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

Weixing Zhang[a], Jörg Holtmann[b], Daniel Strüber[a,c], Regina Hebig[d], Jan-Philipp Steghöfer[e]

[a] *Department of Computer Science & Engineering, Chalmers | University of Gothenburg, Gothenburg, Sweden*
[b] *Independent Researcher (main work conducted at [a]), Paderborn, Germany*
[c] *Department of Software Science, Radboud University, Nijmegen, Netherlands*
[d] *Institute of Computer Science, University of Rostock, Rostock, Germany*
[e] *XITASO GmbH IT & Software Solutions, Augsburg, Germany*

Abstract

In model-driven engineering, developing a textual domain-specific language (DSL) involves constructing a meta-model, which defines an underlying abstract syntax, and a grammar, which defines the concrete syntax for the DSL. We consider a scenario in which the meta-model is manually maintained, which is common in various contexts, such as *blended modeling*, in which several concrete syntaxes co-exist in parallel. Language workbenches such as Xtext support such a scenario, but require the grammar to be manually co-evolved, which is laborious and error-prone.

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

Keywords: Domain-specific Languages, DSL, Grammar, Xtext, Language Evolution, Language Prototyping

1. Introduction

Domain-Specific Languages (DSLs) are a common way to describe certain application domains and to specify the relevant concepts and their relationships (Van Deursen et al., 2000). They are, among many other things, used to describe model transformations (the Operational transformation language of the MOF Query, View, and Transformation — QVTo (Object Management Group, 2016) and the ATLAS Transformation Language — ATL (Eclipse Foundation, 2018)), bibliographies (BibTeX (Paperpile, 2022)), graph models (DOT (Graphviz Authors, 2022)), formal requirements (the Scenario Modeling Language — SML (Greenyer, 2018) and Spectra (Spectra Authors, 2021)), meta-models (Xcore (Eclipse Foundation, 2018)), or web-sites (Xenia (Xenia Authors, 2019)).

In many cases, the syntax of the language that engineers

Email addresses: `weixing.zhang@gu.se` (Weixing Zhang), `joerg_holtmann@gmx.de` (Jörg Holtmann), `danstru@chalmers.se` (Daniel Strüber), `regina.hebig@uni-rostock.de` (Regina Hebig), `jan-philipp.steghoefer@xitaso.com` (Jan-Philipp Steghöfer)

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

A different way to go about constructing DSLs is proposed by model-driven engineering. There, the concepts that are relevant in the domain are captured in a meta-model which defines the *abstract syntax* (see, e.g., (Roy Chaudhuri et al., 2019; Frank, 2013; Mernik et al., 2005)). Different *concrete syntaxes*, e.g., graphical, textual, or form-based, can be defined to describe actual models that adhere to the abstract syntax.

In this paper, we consider the Eclipse ecosystem and Xtext (Bettini, 2016) as its de-facto standard framework for developing textual DSLs. Xtext relies on the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) and uses its Ecore (meta-)modeling facilities as basis. Developing a textual DSL in Xtext involves two main artifacts: a grammar, which defines the concrete syntax of the language, and a meta-model, which defines the abstract syntax. Xtext allows either the grammar or the meta-model to be created first, and then automatically generating the one from the other (or alternatively, writing both manually and aligning them).

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

In this paper, we focus on meta-model-based evolution, for the following rationale: While grammar-based evolution is a common way of developing languages in Xtext, it is not geared for three scenarios that we encountered in the real world, including collaborations with an industrial partner. In particular: 1. Several concrete syntaxes (e.g., visual, textual, tabular) for the same underlying metamodel co-exist and evolve at the same time. This is particular common in the context of blended modeling (Ciccozzi et al., 2019), a timely modeling paradigm. 2. The metamodel comes from some external source (such as a third-party supplier or a standardization committee), which prohibits independent modification. 3. The metamodel is the central artifact of a larger ecosystem of available tools, including. e.g., automated analyses and transformations. As such, the language engineers might prefer to evolve it directly, instead of relying on the, potentially sub-optimal, output of automatically co-evolving it after grammar changes. The real-world case that inspired this paper has aspects of the first two scenarios: we work on a language from an industry partner for which there already exists an evolving metamodel and graphical editor available.

Compared to grammar-based evolution, meta-model-based evolution has one major disadvantage: Co-evolving the grammar after meta-model changes is more complicated than vice versa, as it involves dealing with both abstract and concrete syntax aspects, whereas updating the meta-model after grammar changes only involves abstract syntax aspects. In the state of the art, the updating needs to be done manually, which leads to effort after each evolution step. According to the Xtext textbook (Bettini, 2016), *"the drawback [of manually maintaining the Ecore model] is that you need to keep the Ecore model consistent with your DSL grammar."* The goal of this paper is to substantially mitigate this disadvantage, as we will now explain.

In this paper, we propose a different approach to supporting meta-model-based evolution: Automated synchronization of the grammar based on simple rules, which we call *grammar transformation rules*. Such rules encode typical improvements that are made to a grammar, e.g., changing parentheses layouts, keywords, and orders of rule fragments. Configurations can either be automatically extracted from previous manual edits of the grammar (Zhang et al., 2023), or explicitly created by the language engineer, as an alternative to manually performing redundant changes affecting many places in the grammar. Whenever the meta-model evolves, the same or a slightly modified set of transformation rules can be applied to a fresh grammar that Xtext can automatically generate from the meta-model. The resulting grammar is inherently synchronized with the meta-model, but restores the syntax decisions made in the previous grammar versions, thus avoiding effort for manual synchronization.

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

While our approach clearly unfolds these benefits in the case of evolving languages and complex changes, it does not come for free. For locally-scoped changes, creating a configuration generally leads to more effort than a manual grammar edit and hence, presents an upfront investment that pays off only when the language evolves over time. In a different paper (Zhang et al., 2023), we present an approach for automating the extraction of configurations from user-provided manual edits, thus reducing the initial manual effort to be the same as in the traditional process, while keeping the long-term benefits. Together with the present paper, for the supported kinds of changes, it supports a fully automated process for aligning the grammar after changes to the meta-model.

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

We demonstrate its applicability on seven domain-specific languages from different application areas. We also show its support for language evolution in two cases: 1), we recreate the textual model transformation language QVTo in all four versions of the official standard (Object Management Group, 2016) with only small changes to the configuration of transformation rule applications and with high consistency of the syntax between versions; and 2), we conceived for the automotive systems modeling language EAST-ADL (EAST-ADL Association, 2021) together with an industrial partner a textual concrete syntax (Holtmann et al., 2023), where we initially started with a grammar for a subset of the EAST-ADL meta-model (i.e., textual language version 1) and subsequently evolved the grammar to encompass the full meta-model (i.e., textual language version 2).

The remainder of this paper is structured as follows. First, in Section 2, we provide an overview of the background of this paper, in particular, on metamodel-based textual DSL engineering. In Section 3, we review related research. In Section 4, we define the methodology of this paper. Subsequently, in Section 5, we describe the identified transformation rules, which are the main technical contribution of this paper. Following that, in Section 6, we present our solution of the GrammarTransformer, which implements the identified transformation rules. In Section 7, we present our evaluation. Section 8 is devoted to our discussion, where we address threats to validity, the effort required to use GrammarTransformer, implications for practitioners and researchers, and future work. Finally, in the last section, we conclude.

2. Background: Textual DSL Engineering based on Meta-models

The engineering of textual DSLs can be conducted through the traditional approach of specifying grammars, but also by means of meta-models. Both approaches have commonalities, but also differences (Paige et al., 2014). Like grammars specified by means of the Extended Backus Naur Form (EBNF) (International Organization for Standardization (ISO), 1996), meta-models enable formally specifying how the terms and structures of DSLs are composed. In contrast to grammar specifications, however, meta-models describe DSLs as graph structures and are often used as the basis for graphical or non-textual DSLs. Particularly, the focus in meta-model engineering is on specifying the abstract syntax. The definition of concrete syntaxes is often considered a subsequent DSL engineering step. However, the focus in grammar engineering is directly on the concrete syntax (Kleppe, 2007) and leaves the definition of the abstract syntax to the compiler.

Meta-model-based textual DSLs. There are also examples of textual DSLs that are built with meta-model technology. For example, the Object Management Group (OMG) defines textual DSLs that hook into their meta-model-based Meta Object Facility (MOF) and Unified Modeling Language ecosystems, for example, the Object Constraint Language (OCL) (Object Management Group (OMG), 2014) and the Operational transformation language of the MOF Query, View, and Transformation (QVTo) (Object Management Group, 2016). However, this is done in a cumbersome way: Both the specifications for OCL and QVTo define a meta-model specifying the abstract syntax and a grammar in EBNF specifying the concrete syntax of the DSL. This grammar, in turn, defines a different set of concepts and, therefore, a meta-model for the concrete syntax that is different from the meta-model for the abstract syntax. As Willink (Willink, 2020) points out, this leads to the awkward fact that the corresponding tool implementations such as Eclipse OCL (Eclipse Foundation, 2022a) and Eclipse QVTo (Eclipse Foundation, 2022b) also apply this distinction. That is, both tool implementations require an abstract syntax and a concrete syntax meta-model and, due to their structural divergences, a dedicated transformation between them. Additionally, both tool implementations provide a hand-crafted concrete syntax parser, which implements the actual EBNF grammar. Maintaining these different parts and updating the manually created ones incurs significant effort whenever the language should be evolved.

Xtext. Xtext provides a more streamlined approach to language engineering that envisions the use of a single metamodel for defining the abstract syntax, and an associated grammar for defining the textual concrete syntax. Grammars are defined in a custom, EBNF-based format. Using an Xtext grammar, Xtext applies the ANTLR parser generator framework (Parr, 2022) to derive a parser and all its required inputs. It also generates editors along with syntax highlighting, code validation, and other useful tools.

Figure 1: Instance of the generated grammar for EAST-ADL.

Xtext supports both grammar-based and meta-model-based-evolution in the sense introduced in Section 1. For our considered meta-model-based scenario, Xtext's default workflow requires that after each meta-model change, the grammar has to be manually synchronized (Bettini, 2016), a disadvantage we aim to avoid with our approach. To this end, we rely on a built-in feature of Xtext for automatically deriving a grammar from a meta-model. (we call this grammar *generated grammar* in this paper). This creates a grammar that contains grammar rules for all meta-model elements that are contained in a common root node and resolves references, etc., to a degree (see Section 4.3 for details). This grammar is typically quite verbose, structured extensively using braces, and uses a lot of keywords, as illustrated with the example in Figure 1, depicting an instance of the generated grammar for EAST-ADL. Therefore, generated grammars are intended to be improved before being used in practice (Bettini, 2016). In our approach, we use generated grammars as the starting point for recording and automatically replaying changes made to the grammar, thus avoiding manual synchronization effort.

