projection that resembles text. In particular, no possibly inconsistent intermediate states are allowed, which consequently restricts the user accustomed to classical text editors and their corresponding free editing features.

## 2.2.3 Inconsistency management

Approaches, such as multi-view modeling (MVM) and multi-paradigm modeling (MPM) advocate modeling the engineered system using the most appropriate notations, formalisms, and abstractions. This allows multiple users to be involved in the modeling of the system, and thus, introduces parallelism, which is beneficial for the overall efficiency of the engineering endeavor. Parallelism, however, gives rise to inconsistencies between the design artifacts, compromising the ultimate correctness of the system. Inconsistency has been shown to be an effective heuristic for managing the ultimate correctness of the system [57]. Techniques, such as blended modeling, make use of this assertion by focusing on the early detection of inconsistencies [72] and establishing the proper tolerance mechanisms.

The notion of consistency models and their various alternatives have been well-researched already in early distributed systems. Lamport [73] is the

<sup>14</sup><https://www.eclipse.org/modeling/gmp>

<sup>15</sup><http://strategox.org/Spoofox>

<sup>16</sup><https://www.jetbrains.com/mps>

<sup>17</sup><http://www.melanee.org>

first to describe how multi-processor systems should be constructed to ensure proper execution of programs. His notion of *sequential consistency* allows a relaxation of the locking model by assuming a total order of modifications that distributed nodes are guaranteed to observe. Adve and Gharachorloo [74] describe various relaxations of the sequential consistency model, based on architectural choices on the hardware and software level. *Eventual consistency* has been suggested by Vogel et al. [75] to enable a weaker notion of consistency between distributed participants, by embracing that real consistency can never be achieved. In such settings, distributed participants are characterized by the BASE properties: basic availability, soft state, and eventual consistency. Lately, *strong eventual consistency* (SEC) has been suggested [76] to combine the liveness guarantees of eventual consistency with the safety guarantees of strong consistency. Conflict-free replicated data types [77] are the prime examples of their applications.

Inconsistencies are a well-researched area in software engineering [78], too. Consistency between models can be categorized into two orthogonal dimensions [79]: horizontal and vertical consistency; and syntactic and semantic consistency. Horizontal consistency is concerned with models on the same level of abstraction, whereas vertical consistency is defined between models on different levels of abstraction (typically in model-metamodel contexts) [80]. The majority of inconsistency management techniques rely on syntactic concepts, e.g. synchronization by bi-directional model transformations [81], triple-graph grammars [82], and by version control systems and related mechanisms [83, 84]. However, semantic techniques have been shown to be beneficial in heterogeneous engineering settings [55]. View consistency has been researched in the context of MVM, e.g., in the Vitruvius approach [85], which provides languages for consistency preservation, and defines a model-driven development process for enacting consistency rules.

Finkelstein et al. [86] suggest that inconsistencies are organic elements of any engineering process, and instead of simply removing them from the system, one should apply proper inconsistency management techniques [87]. Such inconsistency management techniques typically entail the activities of detecting, resolving, preventing, and tolerating inconsistencies [88]. Blended modeling heavily relies on the tolerance of inconsistencies. Balzer et al. [89] suggest augmenting inconsistency instances with a state. Inconsistency rules are first deconstructed into appearance and disappearance rules spanning a temporal interval; then, tolerance rules are put in place to trigger repair actions based on temporal constructs. Easterbrook et al. [90] propose a similar technique for temporal inconsistency tolerance in the context of MVM. Inconsistency tolerance is achieved via pairs of pre- and post-conditions relying on a user-defined consistency metric. David et al. [91] introduce various patterns of inconsistency tolerance for implementing such systems.

### 2.2.4 Related secondary literature

This paper reports on the first systematic study on blended modeling. There are, however, secondary studies close to our work that are similar in topic, but differ in terms of motivation and objectives, and are generally limited to a narrower scope.

Torres et al. [92] conduct a systematic literature review with the aim to identify a list of available tools to support model management and provide a categorization of these tools into (i) tools that can provide consistency checking on models of different domains, (ii) tools that can provide consistency checking on models of the same domain, and (iii) tools that do not provide any consistency checking. Furthermore, the authors identify the inconsistency types, strategies to keep the consistency between models of different domains, and the challenges to manage models of different domains. The information retrieved from the primary studies is also complemented with additional data sources (e.g., the official website of the tool). Our study focuses on a broader scope, especially multi-notation and seamless interaction. Torres et al. observe that 35% of their analyzed tools do not provide any consistency checking features, whereas in our study we observe that 64% of the analyzed tools do not support models inconsistencies. Moreover, Torres et al. identify different strategies that have been used to keep models consistent, e.g., by using standard file formats for the models, explicitly modeling dependencies among model elements, mapping model elements to a shared ontology, etc. Our study complements such results by highlighting which inconsistency management strategies involve a manual effort (like keeping a dependency matrix always up to date), a semi-automated procedure (e.g., by specifying a priori consistency constraints and checking them during development), or a completely automated one (e.g., via the automated application of inconsistency resolution procedures).

Iung et al. [93] conduct a systematic mapping study with the aim to identify tools, language workbenches, or frameworks for DSL development. The authors identify 59 tools and they use the feature model proposed by Erdweg et al. [3] for their comparison. The study focuses on the technologies/tools used for DSL development, their license types, the application domains, and the features of the DSL creation process that these tools support. 48 tools support only one notation (graphical or textual), seven tools support two notations (graphical and textual), two tools support three notations, and two tools support four notations. Our study focuses on a broader scope, by extending the set of features on which the comparison is based with features such as synchronization mechanisms, collaborative features, or conformance relaxation. We also contextualize our work on a broader timeline, while the authors focus on the period between 2012–2019. In line with the results of our study, Iung et al. observed that the notations that were more frequently used in combination are *textual* and *graphical*, with the tabular one complementing them. In [93] two language workbenches are identified as particularly relevant for blended modeling: (i) GEMOC Studio, which provides real-time bidirectional synchronization in their generated editors, and (ii) the Whole Platform, which allows language engineers to choose among four different types of notation (i.e., textual, graphical, tabular, and symbolic), and to visualize the different translations among them at the model level.

Franzago et al. [94] and David et al. [95] map the state-of-the-practice of collaborative model-based software engineering. The authors identify and classify collaborative MDSE approaches based on the different categories such as characteristics of the collaborative model editing environments, model versioning mechanisms, model repositories, support for communication and decision making, and more. Additionally, the authors identify limitations and challenges



with respect to the state of the art in collaborative MDSE approaches. Regarding model management, they provide a taxonomy for the management support of collaborative MDSE approaches, collaboration support, and communication support. This study covers some of the aspects that we cover in our systematic mapping study (e.g., conflict detection). However, while this study is mostly focused on the characteristics of the collaborative approaches, we aim toward a classification of tools based on a broader set of features such as synchronization mechanisms and their generation, or conformance relaxation. The results of Franzago et al. and David et al. for collaborative modeling that are confirmed in this study are about: (i) the types of notations, with graphical as the most supported one, followed by textual, (ii) the prevalence of custom/other modeling platforms with respect to Eclipse EMF, (iii) the growth of web-based approaches, (iv) the growth of preventive conflict management, and (v) the prevalence of mechanisms for (semi-)automatically resolving conflicts. We anticipate that 15 out of the 26 tools analyzed in this study support collaborative modeling, with the majority of tools providing off-line collaboration (i.e., a la Git), rather than real-time collaboration (i.e., a la Google Docs); this result is different for academia where, according to Franzago et al. and David et al., researchers focus primarily on real-time collaboration. Another difference with respect to the state of the art in collaborative modeling is that blended modeling tools are primarily parser-based, whereas collaborative modeling approaches tend to be equally distributed between parser-based and projectional approaches. Interestingly, while researchers are recently investigating more on eventual consistency for collaborative modeling [95], in our study we observe that blended modeling tools provide limited support for consistency tolerance that would allow deviations between different notations describing the same model.

Granada et al. [96] map model-based language workbenches that can be used to generate editors for visual DSLs and point out their features and functionalities. The authors identify eight language workbenches for the generation of editors for visual DSL. The features taken into consideration for their analysis are the following: scope, framework, the distinction between abstract and concrete syntax, abstract syntax, concrete syntax, editing capabilities, use of models, automation, usability, and methodological basis. The conclusions point out that the most complete commercial language workbenches are MetaEdit<sup>18</sup> and ObeoDesigner<sup>19</sup>, while the most complete open-source ones are Eugenia<sup>20</sup>, GMF<sup>21</sup>, Graphiti<sup>22</sup>, and Sirius<sup>23</sup>. Our study differs in scope, as we focus on tools that provide multiple notations, not only on tools that can be used to develop editors for a single visual DSL. Indeed, none of the tools identified by Granada et al. support the definition of more than one (visual) concrete syntax for the same abstract syntax; this means that language engineers willing to develop blended modeling environments should either use a dedicated language workbench for blended modeling or suitably combine the languages produced by two or more

<sup>18</sup><https://www.metacase.com/products.html>

<sup>19</sup><https://www.obeodesigner.com/en>

<sup>20</sup><https://www.eclipse.org/epsilon/doc/eugenia>

<sup>21</sup><https://www.eclipse.org/modeling/gmp>

<sup>22</sup><https://www.eclipse.org/graphiti>

<sup>23</sup><https://www.eclipse.org/sirius>

of the language workbenches mentioned above.

Do Nascimento et al. [97] perform a large-scale systematic mapping study on DSLs and their related tools. The tools are categorized into (i) tools for using DSLs, (ii) tools for creating DSLs, and (iii) language workbenches. Our study differs from this work, as we focus on DSLs tool comparison, while the authors provide a brief categorization of DSL tools, and do not go into the details of conducting a comparison of the technical features. It is interesting to note that Do Nascimento et al. observed that tool support for a single DSL is well-studied in the literature, but at that time (2012) there was little knowledge about how to support multiple DSLs and notations in a single modeling environment. They claim that supporting multiple DSLs and multiple notations is fundamental when describing large-scale industrial systems and that methods and tool support are needed for the success of multi-DSL development. Based on the results of our study and the ones on multi-notation modeling (see Section 2.2.1, we can confirm that in the last years, the MDE scientific community actively worked and contributed to filling this research gap.

There are additional studies related to our research that are not systematic in nature, but their takeaways are still relevant. Negm et al. [98] compare 14 language workbenches based on (i) structure (grammar-driven or model-driven), (ii) editor (parser-based or projectional), (iii) language notations (textual, tabular, symbols, or graphical), (iv) semantics (translational or interpretive), and (v) composability language aspects. However, this study is limited to language workbenches and does not cover aspects such as synchronization mechanisms and their generation, or collaborative features. Some of the results obtained by Negm et al. are relevant for blended modeling as well. Firstly, out of nine analyzed parser-based language workbenches, only one (i.e., Ensō) supports both textual and graphical concrete syntaxes; this capability is achieved by having a bidirectional mapping between tokens in the textual representation of the model and elements in the object graph. Moreover, all four considered projection-based language workbenches support multiple concrete syntaxes, with the *Whole* platform and MPS supporting four different syntaxes: textual, graphical, tabular, and symbolic. The main advantage of projection-based workbenches is that they can rely on a shared common representation of all modeling elements (e.g., the AST in MPS), whereas parser-based workbenches have a dedicated parser for each concrete syntax. One of the claimed advantages of parser-based language workbenches (especially the textual ones) is the flexibility with respect to the models' conformance; the textual representation of parser-based models can still be opened and inspected, whereas projectional editors work directly on the abstract representation of the model. Similarly, according to Negm et al., textual parser-based workbenches avoid tool lock-in since the modeler is not limited to using any specific editor and can be easily integrated with other tools.

Erdweg et al. [3] conduct a comparison study of 10 language workbenches participating in the 2013 edition of the Language Workbench Challenge (LWC). The comparison of the language workbenches is based on a feature model that includes: notation, semantics, editor support, validation, testing, and composability, where some of them support multiple notations (fully or partially). The conclusions state that no language workbench realizes all features. However, this study is limited to the language workbenches presented in LWC'13. For

what concerns blended modeling, the results obtained by Erdweg et al. are in line with the ones reported by Negm et al. [98], where projectional language workbenches are better supporting the combination of different concrete syntaxes, with Enzō and MPS again as the ones supporting all types of concrete syntaxes. Erdweg et al. also highlight the need for integrating “different notational styles”, which is at the core of blended modeling.

Merkle [99] conduct a comparison study of textual language workbenches categorizing them into pure text-based and projectional-based with a textual projection. The language workbenches compared in this study are Xtext<sup>24</sup>, TEF<sup>25</sup>, EMFText<sup>26</sup>, and MPS<sup>27</sup>. The language workbenches are compared based on workflow, abstract/concrete syntax, and editor. However, this study is limited to textual language workbenches, while our focus is on tools that provide multiple notations. In the study by Merkle, the only language workbench supporting a combination of concrete syntaxes is TEF (Textual Editing Framework<sup>28</sup>), an Eclipse-based language workbench focusing primarily on textual editors, but with the possibility of embedding them into other editors supporting other concrete syntaxes [9]. Internally, TEF follows the *background parsing* strategy for the textual concrete syntax, where textual models are always represented and edited as plain text, and their parsing is demanded by a background process. TEF also provides some basic form of blending, where modelers can bring up a textual editor from either a graphical or a tree-based editor (e.g., by opening a small overlay window); however, TEF-based modeling tools cannot be considered as blended since model updates the embedded textual editor is not seamlessly integrated into its host editor, and model updates are propagated on-demand to the host editor only when the modeler closes the textual editor.

## 2.3 Study design

The goal of this study is to characterize the state of the art and the state of the practice of modeling tools in relation to blended modeling. More specifically, we formulate such high-level goal by using the Goal-Question-Metric perspectives [100], shown in Table 2.1.

<i>Purpose</i>	Identify, classify, and analyze
<i>Issue</i>	the user-oriented and implementation-oriented characteristics of
<i>Object</i>	existing modeling tools
<i>Context</i>	in relation to the principles of blended modeling,
<i>Viewpoint</i>	from a researcher’s and practitioner’s point of view.

Table 2.1: Goal of this study.

<sup>24</sup><https://www.eclipse.org/Xtext>

<sup>25</sup><https://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html>

<sup>26</sup><https://github.com/DevBoost/EMFText>

<sup>27</sup><https://www.jetbrains.com/mps>

<sup>28</sup><http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html>

### 2.3.1 Process

This research was carried out by following the process shown in Figure 2.2. Our process can be divided into three main phases, all well-established in systematic secondary studies [101][102][103]: planning, conducting and documenting. In the following, we present the three phases of the process.

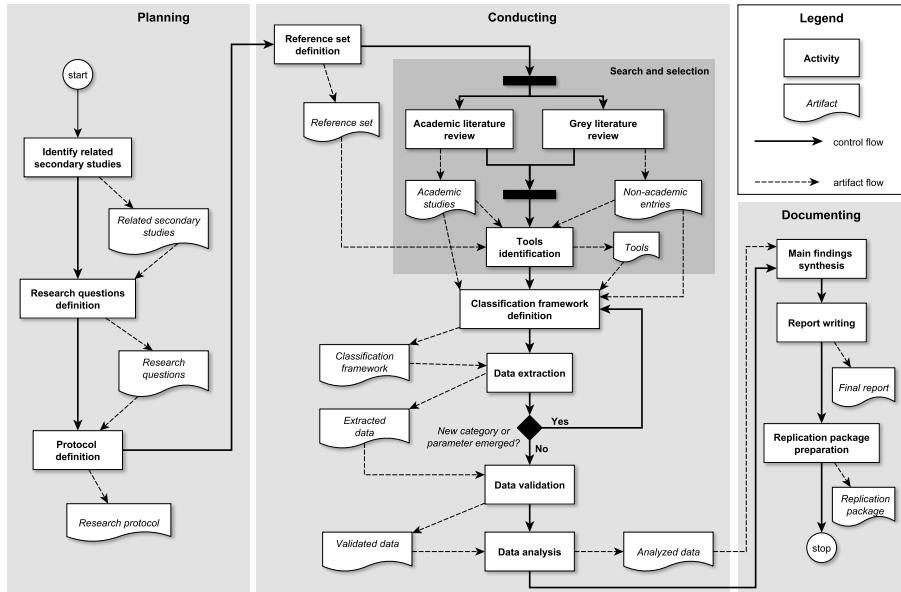


Figure 2.2: Overview of the whole review process

#### 2.3.1.1 Planning

This phase aims at defining the plan for carrying out all the activities of this study. More specifically, we first identified *related secondary studies*, i.e., surveys and literature reviews with a scope similar to the current review’s scope (Section 2.2.4). Subsequently, we formulated the *research questions* (Section 2.3.2), and compiled the *research protocol*.

The research protocol is a document reporting the methodological details of this study. Specifically, the research protocol contains a detailed description of all the steps we followed in the subsequent *Conducting* and *Documenting* phases. To mitigate potential threats to validity and any bias, the research protocol was defined *prior to* conducting the study, and it was reviewed by two experts. The experts were asked to provide feedback on the protocol, particularly on possible unidentified threats to validity, problems in the overall construction of the review, and the appropriateness of the proposed research protocol and final reports for the aim of this study. Both experts are well-established professors of Computer Science, with substantial experience in empirical research.

### 2.3.1.2 Conducting

In this phase, the mapping study is carried out according to the research protocol. More specifically, we carry out the following activities.

**Reference set definition.** The goal of this activity is to identify the modeling tools that could be part of the final set of modeling tools. This set will serve as a guideline for the subsequent steps of the study design, especially formulating the inclusion and exclusion criteria. The inclusion and exclusion criteria will be tested against this set, and thus, the reference set is subject to change until the criteria are not final. We identify the initial reference set based on (i) the modeling tools mentioned in related secondary studies (Section 2.2.4); (ii) the authors' experiences with tools partially supporting blended modeling (e.g., [11, 54]); (iii) searches in generic web search engines; and (iv) knowledge garnered from existing networks of experts, e.g., by accessing forums and mailing lists (e.g., the Eclipse EMF community forum<sup>29</sup>). The results of the subsequent *Search and selection* activity are eventually compared to the reference set for validation purposes. The eventual reference set is composed of the following tools: MagicDraw [43], Eclipse Papyrus [66], MetaEdit+<sup>30</sup>, Umple [40], and the Open Source AADL Tool Environment (OSATE) [104].

**Search and selection.** (Section 2.3.3) The goal of this activity is to identify as many (possibly blended) modeling tools as possible. Two parallel activities are carried out: the *Academic literature review*, and the *Grey literature review*. In both search activities, we perform a combination of automated search, manual search, and backward-forward snowballing [105]. These activities yield two types of artifacts: (i) *Academic studies* (e.g., articles published in scientific journals, and proceedings of scientific conferences) and (ii) *Non-academic entries* (e.g., blog posts, technical reports). Because the subject of this study are the *tools* these artifacts describe, both types of artifacts are screened for a specific *Tool* in the *Tools identification* activity. Here, we manually analyze all academic studies and non-academic entries and identify every modeling tool mentioned in their contents. Moreover, in this activity, we keep track of pointers and links referring to the relevant documentation about each tool (e.g., its official documentation, its wiki-based knowledge base, etc.).

**Classification framework definition.** (Section 2.3.4) The goal of this activity is to define the set of categories and their possible values to classify the identified modeling tools.

**Data extraction.** (Section 2.3.5) The goal of this activity is to collect relevant information about each modeling tool. In this activity, multiple researchers collaboratively (i) read the full text of the relevant documentation of each modeling tool, and (ii) populate the data extraction form with the collected data. Upon the emergence of a new category or new possible value in the domain of previously defined categories, the

<sup>29</sup>[https://www.eclipse.org/forums/index.php?t=thread&frm\\_id=108](https://www.eclipse.org/forums/index.php?t=thread&frm_id=108)

<sup>30</sup><https://www.metacase.com/products.html>

classification framework can be dynamically adapted. In such cases, the previously extracted data entries are updated in accordance with the new framework.

**Data validation.** (Section 2.3.6) To ensure the validity of the extracted data, the tool vendors and knowledgeable experts are contacted to review the data extracted in the previous step.

**Data analysis.** (Section 2.3.7) The goal of this activity is to analyze the extracted data in accordance with the research questions. The activity involves both quantitative and qualitative analyses.

### 2.3.1.3 Documenting

The main activities performed in this phase are: (i) a thorough elaboration on the data analyzed in the previous phase with the aim of discovering the main findings of the study; (ii) reporting the possible threats to validity, especially the ones identified during the definition of the review protocol; and (iii) producing the final report. The final report is evaluated by external reviewers and forms the basis of this article.

The complete *replication package* is available online<sup>31</sup> to allow independent researchers to replicate and verify our study, and to reuse our data for other purposes. The replication package includes the research protocol, the list of all academic and non-academic entries considered in the search and selection phase, the complete list of all identified tools, raw data, the scripts for data analysis, and the details on technical requirements.

## 2.3.2 Research questions

The research questions of this study are reported below.

**RQ1.** *What are the **user-oriented characteristics** of modeling tools most suitable for supporting blended modeling?*

Modeling tools are designed and developed to be adopted by specific users, application domains, and usage scenarios.

By answering this research question, we aim to identify the external characteristics of modeling tools, pertaining to their adoption and usage [11]. Typical examples include: supported (types of) notations, human-computer interfaces, application domains, and addressed user groups.

Practitioners can benefit from the answer to this research question by understanding how specific state-of-the-art tools address their problems, what are their limitations in terms of blended modeling, and how they can be improved.

**RQ2.** *What are the **realization-oriented characteristics** of modeling tools most suitable for supporting blended modeling?*

With the advent of model-based approaches and domain-specific modeling, in particular, several modeling tools are being developed to support

---

<sup>31</sup><https://zenodo.org/record/6402743>

certain levels of blending, formalisms, and semantics. Moreover, until the recent spread of mainstream language workbenches (e.g., Xtext<sup>32</sup>, Sirius<sup>33</sup>, MPS<sup>34</sup>, etc.), the development of such modeling tools had been relatively ad-hoc.

By answering this research question, we aim to identify the internal characteristics of modeling tools, and that, in terms of (i) their features and (ii) the techniques employed to implement those features. Typical examples include: implementation platforms, consistency mechanisms, change propagation, traceability, and the linguistic level of model-to-model correspondence are investigated.

Researchers can benefit from the answer to this research question by understanding the state of the practice on the techniques of blended modeling tools, including the gaps to fill.

The identified research questions drive the whole study, with a special influence on (i) the search and selection, (ii) data extraction (including the definition of the classification framework), and (iii) data and main findings synthesis.

### 2.3.3 Search and selection

The goal of the search and selection phase is to retrieve a representative set of modeling tools supporting multiple modeling notations, as demanded by the principles of blended modeling. First, we perform a *systematic review* of both the academic (i.e., scientific articles published at peer-reviewed academic venues) and grey literature (i.e., websites, online blogs, etc.), and discuss the results in Section 2.3.3.1. The output of these two activities (i.e., academic studies and non-academic entries) is then further analyzed in order to identify the modeling tools either considered, mentioned, or discussed in them (Section 2.3.3.2).

#### 2.3.3.1 Systematic Reviews

We follow the same overall process when reviewing both the academic and grey literature. In this phase, it is fundamental to achieve a good trade-off between the coverage of existing results on the considered topic and having a manageable number of studies to be analyzed [59, 101]. To achieve the above-mentioned trade-off, our search and selection process has been designed as a multi-stage process; this gives us full control over the number and characteristics of the entries being either selected or excluded during the various stages. In the following, we present each step of our systematic review process. In the remainder of this report, we refer to both academic studies and non-academic entries as *primary studies*, unless specifically noted otherwise. The systematic review is divided into three subsequent and complementary steps of (i) automatic search, (ii) application of selection criteria, and (iii) snowballing.

<sup>32</sup><https://www.eclipse.org/Xtext>

<sup>33</sup><https://www.eclipse.org/sirius>

<sup>34</sup><https://www.jetbrains.com/mps>

**Automatic search.** In this step, we automatically inspect all the results returned from a query execution on (i) Google Scholar for academic studies and (ii) the Google Search engine for grey literature. The automatic searches for both academic and non-academic literature are executed in November 2020.

For the *academic literature*, we use *Google Scholar*. We use Google Scholar as the data source for the following main reasons: (i) it is one of the largest and most complete databases and indexing systems for scientific literature; (ii) as reported in [105], the adoption of this data source has proved to be a sound choice to identify the initial set of literature studies for the snowballing process (Section 2.3.3.1), producing a reasonable number of false positives, but no false negatives (thus, no information is lost); (iii) the query results can be automatically processed via already existing tools.

Below we report the search string used in this study. In order to cover as many potentially relevant studies as possible, we defined the search string so that it includes academic studies on blended modeling. The search string can be divided into three main components: the first component captures the model-driven paradigm, the second one captures the focus on multiple entities (e.g., multiple notations) and blending, and the third one is used for ensuring that our results focus on software aspects. To keep the results of this initial search as focused as possible, the query has been applied to the title of the targeted studies.

```
("modeling" OR "modelling" OR "model based"
  OR "model driven")
AND
("multi*" OR "blended")
AND
("notation*" OR "syntax*" OR "editor"
  OR "tool" OR "software")
```

The search string has been tested by executing pilot searches on Google Scholar. At the time of writing, Google Scholar produced a total of 280 hits when searching with the reported search string.

For the *grey literature*, we target the regular *Google Search Engine*. The search engine is selected in accordance with the recommendations for including grey literature in software engineering multi-vocal reviews [59]. The search string used for the academic literature yields mostly academic results even in a general web search. We have, therefore, adapted our search strategy to find non-academic sources. In particular, we identified a number of relevant hits through manual searches early on. These manual hits could be classified as either *lists* (e.g., Wikipedia’s “List of Unified Modeling Language tools”) or *tool-specific pages* (e.g., tool vendor pages or blog posts about how specific tools are used).

We experimented with several search strings to ensure that we find all relevant hits. In particular, we tried to combine different modeling languages and diagram types into one large all-encompassing search string to simplify our search and make it easier to extract results. However, on prototyping this approach, we realized that the OR clauses that we used did not have the desired effect and we did not find the tools we expected, and in particular, not the lists that we expected. In comparison, a search string such as (MARTE) AND



(tool OR editor OR notation OR modelling) yields 162 results on Google, whereas our combined search string that included MARTE<sup>35</sup> and many other languages only yielded 150 results.

Therefore, we decided to carry out an independent search for popular modeling languages. We ran the different searches independently and merged the results later on. We selected the relevant modeling languages using a mixture of expert knowledge, browsing the web pages of well-known modeling tools from the reference set and beyond (e.g., Eclipse Capella<sup>36</sup> and Enterprise Architect<sup>37</sup>), using lists such as Wikipedia's page on "Modeling Languages". We narrowed down the resulting list of around 40 potential modeling languages by searching for (Language Name) AND (tool OR editor OR notation OR modelling) in Google, and analyzing the first ten non-academic hits (i.e., search results that were not academic papers). Since the search term explicitly contains the terms "tool" and "editor", we expected that the Google search engine would include such a tool within the first ten non-academic hits if it exists, and has any practical relevance. Experiments where we checked later result pages for selected searches confirmed this expectation. We thus only included modeling languages for which Google does report a link to a modeling tool. Otherwise, we disregarded it.

To address the large number of hits we would get this way, we limited the search results for each included search string to the first 50 unique results (if less than 50 hits are reported, we collect all of them), which is based on the suggestion from Garousi et al. [59]. The eventual result set included 1,494 hits, typically containing blog posts, user manuals, websites, technical reports, white papers, academic articles, etc.

**Application of Selection criteria.** In this step, the identified potentially relevant entries undergo rigorous filtering based on the application of a set of selection criteria. Following the guidelines for systematic literature review for software engineering [101], we define the set of inclusion and exclusion criteria *a priori*, in order to reduce the likelihood of bias. The potentially relevant entries are rigorously examined by adopting multiple selection rounds in an adaptive reading depth fashion [106]. Specifically, in the first round, the title of the entry is examined. This first step enables us to discard all those papers or web pages that clearly do not fall within the scope of this study. In the second exclusion round, the introduction and conclusion sections are inspected (if present). Finally, the entries are further inspected by considering their full text, in order to ensure that only the ones relevant to answering the research questions are selected. While processing the full text of a paper/web page, we also keep track of all the mentioned modeling tools and consider them in the tools' identification phase (Section 2.3.3.2).

In the following, we detail the set of inclusion and exclusion criteria that guide the selection of the academic and non-academic entries for our systematic review.<sup>38</sup> A potentially relevant entry is selected if it (i) satisfies *all* inclusion

<sup>35</sup><https://www.omg.org/omgmarte>

<sup>36</sup><https://www.eclipse.org/capella>

<sup>37</sup><https://sparxsystems.com/products/ea>

<sup>38</sup>The identifiers used in this section are consistent with those used in the replication package to enable better traceability.

criteria and (ii) does not satisfy *any* of the exclusion criteria. The selection criteria are divided into three categories, namely: *generic* (i.e., they apply for both academic and non-academic studies), *academic-specific*, and *grey-specific*. The decision of adopting three categories of criteria originates from the different nature of the sources we considered (i.e., Google Scholar and the Google Search Engine). By defining three different sets, it is possible to design selection criteria specifically tailored to the specific characteristics of academic and non-academic entries, and hence, improve the overall quality of the selection process.

Generic inclusion criteria:

GEN-I1) Entries on modeling tools, i.e., where models are used as first-class entities and used as a substantial abstraction from the problem domain (e.g., OSATE [104] for modeling hardware/software systems according to the AADL modeling language).

GEN-I2) Entries discussing at least two different notations (possibly for the same abstract syntax). The notations can be of the same type (e.g., both textual).

Generic exclusion criteria:

GEN-E1) Entries on non-modeling tools. For example, articles on IDEs, programming tools, drawing tools, etc.

GEN-E2) Entries that are not in English.

GEN-E3) Duplicates of already included entries.

GEN-E4) Entries that are not available, and hence not analyzable (e.g., the full text of a scientific article is not accessible or the link to a web page is broken).

Exclusion criteria specific to academic sources:

A-E1) Studies in the form of full proceedings and books since they are too broad for being thoroughly analyzed in this phase of the study.

A-E2) Studies that have not been peer-reviewed, as peer-reviewing is the *de facto* standard of quality assurance for scientific literature.

Exclusion criteria specific to grey literature:

G-E1) Web pages reporting exclusively the basic principles of modeling techniques, without mentioning any modeling tool.

G-E2) Web pages reporting exclusively abstract best practices while applying modeling techniques.

G-E3) Web pages reporting an implementation without a discussion of its benefits and/or drawbacks.

G-E4) Academic literature, since such type of studies is considered by a different process in our protocol.

G-E5) Videos, podcasts, and webinars since they are too time-consuming to be considered for this phase of the study.

**Snowballing.** In this step, we complement the preliminary set of academic studies by applying the snowballing procedure [105]. To mitigate a potential bias with respect to the construct validity of the study, backward and forward snowballing is used to complement the automatic search of the academic literature [107]. In particular, this process is carried out by considering the scientific publications selected in the initial automatic search, and subsequently selecting relevant studies among those cited by one of the initially selected ones (backward snowballing). Then, we also perform forward snowballing, i.e., selecting relevant studies among those citing one of the initially selected academic studies [105]. In this context, the *Google Scholar*<sup>39</sup> bibliographic database is adopted to retrieve the studies citing the ones selected through the initial search phase. The final decision about the inclusion of the newly considered publications in the study is based on the application of the selection criteria presented in Section 2.3.3.

### 2.3.3.2 Tool identification

In the tool identification activity, each primary study is manually analyzed and the mentioned modeling tools are identified. This is achieved by investigating the full text of each primary study, and collecting every modeling tool mentioned in it, independently of whether it is blended or not. Then, the set of identified modeling tools is filtered for duplicates, which are subsequently merged, regardless of whether the tool originates from an academic or a non-academic source. After the merge, we obtained a total of 133 modeling tools. For each tool, we have collected the following information: (i) name, (ii) link/reference to official documentation, (ii) organization(s) implementing, maintaining, and supporting the tool, and (iii) tracing information towards all primary studies mentioning the tool.

In order to ensure that the identified tools support us in answering the research questions of this study, we further filter the list of all modeling tools according to a set of selection criteria. Below we report the inclusion and exclusion criteria.

- TI1) The tool allows its users to edit the same model in multiple notations. The user can switch between these notations easily and without an extra processing step (i.e., the tool supports some level of blended modeling). The tool allows a certain degree of temporary inconsistencies. Notations like an overview tree for navigation purposes or any textual representation used for file persistency purposes only are not considered (e.g., XMI).
- TI2) The tool is publicly available (either as an open-source or commercial product).
- TI3) The documentation of the tool is publicly available.
- TE1) The tool is a language workbench. (Our study focuses on modeling tools themselves.)
- TE2) The tool is not available for download as a binary that can be run on current operating systems from an official website or an affiliated platform supporting it (e.g., a GitHub repository).

---

<sup>39</sup><https://scholar.google.com>

TE3) The documentation of the tool is not in English.

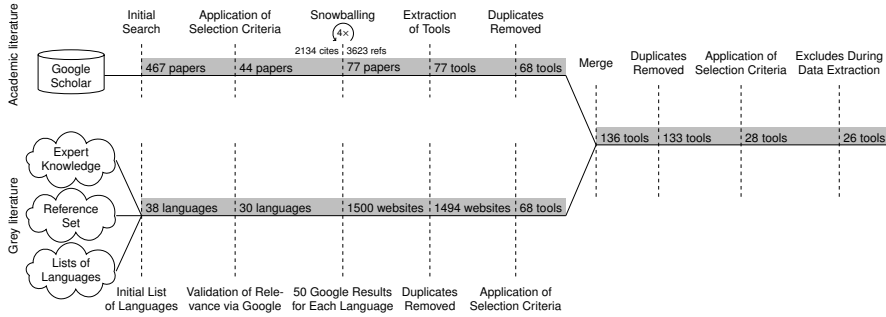


Figure 2.3: Overview of the conducted search and selection steps

A potentially relevant modeling tool is included if it satisfies *all* inclusion criteria (TI1-TI3), and discarded if it satisfies *any* exclusion criterion (TE1-TE2).

To minimize bias, this activity is performed by five researchers and organized as follows. First, two researchers are randomly assigned to each of the potentially relevant tools. Then, the researchers independently apply the tool selection criteria to their assigned tools; each researcher could mark a tool as **included**, **excluded**, **maybe**. For the 12 of 133 tools where at least one researcher indicates an uncertainty (**maybe**), the conflicts are resolved with the intervention of a randomly-assigned third researcher and, when needed, discuss plenary among all researchers involved in this study.

After the final set of modeling tools has been established, we check whether each tool in the reference set is also included in this final set of tools. If all tools in the reference set are indeed included in the final set of tools, we continue with the subsequent phases of the protocol (i.e., data extraction). Otherwise, a dedicated meeting is set up, and a refinement of the systematic review process is designed and conducted again.

Eventually, the final list of modeling tools contains all tools of the reference set.

Figure 2.3 shows the different steps performed in the search and selection phase. Out of the 467 papers in the initial scientific search, 44 papers were included in the snowballing process. The snowballing was performed four times before no more new papers were included. During this process, a total of 2,134 cited and 3,623 referenced papers were reviewed. In summary, 68 distinct tools were extracted from the included papers. For the grey literature part, 30 relevant languages were identified as described above, for which the different search terms yielded 1,494 distinct websites. After applying the selection criteria, 68 tools were included in the tools set. Merging the academic and grey literature parts resulted in 133 distinct tools, of which 30 tools were selected according to the tool selection criteria. Two tools had to be excluded during the data extraction process due to lack of availability or semantically out of scope (see Section 2.3.4). Eventually, 26 modeling tools were sampled, shown in the Referred Tools section at the end of this paper.

Table 2.2: Categories of the classification framework, and their domain.

