CHALMERS
UNIVERSITY OF TECHNOLOGY

Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution

Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution

Weixing Zhang
weixing@chalmers.se
Chalmers | University of Gothenburg
Gothenburg, Sweden

Regina Hebig
regina.hebig@uni-rostock.de
University of Rostock
Rostock, Germany

Daniel Strüber
danstru@chalmers.se
Chalmers | Gothenburg University, Gothenburg, Sweden
Radboud University, Nijmegen, Netherlands

Jan-Philipp Steghöfer
jan-philipp.steghoefer@xitaso.com
Xitaso IT and Software Solutions
Augsburg, Germany

Abstract

When a language evolves, meta-models and associated grammars need to be co-evolved to stay mutually consistent. Previous work has supported the automated migration of a grammar after changes of the meta-model to retain manual *optimizations* of the grammar, related to syntax aspects such as keywords, brackets, and component order. Yet, doing so required the manual specification of optimization rule configurations, which was laborious and error-prone.

In this work, to significantly reduce the manual effort during meta-model and grammar co-evolution, we present an automated approach for extracting optimization rule configurations. The inferred configurations can be used to automatically replay optimizations on later versions of the grammar, thus leading to a fully automated migration process for the supported types of changes. We evaluated our approach on six real cases. Full automation was possible for three of them, with agreement rates between ground truth and inferred grammar between 88% and 67% for the remaining ones.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Model-driven software engineering**.

Keywords: meta-models, grammars, co-evolution

ACM Reference Format:
Weixing Zhang, Regina Hebig, Daniel Strüber, and Jan-Philipp Steghöfer. 2023. Automated Extraction of Grammar Optimization Rule Configurations for Metamodel-Grammar Co-evolution. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623476.3623525

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623525

1 Introduction

Model-driven engineering is an important software engineering paradigm, in which models are considered as primary artifacts during software development [46]. Managing the consistency of artifacts produced during model-driven engineering is a hard problem. When a meta-model is updated, associated artifacts that still refer to the old version of the meta-model, such as model instances [23], model transformations [30], and code generators [37], become outdated and need to be migrated. The overall class of problems addressed here is referred to as *co-evolution* [23] or *coupled evolution* [3] and, due to its practical significance, has led to a large body of work, focusing on automated migration support (see, e.g., [6, 9, 19, 23, 27, 28, 39, 44, 44, 45]).

We consider a scenario in which a meta-model is co-evolved with an associated grammar. Such a scenario is common in cases where the meta-model defines the underlying abstract syntax for a modeling language, and the grammar defines a concrete textual syntax for that language [34]. In the technical space of Eclipse, the meta-model and grammar could be specified using Ecore and Xtext, respectively. In this scenario, there are two situations that lead to co-evolution: First, the meta-model evolves over time, rendering previous versions of the grammar obsolete. Second, in a rapid prototyping context, the meta-model evolves quickly and then requires the grammar to be updated quickly as well.

The main challenge with this scenario stems from two core requirements that typically need to be addressed:

- The updated grammar should be consistent with the new version of the meta-model.
- The updated grammar should incorporate any manual improvements that were made to previous versions of the grammar (e.g., adding and modifying keywords, changing the order of rule components, modifying and omitting brackets).

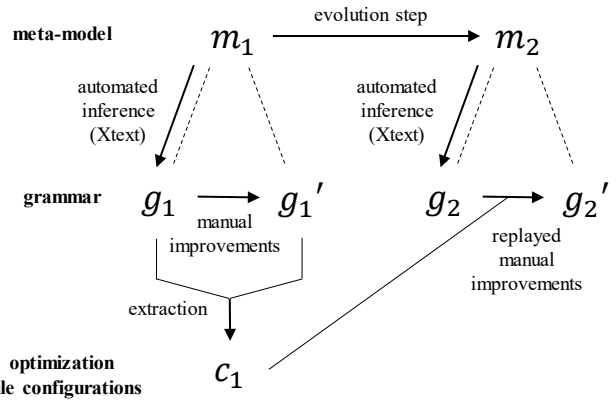

Figure 1. Overview of meta-model/grammar co-evolution; dashed lines indicate mutual consistency. Our contribution is to automate the extraction of c_1, previously done manually.

A tempting solution that addresses the first requirement is to automatically re-generate the entire grammar from the evolved meta-model, which is supported by platforms such as Xtext (in Fig. 1, arrow between m_1 and g_1). Yet, the second requirement renders this solution insufficient, as it leads to a laborious process, in which developers need to manually re-apply their optimizations to the re-generated grammar in every evolution step. For languages with extensive grammars like EAST-ADL, which encompasses approximately 300 grammar rules [12], this process simply seems infeasible when done manually.

We are aware of only one previous work that addresses this problem, a tool called *GrammarOptimizer* [58]. The key idea of GrammarOptimizer was to provide a catalog of grammar optimization rules that can be used to specify and then automatically perform the required changes for moving from a generated grammar to an optimized one. For example, in Fig. 1, the developers use these rules to specify a configuration c_1 that captures the improvements for moving from g_1 to g_1'. We will introduce this tool further in Sect. 2. Yet, the tool in its current version has a major drawback: manually instantiating the grammar optimization rules to specify migrations can be cumbersome and error-prone. Specifying the right configuration for the task at hand involves choosing the right set of rules together with correct values for parameters. This is a complicated configuration process.

In this paper, to considerably reduce this manual specification effort, we present an approach for automating the configuration of grammar optimization rules. Our approach assumes that two versions of the grammar are available: an automatically generated one and a manually optimized one; we call the latter the *target grammar*. In a co-evolution scenario, these grammars come from a previous state in history, and the manual optimizations evident in the target grammar should inform the migration of the grammar towards a new version of the meta-model (in Fig. 1, consider the generated grammar g_1, the target grammar g_1' and the new meta-model version m_2). Our approach can then automatically extract an *optimization rule configuration* that encodes the manual improvements. Technically, our approach works by establishing a mapping between the grammar rules from both grammars and then, per rule, performing a line-by-line comparison to extract invocations of relevant grammar optimization rules with their parametrizations. We automated this process by developing a tool named *ConfigGenerator*.

The extracted configuration can be applied to a newly-generated version of the grammar based on the evolved meta-model. For changes of the types supported by our approach, this entirely avoids any manual effort for specifying and re-applying the manual optimizations. In Fig. 1, after the evolution step that created m_2, replaying the changes between g_1 and g_1' on the generated grammar g_2 using the automatically extracted configuration c_1 leads to the target grammar g_2'. Once the target grammar is available, new and changed meta-model elements may lead to new manual optimizations on top of it. In that case, our approach can be applied after meta-model changes to capture these changes in a new version of the configuration. That way, our approach provides support for meta-model/grammar co-evolution throughout the history of an evolving language.