3. Related Work

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

Grammar Transformation. There are a few works that aim at transforming grammar rules with a focus on XML-based languages. For example, Neubauer et al. (2015, 2017) also mention transformation of grammar rules in Xtext. Their approach XMLText and the scope of their transformation focus only on XML-based languages. They convert an XML schema definition to a meta-model using the built-in capabilities of EMF. Based on that meta-model, they then use an adapted Xtext grammar generator for XML-based languages to provide more human-friendly notations for editing XML files. XMLText thereby acts as a sort of compiler add-on to enable editing in a different notation and to automatically translate to XML and vice versa. In contrast, we develop a post-processing approach that enables the transformation of any Xtext grammar, not only XML-based ones, cf. also our discussion in Section 8).

The approach of Chodarev (2016) shares the same goal and a similar functional principle as XMLText, but uses other technological frameworks. In contrast to XMLText, Chodarev supports more straightforward customization of the target XML language by directly annotating the meta-model that is generated from the XML schema. The same distinction applies here as well: GrammarTransformer enables the transformation of any Xtext grammar and is not restricted to XML-based languages.

Grammar transformation for DSLs in general is addressed by Jouault et al. (2006). They propose an approach to specify a syntax for textual, meta-model-based DSLs with a dedicated DSL called Textual Concrete Syntax, which is based on a meta-model. From such a syntax specification, a concrete grammar and a parser are generated. The approach is similar to a template language restricting the language engineer and thereby, as the authors state, lacks the freedom of grammar specifications in terms of syntax customization options. In contrast, we argue that the GrammarTransformer provides more syntax customization options to achieve a well-accepted textual DSL.

Finally, Novotný (2012) designed a model-driven Xtext pretty printer, which is used for improving the readability of the DSL by means of improved, language-specific, and configurable code formatting and syntax highlighting. In contrast, our GrammarTransformer is not about improving code readability but focused on how to design the DSL itself to be easy to use and user-friendly.

Designing and Evolving Meta-model-based DSLs. Many papers about the design of DSLs focus solely on the construction of the abstract syntax and ignore the concrete syntaxes (e.g., (Roy Chaudhuri et al., 2019; Frank, 2011)), or focus exclusively on graphical notations (e.g.,(Frank, 2013; Tolvanen and Kelly, 2018)). In contrast, the guidelines proposed by Karsai et al. (2009) contain specific ideas about concrete syntax design, e.g., to "balance compactness and comprehensibility". Arguably, the languages automatically generated by Xtext are neither compact nor comprehensible and therefore require manual changes.

Mernik et al. (2005) acknowledge that DSL design is not a sequential process. The paper also mentions the importance of textual concrete syntaxes to support common editing operations as well as the reuse of existing languages. Likewise, van Amstel et al. (2010) describe DSL development as an iterative process and use EMF and Xtext for the textual syntax of the DSL. They also discuss the evolution of the language, and that "it is hard to predict which language features will improve understandability and modifiability without actually using the language". Again, this is an argument for the need to do prototyping when developing a language. Karaila (2009) broadens the scope and also argues for the need for evolving DSLs along with the "engineering environment" they are situated in, including editors and code generators. Pizka and Jürgens (2007) also acknowledge the "constant need for evolution" of DSLs.

There is a lot of research supporting different aspects of language change and evolution. Existing approaches focus on how diverse artifacts can be co-evolved with evolving meta-models, namely the models that are instances of the meta-models (Hebig et al., 2016), OCL constraints that are used to specify static semantics of the language (Khelladi et al., 2017, 2016), graphical editors of the language (Ruscio et al., 2010; Di Ruscio et al., 2011), and model transformations that consume or produce programs of the language (García et al., 2012). Specifically, the evolution of language instances with evolving meta-models is well supported by research approaches. For example, Di Ruscio et al. (Di Ruscio et al., 2011) support language evolution by using model transformations to simultaneously migrate the meta-model as well as model instances.

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

Other Approaches. As we mentioned above, besides Xtext, there are two more approaches that support the generation of EBNF-based grammars and from these the generation of the actual parsers. These are EMFText (Heidenreich et al., 2009) and the Grasland toolkit (Kleppe, 2007), which are both not maintained anymore.

Whereas our work focuses on the Eclipse technology stack based on EMF and Xtext, there are a number of other language workbenches and supporting tools that support the design of DS(M)Ls and their evolution. However, none of these approaches are able to derive grammars directly from meta-models, a prerequisite for the approach to language engineering we propose and the basis of our contribution, GrammarTransformer. Instead, tools like textX (Dejanović et al., 2017) go the other way around and derive the meta-model from a grammar. Langium (TypeFox GmbH, 2022) is the self-proclaimed Xtext successor without the strong binding to Eclipse, but does not support this particular use case just yet and instead focuses on language construction based on grammars. MetaEdit+ (Kelly and Tolvanen, 2018) does not offer a textual syntax for the languages, but instead a generator to create text out of diagrams that are modeled using either tables, matrices, or diagrams. JetBrains MPS (JetBrains, 2022) is based on projectional editing where concrete syntaxes are projections of the abstract syntax. However, these projections are manually defined and not automatically derived from the meta-model as it is the case with Xtext. Finally, Pizka and Jürgens (2007) propose an approach to evolve DSLs including their concrete syntaxes and instances. For that, they present "evolution languages" that evolve the concrete syntax separately. However, they focus on DSLs that are built with classical compilers and not with meta-models.

4. Methodology

In this section, we describe our research methodology, shown in an overview in Figure 2. Our methodology consists of a number of sequential steps, in particular: selecting the case languages, preparing metamodels and grammars (including the exclusion of certain parts of the language), and two iterations of analysis, including extraction of grammar transformation rules and tool development. We now describe all of these steps in detail.

4.1. Selection of Sample DSLs

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

Sources. To find language candidates, we collected well-known languages, such as DOT, and used language collections, such as the Atlantic Zoo (AtlanMod Team, 2019), a list of robotics DSLs (Nordmann et al., 2020), and similar collections (Wikimedia Foundation, 2023; Barash, 2020; Semantic Designs, 2021; Community, 2021; Van Deursen et al., 2000). However, it turned out that the search for suitable examples was not trivial despite these resources. The quality of the meta-models in these collections was often insufficient for our purposes. In many cases, the meta-model structures were too different from the grammars or there was no grammar in either Xtext or in EBNF publicly available as well as no clear syntax definition by other means. We therefore extended our search to also use Github's search feature to find projects in which meta-models and Xtext grammars were present and manually searched the Eclipse Foundation's Git repositories for suitable candidates. Grammars were either taken from the language specifications or from the repositories directly.

Concrete Grammar Reconstruction for BibTeX. In some cases, the syntax of a language is described in detail online,

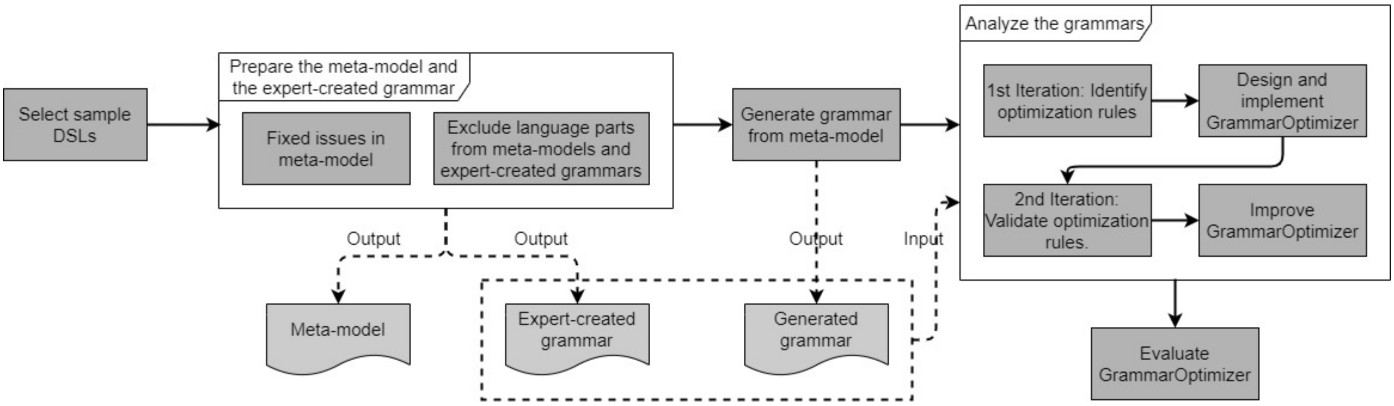

Figure 2: Overview of our methodology.

but no EBNF or Xtext grammar can be found. In our case, this is the language BibTeX. It is a well-known language to describe bibliographic data mostly used in the context of typesetting with LaTeX that is notable for its distinct syntax. In this case, we utilized the available detailed descriptions (Paperpile, 2022) to reconstruct the grammar. To validate the grammar we created, we used a number of examples of bibliographies from (Paperpile, 2022) and from our own collection to check that we covered all relevant cases.

Meta-model Reconstruction for DOT. DOT is a well-known language for the specification of graph models that are input to the graph visualization and layouting tool Graphviz. Since it is an often used language with a relatively simple, but powerful syntax, we decided to include it, even if we could not find a complete meta-model that contains both the graph structures and formatting primitives. The repository that also contains the grammar we ended up using (itemis AG, 2020), e.g., only contains meta-models for font and graph model styles.

Therefore, we used the Xtext grammar that parses the same language as DOT's expert-created grammar to derive a meta-model (itemis AG, 2020). Xtext grammars include more information than an EBNF grammar, such as information about references between concepts of the language. Thus, the fact that the DOT grammar was already formulated in Xtext allowed us to directly generate DOT's Ecore meta-model from this Xtext grammar. This meta-model acquisition method is an exception in this paper. Since this paper focuses on how to transform the generated grammar, we consider this way of obtaining the meta-model acceptable for this one case.

Selected Cases. As a result, we identified a sample of seven DSLs (cf. Table 1), which has a mix of different sources for meta-models and grammars. This convenience sampling consists of a mix of well-known DSLs with lesser-known, but well-developed ones. We believe this breadth of domains and language styles is broad enough to extract a generically applicable set of candidate transformation rules for GrammarTransformer. We analyzed these selected languages in two iterations, the 1st analyzing four of them and the 2nd analyzing the remaining three. In Table 1, we list all seven languages, including information about the meta-model (source and the number of classes in the meta-model) and the expert-created grammar (source and the number of grammar rules).