Category	Definition	Type/Domain
<b>GENERIC</b>		
<b>META</b>		
Tool ID	The internally used ID of the tool.	"T"+[numeric]
Name	The name of the tool.	Free text
Analyzed release	The version of the release the analysis was carried out on.	Free text
<b>TOOL</b>		
First release	Date of the first available release.	Date
Latest release	Date of the latest available release.	Date
Motivation	The self-declared motivation of the tool.	Free text
Open-source	Whether the tool's sources are available openly.	{Yes, No}
Web-based	Whether the tool is web-based.	{Yes, No}
Collaboration	The degree and type of support for collaboration.	{No, Asynchronous, Synchronous}
<b>RQ1: USER-ORIENTED CHARACTERISTICS</b>		
<b>NOTATIONS</b>		
Notation types	Types of notations supported by the tool.	{Textual, Graphical, Tabular, Tree-based, Mixed textual-graphical}
Notation instances (number of)	Sum number of instances of notation types.	Numeric
Embedded notations	Whether there are notations that are embedded into each other.	{Yes, No}
Overlap	The degree of overlap between notations.	{None, Partial, Complete}
<b>VISUALIZATION AND NAVIGATION</b>		
Visualize multiple notations	The ability to visualize more than one notations.	{Yes, No}
Synchronous navigation	Whether the tool supports a synchronous navigation of multiple visualized notations.	{Yes, No}
Navigation among notations	The dynamics of navigation between different notations.	{Immediate, Complex}
<b>FLEXIBILITY</b>		
Flexibility - models	Whether the tool supports temporary inconsistency at the level of the instance models.	{Yes, No}
Flexibility - language	Whether the tool supports temporary inconsistency at the level of the language.	{Yes, No}
Flexibility - persistence	Whether the tools can persist inconsistent models.	{Yes, No}
<b>RQ2: REALIZATION-ORIENTED CHARACTERISTICS</b>		
<b>MAPPING AND PLATFORMS</b>		
Mapping	The way concrete and abstract syntax are mapped.	{Parser-based, Projectional}
Platform	The platform the tool is built on.	{Eclipse, Other}
<b>CHANGE PROPAGATION AND TRACEABILITY</b>		
Change propagation	The dynamics of propagating changes across notations.	{Sequential, Concurrent}
Traceability	Whether the tool supports explicit traceability between notations.	{Yes, No}
<b>INCONSISTENCY MANAGEMENT</b>		
Inconsistency visualization	The degree and way the tool visualizes inconsistencies.	{No, Internal, External}
Inconsistency management type	The way the tool manages inconsistencies.	{On-the-fly, On-demand, Preventive}
Inconsistency management automation	The degree of automation of inconsistency management activities.	{Manual, Partial, Automated, Not applicable}

### 2.3.4 Classification framework definition

Table 2.2 shows the classification framework of this study. The classification framework is composed of three distinct facets; the first facet is about generic characteristics of modeling tools (e.g., release dates, vendor, main motivation for blending notations); the second and third facets directly address research questions RQ1 and RQ2.

We partially reuse the results of previous work [11] related to blended modeling for defining the initial version of the classification framework. Then, as suggested in [102], the customization of the classification framework is performed as follows: (i) firstly we select a random sample of 10 modeling tools, (ii) then two researchers independently extract the data from the 10 modeling tools by using the initial version of the classification framework, (iii) the two researchers then discuss the results of the data extraction with a third researcher, with a special focus on too generic/abstract parameters, parameters which did not fully fit with the characteristics of the tools, parameters with redundant values, and recurrent missing concepts, (iv) the classification framework is customized according to the discussion, and lastly (v) the final version of the classification framework is applied to all remaining modeling tools. It is important to note that when analyzing the remaining 26 tools, the classification framework can still be enriched/updated based on the characteristics of the currently analyzed tool. The details about how we extracted data for each

modeling tool are provided in the next section.

### 2.3.5 Data extraction

The main goal of this activity is to extract relevant data about each modeling tool for answering the research questions. The inputs to this activity are: (i) the set of 28 modeling tools, out of which 26 remained after excluding two additional tools during this phase; and (ii) the textual contents of the academic studies and non-academic entries referring to the tools, and the tools' official documentation (when publicly available). Moreover, when we are not able to collect all relevant data for some specific aspects of a tool (e.g., the internal consistency mechanisms of a proprietary tool) we perform a series of ad-hoc Web searches and contact the support team of the tool for collecting the missing data. For the sake of external verifiability, full tracing information is kept between the extracted data and the considered data sources and it is included in the replication package of the study.<sup>31</sup>

To carry out a rigorous data extraction process, and to ease the control and the subsequent analysis of the extracted data, a predefined data extraction form is designed prior to the data extraction process. The structure of the data extraction form is based on the various categories of the classification framework.

### 2.3.6 Data validation

To ensure the validity of the extracted data, the tool vendors are contacted and the data and the explanation of the reference framework are made available to them. If a tool does not have a clearly identified vendor, we identify knowledgeable experts who published scientific papers related to the tool. The vendors and experts are asked to identify any invalid data related to their tool. The contact is initiated via email with the vendors and experts having an option to ask and discuss the details with our research team. The majority of interactions happened in email. Some vendors and experts preferred a live discussion during a video call, which we also accommodated.

Eventually, we have contacted vendors and experts of 24 tools. The authors of this paper have developed or extensively contributed to the remaining 2 tools, and validated them internally. The validation phase ran for three weeks, between February 28 and March 22, 2022. 69% of tool vendors or experts replied either with minor change suggestions or with the approval of the extracted data. Based on their responses, 3.8% of the data (20 of 520 records) has been updated. The most changes, five, were observed in the model-level flexibility category.

### 2.3.7 Data analysis

The data analysis activity involves collating and summarizing the data, aiming at understanding, analyzing, and classifying the state of the art of modeling tools [108, § 6.5]. The data synthesis is divided into two main phases: vertical analysis and horizontal analysis. In both cases, we perform a combination of content analysis [109] (mainly for categorizing and coding tools under

broad thematic categories) and narrative synthesis [110] (mainly for detailed explanation and interpretation of the findings coming from the content analysis). When performing *vertical analysis*, we analyze the extracted data to find trends and collect information about *each category* of the classification framework. When performing *horizontal analysis*, we analyze the extracted data to explore possible relations *across different categories* of the classification framework.

### 2.3.7.1 Vertical analysis

Depending on the parameters of the classification framework, in this research, we apply both quantitative and qualitative synthesis methods, separately. When considering quantitative data, depending on the specific data to be analyzed, we apply descriptive statistics for a better understanding of the data. When considering qualitative data, we apply the *line of argument* synthesis [102], that is: firstly we analyze each tool individually to document it and tabulate its main features with respect to each specific parameter of the classification framework, then we analyze the set of tools as a whole, to reason on potential patterns and trends. When both quantitative and qualitative analyses are completed, we integrate their results to explain quantitative results by using qualitative results [108, § 6.5]. The results are discussed in Section 2.4.

### 2.3.7.2 Horizontal analysis

Following the best practice of previous secondary studies [94, 95, 111, 112], we explore significant phenomena across pairs of categories as well. We use contingency tables annotated with the Chi-square statistic at  $\alpha = 0.05$ , for identifying statistically significant cases. Following the directions of Haviland [113], we report the p-values of the conventional Chi-square test without Yates's correction for continuity. The results are discussed in Section 2.5.

## 2.4 Results

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna.

Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.

In this section, we elaborate on the findings of this study. First, we discuss the general findings in Section 2.4.1. Then, we elaborate on the two research questions of our study: the user-oriented characteristics (RQ1) and the realization-oriented characteristics (RQ2) of the sampled tools, in Section 2.4.2 and 2.4.3, respectively. In both cases, we contextualize our findings in terms of the three core blended modeling aspects: multi-notation, seamless interaction, and flexibility, as shown in Table 2.3.

Table 2.3: Relationships between blended aspects (BA) and the research questions (RQ) of this study.

	<b>RQ1: User-oriented characteristics</b> (Section 2.4.2)	<b>RQ2: Realization-oriented characteristics</b> (Section 2.4.3)
<b>BA1: Multi-notation</b>	Notations (Section 2.4.2.1)	Mapping and platforms (Section 2.4.3.1)
<b>BA2: Seamless interaction</b>	Visualization and navigation (Section 2.4.2.2)	Change propagation, traceability (Section 2.4.3.2)
<b>BA3: Flexibility</b>	Model/language/persistence flexibility (Section 2.4.2.3)	Inconsistency management and tolerance (Section 2.4.3.3)



Table 2.4: The list of included tools.

Tool			Releases			Info	
ID	Name	Vendor/Maintainer	First	Latest	Analyzed	Open-source	Self-declared motivation
[114]	ADOIT: Community Edition	BOC Products & Services AG	2003	2020	ADOIT:CE based on ADOIT 12.0	No	Enterprise architecture management
[115]	Archi	Beauvoir, P and Sarrodie, JB	2010	2021	4.8.1	Yes	Enterprise architecture
[116]	ARIS	Software AG	2009	2017	2.4d - 7.1.0.1161389	No	Business process modeling
[117]	ASCET Developer	ETAS	2002	2020	7.6.0 Build ID 209	No	"easily combine texts and graphics suiting your programming needs"
[118]	ATGMPM	Université de Montréal	2013	2020	0.8.5	Yes	Multi-paradigm modeling on the web
[106]	BlendedProfile	Mälardalen University	2018	2020	0.3	Yes	Blended modelling for UML profiles
[38]	Boston	View	2015	2020	5.0	No	Fact-based modeling via Object-Role Modeling (ORM)
[119]	Cardanit	ESTECO SpA	2013	2020	Online 007.04.2021.	No	Modeling BPMN with diagrams and tabular views
[120]	Certware	NASA	2013	2016	2.0	Yes	Safety case modeling
[121]	DBDiagrams	Holistics Software	2018	2021	Online 007.04.2021.	No	Visualize textual DB schema definition
[66]	Eclipse Papyrus	The Eclipse Foundation	2008	2020	5.0.0	Yes	Generic-purpose MBSE tool, based on UML and providing support for DSLs via UML Profiles
[122]	Eclipse Process Framework Project	The Eclipse Foundation	2006	2018	1.5.2	Yes	Software process modeling
[43]	MagicDraw	CATIA No Magic	1998	2021	MagicDraw 2021x LTR Enterprise	No	Modelling tool that facilitates analysis and design of Object Oriented (OO) systems and databases. It provides code engineering mechanism (with full round-trip support for Java, C++, C#, CL (MSIL) and CORBA IDL programming languages), as well as database schema modeling, DDL generation and reverse engineering facilities.
[41]	mbdeddr	itemis AG	2012	2018	2018.2.0 based on MPS 2018.2.6	Yes	"Boosting productivity and quality by using extensible DSLs, flexible notations and integrated verification tools."
[115]	MEMO4ADO	OMiLAB	2015	2018	1.10	No	Multi-Perspective Enterprise Modeling
[123]	Modelio	Modelisoft	2011	2020	4.1.0 (202001232131)	Yes	Generic modeling tool for UML, BPMN, ArchiMate, SysML, etc
[104]	OSATE	Carnegie Mellon University	2004	2021	2.9.1	Yes	AADL is a language, with different representations. A textual representation provides a comprehensive view of all details of a system, and graphical if one want to hide some details, and allow for a quick navigation in multiple dimensions.
[124]	QuickDataBaseDiagrams	Dovetail Technologies Ltd	2002	2021	Online 007.04.2021.	No	Modeling DB schemas by text and diagram
[119]	SequenceDiagramOrg	-	2014	2021	Online - 9.1.1	No	Improve the efficiency when creating and working with sequence diagrams by combining text notation scripting and drawing by clicking and dragging in the same model.
[125]	SOM/ADOxx	OMiLAB	1996	2014	SOM 3.0 on ADOxx 1.5	No	Semantic Object Model. Comprehensive approach for object-oriented and semantic modeling of business systems.
[126]	Swimlanes	-	2014	2021	Online 007.04.2021.	No	Visualize sequence diagrams
[39]	TopBraid Composer Maestro Edition	TopQuadrant, Inc	2006	2021	7.1.0	No	"TopBraid Composer™ Maestro Edition (TBC-ME) is a comprehensive Knowledge Graph modeling and SPARQL query tool. In use by thousands of commercial customers, Composer offers robust and comprehensive support for building and testing configurations of rich knowledge graphs."
[127]	UMLet	TU Wien	2002	2018	14.3 Standalone	Yes	Allow textual+visual modeling of UML diagrams
[128]	UMLetino	TU Wien	2013	2018	14.3	Yes	Allow textual+visual modeling of UML diagrams
[40]	Umple	University of Ottawa	2008	2020	Online 1.30.1.5099.60569f335	Yes	Support the convenient modeling across different formalisms. No particular domain targeted, thus, it's a pretty abstract tool.
[129]	USE	Universität Bremen	2007	2020	6.0.0	Yes	System modeling via a subset of UML + OCL

### 2.4.1 Overview

In this section, we review some of the general findings regarding the analyzed blended modeling tools. The list of the included tools is shown in Table 2.4.

**Project age and timeline.** The tools and their respective projects spread over 25 years, with SOM/ADOxx [125] being the oldest tool (first release in 1996) in our sample. On average, the age of the tool projects is 10.6 years ( $\sigma = 5.9$ ). The means of the first and last releases are 2008.8 ( $\sigma = 5.9$ ) and 2019.4 ( $\sigma = 1.8$ ), respectively. These numbers suggest a sample of mature enough tools with sufficient recency in terms of the latest release. Fig. 2.4 provides a visual overview of the age and timeline of tool projects.

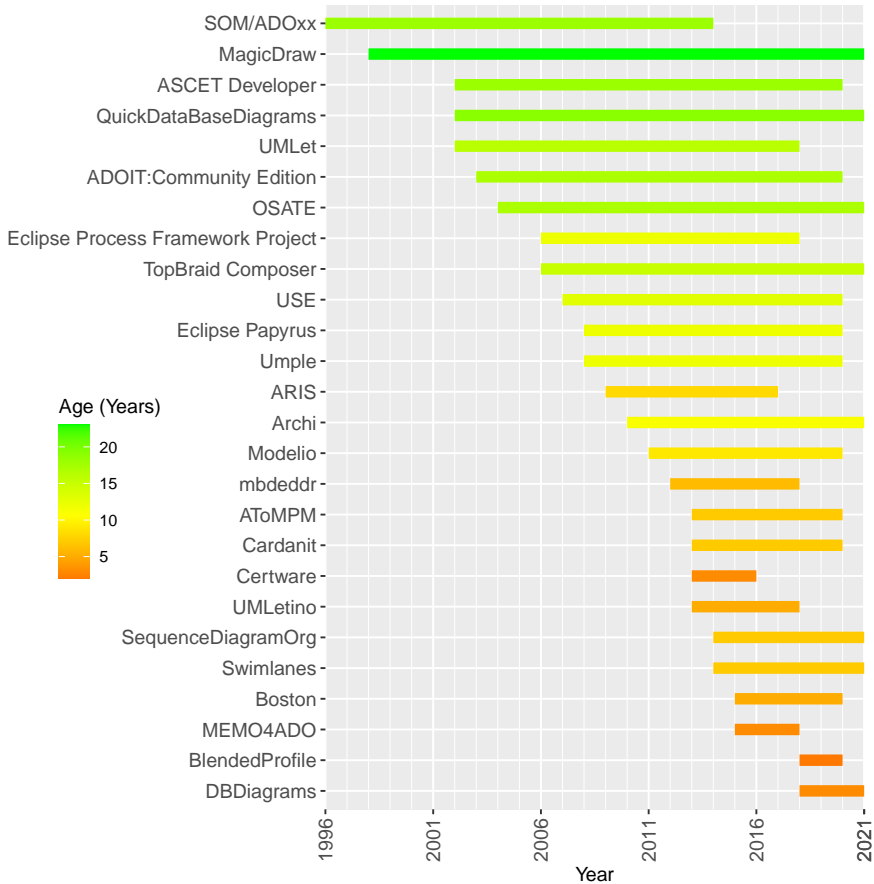


Figure 2.4: Overview of the age of the tool projects, spanned by their respective first and last releases.

**Motivations.** The self-declared motivations of the tools vary greatly. We have recorded the mission statements of the tools and clustered them. General-purpose modeling tools are typical in our sample, usually offering multi-notation

support for UML-based modeling, e.g. Modelio [123], USE [129], and Papyrus [66]. Some of these tools are very specific about their intentions to combine or augment the traditional graphical notation of UML with textual elements, such as UMLet [127] and ETAS ASCET Developer [117]. Among the tools with specific modeling purposes are the ones aiming at process modeling (e.g., SOM/ADOxx [125], ARIS [116]), database modeling (e.g., DB-Diagram [121], QuickDBD [124]), and enterprise architecture (e.g., Archi [115], ADOIT [114]).

**Web-based implementation.** We have found that the majority of the sampled tools, 17 of 26 (65%), are exclusively desktop-based applications, as shown in Table 2.5.

Table 2.5: The web-based nature of tools.

Web-based	#Tools	Tools
No	17 (65%)	[115], [116], [117], [T06], [38], [120], [66], [122], [43], [41], [T15], [123], [104], [125], [39], [127], [129]
Yes	9 (35%)	[114], [118], [119], [121], [124], [T19], [126], [128], [40]

**Open-source.** Half of the sampled tools are released as open-source software (Table 2.6), allowing access to the source code of the tool.

Table 2.6: The open-source nature of tools.

Open-source	#Tools	Tools
No	13 (50%)	[114], [116], [117], [38], [119], [121], [43], [T15], [124], [T19], [125], [126], [39]
Yes	13 (50%)	[115], [118], [T06], [120], [66], [122], [41], [123], [104], [127], [128], [40], [129]

**Collaboration.** Collaborative modeling is the joint creation of a shared representation of a system through means of modeling [94, 95]. Collaboration enables an orchestrated interplay among stakeholders of different domains, and thus, very often, collaboration raises the need for multiple different notations. In real-time collaborative settings, the groupwork of stakeholders happens synchronously. Off-line collaborative settings do not assume synchronicity, but rather stakeholders who work on shared models at different times. As shown in Table 2.7, the majority of tools, 15 of 26 (58%), provides some means of collaboration. Specifically, off-line techniques are typical, accounting for 9 of 15 collaborative tools (60%) or 9 of 26 tools overall (35%), respectively. Finally, 11 of 26 sampled tools (42%) do not support any means of collaboration.

Table 2.7: Support for collaboration.

Collaboration #Tools		Tools
No	11 (42%)	[116], [T06], [120], [66], [122], [T15], [T19], [125], [126], [39], [129]
Yes: Off-line	9 (35%)	[115], [117], [43], [41], [123], [104], [127], [128], [40]
Yes: Real-time	6 (23%)	[114], [118], [38], [119], [121], [124]

## 2.4.2 User-oriented characteristics (RQ1)

In this section, we discuss the findings related to the user-oriented characteristics of the sampled tools. We contextualize our findings in terms of the three aspects of blended modeling tools: the support for multiple notations (Section 2.4.2.1), seamless interaction (Section 2.4.2.2), and flexibility (Section 2.4.2.3).

### 2.4.2.1 Notations

**Notation types.** As shown in Fig. 2.5(a) and Table 2.8, the majority of tools, 5 of 26 (19%), support two types of notation, with additional nine tools supporting three types, and two tools supporting four types.

Every tool, 26 of 26 (100%), features a graphical notation. Textual notations are supported by 19 tools. Additional 13 tools were found with a support for tabular notations, and seven with a support for tree-like notations. This information is detailed in Fig. 2.5(b) and Table 2.9.

Table 2.8: Number of supported notation types.

#Notation types	#Tools	Tools
2	15 (58%)	[114], [116], [117], [118], [119], [120], [121], [T15], [124], [T19], [125], [126], [127], [128], [40]
3	9 (35%)	[115], [T06], [122], [66], [43], [41], [123], [104], [129]
4	2 (8%)	[38], [39]

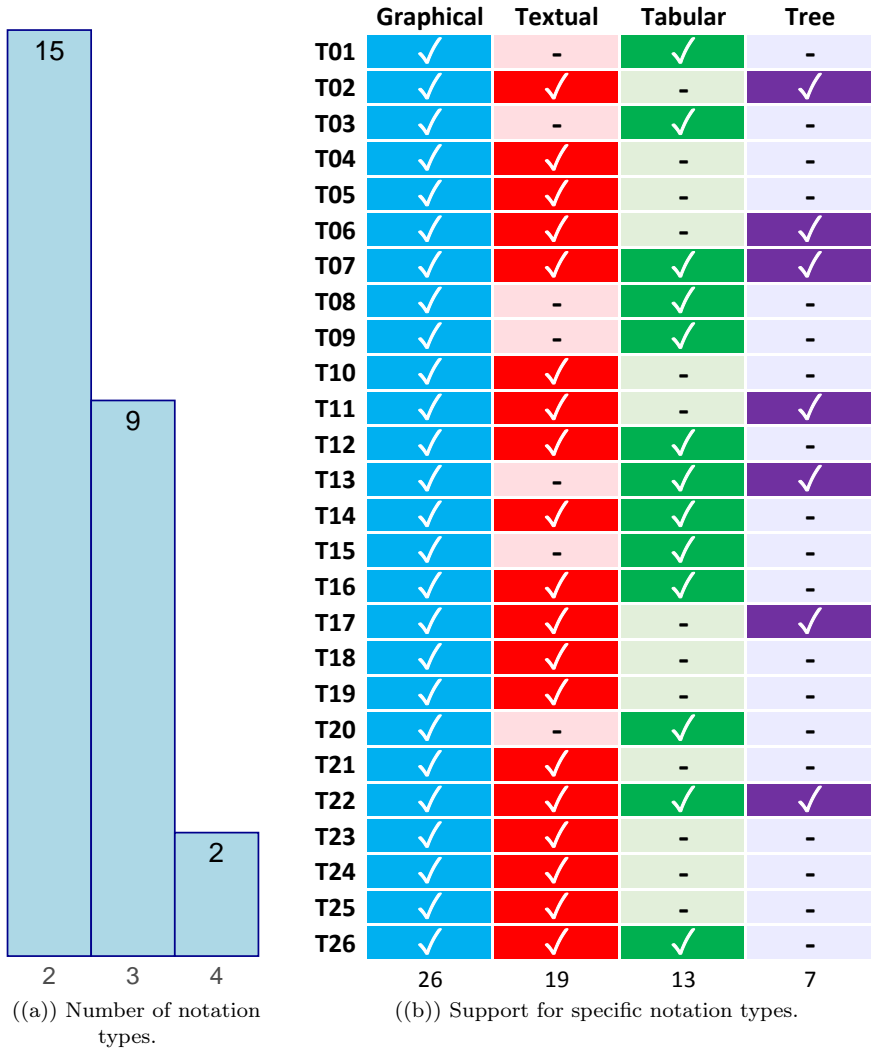


Figure 2.5: Number and combinations of notation types.

Table 2.9: Support for specific notation types.

Notation type	#Tools	Tools
Graphical	26 (100%)	[114], [115], [116], [117], [118], [T06], [38], [119], [120], [121], [66], [122], [43], [41], [T15], [123], [104], [124], [T19], [125], [126], [39], [127], [128], [40], [129]
Textual	19 (73%)	[115], [117], [118], [T06], [38], [121], [122], [66], [41], [123], [104], [124], [T19], [126], [39], [127], [128], [40], [129]
Tabular	13 (50%)	[114], [116], [38], [119], [120], [122], [43], [41], [T15], [123], [125], [39], [129]
Tree	7 (27%)	[115], [T06], [38], [66], [43], [104], [39]

**Embedded notations.** We found a single occurrence of embedded notations, i.e., a host notation being enriched by fragments of another notation (Table 2.10). While the host notation is prevalent during the entirety of the interaction, the embedded notation is accessible in a specific subset of the host notation. For example, in the **Statecharts + Class Diagrams** (SCCD) formalism [130], Class Diagram fragments are used to augment the Statecharts formalism, and provide structural information to compose complex systems.

Table 2.10: Support for embedded languages.

Embedded languages	#Tools	Tools
No	25 (96%)	[114], [115], [116], [117], [118], [T06], [38], [119], [120], [121], [66], [122], [43], [T15], [123], [104], [124], [T19], [125], [126], [39], [127], [128], [40], [129]
Yes	1 (4%)	[41]

**Overlap.** The majority of tools, 21 of 26 (81%), comes with notations that are not fully overlapping. This means that different notations provide different modeling aspects in these tools. An example of full overlap is where a graphical state machine language can render a state machine model with every structural feature; whereas a table only shows which states have transitions to which states.

#### 2.4.2.2 Visualization and navigation

Usability aspects in general are hard to measure. To gain reliable results, it is necessary to conduct a complex user study with concrete tasks, a larger number of participants, interviews and/or surveys, and a thorough evaluation of the answers. This is not feasible in the context of this study and therefore,

Table 2.11: Overlap between notations.

Overlap	#Tools	Tools
Partial	21 (81%)	[114], [115], [116], [117], [118], [T06], [38], [119], [120], [66], [122], [43], [T15], [123], [104], [125], [39], [127], [128], [40], [129]
Complete	5 (19%)	[121], [41], [124], [T19], [126]

we decided to focus on usability aspects that are i) easily measured objectively and ii) specific to blended modeling.

We do not consider the usability of modeling languages themselves as discussed in [131] and [132]. Instead, we focus on the usability of the tools in terms of the topics that are crucial for blended modeling. The idea of blended modeling is to use the notation that is best suited for the current task at hand. This makes it necessary to switch frequently between the available notations. Therefore, for pleasant usability with good support for the user, a tool must offer the possibility to visualize multiple notations side by side and/or provide seamless navigation between notations, or even synchronized navigation. To clarify this more focused view of usability, we use the term “seamless interaction”.

**Visualization of multiple syntaxes.** In general, a blended modeling tool must have the ability to support multiple concrete syntaxes of the same abstract syntax. This parameter, in particular, addresses the possibility of simultaneously viewing multiple notations within a modeling tool, e.g., side-by-side or in an integrated manner such as projectional editors as mbeddr [41] do. All 26 identified tools support the simultaneous view of two or more notations.

**Synchronized navigation.** In addition to the previous parameter, this parameter investigates whether the navigation across multiple notations in the models’ editors is synchronized. For instance, this can be the case in a side-by-side view, if an element in one notation is selected, also its corresponding element in the other notation is selected. Another example of such synchronized navigation is the usage of the double click feature to jump between different views showing corresponding elements but belonging to different notations. As shown in Table 2.12, more than half of the tools, 16 of 26 (62%), provides synchronized navigation facilities.

**Navigation among notations.** Blended modeling tools introduce the benefit that the same model can be viewed and modified using different notations. To enable a fluent modeling experience, the effort required to navigate across notations should be minimal. This binary parameter classifies the effort. It can be either *immediate* (e.g., a click or a keyboard shortcut), or it can involve more complex steps, such as the navigation through multiple (context) menus or wizards. The majority of tools, 20 of 26 (77%), provide immediate navigation from one notation to the other, suggesting a better user experience in terms of seamless interaction.

Table 2.12: Support for synchronized navigation.

Sync'd navigation	#Tools	Tools
Yes	16 (62%)	[115], [116], [118], [T06], [38], [119], [121], [122], [43], [41], [123], [124], [T19], [126], [39], [40]
No	10 (38%)	[114], [117], [120], [66], [T15], [104], [125], [127], [128], [129]

Table 2.13: Navigation among notations.

Navigation	#Tools	Tools
Immediate	20 (77%)	[115], [116], [117], [118], [T06], [38], [119], [121], [66], [122], [43], [41], [T15], [123], [124], [T19], [125], [126], [39], [40]
Complex	6 (23%)	[114], [120], [104], [127], [128], [129]

### 2.4.2.3 Flexibility

Flexibility is the user-related embodiment of tolerating vertical and horizontal inconsistencies [55] at various levels of abstraction in the modeling stack and various modeling facilities. In this study, we specifically consider three types of flexibility, as follows.

**Flexibility – models.** As shown in Table 2.14, the majority of tools, 19 of 26 (73%), does not provide flexibility at the model level. This means that there are no inconsistency tolerance mechanisms in place that would allow deviations between different notations describing the same model. However, a small set of six tools support model-level flexibility.

Table 2.14: Support for model-level flexibility.

Flexibility: models	#Tools	Tools
No	19 (73%)	[114], [115], [116], [118], [T06], [119], [120], [66], [43], [41], [123], [104], [T19], [125], [126], [39], [127], [128], [129]
Yes	7 (27%)	[38], [122], [117], [121], [T15], [124], [40]

**Flexibility – language.** The majority of tools, 22 of 26 (85%), does not provide flexibility at the language-level. (Table 2.15) This means that vertical inconsistencies between model and language (e.g., broken conformance or typing relationships) are not tolerated. We found three exceptions, which



are, however, different from the ones with support for model-level flexibility discussed above: mbeddr [41], OSATE [104], TopBraid Composer [39]. Only a single tool, Umple [40], supports both model- and language-level flexibility.

Table 2.15: Support for language-level flexibility.

Flexibility: language	#Tools	Tools
No	22 (85%)	[114], [115], [116], [117], [118], [T06], [38], [119], [120], [121], [122], [66], [43], [T15], [123], [124], [T19], [125], [126], [127], [128], [129]
Yes	4 (15%)	[41], [104], [39], [40]

**Flexibility – persistence.** The majority of tools, 22 of 26 (85%), does not support persisting inconsistent models. (Table 2.16) Out of the ones with support for persistence-level flexibility, ETAS ASCET Developer [117] and Umple [40] support model-flexibility and flexibility at both levels, respectively. The other two tools with support for persistence-level flexibility are MagicDraw [43] and SequenceDiagramOrg [T19].

Table 2.16: Support for persistence flexibility.

Flexibility: persistence	#Tools	Tools
No	22 (85%)	[114], [115], [116], [118], [T06], [38], [119], [120], [121], [66], [122], [41], [T15], [123], [104], [124], [125], [126], [39], [127], [128], [129]
Yes	4 (15%)	[117], [43], [T19], [40]

## 2.4.3 Realization-oriented characteristics (RQ2)

In this section, we discuss the findings related to the implementation characteristics of the sampled tools. We contextualize our findings in terms of the three aspects of blended modeling tools: the support for multiple notations (Section 2.4.3.1), seamless interaction (Section 2.4.3.2), and flexibility (Section 2.4.3.3).

### 2.4.3.1 Mapping and platforms

**Mapping.** The mapping between abstract syntax and notation is typically implemented either in a parser-based or in a projectional fashion. In *parser-based* approaches, the user modifies the models via different notations, and a parser produces the abstract syntax tree. In *projectional* approaches, however,

the abstract syntax tree is modified directly. Since projectional editors bypass the stages of parser-based editors, they provide support for notations that cannot be easily parsed, but at the same time deliver a different editing experience for textual notations. As shown in Table 2.17, the majority of tools, 22 of 26 (85%), implement a parser-based editor, while four come with projectional facilities.

Table 2.17: Type of mapping.

Mapping	#Tools	Tools
Parser-based	22 (85%)	[115], [116], [117], [118], [T06], [38], [119], [120], [121], [66], [122], [123], [104], [124], [T19], [125], [126], [39], [127], [128], [40], [129]
Projectional	4 (15%)	[114], [43], [41], [T15]

**Platforms.** Eclipse is the only frequently encountered platform in our sample. As shown in Table 2.18, 10 of 26 tools (38%) are built on top of Eclipse, and 18 are built on other, mainly custom platforms. `mbdeddr` [41] is the only MPS-based tool in our sample. One tool, `MagicDraw` [43], also supports more than one platform.

Table 2.18: Platforms of implementation.

Platform	#Tools	Tools
Other	17 (65%)	[114], [116], [118], [38], [119], [121], [43], [T15], [41], [123], [124], [T19], [125], [126], [128], [40], [129]
Eclipse	10 (38%)	[115], [117], [T06], [120], [66], [122], [43], [104], [39], [127]

#### 2.4.3.2 Change propagation and traceability

Change propagation and traceability are the realization-oriented manifestations of the *seamless integration* blended modeling aspect. (See Table 2.3.) During the data extraction phase, however, we have failed to obtain any useful information in these two categories. In the vast majority of cases (exception for ADOIT [114], ARIS [116], the Eclipse Process Framework [122], and `MagicDraw` [43]), we have not found explicit discussions of these concerns, nor any evidence of these concerns being explicit in the tool.

We report these negative results to maintain the symmetry of our classification framework. We suggest replications of this study to be carried out in a conceptual way [133], i.e., attempting to answer the research questions using different methods.

### 2.4.3.3 Inconsistency management

**Inconsistency visualization.** As shown in Table 2.19, the majority of tools, 15 of 26 (58%), does not provide any visualization for inconsistencies. Out of the remaining 11 tools, eight implement an internal visualization mechanism, and three rely on external services.

Table 2.19: Support for inconsistency visualization.

Inconsistency#Tools visualiza- tion	Tools
No	15 (58%) [114], [116], [118], [T06], [119], [120], [122], [43], [104], [T19], [125], [126], [127], [128], [129]
Internal	8 (31%) [115], [38], [117], [121], [41], [123], [124], [40]
External	3 (11%) [66], [T15], [39]

**Inconsistency management type.** The two fundamental approaches to manage inconsistencies are prevention, and allow-and-resolve [57]. Preventive techniques effectively prohibit the emergence of inconsistencies, either by serializing user operations (e.g., via locking), or by constructing the underlying data structures in a way that they can never be inconsistent (e.g., in conflict-free replicated data types (CRDT) [77]. Allow-and-resolve approaches embrace the existence of inconsistencies [86] instead of preventing them. This allows treating inconsistencies with highly sophisticated operations for tolerance [89, 91], and resolution [134, 135, 136]. As shown in Table 2.20, half of the tools, 13 of 26 (50%), prevent inconsistencies. The remaining tools either manage inconsistencies on-the-fly (11 of 26 – 42%) or on-demand (2 of 26 – 8%).

Table 2.20: Support for different inconsistency management types.

Inconsistency#Tools mgmt type	Tools
Preventive	13 (50%) [114], [116], [T06], [119], [122], [43], [T15], [123], [126], [127], [128], [40], [129]
On-the-fly	11 (42%) [115], [117], [118], [38], [120], [121], [41], [124], [T19], [125], [39]
On-demand	2 (8%) [66], [104]

**Inconsistency management automation.** As shown in Table 2.21, 13 of 26 tools (50%) do not provide inconsistency resolution due to their preventive inconsistency management approach. These tools are identical to the ones of the *preventive* category in Table 2.20. Out of the remaining 13 tools, 11

provide some level of automation for resolving inconsistencies, while two tools rely on manual resolution.

Table 2.21: Level of automation of inconsistency management.

Inconsistency automation	#Tools	Tools
Not applicable	13 (50%)	[114], [116], [T06], [119], [122], [43], [T15], [123], [126], [127], [128], [40], [129]
Fully automated	6 (23%)	[38], [104], [124], [T19], [125], [39]
Semi-automated	5 (19%)	[115], [120], [121], [41], [66]
Manual	2 (8%)	[117], [118]

## 2.5 Orthogonal findings

We have analyzed the extracted data for horizontal findings, orthogonal to the vertical analysis reported in the previous section. Specifically for this purpose, we have generated contingency tables for each pair of categories of the classification framework and looked for relevant emerging correlations. In this section, we discuss these findings and contextualize them in terms of the aspects of blended modeling: the support for multiple notations (Section 2.5.1), seamless interaction (Section 2.5.2), and flexibility and inconsistency management (Section 2.5.3); and in the additional aspect of technological trends that are independent from the blended aspects (Section 2.5.4).

### 2.5.1 Number of notation types and Overlap of notations

As shown by the data in Section 2.4.2, the sampled tools support 2.5 types of notation on average. In about 81% of the cases, the overlap between the specific notations is only partial, thus providing a richer way to build models.

**Notation types count vs Web-based nature.** The number of types of notation tends to be higher in desktop tools. Tools with more than two types of notation are exclusively desktop-based. While every web-based tool in our sample provides a maximum of two types of notations, 11 of 17 desktop tools (65%) provide three or more types of notations. We have measured a statistically significant difference at  $p = 0.0064$ .<sup>40</sup>

**Overlap of notations vs Web-based nature.** We found significantly more completely overlapping notations in web-based tools than in desktop-based

<sup>40</sup>For the remainder of the paper,  $\alpha = 0.05$ , unless specifically noted otherwise. Following the directions of Haviland [113], we report the p-values of the conventional Chi-square test without Yates’s correction for continuity.

tools. 4 of 9 web-based tools (44%) come with completely overlapping notations. This ratio is 5.9% in desktop-based tools ( $p = 0.0176$ ). This is in line with the previous observation of web-based tools typically providing fewer types of notation. It is plausible to assume that in desktop tools, the higher number of notation types might result in relevant differences between the notations and, thus, less overlap among them.