To evaluate our approach, we applied the *ConfigGenerator* to six cases of languages whose meta-models and grammars are available: EAST-ADL, Bibtex, Xenia, Xcore, DOT, and SML. The results show that our approach is able to extract complete configurations for three of the cases (EAST-ADL, Bibtex, and Xenia). For these languages, the target grammars yielded by replaying the optimizations are identical with an existing ground truth grammar. For the other three languages, the optimization rates — defined as the agreement between a ground truth grammar and the grammar obtained by replaying — are between 87.5% and 68%. These findings indicate the potential and effectiveness of ConfigGenerator in extracting optimization rules based on the comparison between generated grammar and target grammar.

2 Background

2.1 Xtext and DSL Generation

Eclipse Xtext is a framework for developing software languages, including modeling languages [15]. Xtext offers two approaches for implementing the grammar design of a textual DSL [42]. One approach involves creating an Ecore meta-model to represent domain concepts and their relationships, and then generating an Xtext grammar from the meta-model (in the remainder of the paper, we call it *generated grammar*). The other approach is first to create a grammar and then derive a meta-model from it. The scope addressed in this paper involves the former approach.

Listing 1. Example from Xenia: generated grammar rule `SiteWithModal`.

```
1  SiteWithModal returns SiteWithModal:
2   {SiteWithModal}
3   'SiteWithModal'
4   name=EString
5   '{'
6    ('sites' '{' sites+=SuperSite ( "," sites+=
         SuperSite)* '}' )?
7   '}';
```

In Xtext, grammars are specified in an EBNF (Extended Backus-Naur Form) format, augmented with references and annotations that specify the relationship to the Ecore meta-model. The meta-model represents the abstract syntax for language at hand (classes with their features, including names, attributes, and references), while the augmented EBNF expression describes the concrete syntax and its mapping to specific parts of the meta-model. Listing 1 shows an example of a grammar rule in Xtext, from the context of Xenia [54, 55], one of our evaluation cases. The depicted rule, `SiteWithModel`, contains both traditional EBNF elements for specifying the syntax, as well as several annotations and references. In particular, the `returns` keyword is followed by a reference to the `SiteWithModel` class, and several grammar elements are mapped to attributes (`name`) and references (`sites`) from that class, using the '=' and '+=' operands. By defining grammar rules and associating them with the corresponding meta-model elements, Xtext enables the automatic generation of a parser and other language tools. The parser uses the grammar rules to parse the input code and create an abstract syntax tree (AST) that conforms to the meta-model elements. This AST can then be further processed or used for various purposes in language development.

2.2 GrammarOptimizer and Optimization Rules

We now provide additional details for the GrammarOptimizer tool [58] that, in particular, provides the grammar optimization rules we automatically configure with our approach. Their approach includes 54 optimization rules extracted from seven sample languages, which are used to optimize the generated grammar (explained above). These optimization rules operate on various elements within the grammar, including keywords, curly braces, symbols, and optionality. For example, `AddKeywordToAttr` is used to add a new keyword to a specific attribute, `ChangeBracesToSquare` is used to transform specified curly braces into square brackets, and `RemoveRule` is used to remove unnecessary grammar rules. Their tool is an Eclipse plugin developed in Java.

To use GrammarOptimizer, language engineers need to manually select and configure the optimization rules for performing the intended changes. Given a selected rule, configuring it involves invoking methods of a Java class representing the application of that rule, with parameters such as the name of the relevant grammar rule and involved elements, such as attribute names and keywords. These parameters enable GrammarOptimizer to accurately locate the specific targets in the generated grammar that need to be modified.

Listing 2. Example from Xenia: target grammar rule `SiteWithModal`; all attributes and keywords are now on the same line.

```
1  SiteWithModal:
2   '@' name=ID 'with' 'modal' '(' sites+=
        SuperSite (',' sites+=SuperSite)* ')'
3  ;
```

As an example, consider Listings 1 and 2. Listing 1 shows the grammar rule `SiteWithModal` from Xenia's generated grammar, while Listing 2 shows the version of that rule in the target grammar. We focus on the `name` attribute, which has different types in the two grammars: `EString` and `ID` in the generated and target grammar, respectively. While editing the generated grammar manually to change the type is simple, this change cannot be recovered if the grammar is re-generated after a meta-model change, unless dedicated support is provided.

Hence, the language engineer uses GrammarOptimizer. Doing so involves identifying the relevant optimization rule, in this case, `changeTypeOfAttr`. To configure the optimization rule, the engineer instantiates the `GrammarOptimizer` class which acts as a facade and defines a public method for each of the optimization rules. The `changeTypeOfAttr` method accepts four parameters: the names of the grammar rule, of the attribute name, of the current type, and of the new type. In this example, where the instantiated `GrammarOptimizer` object is named `go`, the configuration of the optimization rule call to modify the type of `name` is as follows: `go.changeTypeOfAttr("SiteWithModal", "name", "EString", "ID")`.

3 Related Work

Recovery of grammars and meta-models. In legacy systems, a common situation is that the underlying grammar or meta-model is absent and has to be recovered from available instances (programs or models, respectively). Available solutions are based on using available compiler sources and language reference manuals [35], evolutionary computing [4, 25], or iteratively provided user input [38]. With a focus on supporting grammar recovery scenarios, Lämmel [32] provides a set of operators for grammar modification, focusing on refactoring, construction, and destruction. Yet, recovery approaches such as those discussed are not applicable to the scenario considered in this paper, in which the grammar instances, after an evolution step, still conform to the old, known grammar.

Co-evolution in MDE contexts. In model-driven engineering, it is well-known that evolutionary changes to an artifact

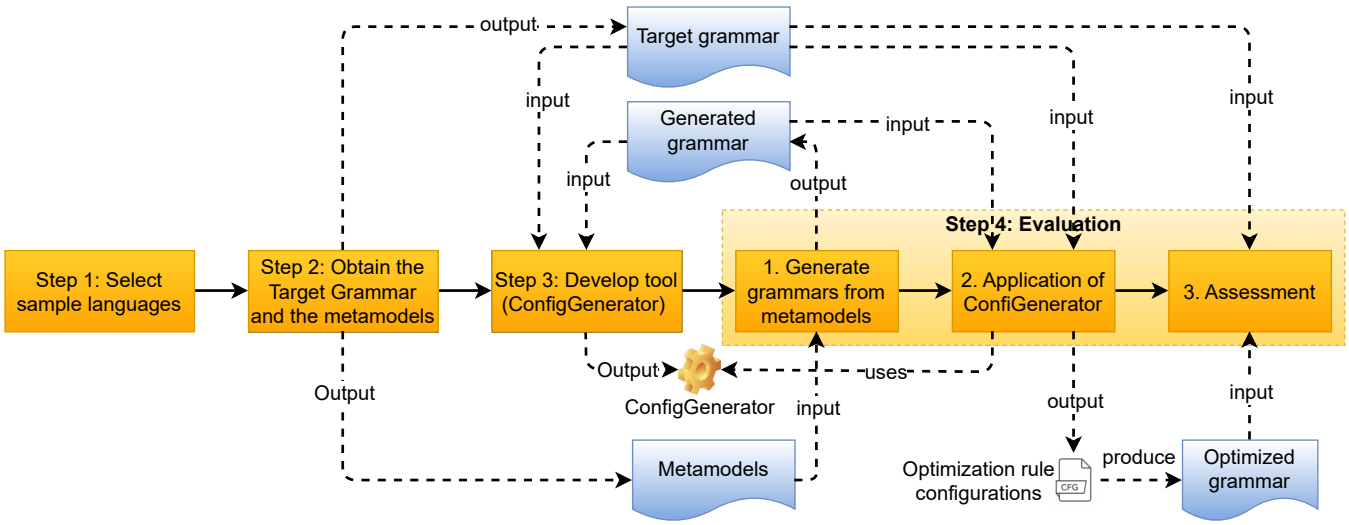

Figure 2. Schematic diagram of the whole process of the research methodology.