4.2. Exclusion of Language Parts for Low-level Expressions

Two of the analyzed languages encompass language parts for expressions, which describe low-level concepts like binary expressions (e.g., addition). We excluded such language parts in ATL and in SML due to several aspects. Both languages distinguish the actual language part and the expression language part already on the meta-model

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. The first set of DSLs was analyzed to derive necessary transformation rules, and the second set to validate the candidate transformation rules and extend them if necessary.

		Meta-model		Expert-created Grammar		Generated Grammar		
Iteration	DSL	Source	Classes[1]	Source	Rules	lines	rules	calls
1st	ATL[2]	Atlantic Zoo (AtlanMod Team, 2019)	30	ATL Syntax (Eclipse Foundation, 2018)	28	275	30	232
	BibTex	Grammarware (Zaytsev, 2013)	48	Self-built Based on (Paperpile, 2022)	46	293	48	188
	DOT	Generated	19	Dot (Graphviz Authors, 2022)	32	125	23	51
	SML[3]	SML repository (Greenyer, 2018)	48	SML repository (Greenyer, 2018)	45	658	96	377
2nd	Spectra	GitHub Repository (Spectra Authors, 2021)	54	GitHub Repository (Spectra Authors, 2021)	58	442	62	243
	Xcore	Eclipse (Eclipse Foundation, 2012)	22	Eclipse (Eclipse Foundation, 2018)	26	243	33	149
	Xenia	Github Repository (Xenia Authors, 2019)	13	Github Repository (Xenia Authors, 2019)	13	84	15	36

[1] After adaptations, containing both classes and enumerations.
[2] Excluding embedded OCL rules.
[3] Excluding embedded SML expressions rules.

level and thereby treat the expression language part differently. The respective expression parts are similarly large than the actual languages (i.e., 56 classes for the embedded OCL part of ATL and 36 classes for the SML scenario expressions meta-model), which implies a high analysis effort. Finally, although having a significantly large meta-model, the embedded OCL part of ATL does not specify the expressions to a sufficient level of detail (e.g., it does not allow to specify binary expressions). Therefore, we excluded such language parts by introducing a fake class `OCLDummy`. The details for the exclusion is described in the supplemental material (Zhang et al., 2024)[1].

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

[1]See folder "Section_4_Methodology"

4.3. Meta-model Preparations and Generating an Xtext Grammar

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

Note that it is sometimes necessary to slightly change the meta-model to enable the generation of the Xtext grammar or to ensure that the compatibility with the expert-created grammar can be reached. These changes are necessary in case the meta-model is already ill-formed for EMF itself (e.g., purely descriptive Ecore files that are not intended for instantiating runtime models) or if it does not adhere to certain assumptions that Xtext makes (e.g., no bidirectional references). The method of metamodel modification is described in detail in our supplementary material (Zhang et al., 2024)[2].

In Table 1, we list how many lines, rules, and calls between rules the generated grammars included for the seven languages.

4.4. Comparing EBNF and Xtext grammars

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

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

Like semantic equivalence of context-free grammars, in general, (Hopcroft, 1969), imitation is undecidable if two arbitrary grammars are considered. However, in the scope of our analysis, we deal with specific cases that come from our evaluation subjects. These are generally of the following form: 1. Two syntactically identical—and thus, inherently semantically equivalent—grammar rules 2. Situations in which a larger rule from the first grammar is, in a controlled way, split up into several rules in the second grammar. For these, we consider them as equivalent based on a careful manual analysis, explained later.

[2]See directory "Section_4_Methodology'.

Listing 1: EBNF rule `edge_stmt` from the expert-created grammar for DOT

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

Listing 2: Xtext rules `EdgeStmtNode` and `EdgeStmtSubgraph` from the transformed generated grammar

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

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

For example, the rule `edge_stmt` shown in Listing 1 is imitated by the combination of the rules `EdgeStmtNode` and `EdgeStmtSubgraph` shown in Listing 2. Merging the Xtext rules to form one rule, like the EBNF counterpart, was not possible in this case, due to the necessity of specifying a distinct return type in Xtext, which is not required in EBNF. In addition, the Xtext rules contain Xtext-specific information for dealing with references and attribute types, which is not present in the EBNF rule.

4.5. Analysis of Grammars

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

Our general approach was similar in both iterations.

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

For example, the expert-created grammar rule `node_stmt` in DOT (see Listing 3) maps to the generated grammar rule `NodeStmt` in Listing 4. Multiple changes are necessary to adjust the generated Xtext grammar rule:

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

Note that in most cases the expert-created grammar was written in EBNF instead of Xtext. For example, the `returns` statement in line 1 of Listing 4 is required for parsing in Xtext. We took that into account when comparing both grammars.

4.5.1. First Iteration: Identify Transformation Rules

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

1. identify the differences between the expert-created grammar and generated grammar of the language;

Listing 3: Non-terminal `node_stmt` in the expert-created grammar of DOT, in EBNF

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

Listing 4: Grammar rule `NodeStmt` in the generated grammar of DOT, in Xtext

```
1  NodeStmt returns NodeStmt:
2          {NodeStmt}
3          'NodeStmt'
4          '{'
5                  ('node' node=NodeId)?
6                  ('attrLists' '{' attrLists+=
                       AttrList ( "," attrLists+=
                       AttrList)* '}' )?
7          '}';
```

2. derive grammar transformation rules that can be applied to change the generated grammar so that the transformed grammar parses the same language as the expert-created grammar.

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

As a result of the analysis, we obtained an initial set of grammar transformation rules, which contained a total of 58 candidate transformation rules. Table 2 summarizes in the second column the number of identified rule candidates and in the second row the number for the first iteration. Since the initial set of grammar transformation rules was a result of an analysis done by multiple authors, it included rules that were partially overlapping and rules that turned out to only affect the grammar's formatting, but not the language specified by the grammar. Thus, we filtered rules that belong to the latter case. For rule candidates that overlapped with each other, we selected a subset of the

Table 2: Summary of identified rules their rule variants and their sources

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

rules as a basis for the next step. This filtering led to a selection of 46 transformation rules (cf. third column in Table 2).

We processed these 46 selected transformation rules to identify required *rule variants* that could be implemented directly by means of one Java class each, which we describe more technically as part of our design and implementation elaboration in Section 6.3. For identifying the rule variants, we focused on the following aspects:

Specification of scope Small changes in the meta-model might lead to a different order of the lines in the generated grammar rules or even a different order of the grammar rules. Therefore, the first step was to define a suitable concept to identify the parts of the generated grammar that can function as the *scope* of an transformation rule, i.e., where it applies. We identified different suitable scopes, e.g., single lines only, specific attributes, specific grammar rules, or even the whole grammar. Initially, we identified separate rule variants for each scope. Note that this also increased the number of rule variants, as for some rule candidates multiple scopes are possible.

Allowing multiple scopes In many cases, selecting only one specific scope for a rule is too limiting. In the example above (Listing 4), pairs of braces in different scopes are removed: in the scope of the attribute `attrLists` in line 6 and in the scope of the containing grammar rule in lines 4 and 7. This illustrates that changes might be applied at multiple places in the grammar at once. When formulating rule variants, we analyzed the rule candidates for their potential to be applied in different scopes. When suitable, we made the scope configurable. This means that only one transformation rule variant is necessary for both cases in the example. Depending on the provided parameters, it will either replace the braces for the rule or for specific attributes.

Composite transformation rules We decided to avoid transformation rule variants that can be replaced or composed out of other rule variants, especially when such compositions were only motivated by very few cases. However, such rules might be added again later if it turns out they are needed more often.

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

4.5.2. Second iteration: Validate Transformation Rules

The last step left us with 46 selected transformation rules from the first iteration (cf. second row in Table 2). We developed a preliminary implementation of GrammarTransformer by implementing the 57 rules variants belonging to these 46 transformation rules (we will describe the implementation in the *Solution* section). To validate this set of transformation rules, we performed a second iteration. In the second iteration, we selected the three DSLs Spectra, Xenia, and Xcore. As in the first

Listing 5: Two attributes in the grammar rule `XOperation` in the generated grammar of Xcore

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

Listing 6: Two attributes in the grammar rule `XOperation` in the expert-created grammar of Xcore

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

iteration, we generated a grammar from the meta-model, analyzed the differences between the generated grammar and the expert-created grammar, and identified transformation rules that need to be applied to the generated grammar to accommodate these differences. In contrast to the first iteration, we aimed at utilizing as many existing transformation rules as possible and only added new rule candidates when necessary.

We configured the preliminary GrammarTransformer for the new languages by specifying which transformation rules to apply on the generated grammar. The execution results showed that the existing transformation rules were sufficient to change the generated grammar of Xenia to imitate the expert-created grammar used as the ground truth. However, we could not fully transform the generated grammar of Xcore and Spectra with the preliminary set of 46 transformation rules from the first iteration. For example, Listing 5 shows two attributes `unordered` and `unique` in the grammar rule `XOperation` in the generated grammar for Xcore. However, in the expert-created grammar, the rule portions for the two attributes each refer to the other attribute in a way that allows using the keywords in several possible orders, as shown in Listing 6. This transformation could not be performed with the transformation rules from the first iteration.

Based on the non-transformed parts of the grammars of Xcore and Spectra, we identified another ten transformation rules for the GrammarTransformer. These ten newly identified transformation rules transform all the non-transformed parts of the grammar of Xcore, including, e.g., transforming the grammar in Listing 5 to Listing 6. These new transformation rules also transform part of the non-transformed parts of the grammar of Spectra. We will interpret the remaining non-transformed parts in the *Evaluation* section. In the end, after two iterations, we identified a total of 56 transformation rules (which will be implemented by a total of 67 rule variants) (cf. fourth row in Table 2).

5. Identified Transformation Rules

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

Table 3 shows some examples of the transformation rules. The rules we implemented can be categorized by the primitives they manipulate: grammar rules, attributes keywords, braces, multiplicities, optionality (a special form of multiplicities), grammar rule calls, import statements, symbols, primitive types, and lines. They either 'add' things (e.g., *AddKeywordToRule*), 'remove' things (e.g., *RemoveOptionality*), or 'change' things (e.g., *ChangeCalledRule*). All transformation rules ensure that the resulting changed grammar is still valid and syntactically correct Xtext.

