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

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 LaTeX.
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. Strüber, who joined the team of authors for the revision. Additionally, I gave presentations on the paper on several occasions.

Role	**Paper A**	**Paper B**	**Paper C**	**Paper D**
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

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 replies 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

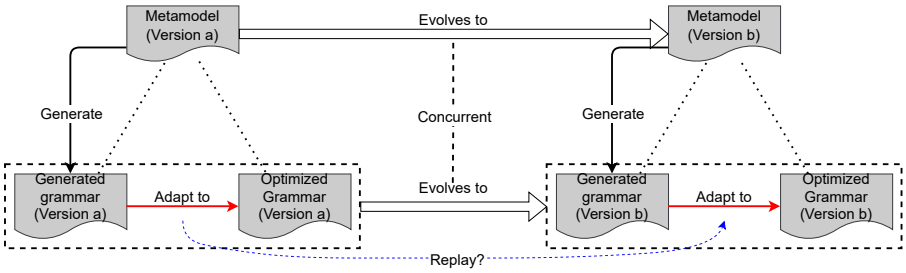

Figure 1.1: In a MDE approach, the grammar is adapted before being put into use, and when the language evolves, the grammar generated from the evolved meta-model needs to be adapted again.

(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 the 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,

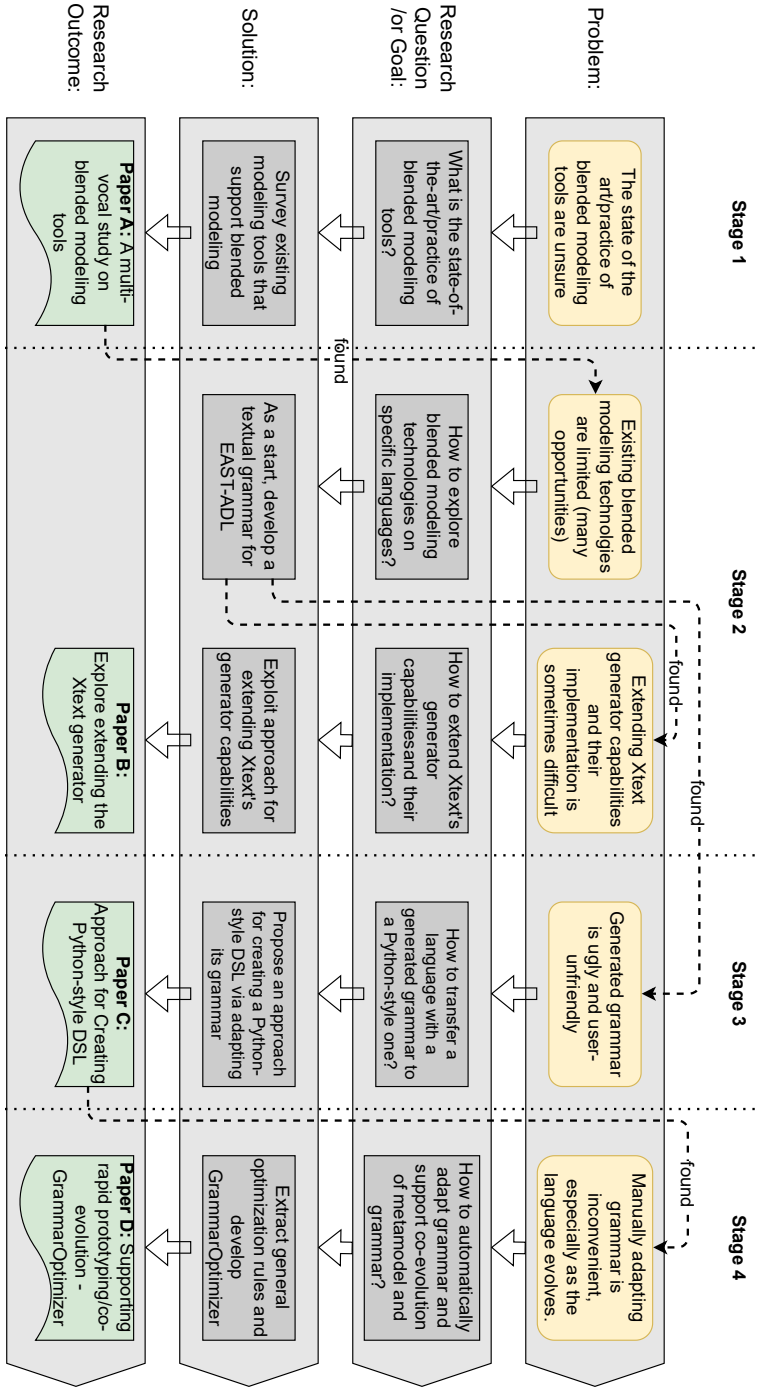

Figure 1.2: The research included in this thesis involves multiple problems. We have implemented different research activities and obtained some research results for these different problems.

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[1], Bibtex[2], DOT[3], SML[4], Spectra[5], Xcore[6], and Xenia[7], 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

[1] `https://eclipse.dev/atl/`
[2] `https://www.bibtex.com/`
[3] `https://graphviz.org/doc/info/lang.html`
[4] `http://scenariotools.org/scenario-modeling-language/`
[5] `http://smlab.cs.tau.ac.il/syntech/spectra/index.html`
[6] `https://wiki.eclipse.org/Xcore`
[7] `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[8]) 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.

[8] `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 mbdeddr [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 mbdeddr (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

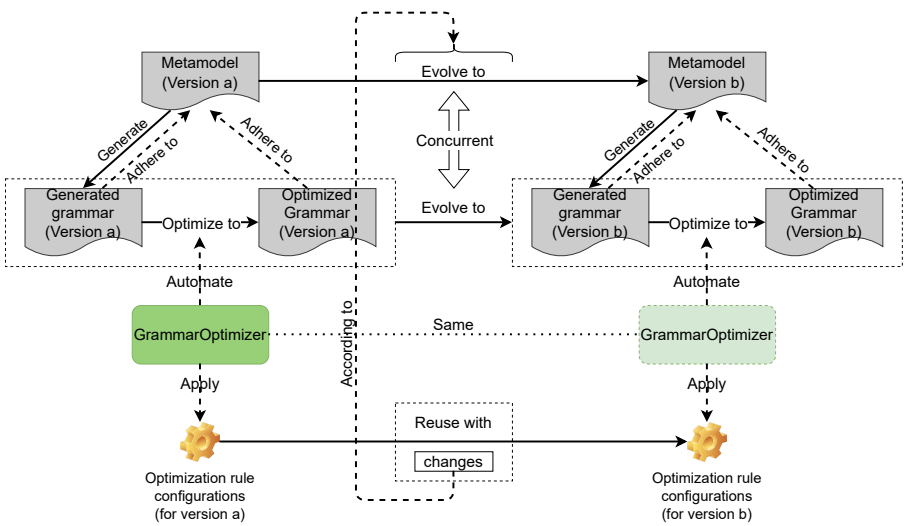

Figure 1.3: Schematic diagram of the proposed solution in **Stage 4** to support the co-evolution of meta-models and grammars.

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.