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. [23] 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 [29], model transformations [30, 31], code generators [37], and graphical editor models [10]. 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 [8]. However, except for *GrammarOptimizer* [58] (described in Sect. 2), on which we build and improve with our contribution, we are not aware of previous work on meta-model/grammar co-evolution.

Model federation [11, 20, 22] deals with 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 that 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.

Automated rule extraction. A line of work focuses on automating the extraction of transformation rules in specific contexts. Model transformation by example [51, 53] is an important paradigm in which entire transformations are recovered from a set of user-provided examples. While the seminal work in this area mostly relied on custom heuristics, recent works have studied applications of AI, in particular, reinforcement learning [17] and deep learning [2]. Apart from these approaches for general transformation inference, there are task-specific approaches, including the refactoring of redundant rules [49, 50] and of mutation operators [47]. These approaches are orthogonal to ours, as we focus on the automated extraction of *configurations* of rules.

From meta-models to graph grammars. Beyond EBNF-style grammars as considered in this paper, grammarware in the broader sense also encompasses graph grammars, which are a rule-based approach for generating instances for a given meta-model, e.g., for testing purposes. A seminal approach by Ehrig et al. [16] supports the generation of a graph grammar in the double-pushout approach to graph rewriting, using advanced transformation features such as negative application conditions. Fürst et al. present an approach that aims to avoid the use of such advanced features that make analysis more complicated, while being sufficient for meta-models with arbitrary multiplicities and inheritance [18].

Text-based merging. Simple cases of our considered scenario could be covered by standard text merging tools, such as Git merge [7]. To this end, the user would perform manual optimizations and re-generation of the grammar in separate branches, and then merge the branches. However, text-based merging operates on the abstraction level of text rather than

grammar structures, which leads to several drawbacks: First, it easily leads to merge conflicts. For example, when the same line is manually optimized (e.g., changing a keyword) and affected by a change in the underlying meta-model (e.g., removing an attribute), a merge conflict arises, whereas our approach supports this example. Second, it does not give an easily inspectable, semantically meaningful overview of the changes, as grammar optimization rules do. In that sense, our approach can be seen as a form of semantic lifting [26] of grammar differences, focused on grammar optimizations.

Grammar convergence. Our contribution bears a connection to grammar convergence [36]. Grammar convergence aims to extract a series of transformations to make two considered grammars syntactically identical, which is similar to our goal. Yet, the grammars in their approach stem from heterogeneous sources (e.g., different parsers for the C++ language), instead of being based on the same underlying source meta-model in several versions, which gives both approaches different knowledge to rely on. A relevant scenario is metalanguage evolution [56], in which the notation used to define the considered languages, instead of the languages themselves, evolves, which necessitates changes in associated artifacts (e.g., parsers). Another one is style normalization for X-to-O mappings, which aims to bridge heterogeneity in different XML different styles when supporting their mapping to object models [33].

4 Methodology

The research methodology in this study consists of the steps shown in Figure 2. The first two steps were performed to prepare the inputs for Step 3, in which we developed the *ConfigGenerator*, and for Step 4, in which we evaluated our approach. All steps are described in the following.

4.1 Step 1: Select Sample Languages

In the first step, we selected appropriate case languages. These chosen languages served as the foundation for our solution and evaluation. Since our goal was to make our approach applicable to real-world DSLs, we needed to select a set of real-world DSLs for which both a grammar and a meta-model were available. In our previous work [58], we identified 9 such DSLs through an extensive search. We decided to directly work with a subset of six of their considered languages–Bibtex, DOT, EAST-ADL (full version), SML, Xcore, and Xenia–, which has the following benefits: First, the considered languages covered a diverse range of domains. Second, we knew from their evaluation that GrammarOptimizer could be used to optimize grammars for these languages. Since GrammarOptimizer was a baseline tool for our approach, working with these languages ensured that any observed issues stem from our approach for automated configuration extraction, and not from our baseline tool. Our reason for selecting a subset was that four of their considered languages had complications that led to a lack of full support (e.g., using OCL as part of the grammar definition). We still included one of the not-fully-supported languages, SML, to study the effect of applying our approach to one case from that category.

4.2 Step 2: Obtain the Target Grammars and Meta-models

After selecting the case languages, we obtained their meta-models and target grammars. The information regarding the meta-models (source and the number of classes) and target grammars (source and the number of grammar rules) is presented in Table 1. We noticed that the meta-models for Bibtex and SML needed adjustments to be effectively used, which we completed in previous work [58]. Therefore, we directly adopted our prepared meta-models for Bibtex, SML, and DOT, and obtained the meta-models for the other three languages from their respective sources.

ConfigGenerator takes two Xtext grammars as input: the target grammar and the generated grammar (i.e., the grammar newly generated from the meta-model). We observed that EAST-ADL and Bibtex did not have original grammars in Xtext, so we directly adopted the optimized grammars from [58] as the target grammars for these two languages. As for the other four languages, their Xtext grammars were already provided in their respective sources, and we simply copied these Xtext grammars as the target grammars.

4.3 Step 3: Develop Tool

In the 3rd step we developed the ConfigGenerator. We developed the initial version of ConfigGenerator based on EAST-ADL. The development of ConfigGenerator involved the implementation of comparisons for different grammar elements. Each time we implemented a comparison method for a specific grammar element (e.g., comparing line orders), we applied it to compare two EAST-ADL grammars and check the selected optimization rules. If the selected optimization rules differed from our expectations, we used the debug mode to identify the reasons behind the differences and fixed them. Once we had the initial version of ConfigGenerator, we applied it to Xenia to refine its implementation. In this context, we considered that target grammars might contain manual modifications that could affect line-by-line matching. For example, in Xenia's target grammar, some different attributes are placed on the same line. This situation impacts our line recognition and then matching. Consequently, when applying ConfigGenerator to Xenia, we developed handling methods for this situation.