Most transformation rules are 'scoped' which means that they only apply to a specific grammar rule or attribute.

Listing 7: Grammar rule `NodeStmt` in the transformed grammar of DOT, in Xtext

```
1  NodeStmt returns NodeStmt:
2      {NodeStmt}
3
4
5          node=NodeId
6          (attrLists+=AttrList)*
7      ;
```

Table 3: Excerpt of implemented grammar transformation rules. A configurable scope ("Config.") means that, depending on provided parameters, the rule either applies globally to a specific grammar rule or to a specific attribute.

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

In other cases, the scope is configurable, depending on the parameters of the transformation rule. For instance, the *RenameKeyword* rule takes a grammar rule and an attribute as a parameter. If both are set, the scope is the given attribute in the given rule. If no attribute is set, the scope is the given grammar rule. If none of the parameters is set, the scope is the entire grammar ("Global"). All occurrences of the given keyword are then renamed inside the respective scope.

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

Likewise, GrammarTransformer contains rules to manipulate the multiplicities in the generated grammars. The meta-models and the expert-created grammars we used as inputs do not always agree about the multiplicity of elements. We provide transformation rules that can address this within the constraints allowed by EMF and Xtext.

For the example in Listing 4, this means that the necessary changes to reach the same language defined in Listing 3 can be implemented using the following GrammarTransformer rules:

- *RemoveBraces* is applied to the grammar rule `NodeStmt` and all of its attributes. This removes all the curly braces ('{' and '}' in lines 4, 6, and 7) within the grammar rule.

- *RemoveKeyword* is applied to the grammar rule `NodeStmt` and all of its attributes. This removes the keywords `'NodeStmt'`, `'node'` and `'attrLists'` (lines 3, 5, and 6) from this grammar rule.

- *RemoveOptionality* is applied to both attributes. This removes the question marks ('?') in lines 5 and 6.

- *convert1toStarToStar* is applied to the attribute `attrLists`. This rule changes line 6. Before this change, this line is "`attrLists+=AttrList ( "," attrLists+=AttrList)*`" (the braces, keyword '`attrLists`' and the optionality '?' have been removed by previous transformation rules). After this change, it becomes `(attrLists+=AttrList)*`. Note that the DOT grammar is specified using a syntax that is slightly different from standard EBNF. In

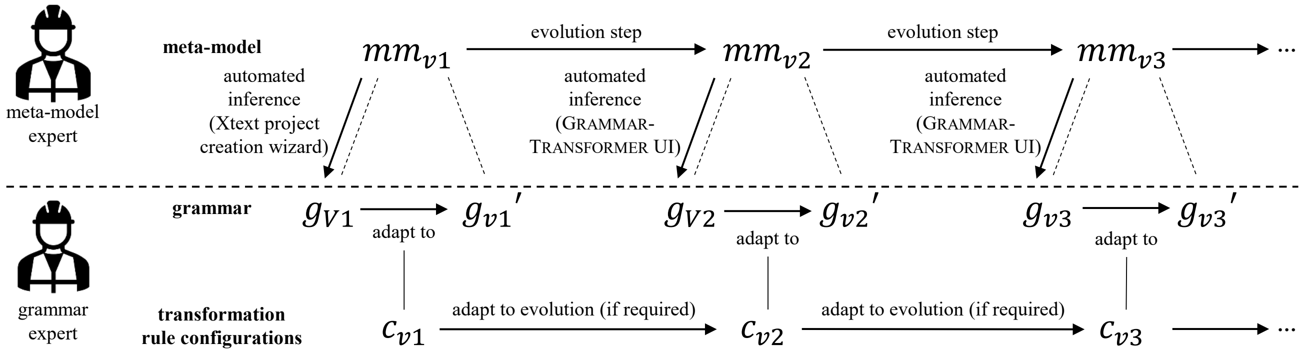

Figure 3: Co-evolution workflow with GrammarTransformer. Dashed lines indicate grammar/meta-model conformance.

that syntax, square brackets ([and]) enclose optional items (Graphviz Authors, 2022).

Note that line 2 in Listing 4 has no effect on the syntax of the grammar but is required by and specific to Xtext, so that we do not adapt such constructs. After the above steps, the grammar rule `NodeStmt` is adapted from Listing 4 to Listing 7.

6. Solution: Design and Implementation

The core of GrammarTransformer is a Java library that offers a simple API to configure transformation rule applications and execute them on Xtext grammars. Language engineers can use that API to create a small program that executes GrammarTransformer, which in turn will produce the transformed grammar. Alternatively, the programs can be generated automatically, using an automated tool (Zhang et al., 2023).

In this section, we first present our envisioned workflow, before describing in detail the specific components of our solution: its grammar representation, the design of transformation rules and configurations, and its execution engine. We wrap up with a comparison to an alternative approach and a discussion of limitations and caveats.

6.1. Language Evolution Workflow

Figure 3 depicts GrammarTransformer's language evolution workflow from a conceptual as well as user point of view. We distinguish between the roles of meta-model expert and grammar expert, which can be held by the same person. The former one takes care of the meta-model evolution, and the latter one takes care of the grammar adaptions and particularly the transformation rule configurations.

For the first meta-model version mm_{v1}, the initial grammar g_{v1} as well as the complete Xtext editor environment are automatically inferred via the Xtext project creation wizard. The initial grammar follows Xtext's default layouting and is not intended to be directly usable. Creating the first usable version g_{v1}' of the grammar is the responsibility of the grammar expert. In our approach, they do so in a way that leads to the creation of a transformation rule configuration c_{v1} that can automatically transform g_{v1} to g_{v1}'. They have two options for doing so: manually writing the configuration, or performing the intended changes manually and then using `ConfigGenerator` (Zhang et al., 2023) to extract the configuration.

Subsequently, the meta-model expert conducts a meta-model evolution step that results in mm_{v2}, leading to a need to co-evolve the grammar. To this end, first, the grammar expert obtains a synchronized version g_{v2} of the grammar, by having it inferred from the meta-model. GrammarTransformer offers a custom user interface to infer g_{v2} without the need to use the Xtext project creation wizard, which would result in a cumbersome workflow due to the generation of the complete editor environment. To replay the previously made concrete syntax changes, the grammar expert re-applies the transformation rule configuration

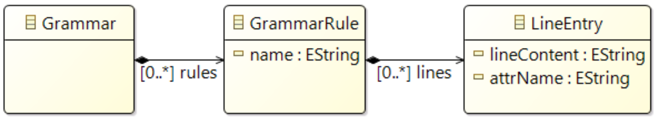

Figure 4: The class design for representing grammar rules.

c_{v1} to g_{v2}. The grammar engineer might then intend to perform further changes to the grammar, for example, to change the concrete syntax for new language elements. To this end, they proceed in the same way as before, either by manually writing a configuration or by automatically inferring one from manual changes.

All further meta-model and grammar co-evolution steps follow the same principle.

6.2. Grammar Representation

We designed GrammarTransformer to parse an Xtext grammar into an internal data structure which is then modified and written out again. This internal representation of the grammar follows the structure depicted in Figure 4. A `Grammar` contains a number of `GrammarRule`s that can be identified by their names. In turn, a `GrammarRule` consists of a sorted list of `LineEntry`s with their textual `lineContent` and an optional `attrName` that contains the name of the attribute defined in the line. Note that we utilize the fact that Xtext generates a new line for each attribute.

6.3. Transformation Rule Design

Internally, all transformation rules derive from the abstract class `TransformationRule` as shown in Figure 5. Derived classes overwrite the `apply()`-method to perform the specific text modifications for this rule. By doing so, the specific rule can access the necessary information through the class members: `grammar` (i.e., the entire grammar representation as explained in Section 6.2 and depicted in Figure 4), `grammarRuleName` (i.e., the name of the specified grammar rule that a user wants to transform exclusively), and `attrName` (i.e., the name of an attribute that a user wants to transform exclusively). Sub-classes can also add additional members if necessary. This architecture makes the GrammarTransformer extensible, as new transformation rules can easily be defined in the future.

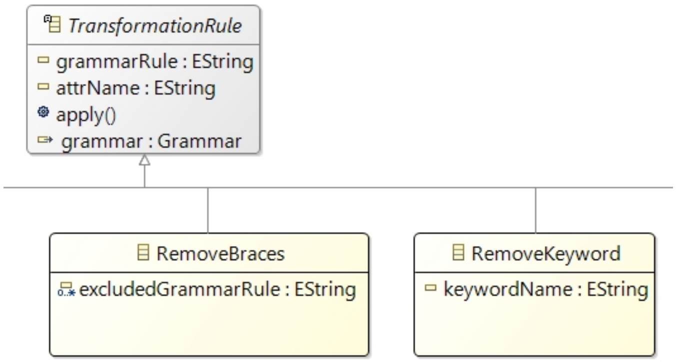

Figure 5: Excerpt of the class diagram for transformation rules.

We built the transformation rules in a model-based manner by first creating the meta-model shown in Figure 5 and then using EMF to automatically generate the class bodies of the transformation rules. This way we only needed to overwrite the `apply()`-method for the concrete rules. Internally, the `apply()`-methods of our transformation rules are implemented using regular expressions. Each transformation rule takes a number of parameters, e.g., the name of the grammar rule to work on or an attribute name to identify the line to work on. In addition, some transformation rules take a list of exceptions to the scope. For example, the transformation rule to remove braces can be applied to a global scope (i.e., all grammar rules) while excluding a list of specific grammar rules from the processing. This allows to configure transformation rule applications in a more efficient way. We implemented all identified transformation rules.[3] For testing, we built a comprehensive test suite, based on the transformed grammars considered in our design methodology. We created one test case per scenario, to ensure that the grammar produced by our implementation after applying a full given configuration to

[3] See folder '1_Source_Code/org.bumble.xtext.grammartransformer' in our supplemental material (Zhang et al., 2024), which contains the 'transformationrule' project with the full implementation.

an Xtext-generated grammar exactly matches an expected ground-truth grammar, for which we previously manually established that it agrees (in the sense of *imitation*) with an expert-created one).