**Notation types count vs open-source nature.** The number of notation types tends to be higher in open-source tools than in commercial ones. Three or more types of notations are supported in 8 of 13 open-source tools (62%), while this number is only 3 of 13 (23%) in commercial tools. However, a deeper look also reveals that the only two tools supporting four types of notations are commercial ones ([38], [39]). While 2 of 13 commercial tools (15%) provide four types of notations, 10 of 13 (77%) of them support only two. These differences are significant at  $p = 0.0105$ . It is plausible to assume that while commercial tool vendors have the capabilities to develop sophisticated tools with many types of notations, they still opt for a more streamlined user experience either due to explicit user requirements, or to minimize the technological risks and improve the maintainability of the tools.

## 2.5.2 Seamless interaction

In terms of seamless interaction, we have found significant relationships between the navigation among notations, their synchronicity, and the presence of inconsistency visualization.

**Navigation among notations vs Synchronous navigation.** We have observed a statistically significant difference ( $p = 4E-4$ ) between the *complexity of navigation among notations*, and the *synchronicity of navigation*. The two features go hand in hand. 16 of 16 tools (100%) with support for synchronous navigation also support immediate navigation across different notations. In contrast, only 4 of 10 tools (40%) without synchronous navigation support immediate navigation. That is, in over half of such tools, navigation between notations becomes a complex and tedious task, significantly impacting the user experience in terms of seamless interaction. Synchronous navigation is more frequently observed in tools with completely overlapping notations. While 5 of 5 tools (100%) with completely overlapping notations operate with synchronous navigation, this ratio is only 11 of 21 (52%) in tools with partially overlapping notations.

**Navigation among notations vs Inconsistency visualization.** We observed that 11 of 11 tools (100%) that support inconsistency visualization operate with immediate navigation; while tools without inconsistency visualization support immediate navigation only in 9 of 15 cases (60%). The difference is significant at  $p = 0.0168$ .

### 2.5.3 Flexibility and inconsistency management

As discussed in Section 2.4.2, flexibility, in general, is sporadically supported by the tools we have sampled. We have found that the three types of flexibility features (model-level, language-level, persistence-level), often correlate with inconsistency management aspects.

**Model-level flexibility vs Inconsistency visualization.** Inconsistency visualization is significantly better supported in tools with model-level flexibility. We have found that 7 of 7 tools (100%) with model-level flexibility also support inconsistency visualization, while this ratio drops to 4 of 19 (21%) in tools without model-level flexibility ( $p = 3\text{E}-4$ ). It is plausible to assume that inconsistency visualization is an enabler to model-level flexibility. Visualizing inconsistencies certainly helps the stakeholders to keep track of inconsistencies and reason about the most appropriate time and approach to resolving them.

**Inconsistency visualization vs collaboration.** Tools with internal inconsistency visualization features are also collaborative tools. This holds for 8 of 26 tools (31%). Conversely, the 11 of 26 tools (42%) without collaborative features do not support internal means of inconsistency visualization. The ratio of collaborative and non-collaborative tools is split almost evenly when inconsistency visualization is not present. 15 of 26 tools (58%) come without inconsistency visualization, out of which seven (27%) support collaboration and eight (31%) lack collaborative features. These relationships are significant at  $p = 0.0047$ . These observations can be explained by the strong relationship between collaboration and inconsistencies: as the lack of collaboration might severely reduce the cases when inconsistencies can appear, tools vendors whose tools do not support collaboration might be less interested in developing internal inconsistency visualization techniques.

### 2.5.4 Technological trends

We have further identified some purely technological trends, orthogonal to the three facets of blended modeling, mainly related to the web-based nature of tools (Table 2.5), their collaborative features (Table 2.7), and their platforms of implementation (Table 2.18).

**Collaboration on the web.** The type of collaboration tends to correlate with the type of client software. 5 of 6 tools (83%) that operate with synchronous (real-time) collaboration, are implemented as web-based tools. In contrast, 7 of 9 tools (78%) that operate with asynchronous (off-line) collaboration, are implemented as desktop tools. The type of client software is nearly evenly split in collaborative tools between web clients (7 of 15 – 47%) and desktop clients (8 of 15 – 53%). However, 9 of 11 non-collaborative tools (82%) are built as desktop applications, and we found only two web-based non-collaborative tools. These differences are significant at  $p = 0.0164$ . These observations are in line with the observations of our previous work [95], especially on the apparent mobilization of collaborative modeling.

**"Modeling" platforms are primarily desktop-based.** We observed that neither of the web-based tools in our sample is implemented on a platform that explicitly aims to provide *modeling* capabilities. In contrast, 11 of 18 desktop tools (61%) are implemented on top of a modeling platform, such as Eclipse (10 of 18 – 56%), JetBrains MPS (1 of 18 – 6%), and other, custom platforms (7 of 18 – 39%). While the web-based tools in our sample leverage web frameworks that provide reusable elements to build front-end and back-end functionality, the lack of *modeling* frameworks tailored to the web are apparent. These differences are significant at  $p = 0.0097$ .

## 2.6 Discussion

The corpus of this paper consists of 68 academic papers and 68 entries of grey literature survey, which eventually resulted in 26 identified tools. Based on the rigorously constructed research protocol, we are reasonably confident in the representativeness of our sample for the field under study.

### 2.6.1 Takeaways

The main takeaway of our investigation is that the state-of-the-art and state-of-the-practice tools only provide *partial and accidental support for blended modeling*. This is not a surprising result, considering the novel and emerging nature of the concept of blended modeling. We have found adequately scaling tools in terms of the number of supported notation types. 11 of 26 tools (42%) provide more than the minimal two notation types (Table 2.8). Various aspects related to flexibility, however, pose a potentially serious obstacle for multi-notation tools to become true blended modeling tools. Only 7 of 26 tools (27%) provide flexibility at the instance model level, i.e., tolerance of horizontal inconsistencies between models (Table 2.14). 4 of 26 tools (15%) support flexibility at the language level, i.e., tolerance of vertical inconsistencies, such as conformance or type discrepancies (Table 2.15). In terms of user experience (UX), and especially seamless interaction, we noticed encouraging signs in cross-notation navigability and inconsistency management automation. 16 of 26 tools (62%) support a synchronized navigation across their supported notations (Table 2.12) and, in 20 of 26 tools (77%), immediate navigation is also available (Table 2.13). This enables a better concert of notations, allowing using them in a truly complementary fashion. 11 of 13 tools (85%) that allow inconsistencies to occur treat them with a substantial level of automation; only 2 of 13 (15%) of such tools (a grand total of 8% – 2 of 26) rely on manual resolution of inconsistencies (Table 2.21).

In terms of *user-oriented characteristics* (RQ1), we observed a strong dominance of graphical notations, supported by 26 of 26 tools (100%), followed by textual (19 of 26 – 73%), tabular (13 of 26 – 50%), and tree-based ones (7 of 26 – 27%) (Table 2.9 and Fig. 2.5(b)). Only 5 of 26 tools (19%) feature a combination of notations that are completely overlapping in terms of modeling language concepts (Table 2.11). This means that multi-notation tools tend to leverage the complementary nature of different types of notation. This is a welcome direction as it opens up for opportunities of a richer modeling

experience, paramount in approaches such as MVM and MPM and, as such, it motivates the efforts of blended modeling.

In terms of *realization-oriented characteristics (RQ2)*, we observed the dominance of parser-based solutions, employed in 22 of 26 tools (85%) (Table 2.17). Evidence suggests that projectional editors align better with multi-view and multi-notation principles [69, 71, 137], which are now the typical modeling settings for complex systems [47]. The average age of tools in our sample is 10.6 years ( $\sigma = 5.9$ ), dating the typical modeling tool earlier than the uptick in research interest in projectional editors.<sup>41</sup> We foresee the support for projectional editors to grow as modeling tools are becoming more complex in their denotational and semantic functionalities. We observed a relatively high support for automation of inconsistency management (Table 2.21). Inconsistency management, and tolerance in particular (Tables 2.14-2.16), are key enablers to the flexibility of modeling tools. Only 2 of 26 tools (8%) come without some level of automation in resolving conflicts and these are either research tools, such as [118], or tools that are explicitly not supporting groupwork, such as [117].

## 2.6.2 Challenges and opportunities

By mapping the state-of-the-art and state-of-the-practice, we have identified challenges and opportunities related to the concept of blended modeling in relation to tools.

**Multi-formalism.** Our study assumed one single underlying abstract syntax and a single underlying formalism, but even with this simplification, the support for multi-notation is sporadic. Multi-formalism, and especially multi-semantics, exacerbates this problem as we anticipate the interest in blended modeling gradually shifting towards more complex domains [13, 55, 61]. We see an opportunity for tool builders and integrators in complex engineering domains that inherently work in an MVM/MPM setup, such as mechatronics, automotive, and robotics, to incorporate blendedness as an enabling concept into their existing tool ecosystems. However, this should be preceded by academic research on extending blended modeling, especially on topics such as coordination between models of different languages [138], and synchronization of abstract and concrete syntax in DSLs [139]. Nevertheless, we expect an early maturation and rapid take-off of blended modeling techniques in an array of applied modeling settings. Therefore, we advise technology transfer entities to closely follow academic and semi-academic advancements to propel the transition of the concept to applied industrial settings.

**Seamless interaction.** As a primary user experience (UX) concern, seamless interaction can make a substantial difference in user satisfaction [140] towards modeling tools. The user-oriented aspects of our study (Section 2.4.2.2) show

<sup>41</sup>A directed search on Google Scholar using the (intitle:"projectional editing" OR intitle:"projectional editor" OR intitle:"projectional editors") OR ("projectional editing" OR "projectional editor" OR "projectional editors") search string suggests an increasing publication output starting from 2013.



that current tools are often equipped with related features (e.g., synchronous navigation among notations). Such tools have the opportunity to provide holistic support for blended modeling. The evaluation and comparison of realization-oriented aspects, however, is certainly a challenge, as demonstrated in Section 2.4.3.2. The scope of our study did not include the development of methods that would allow extracting information about user experience and seamless integration of the different modeling paradigms in blended modeling tools. In general, the evaluation of such user-facing aspects remains a challenge. We encourage researchers to develop methods suitable for extracting the types of information outlined in Section 2.4.3.2; and to further enrich the user-facing aspects, based on Section 2.4.2.2. We suggest facilitating dedicated evaluation events, e.g., hands-on workshops at major conferences, where crowdsourcing models for hands-on experimentation and evaluation are feasible because of the volume of the co-located participants and their significant expertise, such as the Hands-on Workshop on Collaborative Modeling (HoWCoM)<sup>42</sup>, and the workshop on Human Factors in Modeling / Modeling of Human Factors (HuFaMo)<sup>43</sup> at MODELS<sup>44</sup>, as well as the Conference on Human Factors in Computing Systems (CHI)<sup>45</sup>. Explicitly modeled user interfaces [141] and API protocols [142] provide especially good foundations for developing software tools that allow seamless switching between notations. Seamless interaction across textual and graphical notations is especially challenging [138] due to the differences between their respective grammar-based and metamodel-based approaches [143]. Projectional editing [69] provides appropriate means to overcome these limitations, thus, we advise researchers to investigate seamless interaction from this standpoint as well.

**Flexibility.** The flexibility of modeling tools in terms of (temporarily) tolerating inconsistencies, such as violations of well-formedness rules and inter-notation/inter-view discrepancies, is best approached by employing state-of-the-art inconsistency models, such as eventual and strong eventual consistency [77]. Although the scope of this study does not entail the particularities of inconsistency management, we have identified traces and patterns of shortcomings in this aspect. While the majority of tools operate in a preventive inconsistency management fashion (Section 2.4.3.3), they implement prevention in the traditional way, i.e., by prohibiting consistency-breaking operations. Such approaches stem from the limitations of strict consistency, whereas novel developments in the field offer much better inconsistency management and, by extension, better flexibility. Strong eventual consistency (SEC) [77], for example, offers a convenient trade-off between the strictness of strong consistency and the guarantees of eventual consistency. As such, SEC is especially well-suited for tools whose developers are more comfortable with preventive inconsistency management models. Such avenues have been explored in multiple collaborative modeling frameworks, such as lowkey<sup>46</sup>, and C-Praxis [144]. We see an opportunity in developing advanced inconsistency tolerance methods

<sup>42</sup><http://howcom2021.github.io/>

<sup>43</sup><https://www.monash.edu/it/humanise-lab/hufamo21>

<sup>44</sup><http://www.modelsconference.org/>

<sup>45</sup><https://chi2021.acm.org/>

<sup>46</sup><https://github.com/geodes-sms/lowkey>

that work at the semantic level of models, especially if blended modeling is extended to support multiple abstract syntaxes or multiple semantics. Recently, inconsistency management between the data and (meta)model level has been investigated, e.g., by Zaher et al. [145]. Such directions align well with the persistence flexibility aspect of modeling tools, which is sporadically supported currently. In general, we encourage tool builders to treat inconsistencies as first-class citizens and, instead of overspending on resources to prevent them, we suggest appropriately managing them [57, 86].

**The many facets of web-based tools.** The interconnected nature of web-based tools and the advanced communication and networking standards of the Internet align well with building collaborative modeling tools. We observed a tendency of tool builders to use web technologies more in collaborative tools (Section 2.5.4). However, we also observed that web-based tools come with significantly less types of notations (Section 2.5.1), and that *modeling* platforms and frameworks are built for desktop applications (Section 2.5.4). It is possible that the shortage of modeling frameworks and language workbenches with a web-based focus limits the ability of tool vendors to provide rich modeling tools with numerous types of notations and advanced modeling facilities. Modeling platforms such as Eclipse already started providing support for deploying modeling tools onto the web, but this is merely a workaround. We foresee an increasing industrial interest in web-based modeling frameworks, such as WebGME [146], providing researchers of language engineering and language workbenches with opportunities.

**Tools performance assessment.** The current generation of modeling tools is facing challenges to manage large-scale complex models [147, 148]. Given the presence of multiple different notations in blended modeling, estimating tool performance when dealing with large-scale and complex models is crucial for the future technical sustainability of blended modeling. However, in our data analysis, we did not observe that tool builders discuss the performance of their blended modeling tools. We conjecture that this lack of communication is mainly because (i) tool performance is still an open problem in MDE [147], and (ii) there are still no standard benchmarks for objectively and fairly comparing the performance of different modeling tools. We suggest that researchers investigate a shared and open benchmark for assessing the performance of modeling tools when dealing with models of different levels of size and complexity (i.e., from a few up to millions of modeling elements). To avoid bias concerning specific DSMLs or application domains, populating such benchmark should be a community effort, where researchers and tool builders coming from different domains collaborate and contribute their models, language definitions, and requirements (e.g., expected time to open a model with 1M elements, expected time to propagate a model change from a visual syntax to the corresponding textual one, etc.). Having such shared benchmarks will provide practitioners with an evidence-based instrument for comparing similar modeling tools and choosing the best one according to their project and organizational needs. Also, a shared benchmark will help MDE researchers in designing and conducting empirical studies assessing the performance of (blended) modeling tools, thus providing objective and replicable knowledge for addressing the grand challenge

of scalability in Model-Driven Engineering [147].

## 2.7 Threats to validity

The study reported in this paper has been carried out based on a carefully designed protocol. To minimize the threats to validity, we have designed our protocol based on well-established guidelines for systematic studies in software engineering [102, 108, 149] and those for including grey literature by Garousi et al. [150].

We have assessed the quality of our study following the guidelines by Petersen et al. [103] and achieved a 63.6% result. This score is significantly higher than the median and absolute maximum scores (33% and 48%, respectively) reported in [103]. This high score can be mainly attributed to the detailed search strategy; the involvement of external senior consultants in the study design phase; and the involvement of multiple authors in the screening phase, minimizing the number of false inclusions and exclusions.

In the following, we discuss the possible threats to the validity of our study and elaborate on how we have mitigated them.

### 2.7.1 External validity

External validity concerns the generalizability of the results [102] and it is primarily associated with the sampling method. The most severe threat to external validity is the lack of representativeness of the selected tools to the field of interest in general. We have mitigated this threat by an appropriately constructed protocol with two orthogonal concerns. First, our search strategy included manual and automated search steps, with exhaustively iterative backward and forward snowballing. Second, we have carried out this search both for the academic and the grey literature [150].

Another class of threats to external validity can be attributed to the inclusion and exclusion criteria used in the screening. To mitigate these threats, we defined exclusion criteria specific to the type of literature (white or grey) being surveyed. Some threats remain, for example, due to the exclusion of non-peer-reviewed academic material (A-E2 in Section 2.3), and the exclusion of proprietary tools that do not allow experimentation with at least a trial version (GEN-E4). We consider these threats minimal.

### 2.7.2 Internal validity

Internal validity is the extent to which claims are supported by data and it is primarily associated with the study design. We have mitigated this risk by the thorough construction and validation of our protocol. The protocol has been developed by multiple authors with relevant expertise on the topics related to blended modeling. Additionally, the protocol has been validated by an external reviewer with significant expertise in empirical research. We have employed rigorous descriptive statistical methods for orthogonal analysis and validation of the data to further mitigate the threats.

### 2.7.3 Construct validity

Construct validity is concerned with the generalizability of the measures of the study to the investigated concepts, and it is primarily associated with the categories and parameters employed during the data extraction and the subsequent analysis. We have mitigated the threats by mapping the research questions to typical parameters before constructing our search strategy. Consequently, we are reasonably confident about the construction validity of the search strings used in the automatic search steps. We have further minimized the threats in the screening phase by refining the inclusion and exclusion criteria in multiple iterations, to reach unambiguous definitions. Each study was assigned to two researchers randomly, and a third researcher was involved to oversee the results and make the final decisions on the inclusion.

### 2.7.4 Conclusion validity

Conclusion validity is the degree of credibility of the conclusions, based on the relationship between cause and effect. Specifically, in our case, conclusion validity is concerned with the relationship between the conclusions communicated in Sections 2.4.2–5.8 and the extracted data. We mitigated the main threats in two steps. First, considering that different researchers might interpret the same data in different ways, we have documented our research protocol in great detail and made it available along with our datasets and statistical analysis scripts in the publicly available replication package.<sup>31</sup> Second, we have constructed conclusions based only on the available data. Any hypotheses and conjunctures were explicitly marked as such.

## 2.8 Conclusions

In this paper, we have reported the results of our systematic, multi-vocal study on the potential, opportunities, and challenges of the emerging approach of blended modeling. We have reviewed nearly 5,000 academic papers, and nearly 1,500 entries of grey literature. Based on these, we have identified 133 candidate tools, and eventually selected 26 state-of-the-art and state-of-the-practice modeling tools which represent the current spectrum of modeling tools. We defined a classification framework for these tools which we used to map their support for other blended aspects, such as navigation and inconsistency tolerance.

Our findings show that current tooling only provides partial support for the features of blended modeling, in particular for inconsistencies between different notations of the same model. The existing support for automated consistency management is encouraging. We also observe that the overlap between notations is not complete. Projectional editing seems to be a promising avenue for future blended modeling, but most existing tools we reviewed are not projectional. Concerning the challenges, we observe that support for multi-formalism and multi-semantics is still largely lacking. We also see opportunities for improvements when it comes to the seamless integration of the different modeling notations and the evaluation of the user experience. Finally, we identify incorporating “softer” models of consistency that directly use the

semantics of the models to achieve eventual consistency as a promising area of future research.

We foresee a new generation of modeling tools that will take blended modeling further by introducing semantic techniques that will allow basing the modeling workflow on multiple different abstract syntaxes.

As for future work, we are working on implementing a generator that produces blended modeling tools for arbitrary domain-specific languages. These tools will be based on the takeaways of this study as well as on a prototype implementation that already embraces the blended principles by Addazi et al. [54]. We intend to keep our dataset up-to-date and report increments on the efforts made on improving blended modeling. Finally, we plan to develop methods for the evaluation of the user experience of blended modeling tools based on hands-on events and workshops.



# Bibliography

- [114] BOC Products & Services AG (2021) ADOIT:Community Edition. <https://www.adoit-community.com/en/>, Retrieved: 22/05/2021.
- [115] Beauvoir, P and Sarrodie, JB (2021) Archi. <https://www.archimatetool.com/>, Retrieved: 22/05/2021.
- [116] Software AG (2021) ARIS. <https://www.ariscommunity.com/>, Retrieved: 22/05/2021.
- [117] ETAS (2021) ASCET Developer. <https://www.etas.com/en/products/ascet-developer.php>, Retrieved: 22/05/2021.
- [118] Université de Montréal (2021) AToMPM. <https://atomp.github.io/>, Retrieved: 22/05/2021.
- [T06] Mälardalen University (2021) Blended Profile. <http://www.es.mdh.se/ModComp/demo.html>, Retrieved: 09/08/2021.
- [38] View (2021) Boston Professional. <https://www.view.com/index.php/products-menu/boston-professional>, Retrieved: 22/05/2021.
- [119] ESTECO SpA (2021) Cardanit. <https://www.cardanit.com/>, Retrieved: 22/05/2021.
- [120] NASA (2021) certware. <https://nasa.github.io/CertWare/>, Retrieved: 22/05/2021.
- [121] Holistics Software (2021) DBDiagram. <https://dbdiagram.io/home>, Retrieved: 22/05/2021.
- [66] The Eclipse Foundation (2021) Eclipse Papyrus. <https://www.eclipse.org/papyrus/>, Retrieved: 22/05/2021.
- [122] The Eclipse Foundation (2021) Eclipse Process Framework Project. <https://projects.eclipse.org/projects/technology.epf>, Retrieved: 22/05/2021.
- [43] CATIA No Magic (2021) MagicDraw. <https://www.3ds.com/products-services/catia/products/no-magic/>, Retrieved: 22/05/2021.
- [41] itemis AG (2021) mbeddr. <http://mbeddr.com/>, Retrieved: 22/05/2021.

- [T15] OMiLAB (2021) MEMO4ADO. <https://austria.omilab.org/psm/content/memo4ado/info>, Retrieved: 08/09/2021.
- [123] Modelisoft (2021) Modelio. <https://www.modelio.org/>, Retrieved: 22/05/2021.
- [104] Carnegie Mellon University (2021) OSATE. <https://osate.org/>, Retrieved: 22/05/2021.
- [124] Dovetail Technologies Ltd (2021) QuickDataBaseDiagrams. <https://www.quickdatabasediagrams.com/>, Retrieved: 22/05/2021.
- [T19] - (2021) SequenceDiagram.org. <https://sequencediagram.org>, Retrieved: 22/05/2021.
- [125] OMiLAB (2021) SOM/ADOxx. <https://austria.omilab.org/psm/content/som/info?view=home>, Retrieved: 22/05/2021.
- [126] - (2021) Swimlanes.io. <https://swimlanes.io/>, Retrieved: 22/05/2021.
- [39] TopQuadrant, Inc (2021) TopBraid Composer. <https://www.topquadrant.com/products/topbraid-composer/>, Retrieved: 22/05/2021.
- [127] TU Wien (2021) UMLet. <https://www.umlet.com/>, Retrieved: 22/05/2021.
- [128] TU Wien (2021) UMLetino 14.3. <https://www.umletino.com/>, Retrieved: 22/05/2021.
- [40] University of Ottawa (2021) Umple. <https://cruise.umple.org/umple/>, Retrieved: 22/05/2021.
- [129] Universität Bremen (2021) USE – The UML-based Specification Environment. [http://useocl.sourceforge.net/w/index.php/Main\\_Page](http://useocl.sourceforge.net/w/index.php/Main_Page), Retrieved: 22/05/2021.



# Chapter 3

## Paper B

Exploiting Meta-Model Structures in the Generation of  
Xtext Editors

J. Holtmann, J. Steghöfer, W. Zhang

*11th International Conference on Model-Based Software and Systems  
Engineering, SciTePress, 2023, pp. 218-225.*



## Abstract

When generating textual editors for large and highly structured meta-models, it is possible to extend Xtext's generator capabilities and the default implementations it provides. These extensions provide additional features such as formatters and more precise scoping for cross-references. However, for large metamodels in particular, the realization of such extensions typically is a time-consuming, awkward, and repetitive task. For some of these tasks, we motivate, present, and discuss in this position paper automatic solutions that exploit the structure of the underlying metamodel. Furthermore, we demonstrate how we used them in the development of a textual editor for EATXT, a textual concrete syntax for the automotive architecture description language EAST-ADL. This work in progress contributes to our larger goal of building a language workbench for blended modelling.

## 3.1 Introduction

Xtext [151] is a framework for the development of domain-specific languages (DSLs). It can either take an existing meta-model and derive a grammar from it or allows a language engineer to create a grammar directly which is then translated into a meta-model. Once a grammar exists, Xtext can generate editors that integrate seamlessly into the Eclipse IDE and offer many convenient features such as an outline view of the file which is currently edited. On the other hand, it offers extension mechanisms for more advanced editor features.

In practice, using these extensions mechanisms can pose significant technical challenges. For example, auto-formatting or the use of template proposals — both common features in modern editors — are not supported for DSLs based on Xtext out-of-the-box. Despite comprehensive documentation of the corresponding extension mechanisms, these challenges re-occur and have to be solved manually. In particular, implementing these features for sufficiently large languages can be cumbersome and involves a lot of repetitive code.

In other cases where Xtext provides support out-of-the-box, the default implementations provided by Xtext are not always suitable for large DSLs since the performance they provide (e.g., for cross-reference auto-completion) is insufficient for practical purposes or for certain use cases.

In the context of a prototype for a textual variant of the automotive systems modeling language EAST-ADL [15, 152], we implemented several automated solutions for these re-occurring challenges by using custom *Xtext generator fragments* that exploit the structure of the language’s meta-model. These fragments are used when Xtext generates the editors for a DSL. We created fragments for formatting, content-assist, and template proposals. In addition, we created a solution for providing the scope of cross-references automatically suggested by the editor which addresses several short-comings of the default solution. In all cases, the structure of the meta-model provides the necessary support to enable the automation.

In this position paper, we discuss these automated solutions and describe how other DSL engineers can adapt them for their needs. Our solutions are particularly suitable for very large, but highly structured DSLs that are available as a metamodel, but should also translate to other situations in which Xtext is used. Furthermore, we discuss the limitations of the automatic solutions where present. Whereas these automations are not generic enough to add them to Xtext properly, they are interesting for other language engineers and can be applied to other languages as well.

Our work in progress contributes to a larger vision of a language workbench for blended modelling in which editors for different concrete syntaxes for the same abstract syntax co-exist and enable engineers to seamlessly switch between the representations [11]. Especially when used in conjunction with evolving languages where the meta-model structure changes regularly, it is necessary to be able to quickly regenerate the editors with as little manual effort as possible.

## 3.2 Related Work

Neubauer et al. proposed an approach [153] to automatically create modern editors for XML-based DSLs by bridging the “technical spaces” *XMLware*,

*grammarware*, and *modelware*. Their automation mainly focuses on providing automatic customization of textual concrete syntaxes for the target DSL, such as which symbols to use. In contrast, our automation mainly focuses on the enhancement of the editors, such as automatically providing default names for new elements, improved suggestions for cross-references, etc.

Latifaj et al. focused on enabling different stakeholders to perform blended modeling [154] in an automated manner, i.e., using different modeling notations to seamlessly handle overlapping parts of the model [155]. They discuss four challenges to achieving this automation goal in the commercial tool RTist. However, they do not address the challenges of content assistance, template proposals, etc., which are the core challenges discussed in this paper.

Cooper and Kolovos acknowledge some of the challenges we address in this work in their paper on requirements and challenges of blended modelling [156]. For instance, they consider scoping across concrete syntaxes to be a challenge. We address this with our custom `ScopeProvider` as discussed in Section 3.4.4.

## 3.3 Background

In the following, we provide relevant information on Xtext as well as EATXT, the textual variant of EAST-ADL which we used as an exemplar and a case study for our work.

### 3.3.1 Xtext

Xtext is a framework for the development of textual domain-specific languages (DSL) [151]. At its core is a grammar language that allows defining the syntax of a DSL. Xtext can either generate a grammar out of an existing meta-model or the user can specify the grammar directly and the meta-model is generated by Xtext. A grammar can be used as the input to generate editors for the Eclipse IDE or for browsers. In addition, Xtext can generate a language server that allows integrating the DSL into VS.Code or Eclipse Theia.

The generation process is controlled by a workflow for the Modeling Workflow Engine (MWE2) [157]. Xtext provides a *language generator* which can be customized and extended with custom *fragments*. A fragment generates code based on the generator’s configuration, the grammar and the corresponding meta-model. Xtext out-of-the-box provides a number of such fragments, which can be extended or replaced. These fragments add a number of features to the generated editors. For instance, Xtext automatically generates a syntax validator which highlights incorrect syntax directly in the editor and in Eclipse’s “Problems” view. For editors in the Eclipse IDE, support for the outline view is also generated. We use Xtext’s ability to change the standard configuration to add custom fragments that provide better formatting, content-assist and template proposals as described below. These custom fragments are written in Xtend [35].

Xtext splits the generated code into different bundles, distinguishing between the infrastructure for the language itself, the user interface of the editors (bundle name ends with “.ui”) and the integration into the Eclipse IDE (bundle name ends with “.ide”).

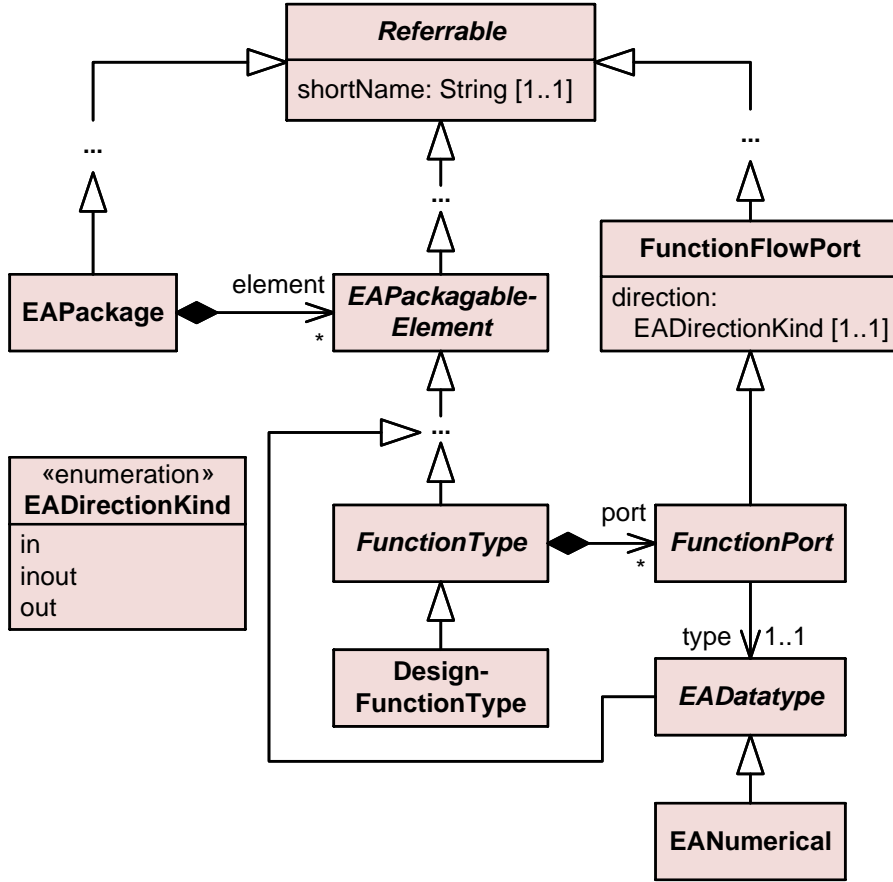


Figure 3.1: EAST-ADL Metamodel Excerpt

### 3.3.2 EAST-ADL and EATXT

EAST-ADL is an automotive systems modeling language [15] and is based on a large metamodel with more than 200 metaclasses and a hierarchy of nested elements describing different aspects of electronic vehicle systems. It can be edited in EATOP, an Eclipse-based editing environment that provides a hierarchical view of an EAST-ADL model along with form- and table-based editing capabilities. EATXT provides a textual syntax for EAST-ADL with the goal to enable blended modeling [11], that is, the ability to switch between the hierarchy-based and the textual representation seamlessly depending on the editing task.

Figure 3.1 depicts an EAST-ADL metamodel excerpt, which we use in this paper as a running example for illustrative purposes. The excerpt contains a set of metaclasses (partially containing attributes) and relationships (i.e., generalizations, compositions, and cross-references), which we explain in the following.

An example of an EATXT file is shown in Figure 3.2. The textual concrete syntax follows the hierarchy and structure of the EAST-ADL metamodel. Meta-

classes in EAST-ADL are represented as blocks delimited by curly braces (e.g., the `FunctionFlowPort WipingCmd` in lines 7–10 as part of the `DesignFunctionType WiperCtrl`). Attributes are represented as lists with the attribute name and the values (e.g., the attribute `direction` in line 8 as part the `FunctionFlowPort WipingCmd`). Cross-references are represented as the reference name followed by the actual path to the reference (e.g., the cross-reference `type` points to “`DataTypes.Integer_uint8`”). In the case of EATXT, the Xtext grammar is specified in such a way that the textual keywords representing the metamodel concepts like metaclasses, attributes, and cross-references are named as in the metamodel (e.g., both the keyword and its corresponding metaclass have the same name `FunctionFlowPort`).

```

1  EAPackage DataTypes {
2      EANumerical Integer_uint8
3  }
4
5  EAPackage FcnDesignTypes {
6      DesignFunctionType WiperCtrl {
7          FunctionFlowPort WipingCmd {
8              direction out;
9              type "DataTypes.Integer_uint8";
10         }
11     }
12
13     DesignFunctionType WiperMotor {
14         FunctionFlowPort WipingCmd {
15             direction in;
16             type "DataTypes.Integer_uint8";
17         }
18     }
19 }

```

Figure 3.2: Excerpt from an EATXT File Specifying a Windshield Wiper Control System

## 3.4 Challenges and Solutions

In this section, we explain our solutions to re-occurring challenges in the development of Xtext-based language workbenches by referring to Figure 3.3, which depicts the coarse-grained architecture of our Xtext-based editor for EATXT.

As described in Section 3.3.2, EAST-ADL is a stable language based on an Ecore metamodel (cf. `eastadl22.ecore` in the plugin `o.e.eatop.eastadl22` in the top-left corner of Figure 3.3). We conceived its textual syntax by automatically

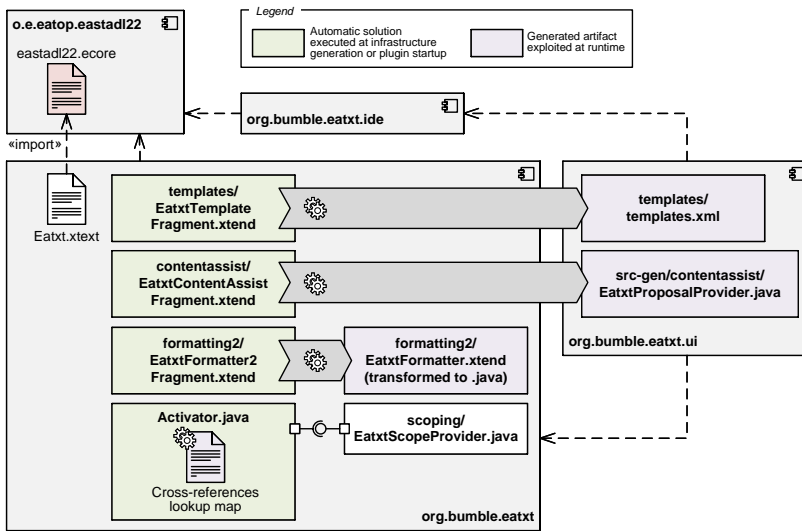


Figure 3.3: EATXT Architecture and Automatic Solutions for Re-occurring Challenges

deriving an initial grammar from that metamodel with Xtext and subsequently adapting the grammar to the stakeholders' needs. Figure 3.3 indicates the resulting grammar as the artifact `Eatxt.xtext` as part of the plugin `org.bumble.eatxt`. In this grammar, we named the production rules and keywords equally as the language concepts in the metamodel (i.e., metaclasses, attributes, and `EReferences`). This enables the exploitation of the metamodel structure and thereby the development of our automatic solutions.

The green Xtend and Java classes that are part of the plugin `org.bumble.eatxt` represent our solutions to the re-occurring challenges described below, exploiting the structure of the metamodel `eastadl22.ecore`. Three of the solutions are custom fragments that are executed by the Xtext language generator (cf. Section 3.3.1) and generate further artifacts (depicted in lilac) that are used in the EATXT runtime. Another solution generates an artifact during the activation of the plugin which can then be exploited at runtime. We provide the implementation in our EATXT development repository [158]. In the following four subsections, we explain and discuss these automatic solutions, the challenges they solve, and the artifacts they generate.