4.4 Step 4: Evaluation

To validate our approach, we applied it to all six languages identified in Step 1. Our goal was to explore whether and to what extent the ConfigGenerator built based on EAST-ADL and Xenia, could also be applied to other DSLs. In our

Table 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.

	Meta-model		Target grammar		Generated grammar			Used in[2]	
DSL	Source	Classes[1]	Source	Rules	lines	rules	calls	Dev.	Eva.
EAST-ADL	EATOP Repository [12]	291	[58]	297	2839	297	3062	YES	YES
BibTex	[58]	48	[58]	43	293	43	188	NO	YES
Xenia	Github Repository [54]	15	Github Repository [55]	13	84	15	36	YES	YES
DOT	[58]	19	Dot [40]	21	125	23	51	NO	YES
Xcore	Eclipse [13]	22	Eclipse [14]	26	243	33	149	NO	YES
SML[3]	[58]	48	SML repository [21]	45	658	96	377	NO	YES

[1] The metrics are assessed after adaptations and contain both classes and enumerations.
[2] These two metrics indicate whether the language is used in the step "Development (Dev.)" or "Evaluation (Eva.)".
[3] The metrics of SML are based on excluding the embedded SML expressions.

previous work [58], we had already shown that an optimization rule configuration created to optimize one version of a language could be reused, with a few changes, for another language version. Thus, we aimed at evaluating whether ConfigGenerator could create a correct optimization rule configuration for a language version given the generated and target grammar. To do so, we performed the following steps.

4.4.1 Step 4.1: Generate Grammars From Meta-models.

When the meta-model was ready, we created an empty EMF project for the language in Eclipse and imported its meta-model. Then, utilizing Xtext, we automatically generated the Xtext grammar from the meta-model. We performed this process for each language.

4.4.2 Step 4.2: Application of ConfigGenerator.

Next, we applied the ConfigGenerator to all of these languages by comparing the target grammar with the generated grammar and extracting the optimization rule configurations. These extracted optimization rule configurations can be used by GrammarOptimizer. We then used the created optimization rule configurations with the GrammarOptimizer on the generated grammars of these languages to automatically create an optimized grammar.

4.4.3 Step 4.3: Assessment.

We conducted a comprehensive comparison between the optimized grammar and the target grammar of each language, based on a one-to-one comparison of corresponding grammar rules.

To assess the similarity between the optimized grammar and the target grammar, in our final step we decided to assess the following metrics which will be listed in Table 2: To provide an impression of the amount of manual adaptation that needed to happen to change the generated grammar to the target grammar, the 4th to 6th columns indicate the number of grammar rules that were modified, removed, and added from the generated grammar to the target grammar. Further, we show how big optimization rule configurations for these languages are. The 2nd column shows the number of lines of optimization rule configurations used in our previous work [58], which are capable of optimizing the generated grammar to achieve an identical state as the target grammar. We referred to the optimization rule configurations used in our previous work [58] as "manual" configuration, since these configurations had to be written by hand. The 3rd column represents the number of lines of the optimization rule configurations extracted by ConfigGenerator. We also listed in columns 7 to 9 the number of grammar rules that were modified, removed, and added from the generated grammar to the optimized grammar. Finally, we aimed to assess how complete the generated optimization rule configurations are. Thus, the last three columns provide statistics on the comparison of the optimized and target grammar, i.e., whether all grammar rules for these languages are the same between the optimized grammar and the target grammar. Specifically, "Same" represents the number of grammar rules that are identical in both grammars, "Diff" represents grammar rules that are not identical, and "Percent" indicates the percentage of grammar rules that are identical between the two grammars.

5 Solution

In this section we present the ConfigGenerator, which creates an optimization rule configuration based on a generated grammar and a target grammar, to enable a re-application of manually defined grammar changes after a meta-model changed and a new grammar was generated. We first introduce and reason about the assumptions we made when building our solution. Afterward, we explain how grammars are compared (rule-to-rule and line-to-line) and how the comparison result is used for generating the configuration.

5.1 Assumptions

Based on the technical reality and practice of Xtext, we made the following assumptions about our solution:

- A grammar rule name is unique across the grammar. Otherwise, Eclipse will prompt "A rule's name has to be unique." error.
- An attribute name is unique within a grammar rule, because the attributes in the generated grammar are unique.
- Attribute names are not modified by users when they manually create a target grammar out of a generated grammar. Otherwise, this may cause the grammar and the meta-model to become incompatible.

5.2 Grammar Comparison Workflow and Grammar Rule Matching

ConfigGenerator selects and parameterizes optimization rules by comparing two input grammars. The selected optimization rules form a configuration that can then be utilized by GrammarOptimizer.

Figure 3 illustrates the internal workflow of ConfigGenerator for selecting the required optimization rules by comparing two grammars. It parses the generated grammar and creates a list Rules_{gen} containing instances of a data structure for each grammar rule. Each instance contains all lines of text that make up that grammar rule. ConfigGenerator does the same for the target grammar to create a list Rules_{tgt}.

ConfigGenerator traverses Rules_{gen}, taking one grammar rule at a time and searching for the grammar rule with the same name in Rules_{tgt}. If no match is found, it indicates that the grammar rule has been deleted in the target grammar, thereby requiring the selection and parameterization of an optimization rule for deleting that grammar rule. If a match is found, a line-by-line comparison is performed between the grammar rules to identify the required optimization rules.

Once the entire traversal of Rules_{gen} is completed, ConfigGenerator performs a reverse traversal. In this reverse traversal, ConfigGenerator retrieves one grammar rule at a time from Rules_{tgt} and searches for the corresponding rule in Rules_{gen}. If a match is found, ConfigGenerator takes no action (as the comparison has already been done in the previous traversal). If no match is found, it signifies that the grammar rule is newly added in the target grammar. In this case, an optimization rule for adding the grammar rule is selected and parameterized.

After both traversals are completed, ConfigGenerator yields an optimization rule configuration with the selected and parameterized optimization rules and writes it into a text file.

5.3 Normalization of Rules_{tgt}

Before performing line-by-line mapping, we need to perform normalization checks and operations on the rules of the target grammar. Because the target grammar may have traces of manual modification that are not conducive to our line-by-line matching. For example, in the generated grammar of Xenia, each of the different attributes and also the `{SiteWithModal}` action, the opening brace, and the closing brace each has its exclusive line, as shown in Listing 1. However, in the target grammar of Xenia, all attributes and keywords of the grammar rule `SiteWithModal` are placed on the same line as shown in Listing 2. This situation hinders row-to-row matching and thus needs to be normalized.