6.4. Configuration

The language engineer has to configure what transformation rules the GrammarTransformer should apply and how. This is supported by the API offered by GrammarTransformer. Listing 8 shows an example of how to configure the transformation rule applications in a method `executeTransformation()`, where the configuration revisits the DOT grammar transformation example transforming Listing 4 into Listing 7. Lines 3 to 6 configure transformation rule applications. For example, line 3 removes all curly braces in the grammar rule *NodeStmt*. The value of the first parameter is set to “NodeStmt”, which means that the operation of removing curly braces will occur in the grammar rule *NodeStmt*. If this first parameter is set to “null”, the operation would be executed for all grammar rules in the grammar. The second parameter is used to indicate the target attribute. Since it is set to “null”, all lines in the targeted grammar rule will be affected. However, if the parameter is set to a name of an attribute, only curly braces in the line containing that attribute will be removed. Finally, the third parameter can be used to indicate names of attributes for which the braces should not be removed. This can be used in case the second parameter is set to “null”.

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

Listing 8: Excerpt of the configuration of GrammarTransformer for the QVTo 1.0 language.)

```
1   public static boolean executeTransformation(\
        grammartransformer go) {
2     ...
3     go.removeBraces("NodeStmt", null, null);
4     go.removeKeyword("NodeStmt", null, null,
          null);
5     go.removeOptionality("NodeStmt", null);
6     go.convert1toStarToStar("NodeStmt", "
          attrLists");
7     ...
8   }
```

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

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

6.5. Execution

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

Figure 6 uses the first transformation operation from Listing 8 removing curly braces as an example to depict how

GrammarTransformer works internally when transforming grammars. The top of the figure shows an example input, which is the grammar rule `NodeStmt` generated from the meta-model of DOT (cf. Listing 4). In the lower right corner, the resulting transformed Xtext grammar rule is illustrated. In both illustrated grammar rule excerpt, blue fonts are the keywords and symbols (braces and commas).

In **Step 1 (initialization)**, GrammarTransformer builds a data structure out of the grammar initially generated by Xtext. That is, it builds a `:Grammar` object containing multiple `:GrammarRule` objects, with each of them containing several `:LineEntry` objects in an ordered list. For example, the `:Grammar` object contains a `:GrammarRule` object with the name `"NodeStmt"`. This `:GrammarRule` object contains seven `:LineEntry` objects, which represent the seven lines of the grammar rule in Listing 4. Three of these `:LineEntry` objects contain at least one curly brace (`" ‘{’ "` or `" ‘}’ "`). These lines are explicitly represented in order to later map relevant transformation rules to them. Figure 6 shows an excerpt of the object structure created for the example with the three line objects for the `NodeStmt` rule.

In **Step 2 (per Transformation Rule)** each transformation rule application is processed by executing the `apply()`-method. For our example, the transformation rule `removeBraces` is applied via the GrammarTransformer API as configured in line 3 of Listing 8.

In **Step 2a (localization of affected grammar rules and lines)**, the grammar rule and lines that need to be changed are located, based on the configuration of the transformation rule application. In the case of our example, the grammar rule `NodeStmt` (cf. line 1 in Listing 4) is identified. Then, all lines of that grammar rule are identified that include a curly brace. For example, the lines represented by `:LineEntry` objects as shown in Figure 6 are identified.

In **Step 2b (change)**, the code uses regular expressions for character-level matching and searching. If it finds curly braces surrounded by single quotes (i.e., `" ‘{’ "` and `" ‘}’ "`), it removes them.

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

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

6.6. Post-Processing vs. Changing Grammar Generation

GrammarTransformer is designed to modify grammars that Xtext generated out of meta-models. An alternative to this post-processing approach is to directly modify the Xtext grammar generator as, e.g., in XMLText (Neubauer et al., 2015, 2017). However, we deliberately chose a post-processing approach, because the application of conventional regular expressions enables the transferability to other recent language development frameworks like Langium (TypeFox GmbH, 2022) or textX (Dejanović et al., 2017), if they support the grammar generation from a meta-model in a future point in time. While the transformation rules implemented in grammar transformer are currently tailored to the structure of Xtext grammars, GrammarTransformer does not technically depend on Xtext and the rules could easily be adapted to a different grammar language. Furthermore, as the implementation of an Xtext grammar generator necessarily depends on many version-specific internal aspects of Xtext, the post-processing approach using regular expressions is considerably more maintainable.

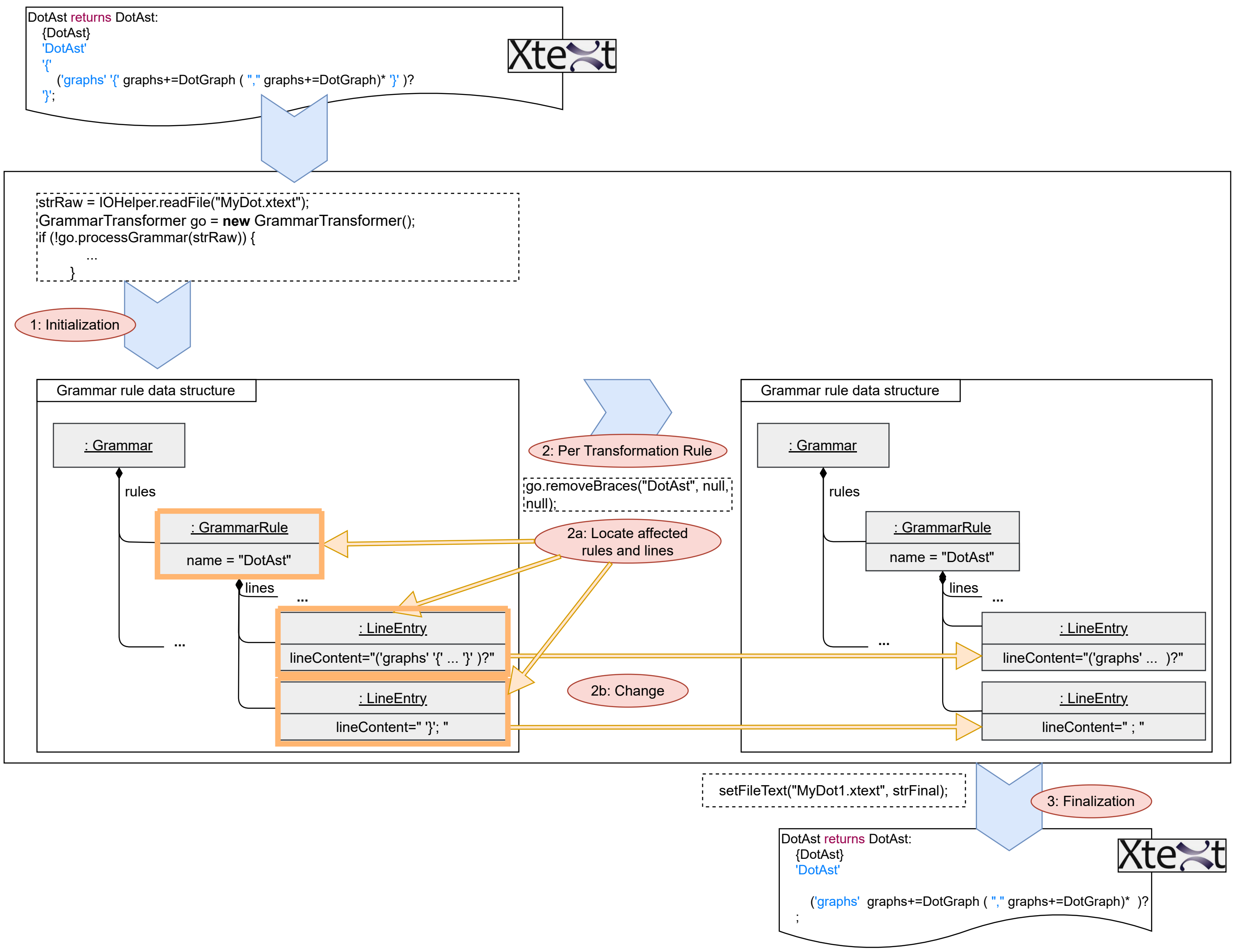

Figure 6: Exemplary Interplay of the Building Blocks of the GrammarTransformer

6.7. Limitations and Caveats

Our solution has the following limitations and caveats.

First, we were not able to completely imitate one of the seven languages. In order to do so, we would have had to provide an transformation rule that would require the GrammarTransformer user to input a multitude of parameter options. This would have strongly increased the effort and reduced the usability to use this one transformation rule, and the rule is only required for this one language. Thus, we argue that a manual post-adaptation is more meaningful for this one case. However, the inherent extensibility of the GrammarTransformer allows to add such an transformation rule if desired. We describe the issue in a more detailed manner in Section 7.1.4, which summarizes the evaluation results for the grammar adaptions of the seven analyzed languages.

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

Third, our solution does not strive to maintain backwards comparability to previous grammar versions—in general, after rule applications, instances of the previous, un-transformed grammar can no longer be parsed. This lack of backwards compatibility is generally desirable, as the alternative would be support for a mixing of old and new grammar elements (e.g., changed keywords and parantheses styles) in the same instance, which would generally be confusing to the user, and lead to issues with parsing and other tool support. However, to reduce manual effort in cases where legacy grammar instances exist, automated co-evolution of grammar instances after grammar changes is generally possible and leads to a promising future work direction (discussed in Section 8.4).

7. Evaluation

In this evaluation, we focus on two research questions:

- *RQ1: Can our solution be used to adapt generated grammars so that they produce the same language as available expert-created grammars?*
  The goal of this question is to validate the claim that our approach can automatically perform the changes that an expert would need to do manually. To this end, we consider languages for which an expert-created grammar exists, and validate the capability of our approach to re-create an equivalent grammar.

- *RQ2: Can our solution support the co-evolution of generated grammars when the meta-model evolves?*
  Our original motivation for the work was to enable evolution and rapid prototyping for textual languages built with a meta-model. The aim here is to evaluate whether our approach is suitable for supporting these evolution scenarios.

In the following, we address both questions. Our supplemental material (Zhang et al., 2024) contains the source code of the implementation as well as all experiments.

7.1. Grammar Adaptation (RQ1)

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

7.1.1. Cases

Our goal is to evaluate whether the GrammarTransformer can be used to transform the generated grammars so that their rules imitate the rules of the expert-created grammars. We reused the meta-model adaptations and generated grammars from Section 4.3. Furthermore, we continued working with the versions of ATL and SML in which parts of their languages were excluded as described in Section 4.2.

7.1.2. Method

For each DSL, we wrote a configuration for the final version of GrammarTransformer which was the result of the work described in Sections 4 to 6. The goal was to transform the generated grammar so as to 'imitate' as many grammar rules as possible from the expert-created grammar of the DSL. Note that this was an iterative process in which we incrementally added new transformation rule applications to the GrammarTransformer's configuration, using the expert-created grammar as a ground truth and using our notion of 'imitation' (cf. Section 4.4) as the gold standard. Essentially, we updated the GrammarTransformer configuration and then ran the tool before analysing the transformed grammar for imitation of the original. We repeated the process and adjusted the GrammarTransformer configuration until the test grammar's rules 'imitated' the expert-created grammar. Note that in the case of *Spectra*, we did not reach that point. We explain this in more detail in Section 7.1.4. For all experiments, we used the set of 56 transformation rules that were identified after the two iterations described in Section 4 and as summarized in Section 5.