### 3.4.1 Template Proposals

The Eclipse IDE provides the possibility of using *code templates* to enter complex code constructs that contain a significant amount of text and are more complicated than simple keywords. For instance, in the case of the Java programming language, the proposal of complete constructors or different kinds of loops is handled by code templates. Eclipse ships with a set of pre-defined code templates and allows adding user-defined templates or customize existing ones. Likewise, Xtext editors also provide the possibility to use and define code templates by supporting *template proposals*, which consist of the



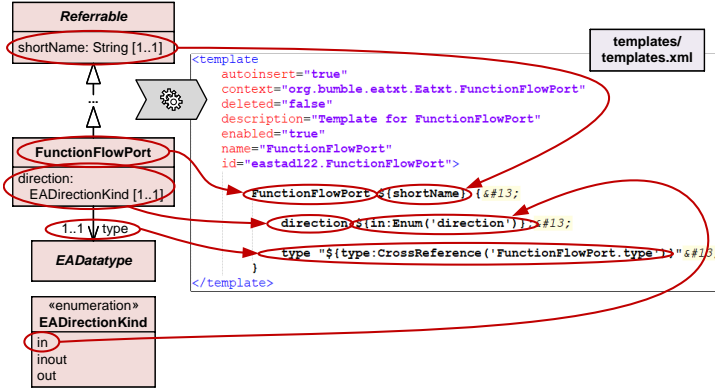


Figure 3.4: Exemplary Generation of a Template Proposal

actual code template and a context type [159]. During the editor generation, Xtext automatically registers such a context type for each production rule and keyword (e.g., the production rule for textual instances of the EAST-ADL metaclass `FunctionFlowPort`), and it provides the context for the code template proposal.

However, the development of template proposals is cumbersome, because the DSL engineer has to define each template manually in an XML file `templates/templates.xml` as part of the UI plugin (cf. top of the plugin `org.bumble.eatxt.ui` on the right-hand side of Figure 3.3) [159]. Moreover, it is recommended practice to define them in the runtime workspace of the Xtext editor through a corresponding dialog and to export the resulting `templates.xml` to the development workspace [159], which impedes rapid prototyping of the template proposals. Particularly, this manual practice is awkward for large metamodels.

As an automatic solution to this challenge in EATXT, we implemented the generator fragment `templates/EatxtTemplateFragment.xtend` as part of the plugin `org.bumble.eatxt` (cf. fragment of the plugin in Figure 3.3). This fragment is executed by the MWE2 workflow and automatically generates the XML file `templates/templates.xml` as part of the plugin `org.bumble.eatxt.ui`. The fragment iterates over the metamodel and generates a template proposal for all metaclasses and all of their mandatory sub-elements (i.e., attributes, containments and their nested structures, as well as cross-references). We restrict the code templates to contain only sub-elements that are mandatory in the metamodel instead of proposing all potential sub-elements, because it might be much effort for the user to delete all proposed optional, but potentially not required, elements.

Figure 3.4 depicts the generation scheme of the fragment using the example of the metaclass `FunctionFlowPort` and its mandatory attribute and mandatory cross-reference. We specify the context type as the name of the production rule, which in the EATXT case is always named the same as the corresponding metaclass (e.g., `org.bumble.eatxt.Eatxt.FunctionFlowPort` for the XML template attribute `context`). The actual code template is then generated as the name of the production rule and metaclass (e.g., `FunctionFlowPort`) followed by a set of further texts as well as opening and closing curly braces.

Beyond proposing simple static texts that would typically not fit to the remainder of the text file and hence would lead to error messages in the editor, the template proposal approach also supports *template variable resolvers* [159]. For simple attributes, we distinguish the different plain data types of attributes (e.g., String, Integer, Float) and corresponding template variable resolvers. For example in Figure 3.4, we translate the String attribute `shortName` of the metaclass `Referrable` to a corresponding template variable resolver `${shortName}`. For enumeration attributes, we generate enumeration template variable resolvers. For example, the value of the attribute `direction` is automatically set to the first literal in of the enumeration `EADirectionKind`. Furthermore, we also support cross-reference template variable resolvers that propose an existing element that fits to the type of the cross-reference. For example, the cross-reference type is translated to the corresponding resolver.

Figure 3.5 depicts a screenshot for the proposal of a code template for the context type `FunctionFlowPort` with its mandatory two attributes, where the default enumeration value of the `direction` attribute as well as a target candidate for the cross-reference `type` is directly proposed. In the case of EATXT, the generated template file contains 194 code templates with a total of more than 1,000 XML lines and covers all metaclasses of the metamodel and their mandatory sub-elements.

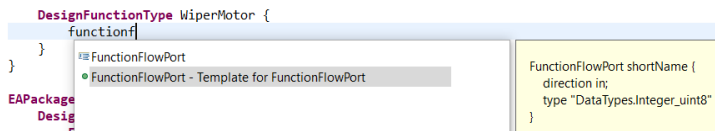


Figure 3.5: Proposed Code Template

As we exploit the concept names of the underlying metamodel for this solution, such a generation of template proposals is restricted to grammars that have the same concept names as the corresponding metamodel (i.e., metaclass names, attribute names, association role names). Thus, the DSL engineer is not allowed to rename the resulting keywords.

### 3.4.2 Content-assist for new Model Elements with Unique Names

Beyond the usual keyword-based content-assist known from IDEs, Xtext provides a customizable approach to provide content-assist for the specification of new model elements by means of *proposal providers* [160]. In this approach, Xtext generates an abstract proposal provider class with default functionality with content-assist methods for all metaclasses as well as an empty concrete subclass that can be customized. To customize the content-assist for a specific model element type (e.g., for the metaclass `FunctionFlowPort` with exemplary instances in lines 7–10 and 14–17 in Figure 3.2), the DSL engineer has to override the corresponding method in a concrete subclass.

In the case of EATXT, we had the requirement to provide content-assist proposals with unique names in the model namespace for all metaclasses with a mandatory `shortName` attribute. Since almost all of the more than

200 metaclasses in the EAST-ADL metamodel have this mandatory attribute due to sub-classing (cf. Figure 3.1), the implementation of the corresponding particular methods in the concrete proposal provider would have been very repetitive.

In order to provide an automatic solution for this challenge, we implemented a fragment `contentassist/EatxtContentAssistFragment.xtend` (cf. plugin `org.bumble.eatxt` in Figure 3.3). Again, this fragment is executed by the MWE2 workflow and exploits the metamodel structure by iterating over all metaclasses with the mandatory attribute and generating the concrete subclass `src-gen/contentassist/EatxtProposalProvider.java` (cf. plugin `org.bumble.eatxt.ui` on the right-hand side of Figure 3.3).

Figure 3.6 depicts the scheme for the generation of the proposal provider methods using the example for the metaclass `FunctionFlowPort`. For each metaclass with the mandatory attribute, this generated proposal provider includes a corresponding overriding content-assist method that proposes a unique name consisting of the corresponding prefix `<metaclassName_>` followed by a randomized number for a new instance of the metaclass. For example, we generate for the metaclass `FunctionFlowPort` a corresponding proposal provider method that proposes the prefix `"FunctionFlowPort_"` followed by the randomized number. Overall, the generated `EatxtProposalProvider` encompasses 188 such methods.

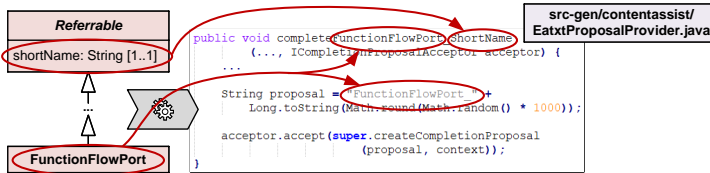


Figure 3.6: Exemplary Generation of a Content-assist Method for Unique Names of new `FunctionFlowPort` Model Elements

Figure 3.7 depicts a screenshot of the content-assist proposal for a new `FunctionFlowPort` instance.

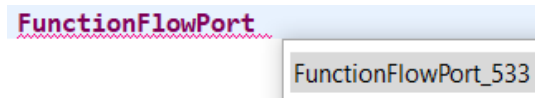


Figure 3.7: Content-assist for a new Model Element

### 3.4.3 Formatters

*Formatting* text documents (e.g., Java source code) in the Eclipse IDE is the process of rearranging the documents' texts without semantically changing their contents. Xtext in principle supports this process as well, but does not provide out-of-the-box formatters [161].

Instead, it requires that the language engineer extends an abstract formatter class and implements a corresponding dispatcher method for any metaclass to be formatted in its text representation. These methods are called when the

formatting process is triggered and apply a series of text replacements for the corresponding model objects. As formatting should treat the text document and its particular contents in a uniform way, the implementation of these methods is highly repetitive.

In order to automate this tedious task and to provide a way of formatting the particular model elements uniformly, we implemented the generator fragment `formatting2/EatxtFormatter2Fragment.xtend` as part of the plugin `org.bumble.eatxt` (cf. fragment of the plugin in Figure 3.3). This fragment is executed by the MWE2 workflow and automatically generates the formatter class `formatting2/EatxtFormatter.xtend` as part of the same plugin. The fragment iterates over the metaclasses of the metamodel and generates the corresponding dispatcher method for each container metaclass. Furthermore, the fragment generates the individual formatting methods for all nested containments or sub-elements of these container classes calls. The generated formatter class in the EATXT case encompasses 51 dispatch methods and 141 calls of the formatting methods for the nested sub-elements.

### 3.4.4 Scoping for Cross-references

Programming languages, DSLs, and metamodels typically provide means to establish links between the semantic concepts defined in the corresponding models (e.g., for specifying the type of a language concept through referencing a different type concept). In the Xtext language development framework, such links are called *cross-references* [162]. The *scoping API* of Xtext provides the means for finding the target of a cross-reference based on its source context [163]. For example, the `FunctionFlowPort` instances in the lines 7–9 and 14–17 in Figure 3.2 are source contexts, and their elements `type` are cross-references pointing to target objects typed by the metaclass `EADatatype` (cf. Figure 3.1). If the target concept is nested in a container hierarchy (e.g., hierarchies of packages like the `EAPackage` instance `DataTypes` in lines 1–3 in Figure 3.2), this procedure for finding a cross-reference target particularly includes the computation of the scope within the nested container hierarchy.

Xtext provides a default out-of-the-box approach for the cross-reference scoping within container hierarchies by means of a *scope provider* [163], where the scope provider also enables content-assist for the cross-reference targets. In this default approach, the fully qualified name of the cross-reference target in the container hierarchy is only proposed by the content-assist if the target is part of a different container. For example in Figure 3.2, the `type` cross-reference in line 9 as part of the container hierarchy `FcnDesignTypes.WiperCtrl.WipingCmd` points to the model element `Integer_uint8` as part of the container `DataTypes`). In contrast, if the target is part of the same container as the source context, then only the plain name of the target is proposed but not its fully qualified name.

In the case of EATXT, a requirement is to provide content-assist that always proposes the fully qualified name. This requires a custom implementation of the scope provider (see also [155] for a different use case requiring such a custom implementation). In such a custom implementation, the DSL engineers have to compute for any source context metaclass the corresponding cross-reference target candidate metaclasses. For example, for the source context metaclass

`FunctionFlowPort` and its cross-reference type, they have to realize that the target candidates are instances of the metaclass `EADatatype` (cf. Figure 3.1). In this context, multiple cross-reference target metaclasses are possible if the source context metaclass has multiple cross-references.

Basically, this computation is straightforward, but needs to consider all cross-references of the underlying metamodel, resulting in a large switch-case statement (i.e., if the source context of a cross-reference is an instance of a certain metaclass, return all candidates that are instances of the metaclasses of all target references). Particularly for large grammars and metamodels, this straightforward custom implementation is very awkward. Beyond that, both the Xtext default approach and custom implementation suffer from performance issues for large grammars and metamodels with many cross-references. In the runtime editor, the target candidate types are always computed on any content-assist keystroke for the given source context type. In the worst case, the lookup needs to traverse the entire switch-case-statement, which consists of all  $n$  possible cases for a complexity of  $O(n)$ .

As an automatic solution for this challenge in EATXT, we generate a cross-references lookup map in the activator of the plugin (cf. `Activator.java` in the plugin `org.bumble.eatxt` in Figure 3.3). This generation traverses the metamodel exactly once with a complexity of  $O(n)$ . This lookup map contains the corresponding type of the cross-reference target for any source context type, and we compute it by iterating over all cross-references in the metamodel. We generate the lookup map during the first activation of the plugin. After that, the `scoping/EatxtScopeProvider.java` accesses it via an interface but does not need to perform the same computation on every cross-reference content-assist keystroke. The lookup map is implemented as a Java `HashMap` whose `get()` method has a complexity of  $O(1)$  in most cases. In the EATXT case, the map encompasses the source context metaclasses and corresponding target metaclasses for 261 cross-references of the EAST-ADL metamodel. Figure 3.8 depicts a screenshot of the content-assist for cross-referencing an existing model element in a different container hierarchy.

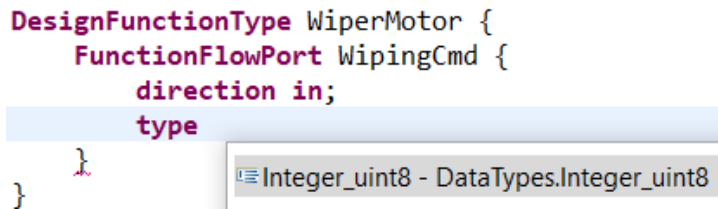


Figure 3.8: Cross-referencing an Existing Model Element

The solution has two advantages, in particular for large grammars and metamodels. First, we automate the awkward custom implementation of a scope provider with a generic solution that exploits the underlying metamodel cross-references structure. Second, it significantly improves the performance of the Xtext scope provider approach by computing a cross-reference lookup table once at plugin activation, instead of computing the target types on every content-assist keystroke for a source context. Thus, the expensive operation takes place only once rather than repeatedly at runtime.

## 3.5 Conclusion and Outlook

Our solutions to the four challenges of template proposals, formatters, content-assist, and scoping have been developed and evaluated in the context of EATXT, but are not only useful there. Instead, we argue that similar approaches are helpful for any highly structured abstract syntax for which Xtext is used to generate editors for a textual concrete syntax. A specific language will benefit from adaptations, especially in the area of scoping. As such, our solutions are not generic enough to contribute to Xtext directly, but will hopefully serve as a blueprint that is useful to other engineers with similar challenges.

We view our contributions in the light of a future language workbench that supports the blended modeling for large DSLs that evolve. As described in our previous work [164], EAST-ADL is such a language and support for its evolution within Eclipse is a relatively new capability. Our work further supports this approach by providing a streamlined way to generate the editors whenever the language changes. When coupled with the generation of a graphical editor (see, e.g., [156]), this brings us closer to the ideal of a *blended modeling language workbench*.

## ACKNOWLEDGEMENTS

Parts of this research were sponsored by Vinnova under grant agreement nr. 2019-02382 as part of the ITEA4 project BUMBLE.

# Chapter 4

## Paper C

Creating Python-style Domain Specific Languages: A Semi-automated Approach and Intermediate Results

W. Zhang, R. Hebig, J. Steghöfer, J. Holtmann

*11th International Conference on Model-Based Software and Systems Engineering, SciTePress, 2023, pp. 210-217.*





## Abstract

Xtext is a well-known domain-specific language design framework and technology. It automatically generates a textual grammar for a language, given a meta-model specified in Ecore. These generated textual grammars are typically not user-friendly. Python-style languages are popular among developers for their usability and conciseness. We aim to propose a systematic approach to transform a DSL with a generated grammar into a Python-style DSL. To achieve this, we analyze the problems of grammars generated with Xtext, based on a lightweight architecture description language. In response to these problems, we propose a general semi-automated grammar adaptation approach. We apply the approach to two other DSLs to validate the generalization of the approach. We also discuss the limitations of this approach and prospects for the future.

## 4.1 Introduction

In contrast to general-purpose programming languages (GPLs) like Java, domain-specific languages (DSLs) are computer languages tailored for a particular application domain [165]. DSLs come in a variety of forms and are employed in a wide range of domains[166]. For example, DSLs are used to improve the level of abstraction and automation in the application development in the robotic domain [167]. But it is important to take into account the workload that goes into creating the DSL [168]. The good news is that there are many tools available for designing and developing DSLs, like JetBrains MPS, MontiCore, Xtext, Racket, etc. [93]. One of them is Xtext [169], an open-source software framework that simplifies the development of DSLs. Xtext can generate a complete DSL infrastructure, including parsers, linkers, compilers, etc. Developers utilize Ecore meta-models to represent domain ideas and their relationships when creating DSLs based on Xtext. From such a meta-model, Xtext generates grammar that specifies the concrete syntax. The editor and its different components are automatically generated from the meta-model of this language as Xtext artifacts.

The grammar generated by Xtext tends to specify languages that are not user-friendly to use. For example, the generated DSLs include default requirements that developers type in all keywords, use braces to an extensive amount, etc. This leads to a high effort in writing when programming in these DSLs. In contrast, the Python [170] language provides a very clean coding style, is renowned for increasing programmer productivity, and is considered easy to learn [36]. Both are properties are also relevant and desirable for DSLs. However, there is currently no systematic approach for converting DSLs to Python-like languages.

In this paper, we introduce an approach to semi-automatically convert textual DSL grammars generated by Xtext to a Python-like DSL. This means that the resulting languages will be easy to code, easy to read, user-friendly, concise, and use whitespace and indents instead of braces [36]. We developed the approach by developing an architectural description language (DemoAdl) as an exemplar for automatically transforming a DSL to a Python-like language. Once our transformation worked for DemoAdl, we evaluated the approach with additional DSLs, *Xenia* and *ACME*. This way we validate the generalizability of our approach and explore its limitations.

## 4.2 Background

As outlined in the introduction, Xtext is a framework for developing DSLs, and Xtext-based Model-Driven Software Engineering (MDSE) is a common solution for developing DSLs [17]. In this solution, the DSL developer uses an Ecore meta-model to describe the concepts of the domain and the relationships between the concepts [171]. For example, “port” becomes a class in the meta-model when using a meta-model to describe concepts in the field of system architecture. Next, the concrete syntax is generated from the meta-model by using the Xtext framework. Xtext provides Modeling Workflow Engine 2 (MWE2) [34], a declarative, externally configurable generator engine. After having a concrete syntax, all Xtext artifacts can be generated by running the

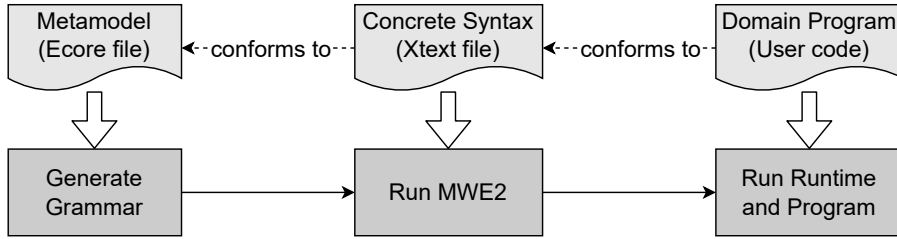


Figure 4.1: The relationship between different artifacts in the Xtext-based MDSE solution.

MWE2 file. These artifacts actually make up the editor for the DSL, including linker, parser, type-checker, etc., common components of editors, and editing support for Eclipse. After the editor is generated, code written in the DSL could be resolved and supported at runtime. Figure 4.1 shows the relationship between the above concepts. When Xtext generates a textual grammar, it will generate a corresponding grammar rule for each class or enumeration in the meta-model and create an attribute in the textual grammar for each attribute or association (i.e., reference or containment) in the meta-model. A grammar rule corresponding to a class may contain multiple attributes, and each attribute occupies a line of text. A grammar rule corresponding to an enumeration contains multiple alternative values, usually on the same line of text.

## 4.3 Methodology

As mentioned above, we developed the approach by developing an architectural description language (i.e., DemoADL) for an example embedded system. The DSL can be used to describe simple embedded system structures, which include hardware components and software components. There are different types of hardware components, and each software component contains a number of sub-components. There are direct links between hardware components and software components, which together make up the system.

We created an Ecore meta-model that describes the domain concepts from this system and generated textual grammar from the meta-model by using Xtext. This constitutes the default grammar for the concrete syntax for DemoAdl. We used this DSL to code an example program with a default style (we called this program a “Default-style program”). We analyzed the shortcomings of the grammar and the program conforms to it. We also created a draft of the program to illustrate what it might look like in a Python-style grammar. We call this program a “Python-like draft”.

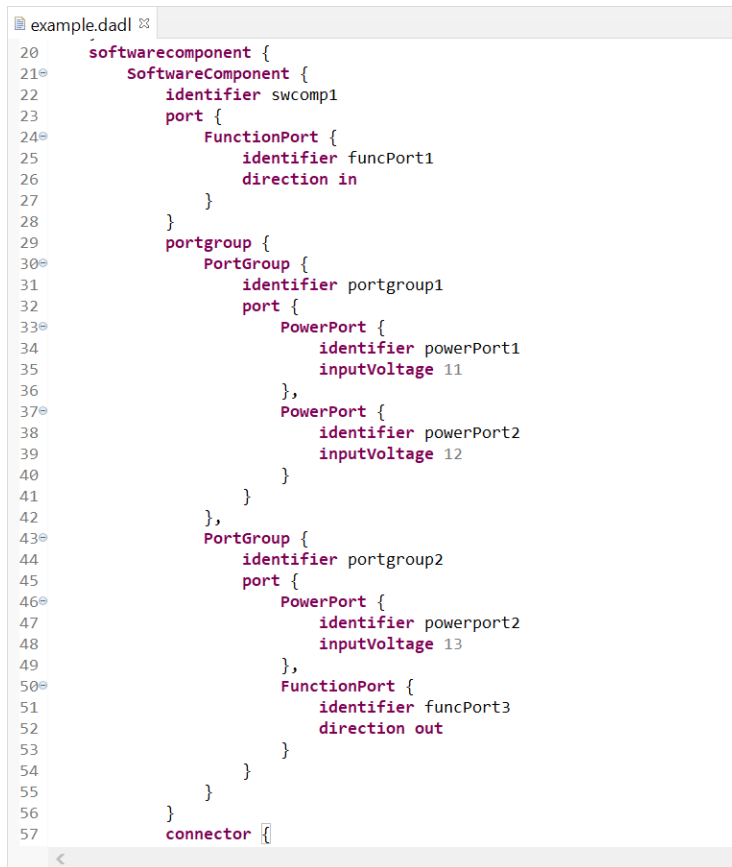
Based on that analysis, we developed a systematic approach to adapt grammars that were generated with Xtext. The approach is based on grammar adaptation rules in response to the problems analyzed from the default-style DSL. We applied these rules to the grammar Xtext generated for DemoAdl. To semi-automate the grammar adaptations, we developed a script in the form of an eclipse Java plug-in. After the adaptation of the grammar, we generated an editor and wrote the same program according to the newly adapted grammar

(we called this code a “Python-like program”). This was to test whether we could successfully parse such a program in the new editor.

To evaluate the generalizability of our approach, we chose two other DSLs. Our selection criteria are that the language 1) has an accessible homepage and 2) with examples on the homepage, and 3) has a downloadable Ecore meta-model. We apply the proposed approach to these two languages to determine whether the script and approach can be applied to create Python-style languages.

## 4.4 Results

We present our analysis, our semi-automated approach, and an evaluation of our approach.



```

example.dadl
20 softwarecomponent {
21   SoftwareComponent {
22     identifier swcomp1
23     port {
24       FunctionPort {
25         identifier funcPort1
26         direction in
27       }
28     }
29     portgroup {
30       PortGroup {
31         identifier portgroup1
32         port {
33           PowerPort {
34             identifier powerPort1
35             inputVoltage 11
36           },
37           PowerPort {
38             identifier powerPort2
39             inputVoltage 12
40           }
41         }
42       },
43       PortGroup {
44         identifier portgroup2
45         port {
46           PowerPort {
47             identifier powerport2
48             inputVoltage 13
49           },
50           FunctionPort {
51             identifier funcPort3
52             direction out
53           }
54         }
55       }
56     }
57   connector {

```

Figure 4.2: A screenshot of part of the domain program with a default Xtext style.

### 4.4.1 Analysis

As mentioned in the methodology section, we first analyzed the shortcomings of the sample DSL (i.e., DemoADL). We illustrate our findings with the help

of a snippet of the default style program in Figure 4.2. We made the following observations about the grammar generated by Xtext:

- **Inappropriate position of identifier.** In Xtext, ‘name’ is the default name for an element’s identifier. However, when we use an attribute named *identifier* to identify an element (e.g., a software component in our case language DemoAdl), then it will default to a normal attribute and be placed within braces. This means that, for example, when we type in keyword *SoftwareComponent*, we have to type in the left brace first and then the attribute identifier. For example, in Figure 4.2 (line 22), attribute *identifier* with value ‘swcomp1’ is used to identify an element *SoftwareComponent*, however, the attribute *identifier* is existing inside of braces in the second line which reduces the brevity of the code. If the same content is expressed in Python, e.g., in Figure 4.3, the identifier value ‘swcomp1’ (line 1) will follow the keyword *SoftwareComponent* to identify the element.
- **Heavy Separation of Code Blocks** The default-style program uses both braces and commas to separate and distinguish code blocks. For example, *PortGroup* ‘portgroup1’ and ‘portgroup2’ are on the same level while they are separated by a comma symbol. Obviously, these commas are redundant, because braces have already been able to separate and distinguish them. In Python, for example in Figure 4.3, ‘portgourp1’ and ‘portgroup2’ are separated by being on different lines though there is no brace or comma for them. This is because there are indents to express hierarchy, and ‘portgourp1’ and ‘portgroup2’ are on the same hierarchy level.
- **Repetitive Keyword.** There are different keywords with the same functional implication. For example, there are two keywords *softwarecomponent* and *SoftwareComponent*, which are highly redundant in function. This also reduced the usability of the language.
- **Nested braces.** There are many nested braces, for example in Figure 4.2, after typing in a keyword, it’s necessary to open a brace. Next, we should type in the keyword *PortGroup* and go into a brace again. These nesting braces are unnecessary and redundant, which increases the complexity of the code. In Python, braces are avoided by having a whitespace-sensitive syntax.

The draft shown in Figure 4.3 shows how the program from Figure 4.2 could appear in a Python-style language.

The envisioned code in Figure 4.3 uses whitespace and indentations to define and separate code blocks, which greatly improves the conciseness of the code. Identifier of e.g. *SoftwareComponent* directly follows after the keyword ‘SoftwareComponent’ (cf. line 1 in Figure 4.3), which makes the code more readable. There are no more braces and the number of keywords and symbols is greatly reduced.

```

1      SoftwareComponent swcompl
2          FunctionPort funcPort1 direction in
3          PortGroup portgroup1
4              PowerPort powerPort1 inputVoltage 11
5              PowerPort powerPort2 inputVoltage 12
6          PortGroup portgroup2
7              PowerPort powerport2 inputVoltage 13
8              FunctionPort funcPort3 direction out
9          PortConnector
10             ports (funcPort1, "swcompl.portgroup1.powerPort1")
11          GroupConnector
12             portgroups (portgroup1, portgroup2)
13

```

Figure 4.3: Snippet of `SoftwareComponent` in the system architecture description example, in Python style

#### 4.4.2 The Semi-automated Approach

As mentioned earlier, we developed a semi-automated approach to adapt a generated grammar to become more like Python. On the one hand, we develop a script that applies some changes automatically. On the other hand, we require the language engineer to take some specific decisions that must be done by hand. The adaptations that are automatically completed by the script are removing braces, repositioning attributes, and removing commas. Hand-crafted adaptations address which keywords need to be refined. In the following we describe these adaptations in more detail:

**Remove Braces and Introduce White-Space Awareness.** Removing all braces directly from the grammar definition may cause errors, such as left recursion errors[172], which will prevent the creation of a textual editor.

Therefore, the functionality of the braces needs to be substituted with the use of whitespace and indents to define code blocks. Thus, we need the language to be whitespace-aware [173]. To this end, the script introduces the following changes:

**Import required features.** Change the reference in the grammar definition file from *org.eclipse.xtext.common.Terminals* to *org.eclipse.xtext.xbase.Xbase*. In addition, import *Xbase* to refer to *EClassifiers* from that model. This gives access to features that allow white-space-aware grammars in Xtext.

**Create BEGIN/END terminals.** Include whitespace-aware blocks in your language by using synthetic tokens in the grammar of the form ‘*synthetic:<terminal name>*’. An example using *BEGIN/END* is shown in Figure 4.4.

**Redefine expression.** Inherits expressions from *Xbase* and redefines the syntax of block expressions by overriding the definition of *xbase::XExpression*.

**Reposition attribute identifier** With the help of the script, we moved the attribute *identifier* from its original position to after the keyword with the same name as the grammar rule. For example, *SoftwareComponent* would

<pre> 27 SoftwareComponent returns SoftwareComponent: 28 {SoftwareComponent} 29 'SoftwareComponent' 30 '{' 31   ('Identifier' identifier=EString)? 32   ('CompID' compID=EString)? 33   ('allocatedTo' allocatedTo={Node EString})? 34   ('port' '{' port+=Port { ',' port+=Port }* ')')? 35   ('portGroup' '{' portGroup+=PortGroup { ',' portGroup+=PortGroup }* ')')? 36   ('connector' '{' connector+=Connector { ',' connector+=Connector }* ')')? 37 '}' </pre>	<pre> 31 SoftwareComponent returns SoftwareComponent: 32 {SoftwareComponent} 33 'SoftwareComponent' 34 identifier=EString 35 BEGIN 36   ('CompID' compID=EString)? 37   ('allocatedTo' allocatedTo={Node EString})? 38   ('port' '{' port+=Port { ',' port+=Port }* ')')? 39   ('portGroup' '{' portGroup+=PortGroup { ',' portGroup+=PortGroup }* ')')? 40   ('connector' '{' connector+=Connector { ',' connector+=Connector }* ')')? 41 END; </pre>
(a) Grammar rule SoftwareComponent before adaptation.	(b) Grammar rule SoftwareComponent after adaptation.

Figure 4.4: Comparison of grammar rules SoftwareComponent before and after adaptation.

be exactly followed by the attribute *identifier*. Here, the script uses regular expressions to find the attribute identifier and move it. If the identifier is called *name*, we do not need to do anything with it.

**Remove Commas** In this step, we removed commas that separate code blocks. For example, if there are multiple *ports* under the same *PortGroup* (cf. line 11 & 12 in Figure 4.5), we do not need to separate these ports with commas because whitespace and indentations are used instead.

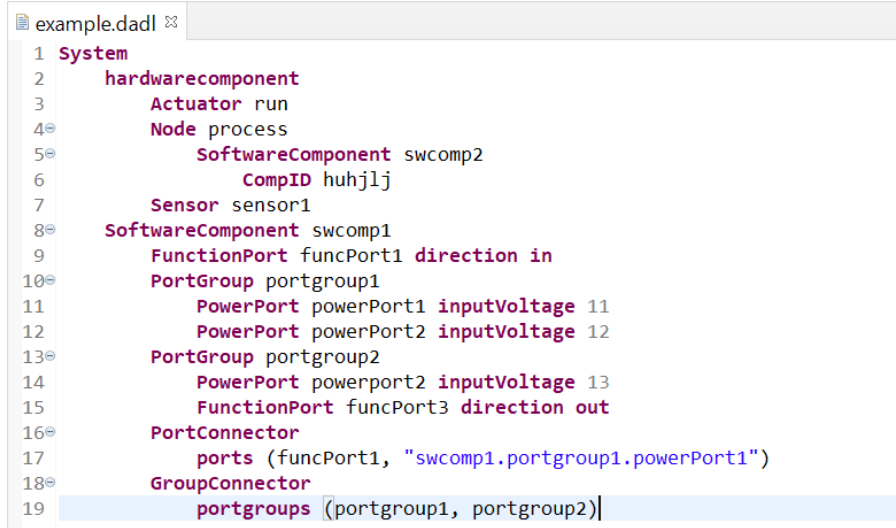
**Refine Keywords** In this step, functionally redundant keywords are manually removed. The aforementioned keywords *SoftwareComponent* and *software-component*, e.g., were functionally redundant and we removed one of them (in our case, we kept the upper case one). We did not implement this removal in the script, because the keywords are language-specific and people make different decisions about what to keep. At the same time, we also did not implement the removal of the *BEGIN/END* related to them in the script. We will address these two automated operations in our future work by configurable rules with a finite set of options. For manually removing functionally redundant keywords, we recommend defining a rule, e.g., to remove the keyword which is written entirely in lowercase. However, our script automatically removes the keyword ‘identifier’, because, in our envisioned draft, the value of the *identifier* should directly follow the keyword (e.g., *SoftwareComponent*) without the existence of the keyword ‘identifier’, which would make the code more concise.

When the script adapts the grammar, it uses regular expressions to search for the target texts in the entire grammar text, and then performs operations on it, including deletion, modification, etc.

### 4.4.3 Evaluation

With the above modifications, the grammar of DemoADL was changed. We generated the Xtext artifacts for the language by running the MWE2 workflow file. We wrote a program (cf. Figure 4.5) that conforms to the newly adapted textual grammar, which was parsed successfully by the generated editor.

To evaluate the usability and generalizability of the proposed approach, we applied it to two additional DSLs, *Xenia* and *ACME*, which we selected based on the criteria described above. The basic information of the two DSLs is shown in Table 4.1.



```

1 System
2   hardwarecomponent
3     Actuator run
4     Node process
5     SoftwareComponent swcomp2
6       CompID huhjlj
7     Sensor sensor1
8     SoftwareComponent swcomp1
9       FunctionPort funcPort1 direction in
10      PortGroup portgroup1
11        PowerPort powerPort1 inputVoltage 11
12        PowerPort powerPort2 inputVoltage 12
13      PortGroup portgroup2
14        PowerPort powerport2 inputVoltage 13
15        FunctionPort funcPort3 direction out
16      PortConnector
17        ports (funcPort1, "swcomp1.portgroup1.powerPort1")
18      GroupConnector
19        portgroups [(portgroup1, portgroup2)]

```

Figure 4.5: Screenshot of part of the example program for the DemoADL language with a Python-like style in eclipse.

Table 4.1: The two languages chosen for the evaluation.

Language	Domain	Language elements	meta-model Source	Homepage
Xenia	Generate web pages	14	[174]	[175]
ACME	SW Architecture description	16	[176]	[177]

**Grammar generation with Xtext** The Ecore meta-model of ACME is from the Atlantic Zoo [176]. To fulfill all technical constraints necessary for the generation of model code and grammar with Xtext, some small adaptations were necessary:

- We filled in the namespace values (i.e., ‘Ns Prefix’ and ‘Ns URI’) for all packages in the meta-model.
- We filled in the value (i.e., ‘Instance Type Name’ for all the types under the package *primitivetypes*.
- We changed the lower-bound value of two attributes under the type *Link* from 1 to 0.

We then generated a textual grammar from both DSLs’ meta-models using the Xtext framework.

**Applying the approach** With the proposed approach, we adapted both DSLs to a Python-like DSL with the help of our script. All manual steps were performed by the first author of this paper. For both *ACME* and *Xenia*, executing these manual steps took less than 30 minutes each.



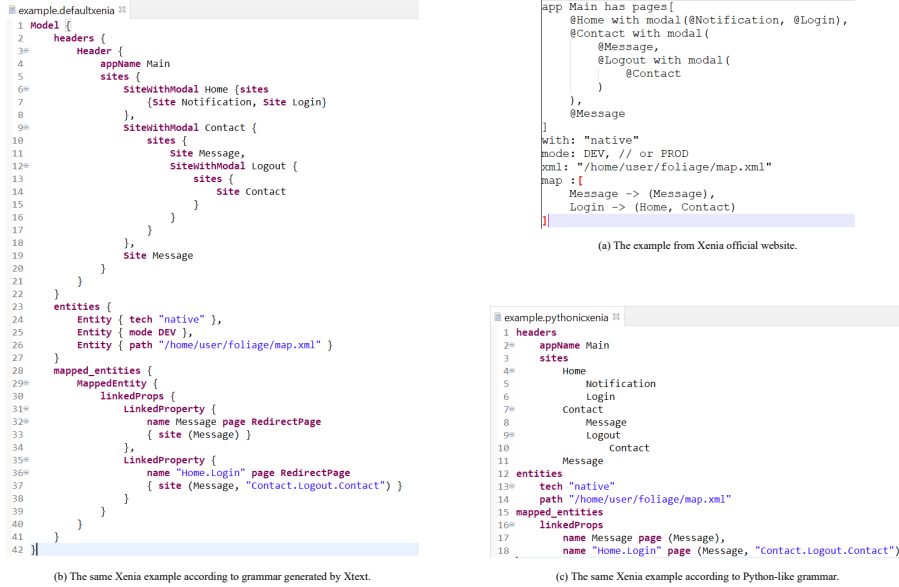


Figure 4.6: Comparison of Xenia programs in different styles (The original example is from <https://github.com/rodchenk/xenia>).