In particular, we begin by examining whether the following situations exist within a grammar rule: 1) different attributes are placed on the same line, 2) an Action with the same name as the grammar rule is combined with any other non-empty string on the same line, and 3) symbols are placed on separate and exclusive lines. If any of these situations are present, normalization is performed. During the normalization process, we gather all lines except the one containing the grammar rule name into a single string, which is then split. Using regular expressions, we separate different attributes into distinct lines, ensuring that each attribute has its own line. Similarly, if there is an Action with the same name as the grammar rule, an opening brace, and a closing brace, we allocate separate lines exclusively for each of them. Additionally, all symbols are placed in adjacent attribute lines rather than being treated as separate lines themselves, e.g., place the symbol `':'` after the attribute `name`.

5.4 Line Matching

As described in Section 5.2, ConfigGenerator matches grammar rules by traversing two lists. After completing the matching of Rules_{gen} and Rules_{tgt}, we need to match the lines between them, forming the foundation for line-to-line comparison. With the exception of attribute lines, all other lines have only one occurrence within the same grammar rule. Therefore, ConfigGenerator only needs to find the corresponding unique line to complete the line matching. For example, to compare the main keyword of the same grammar rule, we search for the main keyword in both the Rules_{gen} and Rules_{tgt}. We call the keyword with the same name as the grammar rule in the generated grammar "main keyword". If both sides find a line containing the main keyword, then the two lines from both sides match each other.

For the matching of attribute lines, we need to recall the assumption set earlier, which states that each attribute within the same grammar rule has a unique name, and language engineers do not modify the names of attributes when modifying the grammar. Therefore, we can use the attribute name as a unique identifier for lines to perform line matching. Specifically, when matching a line, we first take an attribute line from a grammar rule in the generated grammar, and then search for the line with the same attribute name within the corresponding grammar rule in the target grammar, thus completing the line-matching process.

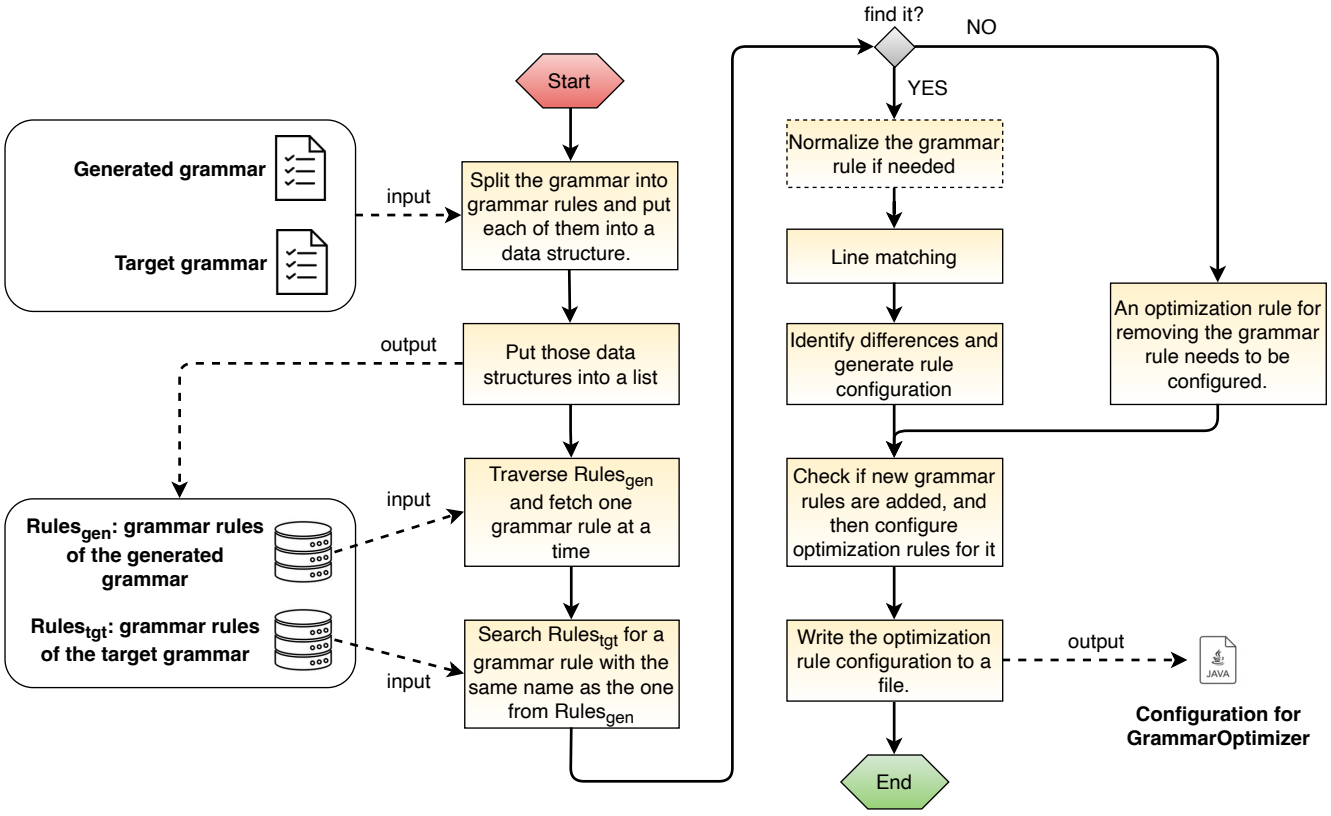

Figure 3. The workflow of extracting optimization rule configurations based on a comparison between the generated grammar and the target grammar.

5.5 Difference Identification, Rule Selection and Parametrization

Once we completed the line-by-line matching as mentioned in the previous section, the next step is to perform the line-by-line comparison. As previously stated, except for attribute lines, the other types of lines are unique within a grammar rule. We only need to compare if they have been removed or renamed. For example, if the ConfigGenerator finds container braces (the outermost braces within a grammar rule) in the grammar rule `Model` in the generated grammar but not in the same grammar rule in the target grammar, it will select and parameterize an optimization rule for removing the braces.

Comparing attribute lines is more complex because they typically consist of multiple elements. Firstly, an attribute line always contains an attribute string which is usually in the form of e.g., `attributeName=typeName`. Additionally, it may include keywords, asterisks indicating multiplicity following parentheses, commas, and curly braces enclosed in single quotation marks, among other elements. The use of regular expressions enables us to identify and distinguish different elements, allowing for their comparison across different grammars. For example, when examining the attribute `ownedComment` in the same grammar rule on both sides, we may observe that in the generated grammar, this attribute is preceded by the keyword `'ownedComment'`, whereas in the target grammar, there is no keyword preceding it. In such a case, an optimization rule named `removeKeyword` would be selected and parameterized.

6 Evaluation

6.1 Results

Table 2 summarizes the results of applying ConfigGenerator to extract optimization rule configurations for different languages (see Table 1 for information about the sources and initially generated grammars of these languages).

The 2nd and 3rd columns show the size of the optimization rules configurations. For comparison, we first show the number of configuration lines in the manually created optimization rule configurations from our previous work [58]. Next to it are the number of configuration lines in the optimization rule configurations that were extracted by ConfigGenerator. For several languages, e.g. BibTex, the automatically extracted configuration had much more lines than the manually written counterpart (with the extreme case of EAST-ADL, where the extracted configuration is up to 100 times as long as the manually written one). This difference

Table 2. Results of applying ConfigGenerator to extract optimization rule configurations for different languages.