To verify whether the transformed grammar imitates the expert-created grammar, we adopted a manual verification method, in which we systematically compared the grammar rules in the transformed grammar with the grammar rules in the expert-created grammar. An expert-created grammar is imitated by an transformed grammar if every grammar rule in it is imitated by one (or several) grammar rules from the transformed grammar. The procedure and results of this step are documented in our supplementary materials (Zhang et al., 2024).[4]

7.1.3. Metrics

To evaluate the transformation results of the GrammarTransformer on the case DSLs, we assessed the following metrics.

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

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

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

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

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

Transformed grammar Metrics about the resulting transformed grammar. We assess

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

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

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

7.1.4. Results

Table 4 shows the results of applying the GrammarTransformer to the seven DSLs. See Table 1 for the corresponding metrics of the initially generated grammars.

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

Limitation regarding Spectra. For one of the languages, Spectra, we were able to come very close to the expert-created grammar. Many grammar rules of Spectra could be nearly imitated. However, we did not implement all grammar rules that would have been necessary to allow the full transformation of Spectra. Listing 9 shows the grammar rule `TemporalPrimaryExpr` in Spectra's generated grammar, while Listing 10 shows what that grammar rule looks like in the expert-created grammar. In order to transform the grammar rule `TemporalPrimaryExpr` from Listing 9 to Listing 10, we need to configure the GrammarTransformer so that it combines the attribute `pointer` and `operator` multiple times, and the default value of the attribute `operator` is different each time. The language engineers using the GrammarTransformer need to input

[4]See directory '2_Supplemental_Material/Section_7_Evaluation'.

Table 4: Result of applying the GrammarTransformer to different DSLs (RQ1)

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

[1] The number includes the calls to dummy OCL and dummy SML expressions.

Listing 9: Example — grammar rule `TemporalPrimaryExpr` in the generated grammar of Spectra

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

Listing 10: Example — grammar rule `TemporalPrimaryExpr` in the expert-created grammar of Spectra

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

multiple parameters to ensure that the GrammarTransformer gets enough information, and this complex transformation requirement only appears in Spectra. Therefore we did not do such an transformation.

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

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

Furthermore, we argue that it is more efficient to configure GrammarTransformer 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 GrammarTransformer would quickly pay off when a language is going through changes, e.g., while rapidly prototyping modifications or when the language is evolving. In the next section (Section 7.2), we evaluate this assumption.

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

7.2. Supporting Evolution (RQ2)

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

7.2.1. Cases

The two cases we are using to evaluate how GrammarTransformer supports the evolution of a DSL are a textual variant of EAST-ADL (EAST-ADL Association, 2021) and QVT Operational (QVTo) (Object Management Group, 2016).

EAST-ADL. EAST-ADL is an architecture description language used in the automotive domain (EAST-ADL Association, 2021). Together with an industrial language engineer for EAST-ADL, we are currently developing a textual notation for version 2.2 of the language (Holtmann et al., 2023). We started this work with a simplified version of the meta-model to limit the complexity of the resulting grammar. In a later step, we switched to the full meta-model. We treat this switch as an evolution step here. The meta-model of EAST-ADL is taken from the EATOP repository (EAST-ADL Association, 2022). The meta-model of the simplified version contains 91 classes and enumerations, and the meta-model of the full version contains 291 classes and enumerations.

QVTo. QVTo is one of the languages in the OMG QVT standard (Object Management Group, 2016). We use the original meta-models available in Ecore format on the OMG website (Object Management Group, 2016). The baseline version is QVTo 1.0 (Object Management Group, 2008) and we simulate evolution to version 1.1 (Object Management Group, 2011), 1.2 (Object Management Group, 2015) and 1.3 (Object Management Group, 2016). Our original intention was to use the Eclipse reference implementation of QVTo (Eclipse Foundation, 2022b), but due to the differences in abstract syntax and concrete syntax (see Section 2), we chose to use the official meta-models instead. We analyzed four versions of QVTo's OMG official Ecore meta-model. There are 50 differences between the meta-models of version 1.0 and 1.1, 29 of which are parts that do not contain OCL (as for ATL as described in Section 4.2, we exclude OCL in our solution for QVTo). These 29 differences include different types, for example, 1) the same set of attributes has different arrangement orders in the same class in different versions of the meta-model; 2) the same class has different superclasses in different versions; 3) the same attribute has different multiplicities in different versions, etc. There are 3 differences between versions 1.1 and 1.2, all of which are from the OCL part. There is only one difference between versions 1.2 and 1.3, and it is about the same attribute having a different lower bound for the multiplicity in the same class in the two versions. Altogether we observed 54 meta-model differences in QVTo between the different versions (cf. the file "Comparison of QVTo metamodel versions" in the folder "Section_7_Evaluation/Subsection_7.2_Support" lists all the metamodel differences).

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

7.2.2. Preparation of the QVTo Case

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

Exclusion of OCL. As described in detail in Section 4.2, we excluded the embedded OCL language part from QVTo. For the meta-model, we introduced a dummy class for OCL, changed all calls to OCL types into calls to that dummy class, and removed the OCL metaclasses from the meta-model.

As described in Section 4.2, excluding a language part such as the embedded OCL from the scope of the investigation also implies that we need to exclude this language part when it comes to judging whether a grammar is imitated. Therefore, we substituted all grammar rules from the excluded OCL part with a placeholder grammar rule called `ExpressionGO` where an OCL grammar rule would have been called. This change allows us to compare the expert-created grammar of the different QVTo versions to the transformed grammar versions.

QVTo Meta-model Adaptations. We found that some non-terminals of QVTo's EBNF grammar are missing in the QVTo meta-model provided by OMG. For example, there is a non-terminal <top_level> in the EBNF grammar, but there is no counterpart for it in the meta-model. Therefore, we need to adapt the meta-model to ensure that it contains all the non-terminals in the EBNF grammar. To ensure that the adaptation of the meta-model is done systematically, we defined seven general adaptation rules that we followed when adapting the meta-models of the different versions. We list these adaptation rules in the supplemental material (Zhang et al., 2024).

As a result, we added 62 classes and enumerations with their corresponding references to each version of the meta-model. Note that this number is high compared to the original number of classes in the meta-model (24 classes).

This massive change was necessary, because the available Ecore meta-models were too abstract to cover all elements of the language. The original meta-model did contain most key concepts, but would not allow to actually specify a complete QVTo transformation. For example, with the original meta-model, it was not possible to represent the scope of a mapping or helper.

These changes enable us to imitate the QVTo grammar. However, they do not bias the results concerning the effects of the observed meta-model evolution as, with exception of a single case, these evolutionary differences are neither erased nor increased by the changes we performed to the meta-model. The exception is a meta-model evolution change between version 1.0 and 1.1 where the class `MappingOperation` has super types `Operation` and `NamedElement`, while the same class in V1.1 does not. The meta-model change performed by us removes the superclass `Operation` from `MappingOperation` in version 1.0. We did this change to prevent conflicts as the attribute *name* would have been inherited multiple times by `MappingOperation`. This in turn would cause problems in the generation process. Thus, only two of the 54 meta-model evolutionary differences could not be studied. The differences and their analysis can be found in the supplemental material (Zhang et al., 2024).

7.2.3. Method

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

Baseline GrammarTransformer Configuration. First, we generated the grammar for the initial version of a language's meta-model (i.e., the simple version for EAST-ADL and version 1.0 for QVTo). Then we defined the configuration of transformation rule applications that allows the GrammarTransformer to modify the generated grammar so that its grammar rules *imitate* the expert-created grammar for each case. Doing so confirmed the observation from the first part of the evaluation that a new language of sufficient complexity requires at least some new transformation rules (see Section 7.1.4). Consequently, we identified the need for four additional transformation rules for QVTo, which we implemented accordingly as part of the GrammarTransformer (this is also summarized in Section 5 in Table 2). This step provided us with a baseline configuration for the GrammarTransformer.

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

7.2.4. Metrics

We documented the metrics used in Section 7.1.3 for EAST-ADL and QVTo in their different versions. In addition, we also documented the following metric:

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

7.2.5. Results

Table 5 shows the results of the evolution cases.

EAST-ADL. Compared with the simplified version of EAST-ADL, the full version is much larger. It contains 291 metaclasses, i.e., 200 metaclasses more than the simple version of EAST-ADL, which leads to a generated grammar with 291 grammar rules and 2,839 non-blank lines in the generated grammar file (cf. Table 5).

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

We only needed to change and add a total of 10 grammar transformation rule applications to complete the transformation of the grammar of full EAST-ADL. For example, we excluded the primary type `String0` from the full version of the EAST-ADL grammar, which led us to add a line of configuration `go.removeRule(String0)`. While this is increasing the GrammarTransformer configuration from the simple EAST-ADL version quite a bit (from 22 transformation rule applications to 31 transformation rule applications), the increase is fairly small given that the meta-model increased massively (with 200 additional metaclasses).

The reason is that our grammar transformation requirements for the simplified version and the full version of EAST-ADL are almost the same. This transformation requirement is mainly based on the look and feel of the language and is provided by an industrial partner. These transformation rule applications have been configured for the simplified version. When we applied them to the generated grammar of the full version of EAST-ADL, we found that we can reuse all of these transformation rule applications. Furthermore, we benefit from the fact that many transformation rule applications are formulated for the scope of the whole grammar and thus can also influence grammar rules added during the evolution step. We do not list a number of grammar rules in a expert-created grammar of EAST-ADL in Table 5, because there is no "original" text grammar of EAST-ADL. Instead, we transform the generated grammar of EAST-ADL according to our industrial partner's requirements for EAST-ADL's textual concrete syntax.

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

However, if we look at the evolution towards versions 1.1, 1.2, and 1.3 we witness that very few changes to the GrammarTransformer configuration are required. In fact, only between 0 and 2 out of the 733 transformation rule applications needed adjustments. This significantly reduces the effort required compared to manually modifying a grammar generated from a new version of the QVTo metamodel, which would require modifying hundreds of lines. The reason is that, even though there are many differences between different versions of the QVTo metamodel, there are only 0 to 2 differences that affect the transformation rule applications.