**Outcome** An example program for the grammar that Xtext generated for *Xenia* is shown in Figure 4.6-(b) while the program for the resulting Python-like grammar of *Xenia* is shown in Figure 4.6-(c). For comparison, two programs correspond to the “app Main” example from the *Xenia* home page [175] (shown in Figure 4.6-(a)). The comparison shows that the program in (c) is much more concise than the one in (b) because there are fewer keywords and nesting. Like Python, the program in (c) uses whitespace and indents to express hierarchy. The comparison to the original program (Figure 4.6-(a)) also shows that the resulting Python-like grammar is much closer in terms of compactness to the actual, intended grammar of *Xenia*.

An example program for the grammar that Xtext generated for *ACME* is shown in Figure 4.8 while the program for the resulting Python-like grammar of *ACME* is shown in Figure 4.9. Similarly, the two programs conform to the example “simple\_cs” from the subpage “An Overview Of Acme” of the *ACME* homepage [177] (shown in Figure 4.7). Again we can see that the Python-like program in (c) contains much fewer lines of code, and there is no need to input braces in the program. Every identifier (here e.g., the name) follows the main keyword to identify a certain structure. Also, the comparison to the original grammar shows that the Python-like style is much closer to the original and intended grammar of *ACME* (Figure 4.7) in terms of compactness.

Note that the original syntaxes of both *Xenia* and *ACME* are quite different in style and neither is originally white-space sensitive. This shows that our approach and the script are applicable to diverse DSLs and can be used by DSL developers to quickly reach a Python-like grammar, which could then be used as a basis for further refinements of the grammar.

---

```

System simple_cs = {
    Component client = { Port send-request; };
    Component server = { Port receive-request; };
    Connector rpc = { Roels { caller, callee}};
    Attachments {
        client.send-request to rpc.caller;
        server.receive-request to rpc.callee;
    }
}

```

---

Figure 4.7: ACME original syntax from [https://www.cs.cmu.edu/~acme/docs/language\\_overview.html](https://www.cs.cmu.edu/~acme/docs/language_overview.html).

## 4.5 Discussion

### 4.5.1 Threats to Validity and Limitations

**Threats to Validity** As discussed in the evaluation part, we applied the proposed approach to the two DSLs Xenia and ACME. The results show that the approach could be successfully used for these two DSLs. However, to ensure generalizability, we plan to apply the approach to additional languages. Although we could show that our method provides a quick way to adapt diverse Xtext-generated grammars to Python-style languages, there are still some limitations.

**Expressiveness of language.** Even after adapting a grammar with our approach, there is often room for further modifications to the grammar to improve the expressiveness of the language itself. In the Xenia example, we can change the keyword *page* to the arrow *->*, indicating a “progressive” or “link” relationship as it was done in the original grammar. Such special keywords are typical for DSLs. Our proposed approach does not include such operations, since they are highly language-specific. We focus currently only on adapting a DSL with a default Xtext-generated grammar to Python style. However, in future work, it might be interesting to support such operations with automation tools.

**Automation of Grammar Modification** We implemented a small script in the form of an Eclipse plugin to simplify grammar modification. However, the functionality of the script is limited. For example, adding a colon *:* after a certain type of keyword is not supported by the script at present. A feasible solution will be to extend the functionality to support various modifications of keywords.

Also, while adapting the grammar, one may encounter problems such as *left-recursion* errors. Other than the focused change to white-space awareness, we currently provide no additional support to help developers avoid changes that lead to these errors.

```

example.myacme ⌕
1 System {
2   name simple_cs
3   componentDeclaration {
4     ComponentInstance {
5       name client
6       instanceOf "="
7       ports {
8         Port { name "send-request" }
9       }
10    },
11    ComponentInstance {
12      name server
13      instanceOf "="
14      ports {
15        Port { name "receive-request" }
16      }
17    }
18  }
19  connectorDeclaration {
20    Connector {
21      name rpc
22      roles {
23        Role { name caller },
24        Role { name callee }
25      }
26    }
27  }
28  attachments {
29    Attachment {
30      comp client port "send-request" con rpc role caller
31    },
32    Attachment {
33      comp server port "receive-request" con rpc role callee
34    }
35  }
36 }

```

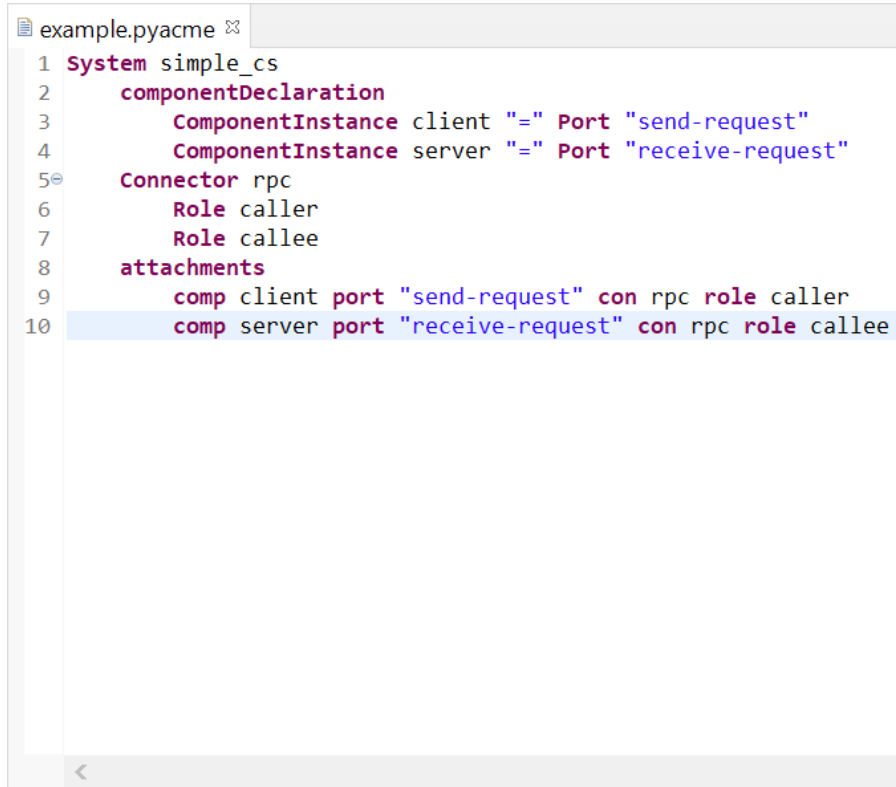
Figure 4.8: The example from Figure 4.7 in the syntax automatically generated by Xtext.

### 4.5.2 Future Work

As we mentioned above, in the future we will apply our approach to more languages to evaluate generalizability. In addition, we plan to apply the approach to larger DSLs, for example, EAST-ADL. We hope that this will also help us refine our approach and extend the capabilities of the script. Namely, we hope that we can address the previously mentioned limited functionality of our script and the semi-automated nature of our approach.

Converting a DSL generated from a meta-model into a Python-like language can improve the language's simplicity, friendliness, and usability. However, these are all issues on the language appearance level. In the future, we will discuss and explore how to design a good DSL, including defining the standards of what a good DSL is.

Another interesting and related topic is blended modeling [54], which refers



```

example.pyacme ✕
1 System simple_cs
2   componentDeclaration
3     ComponentInstance client "=" Port "send-request"
4     ComponentInstance server "=" Port "receive-request"
5   Connector rpc
6     Role caller
7     Role callee
8   attachments
9     comp client port "send-request" con rpc role caller
10    comp server port "receive-request" con rpc role callee

```

Figure 4.9: The example from Figure 4.7 in the Python-like syntax after our tool modified the generated grammar.

to modeling in different notations (such as textual, graphical, etc.) at the same time. When creating a textual language for a DSL that already has a non-textual notation, the approach in this paper can help create a concise, user-friendly textual language in the first stage, thereby improving the user experience of modeling activities [154]. In future work, we plan to evaluate our approach in this context.

## 4.6 Related Work

Since the first version of Xtext was released in 2006 [178], the German company *itemis* released several versions of Xtext. Eysholdt and Behrens from the company briefly introduced the motivation and capabilities of Xtext in [151]. In addition, *itemis* also officially released the Xtext User Guide, which introduces in detail what Xtext is, how to use it and some of its internal principles [17]. This is the technical manual that is the main reference for this paper. Bettini's book [179] goes further and shows how to develop DSLs using Xtext and Xtend. Moreover, the book has a dedicated and independent chapter to introduce Xbase, which is not included in [17] and it supplements the knowledge about Xbase for us.

Mernik et al. in [168] provide empirical data and valuable experience on when and how to develop DSLs, this study refers to these experiences when designing the DSL. Neubauer et al. customized the grammar generated from the Ecore meta-model [153]. However, their purpose was to solve the problem that the Xtext grammar generator did not create rules for meta-model data types. In contrast, the modification in this paper aims to bring the grammar closer to Python. Manual changes were made by Sredojevi to the DSL's grammar definition file in [180]. By including keywords and identifiers, this change primarily addresses the issue that the graphical representation of the meta-model does not fully define the grammar structure. In contrast, our approach replaces the language's overall style by modifying the entire grammar definition file.

## 4.7 Conclusion

Xtext is one of the most commonly used methods for developing DSL. It can be used to generate a textual grammar from an Ecore meta-model. However, this auto-generated grammar has a format that is often considered cumbersome and difficult to work with. Python is a language known for its simplicity, which helps programmers become more productive. We aim to make it easy for DSL developers to create their languages with a Python-style syntax, in the hope that these DSLs would benefit from the advantages of Python's syntax. In this paper, we analyze the primary inadequacies of an auto-generated DSL (i.e., DSL generated by Xtext) which is to describe the architecture of an embedded system. Based on this analysis, this paper presents an approach to semi-automatically change a grammar that was generated with Xtext, so that the DSL becomes a Python-like language. We applied this approach to the design of the aforementioned lightweight example language and obtained an intermediate result, a Python-like structure description language. We applied the approach to two additional DSLs, *Xenia* and *ACME*, to validate the generalizability of our approach. The contribution of this paper is a generalized approach for transforming generated DSLs into Python-like DSLs.

We intend to investigate the practical applications of this approach in the future, e.g., the adaptation of highly sophisticated and large DSLs in industrial settings. In our future efforts, we intend to develop a more complete system for the adaptation of automatically generated grammar. We envision a rule-based system that allows an engineer to configure a number of modifications of the grammar that, together, change the concrete syntax significantly. We believe such a system will contribute to making Xtext more attractive to language engineers who want to combine work on an evolving and rapidly changing language with a user-friendly and compact concrete syntax, a combination that Xtext currently does not support.



# Chapter 5

## Paper D

Supporting Meta-model-based Language Evolution and  
Rapid Prototyping with Automated Grammar Optimiza-  
tion

W. Zhang, J. Holtmann, D. Strüber, R. Hebig, J. Steghöfer

*Revised and Re-submitted to Journal of Systems and Software, 2023.*





## Abstract

In model-driven engineering, developing a textual domain-specific language (DSL) involves constructing a meta-model, which defines an underlying abstract syntax, and a grammar, which defines the concrete syntax for the DSL. Language workbenches such as Xtext allow the grammar to be automatically generated from the meta-model, yet the generated grammar usually needs to be manually optimized to improve its usability. When the meta-model changes during rapid prototyping or language evolution, it can become necessary to re-generate the grammar and optimize it again, causing repeated effort and potential for errors.

In this paper, we present GRAMMAROPTIMIZER, an approach for optimizing generated grammars in the context of meta-model-based language evolution. To reduce the effort for language engineers during rapid prototyping and language evolution, it offers a catalog of configurable *grammar optimization rules*. Once configured, these rules can be automatically applied and re-applied after future evolution steps, greatly reducing redundant manual effort. In addition, some of the supported optimizations can globally change the style of concrete syntax elements, further significantly reducing the effort for manual optimizations. The grammar optimization rules were extracted from a comparison of generated and existing, expert-created grammars, based on seven available DSLs. An evaluation based on the seven languages shows GRAMMAROPTIMIZER's ability to modify Xtext-generated grammars in a way that agrees with manual changes performed by an expert and to support language evolution in an efficient way, with only a minimal need to change existing configurations over time.

## 5.1 Introduction

Domain-Specific Languages (DSLs) are a common way to describe certain application domains and to specify the relevant concepts and their relationships [181]. They are, among many other things, used to describe model transformations (the Operational transformation language of the MOF Query, View, and Transformation — QVTo [182] and the ATLAS Transformation Language — ATL [183]), bibliographies (BibTeX [184]), graph models (DOT [185]), formal requirements (the Scenario Modeling Language — SML [186] and Spectra [187]), meta-models (Xcore [188]), or web-sites (Xenia [189]).

In many cases, the syntax of the language that engineers and developers work with is textual. For example, DOT is based on a clearly defined and well-documented grammar so that a parser can be constructed to translate the input in the respective language into an abstract syntax tree which can then be interpreted.

A different way to go about constructing DSLs is proposed by model-driven engineering. There, the concepts that are relevant in the domain are captured in a meta-model which defines the *abstract syntax* (see, e.g., [168, 190, 191]). Different *concrete syntaxes*, e.g., graphical, textual, or form-based, can be defined to describe actual models that adhere to the abstract syntax.

In this paper, we consider the Eclipse ecosystem and Xtext [2] as its de-facto standard framework for developing textual DSLs. Xtext relies on the Eclipse Modeling Framework (EMF) [4] and uses its Ecore (meta-)modeling facilities as basis. Developing a textual DSL in Xtext involves two main artifacts: a grammar, which defines the concrete syntax of the language, and a meta-model, which defines the abstract syntax. Xtext allows either the grammar or the meta-model to be created first, and then automatically generating the one from the other (or alternatively, writing both manually and aligning them).

Software languages change over time. This is due to *language evolution*, which entails that languages change over time to address new and changed requirements, and due to *rapid prototyping*, which involves many quick iterations on an initial design. In the case of an Xtext-based language, grammar and meta-model need to be modified to stay consistent with each other. We consider two options for evolving a language in Xtext: First, the developers can change the grammar and then use Xtext to automatically create an updated version of the meta-model from it. Second, the developers can change the meta-model then use Xtext to derive an updated version of the grammar from it. We call the first approach *grammar-based evolution*, and the second approach *meta-model-based evolution*.

In this paper, we focus on meta-model-based evolution, for the following rationale: While grammar-based evolution is a common way of developing languages in Xtext, it is not geared for three scenarios that we encountered in the real world, including collaborations with an industrial partner. In particular: 1. Several concrete syntaxes (e.g., visual, textual, tabular) for the same underlying metamodel co-exist and evolve at the same time. This is particularly common in the context of blended modeling [1], a timely modeling paradigm. 2. The metamodel comes from some external source (such as a third-party supplier or a standardization committee), which prohibits independent modification. 3. The metamodel is the central artifact of a larger ecosystem of available

tools, including. e.g., automated analyses, transformations and visualizations. As such, the language engineers might prefer to evolve it directly, instead of relying on the, potentially sub-optimal, output of automatically co-evolving it after grammar changes. The real-world case that inspired this paper has aspects of the first two scenarios: we work on a language from an industry partner for which there already exists an evolving metamodel and graphical editor available.

Compared to grammar-based evolution, meta-model-based evolution has one major disadvantage: Co-evolving the grammar after meta-model changes is more complicated than vice versa, as it involves dealing with both abstract and concrete syntax aspects, whereas updating the meta-model after grammar changes only involves abstract syntax aspects. The goal of this paper is to substantially mitigate this disadvantage, as we will now explain.

One problem that prohibits using a grammar generated from the meta-model directly is that the grammars Xtext automatically generates are not particularly user-friendly. At the same time, the grammars themselves are hard to understand and the languages defined by them are verbose, use many braces, and enforce very strict rules about the presence of keywords and certain constructs. While the usability of DSLs is largely dependent on the right choice of concept names (see, e.g., [192]), the syntax also plays a significant role in how easily a language can be learned. For example, [193] find that *if*-statements are used by novices more accurately if they are written without parentheses and braces. We also find that Xtext tends to add a number of keywords that are not strictly necessary and that make the generated language more verbose without adding clarity.

These issues can be addressed by improving the Xtext-generated grammar. In the state of the art, this is a manual optimization process, in which the user tweaks the grammar to improve its usability, e.g., changing and removing keywords, parentheses, and order of rule elements. However, manual optimization has a significant drawback: Once that the meta-model evolves and the grammar is re-generated, the same optimizations have to be performed again on the generated grammar. This is a time-consuming and error-prone task, even more so when done after any meta-model change. Furthermore, for certain changes that address a larger scope within the grammar (e.g., removing inner parentheses around attributes in every grammar rule), it is a tedious and error-prone manual process even before evolution takes place. Alternatively, instead of auto-generating the grammar when the meta-model evolves, the existing grammar could be manually evolved by new grammar rules and by modifying existing ones. This process is, again, time-consuming and error-prone and can easily lead to inconsistencies.

We propose a different approach: Automated optimization of the generated grammar based on simple rules, which we call *grammar optimization rules*. Instead of modifying the grammar directly, the language engineer creates a set of simple optimization rule applications that modify the grammar file to make the resulting language easier to use and less verbose. Whenever the meta-model changes and the grammar is regenerated, the same or a slightly modified set of optimization rules can be used to update the new grammar to have the same properties as the previous version.

In meta-model-based evolution scenarios, our approach can considerably

reduce the manual effort for optimizations and, consequently, enable faster turnaround times. This is due to two factors that we demonstrate in our evaluation: First, the potential to reuse existing configurations across successive evolution steps. For example, we considered four evolution steps from the history of QVTo. Initially, we created a configuration that fully optimized the generated grammar to be consistent with the expert-created grammar for that evolution step. For the following three iterations, we only needed to modify 2, 0, and 1 configuration lines, respectively, to automatically optimize the generated grammar. Without our approach, language engineers would need to manually modify 228 lines of 66 grammar rules in each evolution step. Second, the availability of powerful rules that enforce a large-scope change affects many grammar rules at the same time. For example, for the EAST-ADL case, modifying the Xtext-generated towards the expert-created grammar required curly braces for all attributes to be removed, while keeping the outer surrounding curly braces for each rule. Performing this change manually entails manually revising 303 rules, whereas it took only one line of configuration in GrammarOptimizer.

While our approach clearly unfolds these benefits in the case of evolving languages and complex changes, it does not come for free. For locally-scoped changes, creating a configuration generally leads to more effort than a manual grammar edit and hence, presents an upfront investment that pays off only when the language evolves over time. In a different paper [45], we present an approach for automating the extraction of configurations from user-provided manual edits, thus reducing the initial manual effort to be the same as in the traditional process while keeping the long-term benefits. Together with the present paper, for the supported kinds of changes, it supports a fully automated process for aligning the grammar after changes to the meta-model.

The contribution of this paper is GRAMMAROPTIMIZER, an approach that modifies a generated grammar by applying a set of configurable, modular, simple optimization rules. It integrates into the workflow of language engineers working with Eclipse, EMF, and Xtext technologies and is able to apply rules to reproduce the textual syntaxes of common, textual DSLs.

We demonstrate its applicability on seven domain-specific languages from different application areas. We also show its support for language evolution in two cases: 1), we recreate the textual model transformation language QVTo in all four versions of the official standard [182] with only small changes to the configuration of optimization rule applications and with high consistency of the syntax between versions; and 2), we conceived for the automotive systems modeling language EAST-ADL [194] together with an industrial partner a textual concrete syntax [195], where we initially started with a grammar for a subset of the EAST-ADL meta-model (i.e., textual language version 1) and subsequently evolved the grammar to encompass the full meta-model (i.e., textual language version 2).

The remainder of this paper is structured as follows. First, in Section 5.2, we provide an overview of the background of this paper, in particular, on metamodel-based textual DSL engineering. In Section 5.3, we review related research. In Section 5.4, we define the methodology of this paper. Subsequently, in Section 5.5, we describe the identified optimization rules, which are the main technical contribution of this paper. Following that, in Section 5.6, we present

our solution of the GRAMMAROPTIMIZER, which implements the identified optimization rules. In Section 5.7, we present our evaluation. Section 5.8 is devoted to our discussion, where we address threats to validity, the effort required to use GRAMMAROPTIMIZER, implications for practitioners and researchers, and future work. Finally, in the last section, we conclude.

## 5.2 Background: Textual DSL Engineering based on Meta-models

As outlined in the introduction, the engineering of textual DSLs can be conducted through the traditional approach of specifying grammars, but also by means of meta-models. Both approaches have commonalities, but also differences [196]. Like grammars specified by means of the Extended Backus Naur Form (EBNF) [197], meta-models enable formally specifying how the terms and structures of DSLs are composed. In contrast to grammar specifications, however, meta-models describe DSLs as graph structures and are often used as the basis for graphical or non-textual DSLs. Particularly, the focus in meta-model engineering is on specifying the abstract syntax. The definition of concrete syntaxes is often considered a subsequent DSL engineering step. However, the focus in grammar engineering is directly on the concrete syntax [198] and leaves the definition of the abstract syntax to the compiler.

**Meta-model-based textual DSLs** There are also examples of textual DSLs that are built with meta-model technology. For example, the Object Management Group (OMG) defines textual DSLs that hook into their meta-model-based Meta Object Facility (MOF) and Unified Modeling Language ecosystems, for example, the Object Constraint Language (OCL) [199] and the Operational transformation language of the MOF Query, View, and Transformation (QVTo) [182]. However, this is done in a cumbersome way: Both the specifications for OCL and QVTo define a meta-model specifying the abstract syntax and a grammar in EBNF specifying the concrete syntax of the DSL. This grammar, in turn, defines a different set of concepts and, therefore, a meta-model for the concrete syntax that is different from the meta-model for the abstract syntax. As Willink [200] points out, this leads to the awkward fact that the corresponding tool implementations such as Eclipse OCL [201] and Eclipse QVTo [202] also apply this distinction. That is, both tool implementations each require an abstract syntax and a concrete syntax meta-model and, due to their structural divergences, a dedicated transformation between them. Additionally, both tool implementations provide a hand-crafted concrete syntax parser, which implements the actual EBNF grammar. Maintaining these different parts and updating the manually created ones incurs significant effort whenever the language should be evolved.

**Grammar generation and Xtext** A much more streamlined approach to language engineering would, instead, use a single meta-model and use this in a model-driven approach to derive the concrete syntax directly from it. With the exception of EMFText [203] and the Grasland toolkit [204] that are both not maintained anymore, Xtext is currently the only textual DSL framework

that allows generating a grammar from a meta-model. Using an EBNF-based Xtext grammar, Xtext applies the ANTLR parser generator framework [205] to derive the actual parser and all its required inputs. It also generates editors along with syntax highlighting, code validation, and other useful tools.

A language engineer has two options when constructing a new language from a meta-model in Xtext:

- **Hand-craft a grammar** that maps syntactical elements of the textual concrete syntax to the concepts of the abstract syntax. This is the way many DSLs have been built in Xtext (e.g., Xcore [188], Spectra [187], and Xenia [189]). However, this approach is not very robust when the meta-model changes since the grammar needs to be adapted manually to that meta-model change.
- **Generate a grammar** from the meta-model using Xtext’s built-in functionality (we call this grammar *generated grammar* in this paper). This creates a grammar that contains grammar rules for all meta-model elements that are contained in a common root node and resolves references, etc., to a degree (see Section 5.4.3 for details). This approach deals very well with meta-model changes and only requires the re-generation of the grammar which is very fast and can be automated. However, the grammar is going to be very verbose, structured extensively using braces, and uses a lot of keywords. Such a situation is shown in Figure 5.1, depicting an instance of the generated grammar for EAST-ADL. This makes it difficult to use such a generated grammar in practice.

In this paper, we focus on making the second option more usable to give language engineers the ability to quickly re-generate their grammars when the meta-model changes, e.g., for rapid prototyping or for language evolution. Thus, we provide the ability to optimize the automatically generated grammars to improve their usability and make them similar in this regard to hand-crafted grammars. We show that this optimization can be re-applied to evolving versions of the language. Our contribution, GRAMMAROPTIMIZER, therefore combines the advantages of both approaches while mitigating their respective disadvantages.

## 5.3 Related Work

In the following, we discuss approaches for grammar optimization, approaches that are concerned with the design and evolution of DSLs, and other approaches.

**Grammar Optimization** There are a few works that aim at optimizing grammar rules with a focus on XML-based languages. For example, [153, 206] also mention optimization of grammar rules in Xtext. Their approach XMLText and the scope of their optimization focus only on XML-based languages. They convert an XML schema definition to a meta-model using the built-in capabilities of EMF. Based on that meta-model, they then use an adapted Xtext grammar generator for XML-based languages to provide more human-friendly notations for editing XML files. XMLText thereby acts as a sort of compiler add-on to enable editing in a different notation and to automatically translate

```

1 EAXML
2 {
3   topLevelPackage
4   {
5     EAPackage
6     {
7       shortName Structure
8       subPackage
9       {
10        EAPackage
11        {
12          shortName DesignPkg
13          subPackage
14          {
15            EAPackage
16            {
17              shortName FcnDesignArchitecture_new
18              element
19              {
20                DesignFunctionType
21                {
22                  shortName FDAWithController_new
23                  part
24                  {
25                    DesignFunctionPrototype
26                    {
27                      shortName wiperCtrlBasic
28                      type "Structure.DesignPkg.FcnDesignArchitecture_new
29                    },
30                    DesignFunctionPrototype
31                    {
32                      shortName wiperCtrlBasic018
33                      type "Structure.DesignPkg.FcnDesignPkg_new.WiperCtrl
34                    }
35                  }
36                },
37                DesignFunctionType
38                {
39                  shortName testtvp0819

```

Figure 5.1: Instance of the generated grammar for EAST-ADL.

to XML and vice versa. In contrast, we develop a post-processing approach that enables the optimization of any Xtext grammar, not only XML-based ones, cf. also our discussion in Section 5.8).

The approach of [207] shares the same goal and a similar functional principle as XMLText, but uses other technological frameworks. In contrast to XMLText, Chodarev supports more straightforward customization of the target XML language by directly annotating the meta-model that is generated from the XML schema. The same distinction applies here as well: GRAMMAROPTIMIZER enables the optimization of any Xtext grammar and is not restricted to XML-based languages.

Grammar optimization for DSLs in general is addressed by [208]. They propose an approach to specify a syntax for textual, meta-model-based DSLs with a dedicated DSL called Textual Concrete Syntax, which is based on a meta-model. From such a syntax specification, a concrete grammar and a parser are generated. The approach is similar to a template language restricting the language engineer and thereby, as the authors state, lacks the freedom of grammar specifications in terms of syntax customization options. In contrast, we argue that the GRAMMAROPTIMIZER provides more syntax customization options to achieve a well-accepted textual DSL.

Finally, [209] designed a model-driven Xtext pretty printer, which is used for improving the readability of the DSL by means of improved, language-specific, and configurable code formatting and syntax highlighting. In contrast, our GRAMMAROPTIMIZER is not about improving code readability but focused on

how to design the DSL itself to be easy to use and user-friendly.

**Designing and Evolving Meta-model-based DSLs** Many papers about the design of DSLs focus solely on the construction of the abstract syntax and ignore the concrete syntaxes (e.g., [190, 210]), or focus exclusively on graphical notations (e.g., [191, 211]). In contrast, the guidelines proposed by [212] contain specific ideas about concrete syntax design, e.g., to “balance compactness and comprehensibility”. Arguably, the languages automatically generated by Xtext are neither compact nor comprehensible and therefore require manual changes.

[168] acknowledge that DSL design is not a sequential process. The paper also mentions the importance of textual concrete syntaxes to support common editing operations as well as the reuse of existing languages. Likewise, [213] describe DSL development as an iterative process and use EMF and Xtext for the textual syntax of the DSL. They also discuss the evolution of the language, and that “it is hard to predict which language features will improve understandability and modifiability without actually using the language”. Again, this is an argument for the need to do prototyping when developing a language. [214] broadens the scope and also argues for the need for evolving DSLs along with the “engineering environment” they are situated in, including editors and code generators. [215] also acknowledge the “constant need for evolution” of DSLs.

There is a lot of research supporting different aspects of language change and evolution. Existing approaches focus on how diverse artifacts can be co-evolved with evolving meta-models, namely the models that are instances of the meta-models [19], OCL constraints that are used to specify static semantics of the language [20, 216], graphical editors of the language [217, 218], and model transformations that consume or produce programs of the language [219]. Specifically, the evolution of language instances with evolving meta-models is well supported by research approaches. For example, Di Ruscio et al. [218] support language evolution by using model transformations to simultaneously migrate the meta-model as well as model instances.

Thus, while these approaches cover a lot of requirements, there is still a need to address the evolution of textual grammars with the change of the meta-model as it happens during rapid prototyping or normal language evolution. This is a challenge, especially since fully generated grammars are usually not suitable for use in practice. This implies that upon changing a meta-model, it is necessary to co-evolve a manually created grammar or a grammar that has been generated and then manually changed. GRAMMAROPTIMIZER has been created to support prototyping and evolution of DSLs and is, therefore, able to support and largely automate these activities.

**Other Approaches** As we mentioned above, besides Xtext, there are two more approaches that support the generation of EBNF-based grammars and from these the generation of the actual parsers. These are EMFText [203] and the Grasland toolkit [204], which are both not maintained anymore.

Whereas our work focuses on the Eclipse technology stack based on EMF and Xtext, there are a number of other language workbenches and supporting tools that support the design of DS(M)Ls and their evolution. However, none of these approaches are able to derive grammars directly from meta-models, a



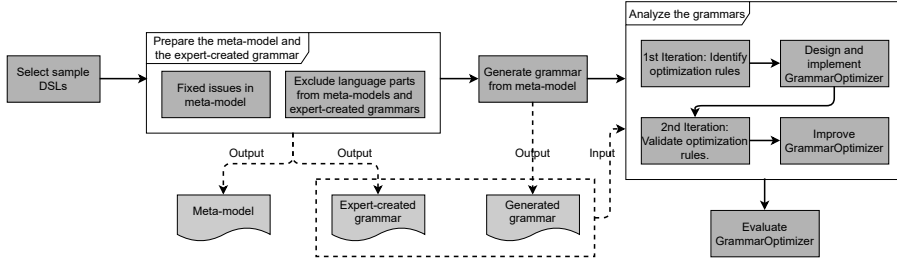


Figure 5.2: Overview of our methodology.

prerequisite for the approach to language engineering we propose and the basis of our contribution, GRAMMAROPTIMIZER. Instead, tools like textX [220] go the other way around and derive the meta-model from a grammar. Langium [221] is the self-proclaimed Xtext successor without the strong binding to Eclipse, but does not support this particular use case just yet and instead focuses on language construction based on grammars. MetaEdit+ [222] does not offer a textual syntax for the languages, but instead a generator to create text out of diagrams that are modeled using either tables, matrices, or diagrams. JetBrains MPS [42] is based on projectional editing where concrete syntaxes are projections of the abstract syntax. However, these projections are manually defined and not automatically derived from the meta-model as it is the case with Xtext. Finally, [215] propose an approach to evolve DSLs including their concrete syntaxes and instances. For that, they present “evolution languages” that evolve the concrete syntax separately. However, they focus on DSLs that are built with classical compilers and not with meta-models.

## 5.4 Methodology

In this section, we describe our research methodology, shown in an overview in Fig. 5.2. Our methodology consists of a number of sequential steps, in particular: selecting the case languages, preparing metamodels and grammars (including the exclusion of certain parts of the language), and two iterations of analysis, including extraction of grammar optimization rules and tool development. We now describe all of these steps in detail.

### 5.4.1 Selection of Sample DSLs

We selected a number of DSLs for which both an expert-created grammar and a meta-model were available. Our key idea was that the expert-created grammar serves as a *ground truth*, specifying what a desirable target of an optimization process would look like. As the starting point for this optimization process, we considered the Xtext-generated grammars for the available meta-models. The goal of our grammar optimization rules was to support an automated transformation to turn the Xtext-generated grammar into the expert-created grammar. By selecting a number of DSLs with a grammar or precise syntax definition from which we could derive such a ground truth, we aimed to generalize the grammar optimization rules so that new languages can be

optimized based on rules that we include in `GRAMMAROPTIMIZER`.

**Sources** To find language candidates, we collected well-known languages, such as DOT, and used language collections, such as the Atlantic Zoo [176], a list of robotics DSLs [223], and similar collections [181, 224, 225, 226, 227]. However, it turned out that the search for suitable examples was not trivial despite these resources. The quality of the meta-models in these collections was often insufficient for our purposes. In many cases, the meta-model structures were too different from the grammars or there was no grammar in either Xtext or in EBNF publicly available as well as no clear syntax definition by other means. We therefore extended our search to also use Github’s search feature to find projects in which meta-models and Xtext grammars were present and manually searched the Eclipse Foundation’s Git repositories for suitable candidates. Grammars were either taken from the language specifications or from the repositories directly.

**Concrete Grammar Reconstruction for BibTeX** In some cases, the syntax of a language is described in detail online, but no EBNF or Xtext grammar can be found. In our case, this is the language BibTeX. It is a well-known language to describe bibliographic data mostly used in the context of typesetting with LaTeX that is notable for its distinct syntax. In this case, we utilized the available detailed descriptions [184] to reconstruct the grammar. To validate the grammar we created, we used a number of examples of bibliographies from [184] and from our own collection to check that we covered all relevant cases.

**Meta-model Reconstruction for DOT** DOT is a well-known language for the specification of graph models that are input to the graph visualization and layouting tool Graphviz. Since it is an often used language with a relatively simple, but powerful syntax, we decided to include it, even if we could not find a complete meta-model that contains both the graph structures and formatting primitives. The repository that also contains the grammar we ended up using [228], e.g., only contains meta-models for font and graph model styles.

Therefore, we used the Xtext grammar that parses the same language as DOT’s expert-created grammar to derive a meta-model [228]. Xtext grammars include more information than an EBNF grammar, such as information about references between concepts of the language. Thus, the fact that the DOT grammar was already formulated in Xtext allowed us to directly generate DOT’s Ecore meta-model from this Xtext grammar. This meta-model acquisition method is an exception in this paper. Since this paper focuses on how to optimize the generated grammar, we consider this way of obtaining the meta-model acceptable for this one case.

**Selected Cases** As a result, we identified a sample of seven DSLs (cf. Table 5.1), which has a mix of different sources for meta-models and grammars. This convenience sampling consists of a mix of well-known DSLs with lesser-known, but well-developed ones. We believe this breadth of domains and language styles is broad enough to extract a generically applicable set of

candidate optimization rules for GRAMMAROPTIMIZER. We analyzed these selected languages in two iterations, the 1st analyzing four of them and the 2nd analyzing the remaining three. In Table 5.1, we list all seven languages, including information about the meta-model (source and the number of classes in the meta-model) and the expert-created grammar (source and the number of grammar rules).

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna.

Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.

### 5.4.2 Exclusion of Language Parts for Low-level Expressions

Two of the analyzed languages encompass language parts for expressions, which describe low-level concepts like binary expressions (e.g., addition). We excluded such language parts in ATL and in SML due to several aspects. Both languages distinguish the actual language part and the expression language part already on the meta-model level and thereby treat the expression language part differently. The respective expression parts are similarly large than the actual languages (i.e., 56 classes for the embedded OCL part of ATL and 36 classes for the SML scenario expressions meta-model), which implies a high analysis effort. Finally, although having a significantly large meta-model, the embedded OCL part of ATL does not specify the expressions to a sufficient level of detail (e.g., it does not allow to specify binary expressions). Therefore, we excluded such language parts by introducing a fake class `OCLDummy`. The details for the exclusion is described in the supplemental material [232]<sup>1</sup>.