Language	Configuration lines		Target grammar[2]			Optimized grammar			Grammar Comparison		
	Manual	Extracted	Change	Remove	Add	Changed	Removed	Added	Same	Diff	Percent
EAST-ADL	31	3378	233	1	12	233	1	12	297	0	100%
BibTex	47	254	43	0	0	43	0	0	43	0	100%
Xenia	74	114	13	2	0	13	2	0	15	0	100%
DOT	79	134	24	3	1	16	3	1	21	3	87.5%
Xcore	307	351	20	14	7	17	14	7	28	12	70.0%
SML	421	369	40	56	8	38	56	8	51	24	68.0%

[1] The numbers in column 2, 4, 5 and 6 were obtained from the supplemental materials of our previous work. [58].
[2] Number of grammar rules that would need to be changed/be removed/be added to create the target grammar out of the generated grammar.

can be explained by the fact that the manually written optimization rule configurations make use of generalizations. I.e., instead of introducing for each grammar rule a new configuration removing curly braces, manually created configurations might just introduce one configuration that applies to all grammar rules. This generalization could in theory be imitated automatically, too. However, it would require additional analysis of the side effects of the generalized rule to make sure that no unintended changes happen. Therefore we, kept this to future work.

The columns for target grammar show the difference between the generated and the target grammar in the form of the number of grammar rules that require a change, require to be removed, and require to be added. These numbers are reported by our previous work. [58]. It can be seen that in all languages the majority of the grammar rules would need to change. This illustrates how different Xtext-generated grammars really are from grammars used in real languages and further illustrates the need to capture and preserve the manual effort made to create grammars.

Table 2 displays the number of changed, removed, and added grammar rules in the optimized grammar compared to the generated grammar in columns 7 to 9. The rightmost three columns compare the differences in grammar rules between the optimized grammar and the target grammar. For EAST-ADL, BibTex, and Xenia, we see the same amount of changed, removed, and added grammar rules as we would have expected judging from the target grammar. However, for DOT, Xcore, and SML the number of changed grammar rules is lower. This is already an indication that the generated optimization rule configuration did not perform the complete adaptation targeted for these three languages.

Finally, the last columns in Table 2 summarize how the optimized grammar compares to the target grammar. The results confirm that the grammar rules in the generated grammars of EAST-ADL, Bibtex, and Xenia have been optimized to be identical to the target grammar using the extracted optimization rule configurations. In the case of DOT, 87.5% of the grammar rules in the optimized grammar are identical to the target grammar. For Xcore and SML, the corresponding figures are 70.0% and 68.0%, respectively. Below we will discuss more in detail, when the ConfigGenerator performed well and when not.

6.2 Capabilities of ConfigGenerator

Although ConfigGenerator cannot optimize all grammar rules in the generated grammars of DOT, Xcore, and SML to achieve an identical state as their target grammars, it still provides the optimization rule configurations which perform the majority of necessary changes. Specifically, configuration rules for the following grammar changes were correctly generated in all cases:

- Removing or renaming individual keywords, including changing the value of `literals` in enumerations.
- Removing grammar rules or attributes.
- Modifying the multiplicities of attributes, including changing optional attributes to mandatory ones.
- Removing braces, including removing braces in attribute lines and container braces.
- Modifying the order of lines in a grammar rule, as long as lines can be identified by attribute names.
- Adding symbols to individual attribute lines.
- Adding rules, including adding terminal rules and primary type rules, as well as completing primary type rules which are to be implemented.
- Removing calls to other rules in grammar rules.
- Changing a specific type in the cross-reference of an attribute.

6.3 Missing Capabilities of ConfigGenerator

For the languages DOT, Xcore, and SML, there are a total of 42 grammar rules with differences between the optimized grammar and the target grammar. These differences can be found in the supplemental materials of this paper [41]. Here, we provide a list of typical cases of these differences.

- Difference in line order in some specific cases. In cases where one of the lines moved contains only a main keyword that has been changed by another configuration

Listing 3. Two attributes in the grammar rule `XOperation` in the *generated* grammar of Xcore

```
1  ...
2  (unordered?='unordered')?
3  (unique?='unique')?
4  ...
```

Listing 4. Two attributes in the grammar rule `XOperation` in the *original* grammar of Xcore

```
1  ...
2  unordered?='unordered' unique?='unique'? |
3  unique?='unique' unordered?='unordered'?
4  ...
```

rule, the reordering of the lines might not work. For example, in Xcore, the order of lines in the `XPackage` grammar rule is different between the optimized grammar and the target grammar. In the optimized grammar, the attribute `annotations` appears after the attribute `name`, while in the target grammar, it appears before the attribute `name`.

- Inconsistent attribute grouping. Listing 3 shows two attributes of the `XOperation` grammar rule in Xcore. In the generated grammar, they are listed one after the other, indicating an order of their appearance. In the target grammar, they are combined together and are in "and" and "or" relationships (as shown in Listing 4), indicating that their order is not predefined. The ConfigGenerator is not able yet, to create a configuration that replicates the occurrence of attributes in the grammar like that.
- Braces not changed to square brackets. In DOT, e.g., the container of the grammar rule `AttrList` in the generated grammar uses square brackets (i.e., `'['` and `']'`), while in the optimized grammar, it uses braces (`'{'` and `'}'`).
- Difference in the position of optionality. In DOT, e.g., there is an attribute `attributes` in a grammar rule where optionality (i.e., `()?`) is handled differently. In the target grammar, the added comma and semicolon are surrounded by `()?`, i.e., `(',' | ';')?`. However, in the optimized grammar, the attribute string is surrounded by `()?`.

The cause of these limitations is that ConfigGenerator uses a line-based text comparison to identify which lines correspond to each other in the generated and in the target grammar and derives optimization rule configurations from this comparison. This limitation can be remedied by relying on a comparison that is based on an abstract syntax tree (AST) instead: the tool could parse both the generated and the target grammar and compare the ASTs, potentially making it more robust to changes in the order of lines and for groups that span multiple lines as these syntactical issues would not be present in the AST. Such an approach would also reduce the reliance on regular expressions which can be a limiting factor as well (see the fourth point in the list above). However, such an implementation is left for future work.

Finally, GrammarOptimizer, adopted without feature-level modification from our previous work [58] has limitations, e.g., rules to transform braces into brackets as needed for the third point mentioned above. While it would be possible to emulate this by first applying a rule that removes the braces and then adds brackets back, we have decided not to use this more complex strategy at this stage and leave the combination of different optimization rules as future work.