For example, version 1.0 of the QVTo meta-model has an attribute called `bindParameter` in the class `VarParameter`, whereas it is called `representedParameter` in version 1.1. This attribute is not needed according to the expert-created grammars, so the GrammarTransformer configuration includes a call to the transformation rule *RemoveAttribute* to remove the grammar line that was generated based on that attribute. The second parameter of the transformation rule *RemoveAttribute* needs to specify the name of the attribute. As a consequence of the evolution, we had to change that name in the transformation rule application. Another example concerns the class `TypeDef`, which contains an attribute `typedef_condition` in version 1.2 of the QVTo meta-model. We added square brackets to it by applying the transformation rule *AddSquareBracketsToAttr* in the grammar transformation. However, in version 1.3 of the QVTo meta-model, the class `TypeDef` does not contain such an attribute, so the transformation rule application *AddSquareBracketsToAttr* was unnecessary.

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

8. Discussion

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

8.1. Threats to Validity

The threats to validity structured according to the taxonomy of Runeson et al. (Runeson and Höst, 2008; Runeson et al., 2012) are as follows.

8.1.1. Construct Validity

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

Furthermore, we had to make some changes to some of the meta-models to be able to generate Xtext grammars out of them at all (cf. Section 4.3) or to introduce certain language constructs required by the textual concrete syntax (cf. Section 7.2.2). These meta-model adaptations might have introduced biased changes and thereby impose a threat to construct validity. However, we reduced these adaptations to a minimum as far as possible to mitigate this threat and documented all of them in our supplemental material (Zhang et al., 2024) to ensure their reproducibility.

8.1.2. Internal Validity

In the evaluation (cf. Section 7), we set up and quantitatively evaluate size and complexity metrics regarding the considered meta-models and grammars as well as regarding the GrammarTransformer configurations for the use cases of one-time grammar adaptations and language evolution. Based on that, we conclude and argue in Sections 7.1.4 and 8.2 about the effort required for creating and evolving languages as well as the effort to create and re-use

Table 5: Result of supporting evolution (RQ2)

	Meta-m.	Generated grammar			Transformed grammar			Grammar rules			Lines in Grammar				
DSL	Classes [1]	lines	rules	calls	lines	rules	calls [2]	mod	add	del	mod	add	del	*#GORA*	*#cORA*
EAST-ADL (simple)	91	755	91	735	767	103	782	70	12	0	517	14	2	22	/
EAST-ADL (full)	291	2,839	291	3,062	2,851	303	3,074	233	12	1	2,046	16	4	31	10
QVTo 1.0	85	1,026	109	910	444	77	181	66	1	33	228	2	580	733	/
QVTo 1.1	85	992	110	836	444	77	181	66	1	34	228	2	546	733	2
QVTo 1.2	85	992	110	836	444	77	181	66	1	34	228	2	546	733	0
QVTo 1.3	85	991	110	835	443	77	180	66	1	34	228	2	546	733	1

[1] The number is after adaptation, and it contains both classes and enumerations.

[2] The number includes the calls to dummy OCL and dummy SML expressions.

GrammarTransformer configurations. These relations might be incorrect. However, the applied metrics provide objective and obvious indications about the particular sizes and complexities and thereby the associated engineering efforts.

8.1.3. External Validity

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

Generalisability is further affected by the question of how representative our cases are for other cases encountered in practice. Our evaluation would be most insightful if the considered languages resemble typical practical cases, instead of corner cases. The fact that we were able to derive rules from a subset of cases that were sufficient for largely—in one case, *entirely*— covering the other cases is a first indication that we did not exclusively deal with corner cases. However, A nuanced assessment of how typical our considered cases are for other cases would require systematic studies of evolution histories of metamodel-driven DSLs, which, to, our knowledge, are not available yet and would be a worthwhile direction for future work.

A related threat is with the software quality of our considered languages. Arguably, a language that was designed following best practices might require less evolution and would then also benefit less from our approach. Our approach is designed for practical use-cases, in which quality issues might be common. By supporting language evolution, our approach can contribute to changes that improve the quality of a language (e.g., introduce clearer keywords, more consistent parenthesis layout). The responsibility to use our tool in such way is with the user of our technique. Offering guidance for language design is an orthogonal issue addressed by other studies (Czech et al., 2018).

8.1.4. Reliability

Our overall procedure to conceive and develop the GrammarTransformer encompassed multiple steps. That is, we first determined the differences between the particular initially generated Xtext grammars and the grammars of the actual languages in two iterations as described in Section 4. This analysis yielded the corresponding identified conceptual grammar transformation rules summarized in Section 5. Based on these identified conceptual grammar transformation rules, we then implemented them as described in Section 6. This procedure imposes multiple threats to reliability. For example, analyzing a different set of languages could have led to a different set of identified transformation rules, which then would have led to a different implementation. Furthermore, analyzing the languages in a different order or as part of different iterations could have led to a different abstraction level of the rules and thereby a different number of rule. Finally, the design decisions that we made during the identification of the conceptual transformation rules and during their implementation could also have led to different kinds of rules or of the implementation. However, we discussed all of these aspects repeatedly amongst all authors to mitigate this threat and documented the results as part of our supplemental material (Zhang et al., 2024) to ensure their reproducibility.

8.2. The Effort of Creating and Evolving a Language with the GrammarTransformer

The results of our evaluation show three things. First, the expert-created grammars of all studied languages differ greatly in appearance from the generated grammars. Thus, in most cases, creating a DSL with Xtext will require the language engineer to perform big changes to the generated grammar. Second, in the case of complex changes, manually writing a GrammarTransformer configuration can lead to considerably less effort for the language engineer compared to manually adapting the grammar. Third, there seems to be a large potential for the reuse of GrammarTransformer configurations between different versions of a language, thus supporting the evolution of textual languages.

These observations can be combined with the experience that most languages evolve with time and that especially DSLs go through a rapid prototyping phase at the beginning where language versions are built for practical evaluation (Wang and Gupta, 2005). Therefore, we conclude that the GrammarTransformer has big potential to save manual effort when it comes to developing DSLs.

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

8.3. Implications for Practitioners and Researchers

Our results have several implications for language engineers and researchers.

Blended Modeling. Ciccozzi et al. (Ciccozzi et al., 2019) coin the term *blended modeling* for the activity of interacting with one model through multiple notations (e.g., both textual and graphical notations), which would increase the usability and flexibility for different kinds of model stakeholders. However, enabling blended modeling shifts more effort to language engineers. This is due to the fact that the realization of the different editors for the different notations requires many manual steps when using conventional modeling frameworks. In this context, Cicozzi and colleagues particularly stress the issue of the manual customization of grammars in the case of meta-model evolution. Thus, as one research direction to enable blended modeling, Ciccozzi et al. formulate the need to automatically generate the different editors from a given meta-model. Our work serves as one building block toward realizing this research direction and opens up the possibility to develop and evolve blended modeling languages that include textual versions.

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

Prevention of Language Flaws. Willink (Willink, 2020) reflects on the version history of the Object Constraint Language (OCL) and the flaws that were introduced during the development of the different OCL 2.x specifications by the Object Management Group (Object Management Group (OMG), 2014). Particularly, he points out that the lack of a parser for the proposed grammar led to several grammar inaccuracies and thereby to ambiguities in the concrete textual syntax. This in turn led to the fact that the concrete syntax and the abstract syntax in the Eclipse OCL implementation (Eclipse Foundation, 2022a) are so divergent that two distinct meta-models with a dedicated transformation between both are required, which also holds for the QVTo specification and its Eclipse implementation (Willink, 2020) (cf. Section 2). The GrammarTransformer will help to prevent and bridge such flaws in language engineering in the future. Xtext already enables the generation of the complete infrastructure for a textual concrete syntax from an abstract syntax represented by a meta-model. Our approach adds the ability to transform the grammar (i.e., the concrete syntax), as we show in the evaluation by deriving an applicable parser with an transformed grammar from the QVTo specification meta-models.

8.4. Future Work

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

Name Changes to Meta-model Elements. In the GrammarTransformer configurations, we currently reference the grammar concepts derived from the meta-model classes and attributes by means of the class and attribute names (cf. Listing 8). Thus, if a meta-model evolution involves many name changes, likewise many changes to transformation rule applications are required. Consequently, we plan as future work to improve the GrammarTransformer with a more flexible concept, in which we more closely align the grammar transformation rule applications with the meta-model based on name-independent references.

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

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

Another interesting idea would be to use artificial intelligence to learn existing examples of grammar transformations in existing languages to provide transformation suggestions for new languages and even automatically create configurations for the GrammarTransformer.

Expression Languages. In this paper, we excluded the expression language parts (e.g., OCL) of two of the example languages (cf. Section 4.2). However, expression languages define low-level concepts and have different kinds of grammars and underlying meta-models than conventional languages. In future work, we want to further explore expression languages specifically, in order to ensure that the GrammarTransformer can be used for these types of syntaxes as well.

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

Co-evolving Model Instances. We also intend to couple GrammarTransformer with an approach for language evolution that also addresses the model instances. In principle, a model instance represented by a textual grammar instance can be read using the old grammar and parsed into an instance of the old meta-model. It can then be transformed, e.g., using QVTo to conform to the new meta-model, and then be serialized again using the new grammar. However, following this approach means that formatting and comments can be lost. Instead, we intend to derive a textual transformation from the differences in the grammars and the transformation rule applications that can be applied to the model instances and maintain formatting and comments as much as possible.

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

9. Conclusion

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

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

EAST-ADL and QVTo.

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

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

Acknowledgements

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

References

A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM Sigplan Notices 35 (2000) 26–36.

Object Management Group, QVT – MOF Query/View/Transformation Specification, 2016. URL: `https://www.omg.org/spec/QVT/`, Accessed February, 2023.

Eclipse Foundation, ATL Syntax, 2018. URL: `https://wiki.eclipse.org/M2M/ATL/Syntax`, Accessed February, 2023.

Paperpile, A complete guide to the BibTeX format, 2022. URL: `https://www.bibtex.com/g/bibtex-format/`, Accessed February, 2023.

Graphviz Authors, Dot language, 2022. URL: `https://graphviz.org/doc/info/lang.html`, Accessed February, 2023.

J. Greenyer, Scenario Modeling Language (SML) Repository, 2018. URL: `https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/`, Accessed February, 2023.

Spectra Authors, Spectra, 2021. URL: `https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/src/tau/smlab/syntech/Spectra.xtext`, Accessed February, 2023.