**Exclusion from the Grammar** In addition, we need to ensure that we can compare the language without the excluded parts to the expert-created grammar. To do so, we derive versions of the expert-created grammars in which these respective language parts are substituted by a dummy grammar rule, e.g., `OCLDummy` in the case of ATL. This dummy grammar rule is then called everywhere where a rule of the excluded language part would have been called.

### 5.4.3 Meta-model Preparations and Generating an Xtext Grammar

The first step of the analysis of any of the languages is to generate an Xtext grammar based on the language's meta-model. This is done by using the Xtext project wizard within Eclipse.

Note that it is sometimes necessary to slightly change the meta-model to enable the generation of the Xtext grammar or to ensure that the compatibility with the expert-created grammar can be reached. These changes are necessary in case the meta-model is already ill-formed for EMF itself (e.g., purely descriptive

<sup>1</sup>See folder "Section\_4\_Methodology"

Table 5.1: DSLs used in this paper, the sources of the meta-model and the grammar used, as well as the size of the meta-model and grammar. The first set of DSLs was analyzed to derive necessary optimization rules, and the second set to validate the candidate optimization rules and extend them if necessary.

Iteration	DSL	Meta-model		Expert-created Grammar		Generated Grammar		
		Source	Classes <sup>1</sup>	Source	Rules	lines	rules	calls
1st	ATL <sup>2</sup>	Atlantic Zoo [176]	30	ATL Syntax [183]	28	275	30	232
	BitTex	Grammarware [229]	48	Self-built Based on [184]	46	293	48	188
	DOT	Generated	19	Dot [185]	32	125	23	51
	SML <sup>3</sup>	SML repository [186]	48	SML repository [186]	45	658	96	377
2nd	Spectra	GitHub Repository [230]	54	GitHub Repository [187]	58	442	62	243
	Xcore	Eclipse [231]	22	Eclipse [188]	26	243	33	149
	Xenia	GitHub Repository [174]	13	GitHub Repository [189]	13	84	15	36

<sup>1</sup> After adaptations, containing both classes and enumerations.  
<sup>2</sup> Excluding embedded OCL rules.  
<sup>3</sup> Excluding embedded SML expressions rules.

Listing 5.1: EBNF rule `edge_stmt` from the expert-created grammar for DOT

```
edge_stmt : (node_id | subgraph) edgeRHS [ attr_list ]
```

Ecore files that are not intended for instantiating runtime models) or if it does not adhere to certain assumptions that Xtext makes (e.g., no bidirectional references). The method of metamodel modification is described in detail in our supplementary material [232]<sup>2</sup>.

In Table 5.1, we list how many lines, rules, and calls between rules the generated grammars included for the seven languages.

#### 5.4.4 Comparing EBNF and Xtext grammars

As a prerequisite for our analysis of grammars, we present a strategy for dealing with a noteworthy aspect of our methodology: in several cases, we dealt with languages where the expert-created grammar was available in EBNF, whereas our contribution targets Xtext, which augments EBNF with additional technicalities, such as cross-references and datatypes. Hence, to validate whether our approach indeed produces grammars that are equivalent to expert-created ones, we needed a concept that allows comparing EBNF to Xtext grammars.

To this end, we introduce the concept of *imitation*. Imitation is a form of semantic equivalence of grammars that abstracts from Xtext-specific technicalities. Specifically, we consider a set of EBNF rules  $\{rr_x | 1 \leq x \leq n\}$  to be *imitated* by a set of Xtext rules  $\{ro_y | 1 \leq y \leq m\}$  if both produce the exact same language, modulo Xtext-specific details. Note that the cardinalities  $n$  and  $m$  may differ due to situations in which one expert-created rule is replaced by several optimized rules in concert, explained below.

Like semantic equivalence of context-free grammars, in general, [233], imitation is undecidable if two arbitrary grammars are considered. However, in the scope of our analysis, we deal with specific cases that come from our evaluation subjects. These are generally of the following form: 1. Two syntactically identical—and thus, inherently semantically equivalent—grammar rules 2. Situations in which a larger rule from the first grammar is, in a controlled way, split up into several rules in the second grammar. For these, we consider them as equivalent based on a careful manual analysis, explained later.

For example, the rule `edge_stmt` shown in Listing 5.1 is imitated by the combination of the rules `EdgeStmtNode` and `EdgeStmtSubgraph` shown in Listing 5.2. Merging the Xtext rules to form one rule, like the EBNF counterpart, was not possible in this case, due to the necessity of specifying a distinct return type in Xtext, which is not required in EBNF. In addition, the Xtext rules contain Xtext-specific information for dealing with references and attribute types, which is not present in the EBNF rule.

---

<sup>2</sup>See directory “Section\_4\_Methodology”.

Listing 5.2: Xtext rules `EdgeStmtNode` and `EdgeStmtSubgraph` from the optimized generated grammar

```
EdgeStmtNode returns EdgeStmtNode:
    {EdgeStmtNode}
    node=NodeId
    (edgeRHS+=EdgeRhs)+
    (attrLists+=AttrList)*
    ;

EdgeStmtSubgraph returns EdgeStmtSubgraph:
    {EdgeStmtSubgraph}
    subgraph=Subgraph
    (edgeRHS+=EdgeRhs)+
    (attrLists+=AttrList)*
    ;
```

### 5.4.5 Analysis of Grammars

We performed the analysis of existing languages in two iterations. The first iteration was purely exploratory. Here we analyzed four of the languages with the aim of finding as many candidate grammar optimization rules as possible. In the second iteration, we selected three additional languages to validate the candidate rules collected from the first iteration, add new rules if necessary, and generalise the existing rules when applicable.

Our general approach was similar in both iterations. Once we had generated a grammar for a meta-model, we created a mapping between that generated grammar and the expert-created grammar of the language. The goal of this mapping was to identify which grammar rules in the generated grammar correspond to which grammar rules in the expert-created grammar. Note that a grammar rule in the generated grammar may be mapped to multiple grammar rules in the expert-created grammar and vice versa. From there, we inspected the generated and expert-created grammars to identify how they differed and which changes would be required to adjust the generated grammar so that it produces the same language as the expert-created grammar, i.e., *imitates* the expert-created grammar rules. We documented these changes per language and summarized them as optimization rule candidates in a spreadsheet.

For example, the expert-created grammar rule `node_stmt` in DOT (see Listing 5.3) maps to the generated grammar rule `NodeStmt` in Listing 5.4. Multiple changes are necessary to adjust the generated Xtext grammar rule:

- Remove all the braces in the grammar rule `NodeStmt`.
- Remove all the keywords in the grammar rule `NodeStmt`.
- Remove the optionality from all the attributes in the grammar rule `NodeStmt`.
- Change the multiplicity of the attribute `attrLists` from `1..*` to `0..*`.

Note that in most cases the expert-created grammar was written in EBNF instead of Xtext. For example, the `returns` statement in line 1 of Listing 5.4 is required for parsing in Xtext. We took that into account when comparing both grammars.

Listing 5.3: Non-terminal `node_stmt` in the expert-created grammar of DOT, in EBNF

```
node_stmt : node_id [ attr_list ]
```

Listing 5.4: Grammar rule `NodeStmt` in the generated grammar of DOT, in Xtext

```
NodeStmt returns NodeStmt:
    {NodeStmt}
    'NodeStmt'
    '{'
        ( 'node' node=NodeId)?
        ( 'attrLists' '{' attrLists+=AttrList ( ","
            attrLists+=AttrList)* '}' )?
    '}' ;
```

#### 5.4.5.1 First Iteration: Identify Optimization Rules

The analysis of the grammars of the four selected DSLs in the first iteration had two concrete purposes:

- identify the differences between the expert-created grammar and generated grammar of the language;
- derive grammar optimization rules that can be applied to change the generated grammar so that the optimized grammar parses the same language as the expert-created grammar.

Please note that it is not our aim to ensure that the optimized grammar itself is identical to the expert-created grammar. Instead, our goal is that the optimized grammar is an *imitation* of the expert-created grammar and therefore is able to parse the same language as the original, usually hand-crafted grammar of the DSL. Each language was assigned to one author who performed the analysis.

As a result of the analysis, we obtained an initial set of grammar optimization rules, which contained a total of 58 candidate optimization rules. Table 5.2 summarizes in the second column the number of identified rule candidates and in the second row the number for the first iteration. Since the initial set of grammar optimization rules was a result of an analysis done by multiple authors, it included rules that were partially overlapping and rules that turned

Listing 5.5: Grammar rule `NodeStmt` in the optimized grammar of DOT, in Xtext

```
NodeStmt returns NodeStmt:
    {NodeStmt}

    node=NodeId
    ( attrLists+=AttrList)*
    ;
```

Table 5.2: Summary of identified rules their rule variants and their sources

Iteration	Rule Candidates	Selected Rules	Rule Variants
Iteration 1	58	46	57
Iteration 2	10	10	10
Intermediate sum	68	56	67
Evaluation	4	4	4
Overall sum	72	60	71

out to only affect the grammar’s formatting, but not the language specified by the grammar. Thus, we filtered rules that belong to the latter case. For rule candidates that overlapped with each other, we selected a subset of the rules as a basis for the next step. This filtering led to a selection of 46 optimization rules (cf. third column in Table 5.2).

We processed these 46 selected optimization rules to identify required *rule variants* that could be implemented directly by means of one Java class each, which we describe more technically as part of our design and implementation elaboration in Section 5.6.2. For identifying the rule variants, we focused on the following aspects:

- **Specification of scope.** Small changes in the meta-model might lead to a different order of the lines in the generated grammar rules or even a different order of the grammar rules. Therefore, the first step was to define a suitable concept to identify the parts of the generated grammar that can function as the *scope* of an optimization rule, i.e., where it applies. We identified different suitable scopes, e.g., single lines only, specific attributes, specific grammar rules, or even the whole grammar. Initially, we identified separate rule variants for each scope. Note that this also increased the number of rule variants, as for some rule candidates multiple scopes are possible.
- **Allowing multiple scopes.** In many cases, selecting only one specific scope for a rule is too limiting. In the example above (Listing 5.4), pairs of braces in different scopes are removed: in the scope of the attribute `attrLists` in line 6 and in the scope of the containing grammar rule in lines 4 and 7. This illustrates that changes might be applied at multiple places in the grammar at once. When formulating rule variants, we analyzed the rule candidates for their potential to be applied in different scopes. When suitable, we made the scope configurable. This means that only one optimization rule variant is necessary for both cases in the example. Depending on the provided parameters, it will either replace the braces for the rule or for specific attributes.
- **Composite optimization rules.** We decided to avoid optimization rule variants that can be replaced or composed out of other rule variants, especially when such compositions were only motivated by very few cases. However, such rules might be added again later if it turns out they are needed more often.



Listing 5.6: Two attributes in the grammar rule `XOperation` in the generated grammar of Xcore

```
...      (unordered?= 'unordered ')?
          (unique?= 'unique ')?
...      
```

While we identified exactly one rule variant for most of the selected optimization rules, we added more than one rule variant for several of the rules. We did this when slight variations of the results were required. For example, we split up the optimization rule `SubstituteBrace` into the three variants, i.e., 1) `ChangeBracesToParentheses`, 2) `ChangeBracesToSquare`, and 3) `ChangeBracesToAngle`. Note that this split-up into variants is a design choice and not an inherent property of the optimization rule, as, e.g., the type of target bracket could be seen as nothing more than a parameter of the rule. As a result, we settled on 57 rule variants for the 46 identified rules (cf. fourth column of second row in Table 5.2).

#### 5.4.5.2 Second iteration: Validate Optimization Rules

The last step left us with 46 selected optimization rules from the first iteration (cf. second row in Table 5.2). We developed a preliminary implementation of `GRAMMAROPTIMIZER` by implementing the 57 rules variants belonging to these 46 optimization rules (we will describe the implementation in the *Solution* section). To validate this set of optimization rules, we performed a second iteration. In the second iteration, we selected the three DSLs Spectra, Xenia, and Xcore. As in the first iteration, we generated a grammar from the meta-model, analyzed the differences between the generated grammar and the expert-created grammar, and identified optimization rules that need to be applied to the generated grammar to accommodate these differences. In contrast to the first iteration, we aimed at utilizing as many existing optimization rules as possible and only added new rule candidates when necessary.

We configured the preliminary `GRAMMAROPTIMIZER` for the new languages by specifying which optimization rules to apply on the generated grammar. The execution results showed that the existing optimization rules were sufficient to change the generated grammar of Xenia to imitate the expert-created grammar used as the ground truth. However, we could not fully transform the generated grammar of Xcore and Spectra with the preliminary set of 46 optimization rules from the first iteration. For example, Listing 5.6 shows two attributes `unordered` and `unique` in the grammar rule `XOperation` in the generated grammar for Xcore. However, in the expert-created grammar, the rule portions for the two attributes each refer to the other attribute in a way that allows using the keywords in several possible orders, as shown in Listing 5.7. This optimization could not be performed with the optimization rules from the first iteration.

Based on the non-optimized parts of the grammars of Xcore and Spectra, we identified another ten optimization rules for the `GRAMMAROPTIMIZER`. These ten newly identified optimization rules optimize all the non-optimized

Listing 5.7: Two attributes in the grammar rule `XOperation` in the expert-created grammar of Xcore

```
...
        unordered?='unordered ' unique?='unique '? |
        unique?='unique ' unordered?='unordered '?
...
```

parts of the grammar of Xcore, including, e.g., optimizing the grammar in Listing 5.6 to Listing 5.7. These new optimization rules also optimize part of the non-optimized parts of the grammar of Spectra. We will interpret the remaining non-optimized parts in the *Evaluation* section. In the end, after two iterations, we identified a total of 56 optimization rules (which will be implemented by a total of 67 rule variants) (cf. fourth row in Table 5.2).

## 5.5 Identified Optimization Rules

In total, we identified 56 distinct optimization rules for the grammar optimization after the 2nd iteration, which we further refined into 67 rule variants (cf. fourth row in Table 5.2). Note that 4 additional rules were identified during the evaluation (this will be interpreted in the *Evaluation* section), increasing the final number of identified optimization rules to 60 (cf. bottom row in Table 5.2) and the final number of rule variants to 71.

Table 5.3 shows some examples of the optimization rules. The rules we implemented can be categorized by the primitives they manipulate: grammar rules, attributes keywords, braces, multiplicities, optionality (a special form of multiplicities), grammar rule calls, import statements, symbols, primitive types, and lines. They either ‘add’ things (e.g., *AddKeywordToRule*), ‘remove’ things (e.g., *RemoveOptionality*), or ‘change’ things (e.g., *ChangeCalledRule*). All optimization rules ensure that the resulting changed grammar is still valid and syntactically correct Xtext.

Most optimization rules are ‘scoped’ which means that they only apply to a specific grammar rule or attribute. In other cases, the scope is configurable, depending on the parameters of the optimization rule. For instance, the *RenameKeyword* rule takes a grammar rule and an attribute as a parameter. If both are set, the scope is the given attribute in the given rule. If no attribute is set, the scope is the given grammar rule. If none of the parameters is set, the scope is the entire grammar (“Global”). All occurrences of the given keyword are then renamed inside the respective scope.

Changes to optionality are used when the generated grammar defines an element as mandatory, but the element should be optional according to the expert-created grammar. This can apply to symbols (such as commas), attributes, or keywords. Additionally, when all attributes in a grammar rule are optional, we have an optimization rule that makes the container braces and all attributes between them optional. This optimization rule allows the user of the language to enter only the grammar rule name and nothing else, e.g., “`EAPackage DataTypes;`”.

Likewise, GRAMMARTOPTIMIZER contains rules to manipulate the multi-

Table 5.3: Excerpt of implemented grammar optimization rules. A configurable scope (“Config.”) means that, depending on provided parameters, the rule either applies globally to a specific grammar rule or to a specific attribute.

Subject	Op.	Rule	Scope
Keyword	Add	<i>AddKeywordToAttr</i> <i>AddKeywordToRule</i> <i>AddKeywordToLine</i>	Attribute Rule Line
	Change	<i>RenameKeyword</i> <i>AddAlternativeKeyword</i>	Config. Rule
Rule	Remove	<i>RemoveRule</i>	Global
	Change	<i>RenameRule</i> <i>AddSymbolToRule</i>	Rule Rule
Optionality	Add	<i>AddOptionalityToAttr</i> <i>AddOptionalityToKeyword</i>	Attribute Config.
Import	Add	<i>AddImport</i>	Global
	Remove	<i>RemoveImport</i>	Global
Brace	Change	<i>ChangeBracesToSquare</i>	Attribute
	Remove	<i>RemoveBraces</i>	Config.

plicities in the generated grammars. The meta-models and the expert-created grammars we used as inputs do not always agree about the multiplicity of elements. We provide optimization rules that can address this within the constraints allowed by EMF and Xtext.

For the example in Listing 5.4, this means that the necessary changes to reach the same language defined in Listing 5.3 can be implemented using the following GRAMMAROPTIMIZER rules:

- *RemoveBraces* is applied to the grammar rule `NodeStmt` and all of its attributes. This removes all the curly braces (`{` and `}` in lines 4, 6, and 7) within the grammar rule.
- *RemoveKeyword* is applied to the grammar rule `NodeStmt` and all of its attributes. This removes the keywords `‘NodeStmt’`, `‘node’` and `‘attrLists’` (lines 3, 5, and 6) from this grammar rule.
- *RemoveOptionality* is applied to both attributes. This removes the question marks (`‘?’`) in lines 5 and 6.
- *convert1toStarToStar* is applied to the attribute `attrLists`. This rule changes line 6. Before this change, this line is `“attrLists+=AttrList ( "," attrLists+=AttrList)*”` (the braces, keyword `‘attrLists’` and the optionality `‘?’` have been removed by previous optimization rules). After this change, it becomes `(attrLists+=AttrList)*`. Note that the DOT grammar is specified using a syntax that is slightly different from standard EBNF. In that syntax, square brackets (`[` and `]`) enclose optional items [185].

Note that line 2 in Listing 5.4 has no effect on the syntax of the grammar but is required by and specific to Xtext, so that we do not adapt such constructs. After the above steps, the grammar rule `NodeStmt` is adapted from Listing 5.4 to Listing 5.5.

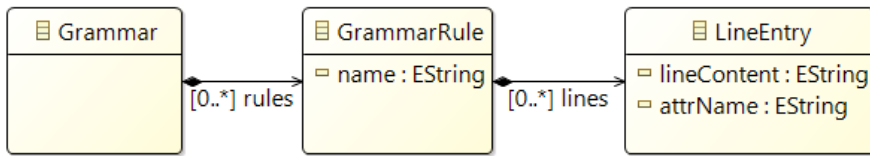


Figure 5.3: The class design for representing grammar rules.

## 5.6 Solution: Design and Implementation

The GRAMMAROPTIMIZER is a Java library that offers a simple API to configure optimization rule applications and execute them on Xtext grammars. The language engineer can use that API to create a small program that executes GRAMMAROPTIMIZER, which in turn will produce the optimized grammar.

### 5.6.1 Grammar Representation

We designed GRAMMAROPTIMIZER to parse an Xtext grammar into an internal data structure which is then modified and written out again. This internal representation of the grammar follows the structure depicted in Figure 5.3. A **Grammar** contains a number of **GrammarRules** that can be identified by their names. In turn, a **GrammarRule** consists of a sorted list of **LineEntry**s with their textual **lineContent** and an optional **attrName** that contains the name of the attribute defined in the line. Note that we utilize the fact that Xtext generates a new line for each attribute.

### 5.6.2 Optimization Rule Design

Internally, all optimization rules derive from the abstract class **OptimizationRule** as shown in Figure 5.4. Derived classes overwrite the **apply()**-method to perform the specific text modifications for this rule. By doing so, the specific rule can access the necessary information through the class members: **grammar** (i.e., the entire grammar representation as explained in Section 5.6.1 and depicted in Figure 5.3), **grammarRuleName** (i.e., the name of the specified grammar rule that a user wants to optimize exclusively), and **attrName** (i.e., the name of an attribute that a user wants to optimize exclusively). Sub-classes can also add additional members if necessary. This architecture makes the GRAMMAROPTIMIZER extensible, as new optimization rules can easily be defined in the future.

We built the optimization rules in a model-based manner by first creating the meta-model shown in Figure 5.4 and then using EMF to automatically generate the class bodies of the optimization rules. This way we only needed to overwrite the **apply()**-method for the concrete rules. Internally, the **apply()**-methods of our optimization rules are implemented using regular expressions. Each optimization rule takes a number of parameters, e.g., the name of the grammar rule to work on or an attribute name to identify the line to work on. In addition, some optimization rules take a list of exceptions to the scope. For example, the optimization rule to remove braces can be applied to a global scope (i.e., all grammar rules) while excluding a list of specific grammar rules

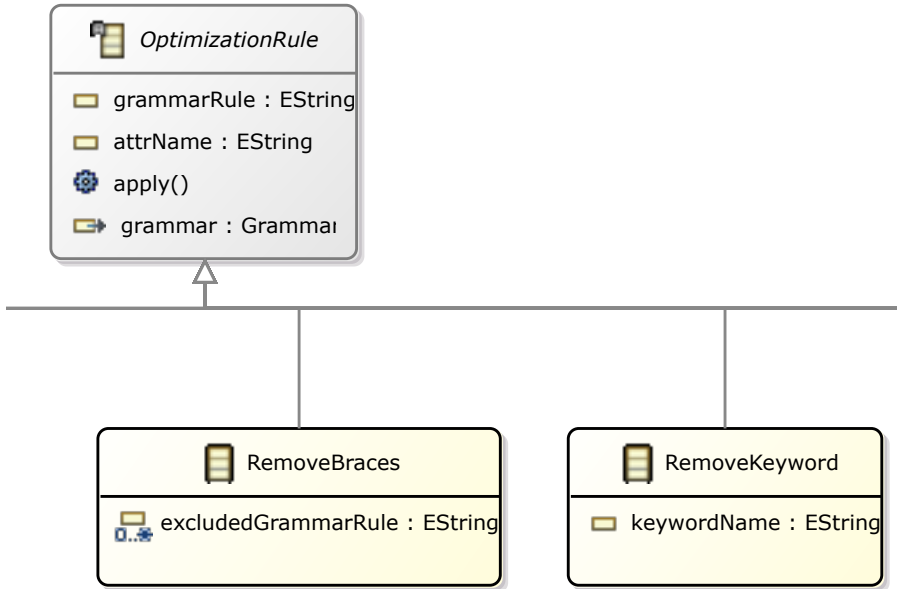


Figure 5.4: Excerpt of the class diagram for optimization rules.

from the processing. This allows to configure optimization rule applications in a more efficient way. We implemented all identified optimization rules.<sup>3</sup> For testing, we built a comprehensive test suite, based on the optimized grammars considered in our design methodology. We created one test case per scenario, to ensure that the grammar produced by our implementation after applying a full given configuration to an Xtext-generated grammar exactly matches an expected ground-truth grammar, for which we previously manually established that it agrees (in the sense of *imitation*) with an expert-created one).

### 5.6.3 Configuration

The language engineer has to configure what optimization rules the **GRAMMAROPTIMIZER** should apply and how. This is supported by the API offered by **GRAMMAROPTIMIZER**. Listing 5.8 shows an example of how to configure the optimization rule applications in a method `executeOptimization()`, where the configuration revisits the DOT grammar optimization example transforming Listing 5.4 into Listing 5.3. The lines 3 to 6 configure optimization rule applications. For example, line 3 removes all curly braces in the grammar rule *NodeStmt*. The value of the first parameter is set to “NodeStmt”, which means that the operation of removing curly braces will occur in the grammar rule *NodeStmt*. If this first parameter is set to “null”, the operation would be executed for all grammar rules in the grammar. The second parameter is used to indicate the target attribute. Since it is set to “null”, all lines in the targeted grammar rule will be affected. However, if the parameter is set to a name of an attribute, only curly braces in the line containing that attribute

<sup>3</sup>See folder ‘1\_Source\_Code/org.bumble.xtext.grammaroptimizer’ in our supplemental material [232], which contains the ‘optimizationrule’ project with the full implementation.

Listing 5.8: Excerpt of the configuration of GRAMMAROPTIMIZER for the QVTo 1.0 language.)

```
public static boolean executeOptimization(GrammarOptimizer go) {
    ...
    go.removeBraces("NodeStmt", null, null);
    go.removeKeyword("NodeStmt", null, null, null);
    go.removeOptionality("NodeStmt", null);
    go.convert1toStarToStar("NodeStmt", "attrLists");
    ...
}
```

will be removed. Finally, the third parameter can be used to indicate names of attributes for which the braces should not be removed. This can be used in case the second parameter is set to “null”.

Similarly, the optimization rule application in line 4 is used to remove all keywords in the grammar rule *NodeStmt*. Again, the second parameter can be used to specify which lines should be affected using an attribute. The third parameter is used to indicate the target keyword. Since it is set to “null”, all keywords in the targeted lines will be removed. However, if the keyword is set, only that keyword will be removed. The last parameter can be used to indicate names of attributes for which the keyword should not be removed. This can be used in case the second parameter is set to “null”.

Line 5 is used to remove the optionality from all lines in the grammar rule *NodeStmt*. If the second parameter gets an argument that carries the name of an attribute, the optionality is removed exclusively from the grammar line specifying the syntax for this attribute.

Finally, line 6 changes the multiplicity of the attribute **attrLists** in the grammar rule *NodeStmt* from  $1..*$  to  $0..*$ . If the second parameter would get the argument “null”, this adaptation would have been executed to all lines representing the respective attributes.

## 5.6.4 Execution

Once the language engineer has configured GRAMMAROPTIMIZER, they can invoke the tool using **GrammarOptimizerRunner** on the command line and providing the paths to the input and output grammars there. Alternatively, instead of invoking GRAMMAROPTIMIZER via the command line and modifying **executeOptimization()**, it is also possible to use JUnit test cases to access the API and optimize grammars in known locations. This is the approach we have followed in order to generate the results presented in this paper.

Figure 5.5 uses the first optimization operation from Listing 5.8 removing curly braces as an example to depict how GRAMMAROPTIMIZER works internally when optimizing grammars. The top of the figure shows an example input, which is the grammar rule *NodeStmt* generated from the meta-model of DOT (cf. Listing 5.4). In the lower right corner, the resulting optimized Xtext grammar rule is illustrated. In both illustrated grammar rule excerpt, blue fonts are the keywords and symbols (braces and commas).

In **Step 1 (initialization)**, GRAMMAROPTIMIZER builds a data structure out of the grammar initially generated by Xtext. That is, it builds a **:Grammar**



(i.e., “ ‘{’ ” and “ ‘}’ ”), it removes them.

Finally, in **Step 3 (finalization)**, the `GRAMMAROPTIMIZER` writes the complete data structure containing the optimized grammar rules to a new file by means of the call `setFileText(...)`.

After the execution of these steps, the optimized versions of the grammar is ready for use. The typical next step is to re-generate the parser, textual editor and other artifacts for the grammar via Xtext. We recommend that the language engineer should systematically test the resulting grammar to check whether it matches their expectations, based on the generated artifacts and a test suite with diverse language instances. After evolution steps, previously developed tests can act as regression tests.

### 5.6.5 Post-Processing vs. Changing Grammar Generation

`GRAMMAROPTIMIZER` is designed to modify grammars that Xtext generated out of meta-models. An alternative to this post-processing approach is to directly modify the Xtext grammar generator as, e.g., in `XMLText` [153, 206]. However, we deliberately chose a post-processing approach, because the application of conventional regular expressions enables the transferability to other recent language development frameworks like `Langium` [221] or `textX` [220], if they support the grammar generation from a meta-model in a future point in time. While the optimization rules implemented in grammar optimizer are currently tailored to the structure of Xtext grammars, `GRAMMAROPTIMIZER` does not technically depend on Xtext and the rules could easily be adapted to a different grammar language. Furthermore, as the implementation of an Xtext grammar generator necessarily depends on many version-specific internal aspects of Xtext, the post-processing approach using regular expressions is considerably more maintainable.

### 5.6.6 Limitations and Caveats

Our solution has the following limitations and caveats.

First, `GRAMMAROPTIMIZER` works on the generated grammar, which is generated from a meta-model. This means that the meta-model must contain all the concepts that the expert-created grammar has. Otherwise, the generated grammar will lack the necessary classes or attributes. This would result in the inability to imitate the expert-created grammar. A feasible solution would be to expand the working scope of the `GRAMMAROPTIMIZER`, e.g., to provide a feature to detect whether all the concepts contained in the expert-created grammar corresponding elements can be found in the meta-model. However, we decided against implementing such a feature for now, as we see the main use case of the `GRAMMAROPTIMIZER` not in imitating existing grammars, but in building and maintaining new DSLs.

Second, we were not able to completely imitate one of the seven languages. In order to do so, we would have had to provide an optimization rule that would require the `GRAMMAROPTIMIZER` user to input a multitude of parameter options. This would have strongly increased the effort and reduced the usability to use this one optimization rule, and the rule is only required for this one language. Thus, we argue that a manual post-adaptation is more meaningful for



this one case. However, the inherent extensibility of the `GRAMMAROPTIMIZER` allows to add such an optimization rule if desired. We describe the issue in a more detailed manner in Section 5.7.1.4, which summarizes the evaluation results for the grammar adaptations of the seven analyzed languages.

Third, our solution is non-commutative, that is, applying the same rules with the same parametrization, but in a different order might lead to different results. For example, if `ChangeBracesToAngle` and `ChangeBracesToSquare` are successively applied to the same grammar rule, the outcome is “last write wins”, i.e., the rule obtains square braces. Users should be aware of this property to ensure that the achieved outcome is consistent with their intended outcome.

Fourth, our solution does not strive to maintain backwards comparability to previous grammar versions—in general, after rule applications, instances of the previous, un-optimized grammar can no longer be parsed. This lack of backwards compatibility is generally desirable, as the alternative would be support for a mixing of old and new grammar elements (e.g., changed keywords and parentheses styles) in the same instance, which would generally be confusing to the user, and lead to issues with parsing and other tool support. However, to reduce manual effort in cases where legacy grammar instances exist, automated co-evolution of grammar instances after grammar changes is generally possible and leads to a promising future work direction (discussed in Section 5.8.4).

## 5.7 Evaluation

In this evaluation, we focus on two research questions:

- *RQ1: Can our solution be used to adapt generated grammars so that they produce the same language as available expert-created grammars?*

The goal of this question is to validate the claim that our approach can automatically perform the changes that an expert would need to do manually. To this end, we consider languages for which an expert-created grammar exists, and validate the capability of our approach to re-create an equivalent grammar.

- *RQ2: Can our solution support the co-evolution of generated grammars when the meta-model evolves?*

Our original motivation for the work was to enable evolution and rapid prototyping for textual languages build with a meta-model. The aim here is to evaluate whether our approach is suitable for supporting these evolution scenarios.

In the following, we address both questions. Our supplemental material [232] contains the source code of the implementation as well as all experiments.

### 5.7.1 Grammar Adaptation (RQ1)

To address the first question, we evaluate the `GRAMMAROPTIMIZER` by transforming the generated grammars of the seven DSLs, so that they parse the same syntax as the expert-created grammars.

### 5.7.1.1 Cases

Our goal is to evaluate whether the GRAMMAROPTIMIZER can be used to optimize the generated grammars so that their rules imitate the rules of the expert-created grammars. We reused the meta-model adaptations and generated grammars from Section 5.4.3. Furthermore, we continued working with the versions of ATL and SML in which parts of their languages were excluded as described in Section 5.4.2.

### 5.7.1.2 Method

For each DSL, we wrote a configuration for the final version of GRAMMAROPTIMIZER which was the result of the work described in Section ???. The goal was to transform the generated grammar so as to ‘imitate’ as many grammar rules as possible from the expert-created grammar of the DSL. Note that this was an iterative process in which we incrementally added new optimization rule applications to the GRAMMAROPTIMIZER’s configuration, using the expert-created grammar as a ground truth and using our notion of ‘imitation’ (cf. Section 5.4.4) as the gold standard. Essentially, we updated the GRAMMAROPTIMIZER configuration and then ran the tool before analysing the optimized grammar for imitation of the original. We repeated the process and adjusted the GRAMMAROPTIMIZER configuration until the test grammar’s rules ‘imitated’ the expert-created grammar. Note that in the case of *Spectra*, we did not reach that point. We explain this in more detail in Section 5.7.1.4. For all experiments, we used the set of 56 optimization rules that were identified after the two iterations described in Section 5.4 and as summarized in Section 5.5.

To verify whether the optimized grammar imitates the expert-created grammar, we adopted a manual verification method, in which we systematically compared the grammar rules in the optimized grammar with the grammar rules in the expert-created grammar. An expert-created grammar is imitated by an optimized grammar if every grammar rule in it is imitated by one (or several) grammar rules from the optimized grammar. The procedure and results of this step are documented in our supplementary materials [232].<sup>4</sup>

### 5.7.1.3 Metrics

To evaluate the optimization results of the GRAMMAROPTIMIZER on the case DSLs, we assessed the following metrics.

**#*GORA*** Number of GRAMMAROPTIMIZER rule applications used for the configuration.

**Grammar rules** The changes in grammar rules performed by the GRAMMAROPTIMIZER when adapting the generated grammar towards the expert-created grammar. We measure these changes in terms of

- mod: Number of modified grammar rules
- add: Number of added grammar rules
- del: Number of deleted grammar rules

---

<sup>4</sup>See directory ‘2\_Supplemental\_Material/Section\_7\_Evaluation’.

**Grammar lines** The changes in the lines of the grammar performed by the GRAMMAROPTIMIZER when adapting the generated grammar towards the expert-created grammar. We measure these changes in terms of

- mod: Number of modified lines
- add: Number of added lines
- del: Number of deleted lines

**Optimized grammar** Metrics about the resulting optimized grammar. We assess

- lines: Number of overall lines
- rules: Number of grammar rules
- calls: Number of calls between grammar rules

**#iGR** Number of grammar rules in the expert-created grammar that were successfully *imitated* by the optimized grammar.

**#niGR** Number of grammar rules in the expert-created grammar that were not *imitated* by the optimized grammar.

#### 5.7.1.4 Results

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna.

Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.

Table 5.4 shows the results of applying the GRAMMAROPTIMIZER to the seven DSLs. See Table 5.1 for the corresponding metrics of the initially generated grammars.

**Imitation** For all case DSLs in the first two iterations except *Spectra*, we were able to achieve a complete adaptation, i.e., we were able to modify the grammar by using GRAMMAROPTIMIZER so that the grammar rules of the optimized grammar *imitate* all grammar rules of the expert-created grammar.

**Limitation regarding Spectra** For one of the languages, *Spectra*, we were able to come very close to the expert-created grammar. Many grammar rules of *Spectra* could be nearly imitated. However, we did not implement all grammar rules that would have been necessary to allow the full optimization of *Spectra*. Listing 5.9 shows the grammar rule **TemporalPrimaryExpr** in *Spectra*’s generated grammar, while Listing 5.10 shows what that grammar rule looks like in the expert-created grammar. In order to optimize the grammar rule **TemporalPrimaryExpr** from Listing 5.9 to Listing 5.10, we need to configure the GRAMMAROPTIMIZER so that it combines the attribute **pointer** and **operator** multiple times, and the default value of the attribute **operator** is different each time. The language engineers using the GRAMMAROPTIMIZER need to input multiple parameters to ensure that the GRAMMAROPTIMIZER

Table 5.4: Result of applying the GrammarOptimizer to different DSLs (RQ1)

DSL	Optimization degree	#GORA	Grammar Rules				Lines in Grammar				Optimized Grammar				#GR	#niGR
			mod	add	del	mod	add	del	lines	rules	calls	1				
ATL	Complete	178	30	0	0	187	0	23	187	30	76	28	0			
BibTeX	Complete	14	47	0	1	291	0	0	291	47	188	46	0			
DOT	Complete	79	24	1	3	112	2	0	114	25	41	13	0			
SML	Complete	421	40	5	56	267	18	2	285	45	121	44	0			
Spectra	Close	585	54	3	8	190	9	13	414	57	223	54	2			
Xcore	Complete	307	20	7	14	179	35	10	214	27	100	25	0			
Xenia	Complete	74	13	0	2	74	0	0	74	13	28	13	0			

<sup>1</sup> The number includes the calls to dummy OCL and dummy SML expressions.