6.4 Usefulness of ConfigGenerator

The results in Section 6.1 indicate that the current version of ConfigGenerator can produce an optimization rule configuration that allows to modify a generated grammar fully into the target grammar for *some* of the languages we tested. In our initial work on GrammarOptimizer [58], we have shown that it is possible to find an optimization rule configuration manually to transform all of the languages we included in our evaluation to the target grammar.

We argue that ConfigGenerator is still a useful tool, even if it cannot fully derive a complete optimization rule configuration for all languages yet. SML, for instance, requires a total of 421 parameterized optimization rule invocations to be fully transformed. Creating all of them manually is a significant effort. ConfigGenerator automatically extracts 369 rules and therefore provides an excellent starting point for a language engineer to complete the optimization rule configuration.

ConfigGenerator is intended for use in scenarios where languages evolve and where they are rapidly prototyped. In such situations, speed is critical and ConfigGenerator increases the speed with which a language engineer can create an optimization rule configuration, even if it requires manual adaptations. We follow the line of argument from our previous work, that making manual changes to a reusable artifact such as an optimization rule configuration is less effort and faster than manually transforming a large grammar repeatedly.

6.5 Threats to Validity

There are several threats to the internal validity of our evaluation. The first stems from the fact that we worked with the slightly adjusted meta-models as well as Xtext versions of target grammars, which were partially not originally written in Xtext, from our previous work [58]. It is possible that these preparation steps introduce differences to the languages and, thus, might have simplified the task of changing the grammar and with that also the configurations that needed to be generated.

A further threat to internal validity concerns correctness: To which extent can we produce grammars that not only syntactically, but also semantically agree with manually changed counterparts? In our evaluation, most generated grammar rules were syntactically identical to their manually written counterparts, which indicates semantic equivalence and thus, correctness in these cases. This applies to three considered languages completely, and to three partially (68-87%). All observed differences were analyzed manually, as reported with details in Sections 6.2 and 6.3. The main practical implication of these cases is that existing grammar instances can no longer be parsed, which makes these inaccuracies easy to spot for the user.

However, analysing semantics and providing correctness guarantees during evolution are intrinsically hard problems. That is because a formalized semantics might not be available (as in our evaluation cases), and, where it is, the semantics might change over time. For example, UML 2 introduced a new Petri-net semantics for sequence diagrams. Supporting such evolution steps in semantics-sensitive way requires specialized approaches for the involved semantic representations (if available). Still, from our experience as language developers [1, 24, 43, 48, 52, 57], changes to semantics of existing language elements are exceedingly rare, and then require careful navigation on part of the developer. Specialized approaches could help, but are outside our scope. The vast majority of changes either add or remove language elements or change the syntax, which is exactly our scope.

Another threat that might make us overestimate the ability of the ConfigGenerator is that we could not build it without consulting real language examples. In consequence, the tool is very likely to work very well for the two used languages EAST-ADL and Xenia. To mitigate these two threats, we made sure to evaluate the tool on four additional languages, to also reduce the impact that changes to meta-models and target grammars of single languages might have had.

Finally, there is a threat to the external validity of generalizability. Of course, using more languages would have given us more insights into how well the ConfigGenerator already works. However, the languages we worked with are fairly different in character, which allows us to cover at least some level of language variety.

7 Conclusion

We presented an approach for supporting the co-evolution of meta-models and associated grammars. Our technical contribution is a technique for automatically extracting *grammar optimization rules*, which capture manual improvements from a previous evolution step, and allow these improvements to be replayed on future versions of the grammar. Our evaluation indicates a perfect coverage for three out of six considered cases – including a large one, namely, EAST-ADL – while showing good coverage with clearly identified limitations in the remaining ones.

We foresee several directions for future work. First, we aim to further improve the coverage of our approach. One idea is to move the grammar comparison to the level of ASTs, rather than lines, which would help to improve support for multiple-line changes. Second, a complementary co-evolution scenario to the one addressed in this paper, which requires support as well, involves migrations of the meta-model after changes to the grammar. Third, we intend to provide support for all-quantified rules (e.g., removing curly braces from all grammar rules) via automated generalization. This would allow to extract considerably more compact and easy-to-read rule configurations. Fourth, a comprehensive evaluation of our technique in concert with the baseline technique from [58] on a full-fledged co-evolution scenario would yield further insight into the practical applicability of our approach. The next step of the work [58] can be to apply GrammarOptimizer to build a language workbench that supports blended modeling [5]. If the automatic extraction capability of this paper can be integrated, it will certainly assist the textual grammar optimization ability of this workbench.

References

[1] Hugo Bruneliere, Jokin Garcia, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, and Jordi Cabot. 2015. On lightweight metamodel extension to support modeling tools agility. In *ECMFA*. Springer, 62–74.

[2] Loli Burgueno, Jordi Cabot, Shuai Li, and Sébastien Gérard. 2022. A generic LSTM neural network architecture to infer heterogeneous model transformations. *Software and Systems Modeling* 21, 1 (2022), 139–156.

[3] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. 2009. Managing dependent changes in coupled evolution. In *Theory and Practice of Model Transformations: Second International Conference, ICMT 2009, Zurich, Switzerland, June 29-30, 2009. Proceedings 2*. Springer, 35–51.

[4] Matej Črepinšek, Marjan Mernik, Faizan Javed, Barrett R Bryant, and Alan Sprague. 2005. Extracting grammar from programs: evolutionary approach. *ACM Sigplan Notices* 40, 4 (2005), 39–46.

[5] Istvan David, Malvina Latifaj, Jakob Pietron, Weixing Zhang, Federico Ciccozzi, Ivano Malavolta, Alexander Raschke, Jan-Philipp Steghöfer, and Regina Hebig. 2023. Blended modeling in commercial and open-source model-driven software engineering tools: A systematic study. *Software and Systems Modeling* 22, 1 (2023), 415–447.

[6] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E Lopez-Herrejon, and Alexander Egyed. 2016. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software* 111 (2016), 281–297.

[7] Git developers. 2023. Git merge. Retrieved June 2023 from https://git-scm.com/docs/git-merge Accessed June, 2023.

[8] Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. 2023. A modeling assistant to manage technical debt in coupled evolution. *Information and Software Technology* 156 (2023), 107146.

[9] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2011. What is needed for managing co-evolution in mde?. In *Proceedings of*

the 2nd International Workshop on Model Comparison in Practice. ACM, 30–38.

[10] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. 2011. 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, 143–162.

[11] Bastien Drouot and Joël Champeau. 2019. Model Federation based on Role Modeling. In *MODELSWARD*. 72–83.

[12] EAST-ADL Association. 2022. EATOP Repository. https://bitbucket.org/east-adl/east-adl/src/Revison/ Accessed February, 2023.

[13] Eclipse Foundation. 2012. Xcore Metamodel. Retrieved May 2022 from https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore Accessed February, 2023.

[14] Eclipse Foundation. 2018. Eclipse Xcore Wiki. Retrieved May 2022 from 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 Accessed February, 2023.