Eclipse Foundation, Eclipse xcore wiki, 2018. URL: `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.

Xenia Authors, Xenia xtext, 2019. URL: `https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext`, Accessed February, 2023.

S. Roy Chaudhuri, S. Natarajan, A. Banerjee, V. Choppella, Methodology to develop domain specific modeling languages, in: Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling, ACM SIGPLAN, 2019, pp. 1–10.

U. Frank, Domain-specific modeling languages: requirements analysis and design guidelines, in: Domain engineering, Springer, 2013, pp. 133–157.

M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM computing surveys (CSUR) 37 (2005) 316–344.

L. Bettini, Implementing domain-specific languages with Xtext and Xtend, 2 ed., Packt Publishing Ltd, 2016.

D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Addison-Wesley Professional, 2008.

F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns, Blended modelling—what, why and how, in: 1^{st} Intl. Workshop on Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS), IEEE, 2019, pp. 425–430. doi:`10.1109/MODELS-C.2019.00068`.

W. Zhang, R. Hebig, D. Strüber, J.-P. Steghöfer, Automated extraction of grammar optimization rule configurations for metamodel-grammar co-evolution, in: 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE'23), 2023.

EAST-ADL Association, East-adl, 2021. URL: `https://www.east-adl.info/`, Accessed February, 2023.

J. Holtmann, J.-P. Steghöfer, W. Zhang, Exploiting meta-model structures in the generation of Xtext editors, in: 11th Intl. Conf. on Model-Based Software and Systems Engineering (MODELSWARD), 2023, pp. 218–225. doi:`10.5220/0011745900003402`.

R. F. Paige, D. S. Kolovos, F. A. Polack, A tutorial on metamodelling for grammar researchers, Science of Computer Programming 96 (2014) 396–416. doi:`10.1016/j.scico.2014.05.007`, selected Papers from the Fifth Intl. Conf. on Software Language Engineering (SLE 2012).

International Organization for Standardization (ISO), Information technology—Syntactic metalanguage—Extended BNF (ISO/IEC 14977:1996), 1996.

A. Kleppe, A language description is more than a metamodel, in: 4th International Workshop on Language Engineering, 2007.

Object Management Group (OMG), Object constraint language 2.x specification, 2014. URL: https://www.omg.org/spec/OCL/, Accessed February, 2023.

E. Willink, Reflections on OCL 2, Journal of Object Technology 19 (2020) 3:1–16. doi:10.5381/jot.2020.19.3.a17.

Eclipse Foundation, Eclipse OCL™ (Object Constraint Language), 2022a. URL: https://projects.eclipse.org/projects/modeling.mdt.ocl, Accessed February, 2023.

Eclipse Foundation, Qvto – eclipsepedia, 2022b. URL: https://wiki.eclipse.org/QVTo, Accessed February, 2023.

T. Parr, ANTLR, 2022. URL: https://www.antlr.org/, Accessed February, 2023.

P. Neubauer, A. Bergmayr, T. Mayerhofer, J. Troya, M. Wimmer, Xmltext: From xml schema to xtext, in: 2015 ACM SIGPLAN Intl. Conf. on Software Language Engineering, 2015, pp. 71–76. doi:10.1145/2814251.2814267.

P. Neubauer, R. Bill, M. Wimmer, Modernizing domain-specific languages with xmltext and intelledit, in: 2017 IEEE 24th Intl. Conf. on Software Analysis, Evolution and Reengineering (SANER), 2017.

S. Chodarev, Development of human-friendly notation for xml-based languages, in: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2016, pp. 1565–1571.

F. Jouault, J. Bézivin, I. Kurtev, Tcs: A dsl for the specification of textual concrete syntaxes in model engineering, in: 5th Intl. Conf. on Generative Programming and Component Engineering, ACM, 2006, p. 249–254. doi:10.1145/1173706.1173744.

M. Novotný, Model-driven Pretty Printer for Xtext Framework, Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2012.

U. Frank, Some guidelines for the conception of domain-specific modelling languages, in: Enterprise Modelling and Information Systems Architectures (EMISA 2011), Gesellschaft für Informatik eV, 2011, pp. 93–106.

J.-P. Tolvanen, S. Kelly, Effort used to create domain-specific modeling languages, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2018, pp. 235–244.

G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, S. Völkel, Design guidelines for domain specific languages, in: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09), TR no B-108, Helsinki School of Economics, Orlando, Florida, USA, 2009. URL: http://arxiv.org/abs/1409.2378.

M. van Amstel, M. van den Brand, L. Engelen, An exercise in iterative domain-specific language design, in: Proceedings of the joint ERCIM workshop on software evolution (EVOL) and international workshop on principles of software evolution (IWPSE), 2010, pp. 48–57.

M. Karaila, Evolution of a domain specific language and its engineering environment–lehman's laws revisited, in: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling, 2009, pp. 1–7.

M. Pizka, E. Jürgens, Tool-supported multi-level language evolution, in: Software and Services Variability Management Workshop, volume 3, 2007, pp. 48–67.

R. Hebig, D. E. Khelladi, R. Bendraou, Approaches to co-evolution of metamodels and models: A survey, IEEE Transactions on Software Engineering 43 (2016) 396–414.

D. E. Khelladi, R. Bendraou, R. Hebig, M.-P. Gervais, A semi-automatic maintenance and co-evolution of OCL constraints with (meta) model evolution, Journal of Systems and Software 134 (2017) 242–260.

D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, M.-P. Gervais, Metamodel and constraints co-evolution: A semi automatic maintenance of OCL constraints, in: International Conference on Software Reuse, Springer, 2016, pp. 333–349.

D. D. Ruscio, R. Lämmel, A. Pierantonio, Automated co-evolution of gmf editor models, in: International conference on software language engineering, Springer, 2010, pp. 143–162.

D. Di Ruscio, L. Iovino, A. Pierantonio, What is needed for managing co-evolution in mde?, in: Proceedings of the 2nd International Workshop on Model Comparison in Practice, 2011, pp. 30–38.

J. García, O. Diaz, M. Azanza, Model transformation co-evolution: A semi-automatic approach, in: International conference on software language engineering, Springer, 2012, pp. 144–163.

F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende, Derivation and refinement of textual syntax for models, in: European Conf. on Model Driven Architecture—Foundations and Applications (ECMDA-FA), volume 5562 of *LNCS*, Springer, 2009, pp. 114–129. doi:10.1007/978-3-642-02674-4_9.

A. Kleppe, Towards the generation of a text-based ide from a language metamodel, in: European Conf. on Model Driven Architecture—Foundations and Applications (ECMDA-FA), volume 4530 of *LNCS*, Springer, 2007, pp. 114–129. doi:10.1007/978-3-540-72901-3_9.

I. Dejanović, R. Vaderna, G. Milosavljević, Ž. Vuković, Textx: A python tool for domain-specific languages implementation, Knowledge-Based Systems 115 (2017) 1–4. doi:10.1016/j.knosys.2016.10.023.

TypeFox GmbH, Langium, 2022. URL: https://langium.org/, Accessed February, 2023.

S. Kelly, J.-P. Tolvanen, Collaborative creation and versioning of modeling languages with metaedit+, in: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2018, pp. 37–41.

JetBrains, MPS: The Domain-Specific Language Creator by JetBrains, 2022. URL: https://www.jetbrains.com/mps/, Accessed February, 2023.

AtlanMod Team, Atlantic zoo, 2019. URL: https://github.com/atlanmod/atlantic-zoo, Accessed February, 2023.

A. Nordmann, N. Hochgeschwender, D. Wigand, S. Wrede, An overview of domain-specific languages in robotics, 2020. URL: https://corlab.github.io/dslzoo/all.html, Accessed February, 2023.

I. Wikimedia Foundation, Wikipedia page of domain specific language, 2023. URL: https://en.wikipedia.org/wiki/Domain-specific_language, Accessed February, 2023.

M. Barash, Zoo of domain-specific languages, 2020. URL: http://dsl-course.org/, Accessed February, 2023.

I. Semantic Designs, Domain specific languages, 2021. URL: http://www.semdesigns.com/products/DMS/DomainSpecificLanguage.html, Accessed February, 2023.

D. Community, Financial domain-specific language listing, 2021. URL: http://dslfin.org/resources.html, Accessed February, 2023.

itemis AG, Dot xtext grammar, 2020. URL: https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext, Accessed February, 2023.

V. Zaytsev, Grammarware bibtex metamodel, 2013. URL: https://github.com/grammarware/slps/blob/master/topics/grammars/bibtex/bibtex-1/BibTeX.ecore, Accessed February, 2023.

Spectra Authors, Spectra metamodel, 2021. URL: https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/model/generated/Spectra.ecore, Accessed February, 2023.

Eclipse Foundation, Xcore metamodel, 2012. URL: https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore, Accessed February, 2023.

Xenia Authors, Xenia metmodel, 2019. URL: https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore, Accessed February, 2023.

W. Zhang, J. Holtmann, D. Strüber, R. Hebig, J.-P. Steghöfer, Grammartransformer_data: Formal release, 2024. doi:10.5281/zenodo.10683827, Accessed February 23, 2024.

J. E. Hopcroft, On the equivalence and containment problems for context-free languages, Mathematical systems theory 3 (1969) 119–124.

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

Object Management Group, QVT – MOF Query/View/Transformation Specification Version 1.0, 2008. URL: https://www.omg.org/spec/QVT/1.0/, Accessed February, 2023.

Object Management Group, QVT – MOF Query/View/Transformation Specification Version 1.1, 2011. URL: https://www.omg.org/spec/QVT/1.1/, Accessed February, 2023.

Object Management Group, QVT – MOF Query/View/Transformation Specification Version 1.2, 2015. URL: https://www.omg.org/spec/QVT/1.2/, Accessed February, 2023.

Object Management Group, QVT – MOF Query/View/Transformation Specification Version 1.3, 2016. URL: https://www.omg.org/spec/QVT/1.3/, Accessed February, 2023.

P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2008) 131–164. doi:10.1007/s10664-008-9102-8.

P. Runeson, M. Höst, R. Austen, B. Regnell, Case Study Research in Software Engineering — Guidelines and Examples, 1^{st} ed., Wiley, 2012.

G. Czech, M. Moser, J. Pichler, Best practices for domain-specific modeling. a systematic mapping study, in: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2018, pp. 137–145.

Q. Wang, G. Gupta, Rapidly prototyping implementation infrastructure of domain specific languages: a semantics-based approach, in: Proceedings of the 2005 ACM symposium on Applied computing, 2005, pp. 1419–1426.