Listing 5.9: Example — grammar rule `TemporalPrimaryExpr` in the generated grammar of Spectra

`TemporalPrimaryExpr` returns `TemporalPrimaryExpr` :

```
{TemporalPrimaryExpr}
'TemporalPrimaryExpr'
'{'
    ('operator' operator=EString)?
    ('predPatt' predPatt=[PredicateOrPatternReferrable
    |EString])?
    ('pointer' pointer=[Referrable|EString])?
    ('regexpPointer' regexpPointer=[DefineRegExpDecl|
    EString])?
    ('predPattParams' '{' predPattParams+=
    TemporalExpression ( "," predPattParams+=
    TemporalExpression)* '}' )?
    ('tpe' tpe=TemporalExpression)?
    ('index' '{' index+=TemporalExpression ( "," index
    +=TemporalExpression)* '}' )?
    ('temporalExpression' temporalExpression=
    TemporalExpression)?
    ('regexp' regexp=RegExp)?
' }';
```

gets enough information, and this complex optimization requirement only appears in Spectra. Therefore we did not do such an optimization.

**Size of the Changes** It is worth noting that the number of optimization rule applications is significantly larger than the number of grammar rules for all cases but BibTeX. This indicates that the effort required to describe the optimizations once is significant. However, the actual changes to the grammar, e.g., in terms of modified lines in the grammar are in most cases comparable to the number of optimization rule applications (e.g., for ATL with 178 optimization rule applications and 187 changed lines in the grammar) or even much larger (e.g., for BibTeX with 14 optimization rule applications and 291 modified lines). Note that the number of changed, added, and deleted lines is also an underestimation of the number of necessary changes, as many lines will be changed in multiple ways, e.g., by changing keywords and braces in the same line. This explains why for some languages the number of optimization rule applications is bigger than the number of changed lines (e.g., for SML we specified 421 optimization rule applications which changed, added, and deleted 287 lines in the grammar).

**Effort for the Language Engineer** We acknowledge that the number of optimization rule applications that are necessary to adapt a generated grammar to imitate the expert-created grammar indicates that it is more effort to configure GRAMMAROPTIMIZER than to apply the desired change in the grammar manually once. However, even with that assumption, we argue that the effort of configuring GRAMMAROPTIMIZER is in the same order of magnitude as the effort of applying the changes manually to the grammar.

Furthermore, we argue that it is more efficient to configure GRAMMAROPTIMIZER once than to manually rewrite grammar rules every time the language

Listing 5.10: Example — grammar rule `TemporalPrimaryExpr` in the expert-created grammar of Spectra

```
TemporalPrimaryExpr returns TemporalExpression :
  Constant | '(' QuantifierExpr ')' | {TemporalPrimaryExpr}
  (predPatt=[PredicateOrPatternReferrable]
  '(' predPattParams+=TemporalInExpr (',' predPattParams+=
    TemporalInExpr)* ')' | '()' ) |
  operator=('-'|'!') tpe=TemporalPrimaryExpr |
  pointer=[Referrable]('[' index+=TemporalInExpr '])* |
  operator='next' '(' temporalExpression=TemporalInExpr ')' ,
  |
  operator='regexp' '(' (regexp=RegExp | regexpPointer=[
    DefineRegExpDecl]) ')' |
  pointer=[Referrable] operator='.all' |
  pointer=[Referrable] operator='.any' |
  pointer=[Referrable] operator='.prod' |
  pointer=[Referrable] operator='.sum' |
  pointer=[Referrable] operator='.min' |
  pointer=[Referrable] operator='.max');
```

changes – under the assumption that the configuration can be reused for new versions of the grammar. In that case, the effort invested in configuring GRAMMAROPTIMIZER would quickly pay off when a language is going through changes, e.g., while rapidly prototyping modifications or when the language is evolving. In the next section (Section 5.7.2), we evaluate this assumption.

In terms of reusability of the configurable optimization rules, we observe that most of the languages we cover require at least one *unique* optimization rule that is not needed by any other language. This applies to DOT, BibTeX, and ATL with one unique optimization rule, each. Spectra was our most complicated case with six unique rules, whereas Xcore requires four and SML requires five unique rules. This indicates that using GRAMMAROPTIMIZER for a new language might require effort by implementing a few new optimization rules. However, we argue that this effort will be reduced as more optimization rules are added to GRAMMAROPTIMIZER and that, in particular for evolving languages, the small investment to create a new optimization rule will pay off quickly.

## 5.7.2 Supporting Evolution (RQ2)

To address the second question, we evaluate the GRAMMAROPTIMIZER on two languages’ evolution histories: The industrial case of EAST-ADL and the evolution of the DSL QVTo. We focus on the question to what degree a configuration of the GRAMMAROPTIMIZER that was made for one language version can be applied to a new version of the language.

### 5.7.2.1 Cases

The two cases we are using to evaluate how GRAMMAROPTIMIZER supports the evolution of a DSL are a textual variant of EAST-ADL [194] and QVT Operational (QVTo) [182].

**EAST-ADL** EAST-ADL is an architecture description language used in the automotive domain [194]. Together with an industrial language engineer for EAST-ADL, we are currently developing a textual notation for version 2.2 of the language [195]. We started this work with a simplified version of the meta-model to limit the complexity of the resulting grammar. In a later step, we switched to the full meta-model. We treat this switch as an evolution step here. The meta-model of EAST-ADL is taken from the EATOP repository [16]. The meta-model of the simplified version contains 91 classes and enumerations, and the meta-model of the full version contains 291 classes and enumerations.

**QVTo** QVTo is one of the languages in the OMG QVT standard [182]. We use the original meta-models available in Ecore format on the OMG website [182]. The baseline version is QVTo 1.0 [234] and we simulate evolution to version 1.1 [235], 1.2 [236] and 1.3 [237]. Our original intention was to use the Eclipse reference implementation of QVTo [202], but due to the differences in abstract syntax and concrete syntax (see Section 5.2), we chose to use the official meta-models instead. We analyzed four versions of QVTo’s OMG official Ecore meta-model. There are 50 differences between the meta-models of version 1.0 and 1.1, 29 of which are parts that do not contain OCL (as for ATL as described in Section 5.4.2, we exclude OCL in our solution for QVTo). These 29 differences include different types, for example, 1) the same set of attributes has different arrangement orders in the same class in different versions of the meta-model; 2) the same class has different superclasses in different versions; 3) the same attribute has different multiplicities in different versions, etc. There are 3 differences between versions 1.1 and 1.2, all of which are from the OCL part. There is only one difference between versions 1.2 and 1.3, and it is about the same attribute having a different lower bound for the multiplicity in the same class in the two versions. Altogether we observed 54 meta-model differences in QVTo between the different versions (cf. the file “Comparison of QVTo meta-model versions” in the folder “Section\_7\_Evaluation/Subsection\_7.2\_Support” lists all the metamodel differences).

The OMG website provides an EBNF grammar for each version of QVTo, which is the basis for our imitations of the QVTo languages. Among them, versions 1.0, 1.1, and 1.2 share the same EBNF grammar for the QVTo part except for the OCL parts, despite the differences in the meta-model. The EBNF grammar of QVTo in version 1.3 is different from the other three versions.

### 5.7.2.2 Preparation of the QVTo Case

In contrast to the EAST-ADL case, we needed to perform some preparations of the grammar and the meta-model to study the QVTo case. All adaptations were done the same way on all versions of QVTo.

**Exclusion of OCL** As described in detail in Section 5.4.2, we excluded the embedded OCL language part from QVTo. For the meta-model, we introduced a dummy class for OCL, changed all calls to OCL types into calls to that dummy class, and removed the OCL metaclasses from the meta-model.

As described in Section 5.4.2, excluding a language part such as the embedded OCL from the scope of the investigation also implies that we need

to exclude this language part when it comes to judging whether a grammar is imitated. Therefore, we substituted all grammar rules from the excluded OCL part with a placeholder grammar rule called **ExpressionG0** where an OCL grammar rule would have been called. This change allows us to compare the expert-created grammar of the different QVTo versions to the optimized grammar versions.

**QVTo Meta-model Adaptations** We found that some non-terminals of QVTo’s EBNF grammar are missing in the QVTo meta-model provided by OMG. For example, there is a non-terminal `<top_level>` in the EBNF grammar, but there is no counterpart for it in the meta-model. Therefore, we need to adapt the meta-model to ensure that it contains all the non-terminals in the EBNF grammar. To ensure that the adaptation of the meta-model is done systematically, we defined seven general adaptation rules that we followed when adapting the meta-models of the different versions. We list these adaptation rules in the supplemental material [232].

As a result, we added 62 classes and enumerations with their corresponding references to each version of the meta-model. Note that this number is high compared to the original number of classes in the meta-model (24 classes). This massive change was necessary because the available Ecore meta-models were too abstract to cover all elements of the language. The original meta-model did contain most key concepts, but would not allow to actually specify a complete QVTo transformation. For example, with the original meta-model, it was not possible to represent the scope of a mapping or helper.

These changes enable us to imitate the QVTo grammar. However, they do not bias the results concerning the effects of the observed meta-model evolution as, with the exception of a single case, these evolutionary differences are neither erased nor increased by the changes we performed to the meta-model. The exception is a meta-model evolution change between version 1.0 and 1.1 where the class `MappingOperation` has super types `Operation` and `NamedElement`, while the same class in V1.1 does not. The meta-model change performed by us removes the superclass `Operation` from `MappingOperation` in version 1.0. We did this change to prevent conflicts as the attribute *name* would have been inherited multiple times by `MappingOperation`. This in turn would cause problems in the generation process. Thus, only two of the 54 meta-model evolutionary differences could not be studied. The differences and their analysis can be found in the supplemental material [232].

### 5.7.2.3 Method

To evaluate how GRAMMAROPTIMIZER supports the evolution of meta-models we look at the effort that is required to update the optimization rule applications after an update of the meta-models of EAST-ADL and QVTo.

**Baseline GRAMMAROPTIMIZER Configuration** First, we generated the grammar for the initial version of a language’s meta-model (i.e., the simple version for EAST-ADL and version 1.0 for QVTo). Then we defined the configuration of optimization rule applications that allows the GRAMMAROPTIMIZER



to modify the generated grammar so that its grammar rules *imitate* the expert-created grammar for each case. Doing so confirmed the observation from the first part of the evaluation that a new language of sufficient complexity requires at least some new optimization rules (see Section 5.7.1.4). Consequently, we identified the need for four additional optimization rules for QVTo, which we implemented accordingly as part of the GRAMMAROPTIMIZER (this is also summarized in Section 5.5 in Table 5.2). This step provided us with a baseline configuration for the GRAMMAROPTIMIZER.

**Evolution** For the following language versions, i.e., the full version of EAST-ADL and QVTo 1.1, we then generated the grammar from the corresponding version of the meta-model and applied the GRAMMAROPTIMIZER with the configuration of the previous version (i.e., simple EAST-ADL and QVTo 1.0). We then identified whether this was already sufficient to *imitate* the language’s grammar or whether changes and additions to the optimization rule applications were required. We continued adjusting the optimization rule applications accordingly to gain a GRAMMAROPTIMIZER configuration valid for the new version (full EAST-ADL and QVTo 1.1, respectively). For QVTo, we repeated that process two more times: For QVTo 1.2, we took the configuration of QVTo 1.1 as a baseline, and for QVTo 1.3, we took the configuration of QVTo 1.2 as a baseline.

#### 5.7.2.4 Metrics

We documented the metrics used in Section 5.7.1.3 for EAST-ADL and QVTo in their different versions. In addition, we also documented the following metric:

**#cORA** The number of changed, added, and deleted optimization rule applications compared to the previous language version.

#### 5.7.2.5 Results

Table 5.5 shows the results of the evolution cases.

**EAST-ADL** Compared with the simplified version of EAST-ADL, the full version is much larger. It contains 291 metaclasses, i.e., 200 metaclasses more than the simple version of EAST-ADL, which leads to a generated grammar with 291 grammar rules and 2,839 non-blank lines in the generated grammar file (cf. Table 5.5).

The 22 optimization rule applications for the simple version of EAST-ADL already change the grammar significantly, causing modifications of all 91 grammar rules and changes in nearly every line of the grammar. This also illustrates how massive the changes to the generated grammar are to reach the desired grammar. The number of changes is even larger with the full version of EAST-ADL.

We only needed to change and add a total of 10 grammar optimization rule applications to complete the optimization of the grammar of full EAST-ADL. For example, we excluded the primary type `String0` from the full version of the EAST-ADL grammar, which led us to add a line of configuration `go.removeRule(String0)`. While this is increasing the GRAMMAROPTIMIZER

configuration from the simple EAST-ADL version quite a bit (from 22 optimization rule applications to 31 optimization rule applications), the increase is fairly small given that the meta-model increased massively (with 200 additional metaclasses).

The reason is that our grammar optimization requirements for the simplified version and the full version of EAST-ADL are almost the same. This optimization requirement is mainly based on the look and feel of the language and is provided by an industrial partner. These optimization rule applications have been configured for the simplified version. When we applied them to the generated grammar of the full version of EAST-ADL, we found that we can reuse all of these optimization rule applications. Furthermore, we benefit from the fact that many optimization rule applications are formulated for the scope of the whole grammar and thus can also influence grammar rules added during the evolution step. We do not list a number of grammar rules in a expert-created grammar of EAST-ADL in Table 5.5, because there is no “original” text grammar of EAST-ADL. Instead, we optimize the generated grammar of EAST-ADL according to our industrial partner’s requirements for EAST-ADL’s textual concrete syntax.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna.

Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi.

**QVTo** The baseline configuration of the GRAMMAROPTIMIZER for QVTo includes 733 optimization rule applications, which is a lot given that the expert-created grammar of QVTo 1.0 has 115 non-terminals. Note that the optimized grammar has even fewer grammar rules (77) as some of the rules in the optimized grammar *imitate* multiple rules from the expert-created grammar at once. This again is a testament to how different the expert-created grammar is from the generated one (over 228 lines in the grammar are modified, 2 lines are added, and 580 lines are deleted by these 733 optimization rule applications).

However, if we look at the evolution towards versions 1.1, 1.2, and 1.3 we witness that very few changes to the GRAMMAROPTIMIZER configuration are required. In fact, only between 0 and 2 out of the 733 optimization rule applications needed adjustments. This significantly reduces the effort required compared to manually modifying a grammar generated from a new version of the QVTo metamodel, which would require modifying hundreds of lines. The reason is that, even though there are many differences between different versions of the QVTo meta-model, there are only 0 to 2 differences that affect the optimization rule applications. For example, version 1.0 of the QVTo meta-model has an attribute called `bindParameter` in the class `VarParameter`, whereas it is called `representedParameter` in version 1.1. This attribute is not needed according to the expert-created grammars, so the GRAMMAROPTIMIZER configuration includes a call to the optimization rule *RemoveAttribute* to remove the grammar line that was generated based on that attribute. The second parameter of the optimization rule *RemoveAttribute* needs to specify the name of the attribute. As a consequence of the evolution, we had to change that name in

Table 5.5: Result of supporting evolution (RQ2)

DSL	Meta-m. Classes <sup>1</sup>	Generated grammar lines	grammar rules	calls	Optimized grammar lines	rules	calls <sup>2</sup>	Grammar rules mod	add	del	Lines in Grammar mod	add	del	#GORA	#cORA
EAST-ADL (simple)	91	735	91	735	767	103	782	70	12	0	517	14	2	22	/
EAST-ADL (full)	291	2,839	291	3,062	2,851	303	3,074	233	12	1	2,046	16	4	31	10
QVTo 1.0	85	1,026	109	910	444	77	181	66	1	33	228	2	580	733	/
QVTo 1.1	85	992	110	836	444	77	181	66	1	34	228	2	546	733	2
QVTo 1.2	85	992	110	836	444	77	181	66	1	34	228	2	546	733	0
QVTo 1.3	85	991	110	835	443	77	180	66	1	34	228	2	546	733	1

<sup>1</sup> The number is after adaptation, and it contains both classes and enumerations.  
<sup>2</sup> The number includes the calls to dummy OCL and dummy SML expressions.

the optimization rule application. Another example concerns the class `TypeDef`, which contains an attribute `typedef_condition` in version 1.2 of the QVTo meta-model. We added square brackets to it by applying the optimization rule *AddSquareBracketsToAttr* in the grammar optimization. However, in version 1.3 of the QVTo meta-model, the class `TypeDef` does not contain such an attribute, so the optimization rule application *AddSquareBracketsToAttr* was unnecessary.

Most of the differences between different versions of the meta-model do not lead to changes in the optimization rule applications. For example, the multiplicity of the attribute `when` in the class `MappingOperation` is different in version 1.0 and 1.1. We used *RemoveAttribute* to remove the attribute during the optimization of grammar version 1.0. The same command can still be used in version 1.1, as the removal operation does not need to consider the multiplicity of an attribute. Therefore, this difference does not affect the configuration of optimization rule applications.

## 5.8 Discussion

In the following, we discuss the threats to the validity of the evaluation, different aspects of the GRAMMAROPTIMIZER, and future work implied by the current limitations.

### 5.8.1 Threats to Validity

The threats to validity structured according to the taxonomy of Runeson et al. [238, 239] are as follows.

#### 5.8.1.1 Construct Validity

We limited our analysis to languages for which we could find meta-models in the Ecore format. Some of these meta-models were not “official”, in the sense that they had been reconstructed from a language in order to include them in one of the “zoos”. An example of that is the meta-model for BibTeX we used in our study. In the case of the DOT language, we reconstructed the meta-model from an Xtext grammar we found online. We adopted a reverse-engineering strategy where we generated the meta-model from the expert-created grammar and then generated a new grammar out of this meta-model. This poses a threat to validity since many of the languages we looked at can be considered “artificial” in the sense that they were not developed based on meta-models. However, we do not think this affects the construct validity of our analysis since our purpose is to analyze what changes need to be made from an Xtext grammar file that has been generated. In addition, we address this threat to validity by also including a number of languages (e.g., Xenia and Xcore) that are based on meta-models and using the meta-models provided by the developers of the language.

Furthermore, we had to make some changes to some of the meta-models to be able to generate Xtext grammars out of them at all (cf. Section 5.4.3) or to introduce certain language constructs required by the textual concrete syntax (cf. Section 5.7.2.2). These meta-model adaptations might have introduced

biased changes and thereby imposed a threat to construct validity. However, we reduced these adaptations to a minimum as far as possible to mitigate this threat and documented all of them in our supplemental material [232] to ensure their reproducibility.

#### 5.8.1.2 Internal Validity

In the evaluation (cf. Section 5.7), we set up and quantitatively evaluate size and complexity metrics regarding the considered meta-models and grammars as well as regarding the GRAMMAROPTIMIZER configurations for the use cases of one-time grammar adaptations and language evolution. Based on that, we conclude and argue in Section 5.7.1.4 and Section 5.8.2 about the effort required for creating and evolving languages as well as the effort to create and re-use GRAMMAROPTIMIZER configurations. These relations might be incorrect. However, the applied metrics provide objective and obvious indications about the particular sizes and complexities and thereby the associated engineering efforts.

#### 5.8.1.3 External Validity

As discussed in the analysis part, we analyzed a total of seven DSLs to identify generic optimization rules. Whereas we believe that we have achieved significant coverage by selecting languages from different domains and with very different grammar structures, we cannot deny that analysis of further languages could have led to more optimization rules. However, due to the extensible nature of GRAMMAROPTIMIZER, the practical impact of this threat to generalisability is low since it is easy to add additional generic optimization rules once more languages are analyzed.

#### 5.8.1.4 Reliability

Our overall procedure to conceive and develop the GRAMMAROPTIMIZER encompassed multiple steps. That is, we first determined the differences between the particular initially generated Xtext grammars and the grammars of the actual languages in two iterations as described in Section 5.4. This analysis yielded the corresponding identified conceptual grammar optimization rules summarized in Section 5.5. Based on these identified conceptual grammar optimization rules, we then implemented them as described in Section 5.6. This procedure imposes multiple threats to reliability. For example, analyzing a different set of languages could have led to a different set of identified optimization rules, which then would have led to a different implementation. Furthermore, analyzing the languages in a different order or as part of different iterations could have led to a different abstraction level of the rules and thereby a different number of rules. Finally, the design decisions that we made during the identification of the conceptual optimization rules and during their implementation could also have led to different kinds of rules or implementation. However, we discussed all of these aspects repeatedly amongst all authors to mitigate this threat and documented the results as part of our supplemental material [232] to ensure their reproducibility.

### 5.8.2 The Effort of Creating and Evolving a Language with the GRAMMAROPTIMIZER

The results of our evaluation show three things. First, the expert-created grammars of all studied languages differ greatly in appearance from the generated grammars. Thus, in most cases, creating a DSL with Xtext will require the language engineer to perform big changes to the generated grammar. Second, depending on the language, using the GRAMMAROPTIMIZER for a single version of the language may or may not be more effort for the language engineer, compared to manually adapting the grammar. Third, there seems to be a large potential for the reuse of GRAMMAROPTIMIZER configurations between different versions of a language, thus supporting the evolution of textual languages.

These observations can be combined with the experience that most languages evolve with time and that especially DSLs go through a rapid prototyping phase at the beginning where language versions are built for practical evaluation [240]. Therefore, we conclude that the GRAMMAROPTIMIZER has big potential to save manual effort when it comes to developing DSLs.

Additionally, a topic worth mentioning is how the involvement of different people and their skill sets affect the effort when creating and reusing optimization rule configurations. For example, in case updates to an existing configuration are needed after an evolution step, the maintainers need to understand the optimization rule configuration of the previous version, which could take a new contributor more time than the original contributor. Assessing the impact of this aspect is a subject for future work.

### 5.8.3 Implications for Practitioners and Researchers

Our results have several implications for language engineers and researchers.

**Impact on Textual Language Engineering** Our work might have an impact on the way DSL engineers create textual DSLs nowadays. That is, instead of specifying grammars and thereby having to be EBNF experts, the GRAMMAROPTIMIZER also enables engineers familiar with meta-modeling to conceive well-engineered meta-models and to semi-automatically generate user-friendly grammars from them. Furthermore, Kleppe [204] compiles a list of advantages of approaches like the GRAMMAROPTIMIZER, among them two that apply especially to our solution: 1) the GRAMMAROPTIMIZER provides flexibility for the DSL engineering process, as it is no longer necessary to define the kind of notation used for the DSL at the very beginning as well as 2) the GRAMMAROPTIMIZER enables rapid prototyping of textual DSLs based on meta-models.

**Blended Modeling** Ciccozzi et al. [1] coin the term *blended modeling* for the activity of interacting with one model through multiple notations (e.g., both textual and graphical notations), which would increase the usability and flexibility for different kinds of model stakeholders. However, enabling blended modeling shifts more effort to language engineers. This is due to the fact that the realization of the different editors for the different notations requires many manual steps when using conventional modeling frameworks. In this

context, Cicozzi and colleagues particularly stress the issue of the manual customization of grammars in the case of meta-model evolution. Thus, as one research direction to enable blended modeling, Cicozzi et al. formulate the need to automatically generate the different editors from a given meta-model. Our work serves as one building block toward realizing this research direction and opens up the possibility of developing and evolving blended modeling languages that include textual versions.

A relevant question is to which extent our approach enables cost savings in a larger context, as the cost for evolving the existing tools and applications working with existing languages might be higher than the cost for evolving the languages themselves. We benefit from the extensive tool support offered by Xtext, which can automatically re-generate large parts of the available textual editor after changes in the underlying grammar, including features such as, e.g., auto-formatting, auto-completion, and syntax highlighting. In consequence, by supporting automated grammar changes (in particular, after evolution steps), we also save effort for the overall adaptation of the textual editor. However, in MDE contexts, other applications and tools typically refer to the metamodel, instead of the grammar, and hence, are outside our scope.

**Prevention of Language Flaws** Willink [200] reflects on the version history of the Object Constraint Language (OCL) and the flaws that were introduced during the development of the different OCL 2.x specifications by the Object Management Group [199]. Particularly, he points out that the lack of a parser for the proposed grammar led to several grammar inaccuracies and thereby to ambiguities in the concrete textual syntax. This, in turn, led to the fact that the concrete syntax and the abstract syntax in the Eclipse OCL implementation [201] are so divergent that two distinct meta-models with a dedicated transformation between both are required, which also holds for the QVTo specification and its Eclipse implementation [200] (cf. Section 5.2). The GRAMMAROPTIMIZER will help to prevent and bridge such flaws in language engineering in the future. Xtext already enables the generation of the complete infrastructure for a textual concrete syntax from an abstract syntax represented by a meta-model. Our approach adds the ability to optimize the grammar (i.e., the concrete syntax), as we show in the evaluation by deriving an applicable parser with an optimized grammar from the QVTo Specification meta-models.

#### 5.8.4 Future Work

The GRAMMAROPTIMIZER is a first step in the direction of supporting the evolution of textual grammars for DSLs. However, there are, of course, still open questions and challenges that we discuss in the following.

**Name Changes to Meta-model Elements** In the GRAMMAROPTIMIZER configurations, we currently reference the grammar concepts derived from the meta-model classes and attributes by means of the class and attribute names (cf. Listing 5.8). Thus, if a meta-model evolution involves many name changes, likewise many changes to optimization rule applications are required. Consequently, we plan as future work to improve the GRAMMAROPTIMIZER with a more flexible concept, in which we more closely align the grammar

optimization rule applications with the meta-model based on name-independent references.

**More Efficient Rules and Libraries** We think that there is a lot of potential to make the available set of optimization rules more efficient. This could for example be done by providing libraries of more complex, recurring changes that can be reused. Such a library can contain a default set of optimization rule configurations to make the generated grammar follow a particular style (e.g., mimicking an existing language, to be appealing to users of that language). Language engineers can use it as a basis and with minimal effort define optimization rule configurations that perform DSL-specific changes. Such a change might make the application of the `GRAMMAROPTIMIZER` attractive even in those cases where no evolution of the language is expected. While this use-case still requires effort for defining configurations, the overall effort compared to manual editing can be reduced especially in cases with applicable large-scoped rules that, e.g., globally change the parenthesis style in the grammar.

In addition, the API of `GRAMMAROPTIMIZER` could be changed to a fluent version where the optimization rule application is configured via method calls before they are executed instead of using the current API that contains many `null` parameters. This could also lead to a reduction of the number of grammar optimization rule applications that need to be executed since some executions could be performed at the same time.

Another interesting idea would be to use artificial intelligence to learn existing examples of grammar optimizations in existing languages to provide optimization suggestions for new languages and even automatically create configurations for the `GRAMMAROPTIMIZER`.

**Expression Languages** In this paper, we excluded the expression language parts (e.g., OCL) of two of the example languages (cf. Section 5.4.2). However, expression languages define low-level concepts and have different kinds of grammars and underlying meta-models than conventional languages. In future work, we want to further explore expression languages specifically, in order to ensure that the `GRAMMAROPTIMIZER` can be used for these types of syntaxes as well.

**Visualization of Configuration** Currently, we configure the `GRAMMAROPTIMIZER` by calling the methods of optimization rules, which is a code-based way of working. In the future, we intend to improve the tooling for `GRAMMAROPTIMIZER` and embed the current library into a more sophisticated workbench that allows the language engineer to select and parameterize optimization rule applications either using a DSL or a graphical user interface and provides previews of the modified grammar as well as a view of what valid instances of the language look like.

**Co-evolving Model Instances** We also intend to couple `GRAMMAROPTIMIZER` with an approach for language evolution that also addresses the model instances. In principle, a model instance represented by a textual grammar instance can be read using the old grammar and parsed into an instance of



the old meta-model. It can then be transformed, e.g., using QVTo to conform to the new meta-model, and then be serialized again using the new grammar. However, following this approach means that formatting and comments can be lost. Instead, we intend to derive a textual transformation from the differences in the grammars and the optimization rule applications that can be applied to the model instances and maintain formatting and comments as much as possible.

**Alternative implementation strategy** Our implementation strategy relies on the format of textual grammars produced by Xtext, which is stable across recent versions of Xtext. This implementation strategy was suitable for positively answering our evaluation questions and thus, substantiating the scientific contribution of our paper. An alternative, arguably more elegant implementation strategy would be to use Xtext’s abstract syntax tree representation of the grammar. A benefit of such an implementation would be that it would be more robust in case that the output format of Xtext changes, rendering it a desirable direction for future work.

## 5.9 Conclusion

In this paper, we have presented GRAMMAROPTIMIZER, a tool that supports language engineers in the rapid prototyping and evolution of textual domain-specific languages that are based on meta-models. GRAMMAROPTIMIZER uses a number of optimization rules to modify a grammar generated by Xtext from a meta-model. These optimization rules have been derived from an analysis of the difference between the actual and the generated grammars of seven DSLs.

We have shown how GRAMMAROPTIMIZER can be used to modify grammars generated by Xtext based on these optimization rules. This automation is particularly useful while a language is being developed to allow for rapid prototyping without cumbersome manual configuration of grammars and when the language evolves. We have evaluated GRAMMAROPTIMIZER on seven grammars to gauge the feasibility and effort required for defining the optimization rules. We have also shown how GRAMMAROPTIMIZER supports evolution with the examples of EAST-ADL and QVTo.

Overall, our tool enables language engineers to use a meta-model-based language engineering workflow and still produce high-quality grammars that are very close in quality to hand-crafted ones. We believe that this will reduce the development time and effort for domain-specific languages and will allow language engineers and users to leverage the advantages of using meta-models, e.g., in terms of modifiability and documentation.

In future work, we plan to extend GRAMMAROPTIMIZER into a more full-fledged language workbench that supports advanced features like refactoring of meta-models, a “what you see is what you get” view of the optimization of the grammar, and the ability to co-evolve model instances alongside the underlying language. We will also explore the integration into workflows that generate graphical editors in order to enable blended modeling.

## Acknowledgements

This work has been sponsored by Vinnova under grant number 2019-02382 as part of the ITEA 4 project *BUMBLE*.

# Bibliography

- [1] F. Ciccozzi, M. Tichy, H. Vangheluwe, and D. Weyns, “Blended modelling—what, why and how,” in *1<sup>st</sup> Intl. Workshop on Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS)*. IEEE, 2019, pp. 425–430.
- [2] Eclipse Foundation, “Xtext language development framework,” 2023, Accessed February, 2023. [Online]. Available: <https://www.eclipse.org/Xtext/>
- [3] S. Erdweg *et al.*, “Evaluating and comparing language workbenches: Existing results and benchmarks for the future,” *Comput. Lang. Syst. Struct.*, vol. 44, pp. 24–47, 2015.
- [4] Eclipse Foundation, “Eclipse Modeling Framework (EMF),” 2023, Accessed February, 2023. [Online]. Available: <https://www.eclipse.org/modeling/emf/>
- [5] L. Addazi, F. Ciccozzi, P. Langer, and E. Posse, “Towards seamless hybrid graphical–textual modelling for uml and profiles,” in *Modelling Foundations and Applications: 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings 13*. Springer, 2017, pp. 20–33.
- [6] C. Atkinson and R. Gerbig, “Harmonizing textual and graphical visualizations of domain specific models,” in *Proceedings of the Second Workshop on Graphical Modeling Language Development*, 2013, pp. 32–41.
- [7] F. Pérez Andrés, J. De Lara, and E. Guerra, “Domain specific languages with graphical and textual views,” in *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers 3*. Springer, 2008, pp. 82–97.
- [8] A. Charfi, A. Schmidt, and A. Spriestersbach, “A Hybrid Graphical and Textual Notation and Editor for UML Actions,” in *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009*, ser. LNCS, vol. 5562. Springer, 2009, pp. 237–252.
- [9] M. Scheidgen, “Textual Modelling Embedded into Graphical Modelling,” in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008*, ser. LNCS, vol. 5095. Springer, 2008, pp. 153–168.

- [10] S. Maro, J. Steghöfer, A. Anjorin, M. Tichy, and L. Gelin, “On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015*. ACM, 2015, pp. 1–12.
- [11] F. Cicciozzi, M. Tichy, H. Vangheluwe, and D. Weyns, “Blended Modelling - What, Why and How,” in *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019*. IEEE, 2019, pp. 425–430.
- [12] R. von Hanxleden, E. A. Lee, C. Motika, and H. Fuhrmann, “Multi-view Modeling and Pragmatics in 2020 – Position Paper on Designing Complex Cyber-Physical Systems,” in *Large-Scale Complex IT Systems. Development, Operation and Management - 17th Monterey Workshop 2012*, ser. LNCS, vol. 7539. Springer, 2012, pp. 209–223.
- [13] P. J. Mosterman and H. Vangheluwe, “Computer Automated Multi-Paradigm Modeling: An Introduction,” *Simul.*, vol. 80, no. 9, pp. 433–450, 2004.
- [14] -, “SequenceDiagram.org,” <https://sequencediagram.org>, 2021, Retrieved: 22/05/2021.
- [15] P. Cuenot, D. Chen, S. Gerard, H. Lönn, M.-O. Reiser, D. Servat, C.-J. Sjostedt, R. T. Kolagari, M. Torngren, and M. Weber, “Managing complexity of automotive electronics using the East-ADL,” in *12th IEEE Intl. Conf. on Engineering Complex Computer Systems (ICECCS 2007)*. IEEE, 2007, pp. 353–358.
- [16] EAST-ADL Association, “EATOP Repository,” 2022, Accessed February, 2023. [Online]. Available: <https://bitbucket.org/east-adl/east-adl/src/Revision/>
- [17] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, and S. Zarnekow, “Xtext user guide,” 2008, <http://www.eclipse.org/Xtext/documentation/>, Last accessed Nov 2022.
- [18] A. Kleppe, “Towards the generation of a text-based ide from a language metamodel,” in *Model Driven Architecture-Foundations and Applications: Third European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proceedings 3*. Springer, 2007, pp. 114–129.
- [19] R. Hebig, D. E. Khelladi, and R. Bendraou, “Approaches to co-evolution of metamodels and models: A survey,” *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, 2016.
- [20] D. E. Khelladi, R. Bendraou, R. Hebig, and M.-P. Gervais, “A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution,” *Journal of Systems and Software*, vol. 134, pp. 242–260, 2017.

- [21] A. Kusel, J. Ettlstorfer, E. Kapsammer, W. Retschitzegger, W. Schwinger, and J. Schönböck, “Consistent co-evolution of models and transformations,” in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 2015, pp. 116–125.
- [22] D. E. Khelladi, H. H. Rodriguez, R. Kretschmer, and A. Egyed, “An exploratory experiment on metamodel-transformation co-evolution,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 576–581.
- [23] T. Lombardi, V. Cortellessa, A. Pierantonio, and I. Model, “Co-evolution of metamodel and generators: Higher-order templating to the rescue.” *J. Object Technol.*, vol. 20, no. 3, pp. 7–1, 2021.
- [24] D. Di Ruscio, R. Lämmel, and A. Pierantonio, “Automated co-evolution of gmf editor models,” in *Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers 3*. Springer, 2011, pp. 143–162.
- [25] D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, “A modeling assistant to manage technical debt in coupled evolution,” *Information and Software Technology*, vol. 156, p. 107146, 2023.
- [26] C. Guychard, S. Guerin, A. Koudri, A. Beugnard, and F. Dagnat, “Conceptual interoperability through models federation,” in *Semantic Information Federation Community Workshop*, 2013, p. 23.
- [27] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard, “Addressing modularity for heterogeneous multi-model systems using model federation,” in *Companion Proceedings of the 15th International Conference on Modularity*, 2016, pp. 206–211.
- [28] B. Drouot and J. Champeau, “Model federation based on role modeling,” in *MODELSWARD*, 2019, pp. 72–83.
- [29] V. Garousi and M. V. Mäntylä, “When and what to automate in software testing? a multi-vocal literature review,” *Information and Software Technology*, vol. 76, pp. 92–117, 2016.
- [30] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, “Benefitting from the grey literature in software engineering research,” in *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 385–413.
- [31] M. Petticrew and H. Roberts, *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons, 2008.
- [32] F. Kamei, I. Wiese, C. Lima, I. Polato, V. Nepomuceno, W. Ferreira, M. Ribeiro, C. Pena, B. Cartaxo, G. Pinto *et al.*, “Grey literature in software engineering: A critical review,” *Information and Software Technology*, vol. 138, p. 106609, 2021.
- [33] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.