[15] Eclipse Foundation. 2023. Xtext. Retrieved June 2023 from https://www.eclipse.org/Xtext/index.html Accessed June, 2023.

[16] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. 2009. Generating instance models from meta models. *Software & Systems Modeling* 8 (2009), 479–500.

[17] Martin Eisenberg, Hans-Peter Pichler, Antonio Garmendia, and Manuel Wimmer. 2021. Towards reinforcement learning for in-place model transformations. In *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 82–88.

[18] Luka Fürst, Marjan Mernik, and Viljan Mahnič. 2015. Converting metamodels to graph grammars: doing without advanced graph grammar features. *Software & Systems Modeling* 14 (2015), 1297–1317.

[19] Sinem Getir, Lars Grunske, Christian Karl Bernasko, Verena Käfer, Tim Sanwald, and Matthias Tichy. 2015. CoWolf–A generic framework for multi-view co-evolution and evaluation of models. In *Theory and Practice of Model Transformations: 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings 8*. Springer, 34–40.

[20] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity*. 206–211.

[21] Joel Greenyer. 2018. Scenario Modeling Language (SML) Repository. Retrieved May 2022 from https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/ Accessed February, 2023.

[22] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. 2013. Conceptual interoperability through models federation. In *Semantic Information Federation Community Workshop*. 23.

[23] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2016. Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering* 43, 5 (2016), 396–414.

[24] Jörg Holtmann, Jan-Philipp Steghöfer, and Henrik Lönn. 2022. Migrating from proprietary tools to open-source software for EAST-ADL metamodel generation and evolution. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 7–11.

[25] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R Bryant. 2008. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology* 50, 9-10 (2008), 948–968.

[26] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2011. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 163–172.

[27] Wael Kessentini and Vahid Alizadeh. 2020. Interactive metamodel/-model co-evolution using unsupervised learning and multi-objective search. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. IEEE, 68–78.

[28] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2019. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology* 106 (2019), 49–67.

[29] Djamel Eddine Khelladi, Reda Bendraou, Regina Hebig, and Marie-Pierre Gervais. 2017. A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution. *Journal of Systems and Software* 134 (2017), 242–260.

[30] Djamel Eddine Khelladi, Horacio Hoyos Rodriguez, Roland Kretschmer, and Alexander Egyed. 2017. An exploratory experiment on metamodel-transformation co-evolution. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 576–581.

[31] Angelika Kusel, Jürgen Etzlstorfer, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger, and Johannes Schönböck. 2015. Consistent co-evolution of models and transformations. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 116–125.

[32] Ralf Lämmel. 2001. Grammar adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity: International Symposium of Formal Methods Europe Berlin, Germany, March 12–16, 2001 Proceedings*. Springer, 550–570.

[33] Ralf Lämmel. 2007. Style normalization for canonical X-to-O mappings. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 31–40.

[34] Ralf Lämmel. 2018. *Software Languages*. Springer.

[35] Ralf Lämmel and Chris Verhoef. 2001. Cracking the 500-language problem. *IEEE software* 18, 6 (2001), 78–88.

[36] Ralf Lämmel and Vadim Zaytsev. 2009. An introduction to grammar convergence. In *Integrated Formal Methods: 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings 7*. Springer, 246–260.

[37] Tiziano Lombardi, Vittorio Cortellessa, Alfonso Pierantonio, and In Model. 2021. Co-evolution of Metamodel and Generators: Higher-order Templating to the Rescue. *J. Object Technol.* 20, 3 (2021), 7–1.

[38] Jesús J López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. 2015. Example-driven meta-model development. *Software & Systems Modeling* 14 (2015), 1323–1347.

[39] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. 2012. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST* 42 (2012), 13 pages.

[40] miklossy, nyssen, prggz, and mwienand. 2020. Dot Xtext grammar. Retrieved May 2020 from https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext Accessed February, 2023.

[41] OSF. [n. d.]. *Replication Package*. https://osf.io/6b2mf/?view_only=5cda50c7a6a9497eb65136fc443266cc

[42] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. 2014. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming* 96 (2014), 396–416. https://doi.org/10.1016/j.scico.2014.05.007 Selected Papers from the Fifth Intl. Conf. on Software Language Engineering (SLE 2012).

[43] Dennis Priefer, Wolf Rost, Daniel Strüber, Gabriele Taentzer, and Peter Kneisel. 2021. Applying MDD in the content management system domain: Scenarios, tooling, and a mixed-method empirical assessment. *Software and Systems Modeling* 20 (2021), 1919–1943.

[44] Louis M Rose, Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. 2010. Model migration with Epsilon Flock. In *Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings 3*. Springer, 184–198.

[45] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona A Polack. 2009. An analysis of approaches to model migration. In *Proc. Joint MoDSE-MCCM Workshop*. 6–15.

[46] Thomas Stahl and Markus Völter. 2006. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc.

[47] Daniel Strüber. 2017. Generating efficient mutation operators for search-based model-driven engineering. In *Theory and Practice of Model Transformation: 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings 10*. Springer, 121–137.

[48] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. 2017. Henshin: A usability-focused framework for EMF model transformation development. In *ICGT*. Springer, 196–208.

[49] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: formal foundation and application. *Formal Aspects of Computing* 30 (2018), 133–162.

[50] Daniel Strüber and Stefan Schulz. 2016. A tool environment for managing families of model transformation rules. In *Graph Transformation: 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings 9*. Springer, 89–101.

[51] Dániel Varró. 2006. Model transformation by example. In *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings 9*. Springer, 410–424.

[52] Steffen Vaupel, Daniel Strüber, Felix Rieger, and Gabriele Taentzer. 2015. Agile bottom-up development of domain-specific IDEs for model-driven development. In *FlexMDE'15: Workshop on Flexible Model Driven Engineering*. CEUR-WS.org, 12–21.

[53] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. 2007. Towards model transformation generation by-example. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 285b–285b.

[54] Xenia Authors. 2019. Xenia Metmodel. Retrieved May 2022 from https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore Accessed February, 2023.

[55] Xenia Authors. 2019. Xenia Xtext. Retrieved May 2022 from https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext Accessed February, 2023.

[56] Vadim Zaytsev. 2012. Language Evolution, Metasyntactically. *Electronic Communications of the EASST* 49 (2012), 17 pages.

[57] Weixing Zhang, Regina Hebig, Jan-Philipp Steghöfer, and Jörg Holtmann. 2023. Creating Python-Style Domain Specific Languages: A Semi-Automated Approach and Intermediate Results.. In *MODELSWARD*. 210–217.

[58] Weixing Zhang, Jörg Holtmann, Regina Hebig, and Jan-Philipp Steghöfer. 2023. Meta-model-based Language Evolution and Rapid Prototyping with Automate Grammar Optimization. Preprint available at SSRN: https://ssrn.com/abstract=4379232.

Received 2023-07-07; accepted 2023-09-01