- [34] Eclipse Foundation, “Mwe2,” 2022, [https://www.eclipse.org/Xtext/documentation/306\\_mwe2.html](https://www.eclipse.org/Xtext/documentation/306_mwe2.html). Last accessed Nov 2022.
- [35] —, “Xtend,” 2023, <https://www.eclipse.org/xtend/>. Last accessed Jan 2023.
- [36] J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, and D. Tuytens, “A comparative study of high-productivity high-performance programming languages for parallel metaheuristics,” *Swarm and Evolutionary Computation*, vol. 57, p. 100720, 2020.
- [37] G. D. Crnkovic, “Constructive research and info-computational knowledge generation,” in *Model-Based Reasoning in Science and Technology: Abduction, Logic, and Computational Discovery*. Springer, 2010, pp. 359–380.
- [38] View, “Boston Professional,” <https://www.view.com/index.php/products-menu/boston-professional>, 2021, Retrieved: 22/05/2021.
- [39] TopQuadrant, Inc, “TopBraid Composer,” <https://www.topquadrant.com/products/topbraid-composer/>, 2021, Retrieved: 22/05/2021.
- [40] University of Ottawa, “Umple,” <https://cruise.umple.org/umple/>, 2021, Retrieved: 22/05/2021.
- [41] itemis AG, “mbeddr,” <http://mbeddr.com/>, 2021, Retrieved: 22/05/2021.
- [42] JetBrains, “MPS: The Domain-Specific Language Creator by JetBrains,” 2022, Accessed February, 2023. [Online]. Available: <https://www.jetbrains.com/mps/>
- [43] CATIA No Magic, “MagicDraw,” <https://www.3ds.com/products-services/catia/products/no-magic/>, 2021, Retrieved: 22/05/2021.
- [44] Eclipse Foundation, “Language Implementation,” 2023, Accessed February, 2023. [Online]. Available: [https://eclipse.dev/Xtext/documentation/303\\_runtime\\_concepts.html](https://eclipse.dev/Xtext/documentation/303_runtime_concepts.html)
- [45] W. Zhang, R. Hebig, D. Strüber, and J.-P. Steghöfer, “Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution,” in *16th ACM SIGPLAN International Conference on Software Language Engineering (SLE’23)*, 2023.
- [46] W. Zhang, J.-P. Steghöfer, R. Hebig, and D. Strüber, “A rapid prototyping language workbench for textual dsls based on xtext: Vision and progress,” *arXiv preprint arXiv:2309.04347*, 2023.
- [47] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil, “A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems,” in *Proceedings of the International Conference on Embedded Software, EMSOFT 2013*. IEEE, 2013, pp. 10:1–10:10.

- [48] C. Atkinson and T. Kühne, “Reducing accidental complexity in domain models,” *Softw. Syst. Model.*, vol. 7, no. 3, pp. 345–359, 2008.
- [49] M. Broy, “Software and System Modeling: Structured Multi-view Modeling, Specification, Design and Implementation,” in *Conquering Complexity*. Springer, 2012, pp. 309–372.
- [50] L. Huning, T. Osterkamp, M. Schaarschmidt, and E. Pulvermüller, “Seamless integration of hardware interfaces in UML-based MDSE tools,” in *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*. SCITEPRESS, 2021, pp. 233–244.
- [51] Z. Gu, S. Wang, S. Kodase, and K. G. Shin, “An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software,” in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*. IEEE Computer Society, 2003, pp. 78–81.
- [52] G. Yang, X. Zhou, and Y. Lian, “Constraint-Based Consistency Checking for Multi-View Models of Cyber-Physical System,” in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017*. IEEE, 2017, pp. 370–376.
- [53] M. Schulze, J. Weiland, and D. Beuche, “Automotive model-driven development and the challenge of variability,” in *16th International Software Product Line Conference, SPLC ’12, Salvador, Brazil - September 2-7, 2012, Volume 1*. ACM, 2012, pp. 207–214.
- [54] L. Addazi and F. Ciccozzi, “Blended graphical and textual modelling for UML profiles: A proof-of-concept implementation and experiment,” *J. Syst. Softw.*, vol. 175, p. 110912, 2021.
- [55] K. Vanherpen, J. Denil, I. David, P. D. Meulenaere, P. J. Mosterman, M. Törngren, A. Qamar, and H. Vangheluwe, “Ontological reasoning for consistency in the design of cyber-physical systems,” in *1st International Workshop on Cyber-Physical Production Systems, CPPS@CPSWeek 2016*. IEEE, 2016, pp. 1–8.
- [56] J. Reineke, C. Stergiou, and S. Tripakis, “Basic problems in multi-view modeling,” *Softw. Syst. Model.*, vol. 18, no. 3, pp. 1577–1611, 2019.
- [57] I. David, “A Foundation for Inconsistency Management in Model-Based Systems Engineering,” Ph.D. dissertation, University of Antwerp, Belgium, Middelheimlaan 1, 2020 Antwerpen, Belgium, 2019.
- [58] H. R. Rothstein and S. Hopewell, “Grey literature,” *The handbook of research synthesis and meta-analysis*, vol. 2, pp. 103–125, 2009.
- [59] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering,” *Inf. Softw. Technol.*, vol. 106, pp. 101–121, 2019.

- [60] ISO/IEC/IEEE, “Systems and software engineering – architecture description,” *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pp. 1–46, 1 2011.
- [61] A. Cicchetti, F. Ciczozzi, and A. Pierantonio, “Multi-view approaches for software and system modelling: a systematic literature review,” *Softw. Syst. Model.*, vol. 18, no. 6, pp. 3207–3233, 2019.
- [62] J. Corley, E. Syriani, H. Ergin, and S. Van Mierlo, *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global, 2016, no. 7, ch. Cloud-based Multi-View Modeling Environments, pp. 120–139.
- [63] P. Carreira, V. Amaral, and H. Vangheluwe, *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*. Springer Nature, 2020.
- [64] H. Vangheluwe, J. de Lara, and P. J. Mosterman, “An introduction to multi-paradigm modelling and simulation,” in *Proceedings of the AIS’2002 conference (AI, Simulation and Planning in High Autonomy Systems)*, 2002, pp. 9–20.
- [65] B. Ries, A. Capozucca, and N. Guelfi, “Messir: a text-first DSL-based approach for UML requirements engineering (tool demo),” in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018*. ACM, 2018, pp. 103–107.
- [66] The Eclipse Foundation, “Eclipse Papyrus,” <https://www.eclipse.org/papyrus/>, 2021, Retrieved: 22/05/2021.
- [67] C.-L. Lazăr, “Integrating Alf editor with Eclipse UML editors,” *Studia Universitatis Babes-Bolyai, Informatica*, vol. 56, no. 3, 2011.
- [68] O. van Rest, G. Wachsmuth, J. R. H. Steel, J. G. Süß, and E. Visser, “Robust Real-Time Synchronization between Textual and Graphical Editors,” in *Theory and Practice of Model Transformations - 6th International Conference, ICMT@STAF 2013*, ser. LNCS, vol. 7909. Springer, 2013, pp. 92–107.
- [69] M. Völter, J. Siegmund, T. Berger, and B. Kolb, “Towards User-Friendly Projectional Editors,” in *Software Language Engineering - 7th International Conference, SLE 2014*, ser. LNCS, vol. 8706. Springer, 2014, pp. 41–61.
- [70] C. Simonyi, “The Death of Computer Languages, The Birth of Intentional Programming,” Tech. Rep. MSR-TR-95-52, September 1995.
- [71] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, “Efficiency of projectional editing: a controlled experiment,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*. ACM, 2016, pp. 763–774.
- [72] I. David, J. Denil, and H. Vangheluwe, “Process-oriented Inconsistency Management in Collaborative Systems Modeling,” in *16th International Industrial Simulation Conference 2018, ISC 2018*. Eurosis, 2018, pp. 54–61.



- [73] L. Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.
- [74] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *Computer*, vol. 29, no. 12, pp. 66–76, 1996.
- [75] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.
- [76] V. Balesgas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Pregoça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015*. ACM, 2015, pp. 6:1–6:16.
- [77] M. Shapiro, N. M. Pregoça, C. Baquero, and M. Zawirski, “Conflict-Free Replicated Data Types,” in *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011*, ser. Lecture Notes in Computer Science, vol. 6976. Springer, 2011, pp. 386–400.
- [78] G. Spanoudakis and A. Zisman, “Inconsistency management in software engineering: Survey and open research issues,” in *Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals*. World Scientific, 2001, pp. 329–380.
- [79] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen, “A methodology for specifying and analyzing consistency of object-oriented behavioral models,” in *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001*. ACM, 2001, pp. 186–195.
- [80] K. Vanherpen, “A Contract-based Approach for Multi-viewpoint Consistency in the Concurrent Design of Cyber-physical Systems,” Ph.D. dissertation, University of Antwerp, Belgium, Middelheimlaan 1, 2020 Antwerpen, Belgium, 2018.
- [81] P. Stevens, “Maintaining consistency in networks of models: bidirectional transformations in the large,” *Softw. Syst. Model.*, vol. 19, no. 1, pp. 39–65, 2020.
- [82] H. Giese and R. Wagner, “Incremental Model Synchronization with Triple Graph Grammars,” in *Model Driven Engineering Languages and Systems, 9th International Conference, MODELS 2006*, ser. LNCS, vol. 4199. Springer, 2006, pp. 543–557.
- [83] T. Kehrler, U. Kelter, and G. Taentzer, “Consistency-preserving edit scripts in model versioning,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*. IEEE, 2013, pp. 191–201.
- [84] S. Kelly, “Collaborative modelling with version control,” in *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops*, ser. LNCS, vol. 10748. Springer, 2017, pp. 20–29.

- [85] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. H. Reussner, “Enabling consistency in view-based system development - the vitruvius approach,” *J. Syst. Softw.*, vol. 171, p. 110815, 2021.
- [86] A. Finkelstein, “A Foolish Consistency: Technical Challenges in Consistency Management,” in *Database and Expert Systems Applications, 11th International Conference, DEXA 2000*, ser. LNCS, vol. 1873. Springer, 2000, pp. 1–5.
- [87] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, “Inconsistency Handling in Multiperspective Specifications,” *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 569–578, 1994.
- [88] B. Nuseibeh, S. M. Easterbrook, and A. Russo, “Making inconsistency respectable in software development,” *J. Syst. Softw.*, vol. 58, no. 2, pp. 171–180, 2001.
- [89] R. Balzer, “Tolerating Inconsistency,” in *Proceedings of the 13th International Conference on Software Engineering*. IEEE/ACM, 1991, pp. 158–165.
- [90] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh, “Coordinating Distributed ViewPoints: the anatomy of a consistency check,” *Concurrent Engineering*, vol. 2, no. 3, pp. 209–222, 1994.
- [91] I. David, E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, and K. Vanherpen, “Towards Inconsistency Tolerance by Quantification of Semantic Inconsistencies,” in *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016)*, ser. CEUR Workshop Proceedings, vol. 1717. CEUR-WS.org, 2016, pp. 35–44.
- [92] W. Torres, M. G. J. van den Brand, and A. Serebrenik, “A systematic literature review of cross-domain model consistency checking by model management tools,” *Softw. Syst. Model.*, vol. 20, no. 3, pp. 897–916, 2021.
- [93] A. Iung, J. Carbonell, L. Marchezan, E. M. Rodrigues, M. Bernardino, F. P. Basso, and B. Medeiros, “Systematic mapping study on domain-specific language development tools,” *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 4205–4249, 2020.
- [94] M. Franzago, D. D. Ruscio, I. Malavolta, and H. Muccini, “Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map,” *IEEE Trans. Software Eng.*, vol. 44, no. 12, pp. 1146–1175, 2018.
- [95] I. David, K. Aslam, S. Faridmoayer, I. Malavolta, E. Syriani, and P. Lago, “Collaborative Model-Driven Software Engineering: A Systematic Update,” in *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021*. IEEE, 2021, pp. 273–284.

- [96] D. Granada, J. M. Vara, F. J. P. Blanco, and E. Marcos, “Model-based Tool Support for the Development of Visual Editors - A Systematic Mapping Study,” in *Proceedings of the 12th International Conference on Software Technologies, ICISOFT 2017*. SciTePress, 2017, pp. 330–337.
- [97] L. M. do Nascimento, D. L. Viana, P. Neto, D. Martins, V. C. Garcia, and S. Meira, “A systematic mapping study on domain-specific languages,” in *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012, pp. 179–187.
- [98] E. Negm, S. Makady, and A. Salah, “Survey on Domain Specific Languages Implementation Aspects,” *International Journal of Advanced Computer Science and Applications*, vol. 10, 2019.
- [99] B. Merkle, “Textual modeling tools: overview and comparison of language workbenches,” in *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010*. ACM, 2010, pp. 139–148.
- [100] V. R. Basili, G. Caldiera, and H. D. Rombach, “The Goal Question Metric Approach,” in *Encyclopedia of Software Engineering*. Wiley, 1994, vol. 2, pp. 528–532.
- [101] B. A. Kitchenham and P. Brereton, “A systematic review of systematic review process research in software engineering,” *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2049–2075, 2013.
- [102] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012.
- [103] K. Petersen, S. Vakkalanka, and L. Kuzniarz, “Guidelines for conducting systematic mapping studies in software engineering: An update,” *Information and software technology*, vol. 64, pp. 1–18, 2015.
- [104] Carnegie Mellon University, “OSATE,” <https://osate.org/>, 2021, Retrieved: 22/05/2021.
- [105] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14*. ACM, 2014, pp. 38:1–38:10.
- [106] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, “Systematic mapping studies in software engineering,” in *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008*, ser. Workshops in Computing. BCS, 2008.
- [107] T. Greenhalgh and R. Peacock, “Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources,” *BMJ*, vol. 331, no. 7524, pp. 1064–1065, 2005.
- [108] B. A. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering, Version 2.3,” Keele University and University of Durham, EBSE Technical Report EBSE-2007-01, 2007.

- [109] R. Franzosi, *Quantitative narrative analysis*. Sage, 2010, no. 162.
- [110] M. Rodgers, A. Sowden, M. Petticrew, L. Arai, H. Roberts, N. Britten, and J. Popay, "Testing methodological guidance on the conduct of narrative synthesis in systematic reviews: effectiveness of interventions to promote smoke alarm ownership and function," *Evaluation*, vol. 15, no. 1, pp. 49–73, 2009.
- [111] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, 2019.
- [112] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of UML models: a systematic review of research and practice," *Softw. Syst. Model.*, vol. 18, no. 3, pp. 2313–2360, 2019.
- [113] M. G. Haviland, "Yates's correction for continuity and the analysis of  $2 \times 2$  contingency tables," *Statistics in medicine*, vol. 9, no. 4, pp. 363–367, 1990.
- [114] BOC Products Services AG, "ADOIT:Community Edition," <https://www.adoit-community.com/en/>, 2021, Retrieved: 22/05/2021.
- [115] P. Beauvoir and J.-B. Sarrodie, "Archi," <https://www.archimatetool.com/>, 2021, Retrieved: 22/05/2021.
- [116] Software AG, "ARIS," <https://www.ariscommunity.com/>, 2021, Retrieved: 22/05/2021.
- [117] ETAS, "ASCET Developer," <https://www.etas.com/en/products/ascet-developer.php>, 2021, Retrieved: 22/05/2021.
- [118] E. Syriani and H. Vangheluwe, "AToMPM," <https://atompmp.github.io/>, 2021, Retrieved: 22/05/2021.
- [119] ESTECO SpA, "Cardanit," <https://www.cardanit.com/>, 2021, Retrieved: 22/05/2021.
- [120] NASA, "certware," <https://nasa.github.io/CertWare/>, 2021, Retrieved: 22/05/2021.
- [121] Holistics Software, "DBDiagram," <https://dbdiagram.io/home>, 2021, Retrieved: 22/05/2021.
- [122] The Eclipse Foundation, "Eclipse Process Framework Project," <https://projects.eclipse.org/projects/technology.epf>, 2021, Retrieved: 22/05/2021.
- [123] Modelisof, "Modelio," <https://www.modelio.org/>, 2021, Retrieved: 22/05/2021.
- [124] Dovetail Technologies Ltd, "QuickDataBaseDiagrams," <https://www.quickdatabasediagrams.com/>, 2021, Retrieved: 22/05/2021.

- [125] OMiLAB, “SOM/ADOxx,” <https://austria.omilab.org/psm/content/som/info?view=home>, 2021, Retrieved: 22/05/2021.
- [126] -, “Swimlanes.io,” <https://swimlanes.io/>, 2021, Retrieved: 22/05/2021.
- [127] M. Auer and T. Tschurtschenthaler, “UMLet,” <https://www.umlet.com/>, 2021, Retrieved: 22/05/2021.
- [128] —, “UMLetino 14.3,” <https://www.umletino.com/>, 2021, Retrieved: 22/05/2021.
- [129] Universität Bremen, “USE – The UML-based Specification Environment,” [http://useocl.sourceforge.net/w/index.php/Main\\_Page](http://useocl.sourceforge.net/w/index.php/Main_Page), 2021, Retrieved: 22/05/2021.
- [130] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe, “SCCD: SCXML extended with class diagrams,” in *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*, vol. 2, 2016, pp. 1–2.
- [131] D. L. Moody, “The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering,” *IEEE Trans. Software Eng.*, vol. 35, no. 6, pp. 756–779, 2009.
- [132] A. Barisic, V. Amaral, and M. Goulão, “Usability Evaluation of Domain-Specific Languages,” in *8th International Conference on the Quality of Information and Communications Technology, QUATIC 2012*. IEEE, 2012, pp. 342–347.
- [133] A. R. Dennis and J. S. Valacich, “A replication manifesto,” *AIS Trans. Replication Res.*, vol. 1, p. 1, 2015.
- [134] T. Mens, R. V. D. Straeten, and M. D’Hondt, “Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis,” in *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, ser. LNCS, vol. 4199. Springer, 2006, pp. 200–214.
- [135] C. Nentwich, W. Emmerich, and A. Finkelstein, “Consistency Management with Repair Actions,” in *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2003, pp. 455–464.
- [136] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook, and J. Rieke, “Management of cross-domain model consistency during the development of advanced mechatronic systems,” in *DS 58-6: Proceedings of ICED 09, the 17th International Conference on Engineering Design*, ser. ICED, vol. 6, no. 2, 2009, pp. 1–12.
- [137] M. Voelter, “Language and IDE Modularization and Composition with MPS,” in *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011*, ser. LNCS, vol. 7680. Springer, 2011, pp. 383–430.

- [138] L. Engelen and M. van den Brand, “Integrating Textual and Graphical Modelling Languages,” *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 105–120, 2010.
- [139] I. Ráth, A. Ökrös, and D. Varró, “Synchronization of abstract and concrete syntax in domain-specific modeling languages - By mapping models and live transformations,” *Softw. Syst. Model.*, vol. 9, no. 4, pp. 453–471, 2010.
- [140] B. H. Wixom and P. A. Todd, “A Theoretical Integration of User Satisfaction and Technology Acceptance,” *Inf. Syst. Res.*, vol. 16, no. 1, pp. 85–102, 2005.
- [141] E. Syriani, D. Riegelhaupt, B. Barroca, and I. David, “Generation of Custom Textual Model Editors,” *Modelling*, vol. 2, no. 4, pp. 609–625, 2021.
- [142] S. Van Mierlo, Y. Van Tendeloo, I. David, B. Meyers, A. Gebremichael, and H. Vangheluwe, “A multi-paradigm approach for modelling service interactions in model-driven engineering processes,” in *Proceedings of the Model-driven Approaches for Simulation Engineering Symposium, SpringSim (Mod4Sim) 2018*. ACM, 2018, pp. 6:1–6:12.
- [143] T. Gjørseter, A. Prinz, and M. Scheidgen, “Meta-model or Grammar? Methods and Tools for the Formal Definition of Languages,” in *Nordic Workshop on Model Driven Engineering (NW-MoDE 2008)*, 2008, pp. 67–82.
- [144] J. Michaux, X. Blanc, M. Shapiro, and P. Sutra, “A semantically rich approach for collaborative model edition,” in *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*. ACM, 2011, pp. 1470–1475.
- [145] M. Zaheri, M. Famelis, and E. Syriani, “Towards Checking Consistency-Breaking Updates between Models and Generated Artifacts,” in *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion*. IEEE, 2021, pp. 400–409.
- [146] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. Lédeczi, “Next generation (meta) modeling: web-and cloud-based collaborative tool infrastructure.” *MPM@ MoDELS*, vol. 1237, pp. 41–60, 2014.
- [147] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio, “Grand challenges in model-driven engineering: an analysis of the state of the research,” *Softw. Syst. Model.*, vol. 19, no. 1, pp. 5–13, 2020.
- [148] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. de Lara, I. Ráth, D. Varró, M. Tisi, and J. Cabot, “A research roadmap towards achieving scalability in model driven engineering,” in *Proceedings of the Workshop on Scalability in Model Driven Engineering*. ACM, 2013, p. 2.

- [149] H. Zhang, M. A. Babar, and P. Tell, “Identifying relevant studies in software engineering,” *Information and Software Technology*, vol. 53, no. 6, pp. 625–637, 2011.
- [150] V. Garousi and J. M. Fernandes, “Highly-cited papers in software engineering: The top-100,” *Inf. Softw. Technol.*, vol. 71, pp. 108–128, 2016.
- [151] M. Eysholdt and H. Behrens, “Xtext: implement your language faster than the quick and dirty way,” in *ACM Intl. Conf. on Object oriented programming systems languages and applications companion*, 2010, pp. 307–309.
- [152] V. Debruyne, F. Simonot-Lion, and Y. Trinquet, “EAST-ADL—an architecture description language,” in *IFIP World Computer Congress, TC 2*. Springer, 2004, pp. 181–195.
- [153] P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, and M. Wimmer, “XMLText: From XML schema to xtext,” in *2015 ACM SIGPLAN Intl. Conf. on Software Language Engineering*, 2015, pp. 71–76.
- [154] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J.-P. Steghöfer, and R. Hebig, “Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study,” *Software & Systems Modeling*, 2022.
- [155] M. Latifaj, F. Ciccozzi, M. Mohlin, and E. Posse, “Towards Automated Support for Blended Modelling of UML-RT Embedded Software Architectures,” in *ECSA (Companion)*, 2021.
- [156] J. Cooper and D. Kolovos, “Engineering hybrid graphical-textual languages with sirius and xtext: Requirements and challenges,” in *2019 ACM/IEEE 22nd Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) Companion Proceedings*. IEEE, 2019, pp. 322–325.
- [157] Eclipse Foundation, “Xtext reference documentation: Configuration,” 2023, [https://www.eclipse.org/Xtext/documentation/302\\_configuration.html](https://www.eclipse.org/Xtext/documentation/302_configuration.html). Last accessed Jan 2023.
- [158] J. Holtmann, J.-P. Steghöfer, and W. Zhang, “EATXT implementation excerpt,” 2023, <https://github.com/joerg-holtmann/EATXT4MODELSWARD23>. Last accessed Jan 2023.
- [159] Eclipse Foundation, “Xtext reference documentation: Template proposals,” 2023, [https://www.eclipse.org/Xtext/documentation/310\\_eclipse\\_support.html#templates](https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#templates). Last accessed Jan 2023.
- [160] —, “Xtext reference documentation: Content assist,” 2023, [https://www.eclipse.org/Xtext/documentation/310\\_eclipse\\_support.html#content-assist](https://www.eclipse.org/Xtext/documentation/310_eclipse_support.html#content-assist). Last accessed Jan 2023.
- [161] —, “Xtext reference documentation: Formatting,” 2023, [https://www.eclipse.org/Xtext/documentation/303\\_runtime\\_concepts.html#formatting](https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#formatting). Last accessed Jan 2023.

- [162] —, “Xtext reference documentation: Cross-references,” 2023, [https://www.eclipse.org/Xtext/documentation/301\\_grammarlanguage.html#cross-references](https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#cross-references). Last accessed Jan 2023.
- [163] —, “Xtext reference documentation: Scoping,” 2023, [https://www.eclipse.org/Xtext/documentation/303\\_runtime\\_concepts.html#scoping](https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping). Last accessed Jan 2023.
- [164] J. Holtmann, J. Steghöfer, and H. Lönn, “Migrating from proprietary tools to open-source software for EAST-ADL metamodel generation and evolution,” in *25th Intl. Conf. on Model Driven Engineering Languages and Systems (MODELS) Companion Proceedings*, T. Kühn and V. Sousa, Eds. ACM, 2022, pp. 7–11. [Online]. Available: <https://doi.org/10.1145/3550356.3559084>
- [165] T. Kosar, S. Bohra, and M. Mernik, “Domain-specific languages: A systematic mapping study,” *IST*, vol. 71, pp. 77–91, 2016.
- [166] L. M. do Nascimento, D. L. Viana, P. Neto, D. Martins, V. C. Garcia, and S. Meira, “A systematic mapping study on domain-specific languages,” in *ICSEA*, 2012, pp. 179–187.
- [167] A. Nordmann, N. Hochgeschwender, and S. Wrede, “A survey on domain-specific languages in robotics,” in *SIMPAR*. Springer, 2014, pp. 195–206.
- [168] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages,” *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.
- [169] Eclipse Foundation, “Xtext homepage,” 2022, <https://www.eclipse.org/Xtext/>. Last accessed Nov 2022.
- [170] Python Community, “Python homepage,” 2022, <https://www.python.org/>. Last accessed Nov 2022.
- [171] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.
- [172] L. Bettini, “Type errors for the ide with xtext and xsemantics,” *Open Computer Science*, vol. 9, no. 1, pp. 52–79, 2019.
- [173] Eclipse Foundation, “Whitespace-awareness,” 2022, [https://www.eclipse.org/Xtext/documentation/307\\_special\\_languages.html](https://www.eclipse.org/Xtext/documentation/307_special_languages.html). Last accessed Nov 2022.
- [174] M. Rodchenkov, “Xenia metamodel,” 2019, <https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore>, Last accessed Nov 2022.
- [175] —, “Xenia homepage,” 2020, <https://github.com/rodchenk/xenia/>, Last accessed Nov 2022.
- [176] AtlanMod, “Atlantic zoo,” 2019, <https://github.com/atlanmod/atlantic-zoo>, Last accessed Nov 2022.



- [177] ABLE Group, “Acme homepage,” 2011, at Carnegie Mellon University, [https://acme.able.cs.cmu.edu/docs/language\\_overview.html](https://acme.able.cs.cmu.edu/docs/language_overview.html), Last accessed Nov 2022.
- [178] S. Efftinge and M. Völter, “oaw xtext: A framework for textual dsls,” in *Workshop on Modeling Symposium at Eclipse Summit*, vol. 32, no. 118, 2006.
- [179] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [180] D. Sredojević, D. Okanović, M. Vidaković, D. Mitrović, and M. Ivanović, “Domain specific agent-oriented programming language based on the xtext framework,” in *ICIST*, 2015, pp. 8–11.
- [181] A. Van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *ACM Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.
- [182] Object Management Group, “QVT – MOF Query/View/Transformation Specification,” 2016, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/QVT/>
- [183] Eclipse Foundation, “ATL Syntax,” 2018, Accessed February, 2023. [Online]. Available: <https://wiki.eclipse.org/M2M/ATL/Syntax>
- [184] Paperpile, “A complete guide to the BibTeX format,” 2022, Accessed February, 2023. [Online]. Available: <https://www.bibtex.com/g/bibtex-format/>
- [185] Graphviz Authors, “Dot language,” 2022, Accessed February, 2023. [Online]. Available: <https://graphviz.org/doc/info/lang.html>
- [186] J. Greenyer, “Scenario Modeling Language (SML) Repository,” 2018, Accessed February, 2023. [Online]. Available: <https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/>
- [187] Spectra Authors, “Spectra,” 2021, Accessed February, 2023. [Online]. Available: <https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/src/tau/smlab/syntech/Spectra.xtext>
- [188] Eclipse Foundation, “Eclipse xcore wiki,” 2018, Accessed February, 2023. [Online]. Available: <https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/src/org/eclipse/emf/ecore/xcore/Xcore.xtext>
- [189] Xenia Authors, “Xenia xtext,” 2019, Accessed February, 2023. [Online]. Available: <https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext>
- [190] S. Roy Chaudhuri, S. Natarajan, A. Banerjee, and V. Choppella, “Methodology to develop domain specific modeling languages,” in *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling*. ACM SIGPLAN, 2019, pp. 1–10.

- [191] U. Frank, “Domain-specific modeling languages: requirements analysis and design guidelines,” in *Domain engineering*. Springer, 2013, pp. 133–157.
- [192] D. Albuquerque, B. Cafeo, A. Garcia, S. Barbosa, S. Abrahão, and A. Ribeiro, “Quantifying usability of domain-specific languages: An empirical study on software maintenance,” *Journal of Systems and Software*, vol. 101, pp. 245–259, 2015.
- [193] A. Stefik and S. Siebert, “An empirical investigation into programming language syntax,” *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–40, 2013.
- [194] EAST-ADL Association, “East-adl,” 2021, Accessed February, 2023. [Online]. Available: <https://www.east-adl.info/>
- [195] J. Holtmann, J.-P. Steghöfer, and W. Zhang, “Exploiting meta-model structures in the generation of Xtext editors,” in *11th Intl. Conf. on Model-Based Software and Systems Engineering (MODELSWARD)*, 2023, pp. 218–225.
- [196] R. F. Paige, D. S. Kolovos, and F. A. Polack, “A tutorial on metamodeling for grammar researchers,” *Science of Computer Programming*, vol. 96, pp. 396–416, 2014, selected Papers from the Fifth Intl. Conf. on Software Language Engineering (SLE 2012).
- [197] International Organization for Standardization (ISO), “Information technology—Syntactic metalanguage—Extended BNF (ISO/IEC 14977:1996),” 1996.
- [198] A. Kleppe, “A language description is more than a metamodel,” in *4th International Workshop on Language Engineering*, 2007.
- [199] Object Management Group (OMG), “Object constraint language 2.x specification,” 2014, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/OCL/>
- [200] E. Willink, “Reflections on OCL 2,” *Journal of Object Technology*, vol. 19, no. 3, pp. 3:1–16, 2020.
- [201] Eclipse Foundation, “Eclipse OCL™ (Object Constraint Language),” 2022, Accessed February, 2023. [Online]. Available: <https://projects.eclipse.org/projects/modeling.mdt.ocl>
- [202] —, “Qvto – eclipsepedia,” 2022, Accessed February, 2023. [Online]. Available: <https://wiki.eclipse.org/QVTo>
- [203] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, “Derivation and refinement of textual syntax for models,” in *European Conf. on Model Driven Architecture—Foundations and Applications (ECMDA-FA)*, ser. LNCS, vol. 5562. Springer, 2009, pp. 114–129.

- [204] A. Kleppe, “Towards the generation of a text-based ide from a language metamodel,” in *European Conf. on Model Driven Architecture—Foundations and Applications (ECMDA-FA)*, ser. LNCS, vol. 4530. Springer, 2007, pp. 114–129.
- [205] T. Parr, “ANTLR,” 2022, Accessed February, 2023. [Online]. Available: <https://www.antlr.org/>
- [206] P. Neubauer, R. Bill, and M. Wimmer, “Modernizing domain-specific languages with xmltext and intelledit,” in *2017 IEEE 24th Intl. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017.
- [207] S. Chodarev, “Development of human-friendly notation for xml-based languages,” in *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2016, pp. 1565–1571.
- [208] F. Jouault, J. Bézivin, and I. Kurtev, “Tcs: A dsl for the specification of textual concrete syntaxes in model engineering,” in *5th Intl. Conf. on Generative Programming and Component Engineering*. ACM, 2006, p. 249–254.
- [209] M. Novotný, “Model-driven pretty printer for xtext framework,” Master’s thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2012.
- [210] U. Frank, “Some guidelines for the conception of domain-specific modelling languages,” in *Enterprise Modelling and Information Systems Architectures (EMISA 2011)*. Gesellschaft für Informatik eV, 2011, pp. 93–106.
- [211] J.-P. Tolvanen and S. Kelly, “Effort used to create domain-specific modeling languages,” in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pp. 235–244.
- [212] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, “Design guidelines for domain specific languages,” in *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM’ 09)*, no. TR no B-108. Orlando, Florida, USA: Helsinki School of Economics, October 2009. [Online]. Available: <http://arxiv.org/abs/1409.2378>
- [213] M. van Amstel, M. van den Brand, and L. Engelen, “An exercise in iterative domain-specific language design,” in *Proceedings of the joint ERCIM workshop on software evolution (EVOL) and international workshop on principles of software evolution (IWPSE)*, 2010, pp. 48–57.
- [214] M. Karaila, “Evolution of a domain specific language and its engineering environment—lehman’s laws revisited,” in *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009, pp. 1–7.
- [215] M. Pizka and E. Jürgens, “Tool-supported multi-level language evolution,” in *Software and Services Variability Management Workshop*, vol. 3, 2007, pp. 48–67.

- [216] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, “Metamodel and constraints co-evolution: A semi automatic maintenance of OCL constraints,” in *International Conference on Software Reuse*. Springer, 2016, pp. 333–349.
- [217] D. D. Ruscio, R. Lämmel, and A. Pierantonio, “Automated co-evolution of gmf editor models,” in *International conference on software language engineering*. Springer, 2010, pp. 143–162.
- [218] D. Di Ruscio, L. Iovino, and A. Pierantonio, “What is needed for managing co-evolution in mde?” in *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, 2011, pp. 30–38.
- [219] J. García, O. Diaz, and M. Azanza, “Model transformation co-evolution: A semi-automatic approach,” in *International conference on software language engineering*. Springer, 2012, pp. 144–163.
- [220] I. Dejanović, R. VADERNA, G. Milosavljević, and Ž. Vuković, “Textx: A python tool for domain-specific languages implementation,” *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017.
- [221] TypeFox GmbH, “Langium,” 2022, Accessed February, 2023. [Online]. Available: <https://langium.org/>
- [222] S. Kelly and J.-P. Tolvanen, “Collaborative creation and versioning of modeling languages with metaedit+,” in *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2018, pp. 37–41.
- [223] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, “An overview of domain-specific languages in robotics,” 2020, Accessed February, 2023. [Online]. Available: <https://corlab.github.io/dslzoo/all.html>
- [224] I. Wikimedia Foundation, “Wikipedia page of domain specific language,” 2023, Accessed February, 2023. [Online]. Available: [https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)
- [225] M. Barash, “Zoo of domain-specific languages,” 2020, Accessed February, 2023. [Online]. Available: <http://dsl-course.org/>
- [226] I. Semantic Designs, “Domain specific languages,” 2021, Accessed February, 2023. [Online]. Available: <http://www.semdesigns.com/products/DMS/DomainSpecificLanguage.html>
- [227] D. Community, “Financial domain-specific language listing,” 2021, Accessed February, 2023. [Online]. Available: <http://dslfn.org/resources.html>
- [228] miklossy, nyssen, prggz, and mwienand, “Dot xtext grammar,” 2020, Accessed February, 2023. [Online]. Available: <https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext>

- [229] V. Zaytsev, “Grammarware bibtex metamodel,” 2013, Accessed February, 2023. [Online]. Available: <https://github.com/grammarware/slps/blob/master/topics/grammars/bibtex/bibtex-1/BibTeX.ecore>
- [230] Spectra Authors, “Spectra metamodel,” 2021, Accessed February, 2023. [Online]. Available: <https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/model/generated/Spectra.ecore>
- [231] Eclipse Foundation, “Xcore metamodel,” 2012, Accessed February, 2023. [Online]. Available: <https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore>
- [232] W. Zhang, J. Holtmann, D. Strüber, R. Hebig, and J.-P. Steghöfer, “Grammaroptimizer\_data: Formal release,” Feb. 2023, Accessed October, 2023.
- [233] J. E. Hopcroft, “On the equivalence and containment problems for context-free languages,” *Mathematical systems theory*, vol. 3, no. 2, pp. 119–124, 1969.
- [234] Object Management Group, “QVT – MOF Query/View/Transformation Specification Version 1.0,” 2008, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/QVT/1.0/>
- [235] —, “QVT – MOF Query/View/Transformation Specification Version 1.1,” 2011, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/QVT/1.1/>
- [236] —, “QVT – MOF Query/View/Transformation Specification Version 1.2,” 2015, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/QVT/1.2/>
- [237] —, “QVT – MOF Query/View/Transformation Specification Version 1.3,” 2016, Accessed February, 2023. [Online]. Available: <https://www.omg.org/spec/QVT/1.3/>
- [238] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2008.
- [239] P. Runeson, M. Höst, R. Austen, and B. Regnell, *Case Study Research in Software Engineering — Guidelines and Examples*, 1st ed. Wiley, 2012.
- [240] Q. Wang and G. Gupta, “Rapidly prototyping implementation infrastructure of domain specific languages: a semantics-based approach,” in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005, pp. 1419–1426.

