

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

Weixing Zhang<sup>a</sup>, Jörg Holtmann<sup>b</sup>, Daniel Strüber<sup>a,c</sup>, Regina Hebig<sup>d</sup>, Jan-Philipp Steghöfer<sup>e</sup>

<sup>a</sup>Department of Computer Science & Engineering, Chalmers / University of Gothenburg, Gothenburg, Sweden

<sup>b</sup>Independent Researcher (main work conducted at <sup>a</sup>), Paderborn, Germany

<sup>c</sup>Department of Software Science, Radboud University, Nijmegen, Netherlands

<sup>d</sup>Institute of Computer Science, University of Rostock, Rostock, Germany

<sup>e</sup>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 transfor-

mation 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](mailto:weixing.zhang@gu.se) (Weixing Zhang),  
[joerg\\_holtmann@gmx.de](mailto:joerg_holtmann@gmx.de) (Jörg Holtmann), [danstru@chalmers.se](mailto:danstru@chalmers.se)  
(Daniel Strüber), [regina.hebig@uni-rostock.de](mailto:regina.hebig@uni-rostock.de) (Regina Hebig),  
[jan-philipp.steghoefer@xitaso.com](mailto:jan-philipp.steghoefer@xitaso.com) (Jan-Philipp Steghöfer)

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

22 A different way to go about constructing DSLs is  
23 proposed by model-driven engineering. There, the con-  
24 cepts that are relevant in the domain are captured in a  
25 meta-model which defines the *abstract syntax* (see, e.g.,  
26 (Roy Chaudhuri et al., 2019; Frank, 2013; Mernik et al.,  
27 2005)). Different *concrete syntaxes*, e.g., graphical, textual,  
28 or form-based, can be defined to describe actual models  
29 that adhere to the abstract syntax.

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

42 Software languages change over time. This is due to  
43 *language evolution*, which entails that languages change  
44 over time to address new and changed requirements, and  
45 due to *rapid prototyping*, which involves many quick itera-  
46 tions on an initial design. In the case of an Xtext-based  
47 language, grammar and meta-model need to be modified to  
48 stay consistent with each other. We consider two options  
49 for evolving a language in Xtext: First, the developers  
50 can change the grammar and then use Xtext to automati-  
51 cally create an updated version of the meta-model from it.  
52 Second, the developers can change the meta-model then  
53 use Xtext to update the grammar. We call the first ap-  
54 proach *grammar-based evolution*, and the second approach

*meta-model-based evolution*. 55

56 In this paper, we focus on meta-model-based evolution,  
57 for the following rationale: While grammar-based evolution  
58 is a common way of developing languages in Xtext, it  
59 is not geared for three scenarios that we encountered in  
60 the real world, including collaborations with an industrial  
61 partner. In particular: 1. Several concrete syntaxes (e.g.,  
62 visual, textual, tabular) for the same underlying metamodel  
63 co-exist and evolve at the same time. This is particular  
64 common in the context of blended modeling (Ciccozzi et al.,  
65 2019), a timely modeling paradigm. 2. The metamodel  
66 comes from some external source (such as a third-party  
67 supplier or a standardization committee), which prohibits  
68 independent modification. 3. The metamodel is the central  
69 artifact of a larger ecosystem of available tools, including  
70 e.g., automated analyses and transformations. As such,  
71 the language engineers might prefer to evolve it directly,  
72 instead of relying on the, potentially sub-optimal, output of  
73 automatically co-evolving it after grammar changes. The  
74 real-world case that inspired this paper has aspects of  
75 the first two scenarios: we work on a language from an  
76 industry partner for which there already exists an evolving  
77 metamodel and graphical editor available.

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

91 In this paper, we propose a different approach to support-  
92 ing meta-model-based evolution: Automated synchroniza-

tion 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 en-

tails 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

169 version 2).

170 The remainder of this paper is structured as follows.  
171 First, in Section 2, we provide an overview of the back-  
172 ground of this paper, in particular, on metamodel-based  
173 textual DSL engineering. In Section 3, we review related  
174 research. In Section 4, we define the methodology of this  
175 paper. Subsequently, in Section 5, we describe the iden-  
176 tified transformation rules, which are the main technical  
177 contribution of this paper. Following that, in Section 6,  
178 we present our solution of the GRAMMARTransformer,  
179 which implements the identified transformation rules. In  
180 Section 7, we present our evaluation. Section 8 is devoted  
181 to our discussion, where we address threats to validity, the  
182 effort required to use GRAMMARTransformer, implica-  
183 tions for practitioners and researchers, and future work.  
184 Finally, in the last section, we conclude.

## 185 2. Background: Textual DSL Engineering based on 186 Meta-models

187 The engineering of textual DSLs can be conducted  
188 through the traditional approach of specifying grammars,  
189 but also by means of meta-models. Both approaches have  
190 commonalities, but also differences (Paige et al., 2014). Like  
191 grammars specified by means of the Extended Backus Naur  
192 Form (EBNF) (International Organization for Standardiza-  
193 tion (ISO), 1996), meta-models enable formally specifying  
194 how the terms and structures of DSLs are composed. In  
195 contrast to grammar specifications, however, meta-models  
196 describe DSLs as graph structures and are often used as  
197 the basis for graphical or non-textual DSLs. Particularly,  
198 the focus in meta-model engineering is on specifying the  
199 abstract syntax. The definition of concrete syntaxes is  
200 often considered a subsequent DSL engineering step. How-  
201 ever, the focus in grammar engineering is directly on the  
202 concrete syntax (Kleppe, 2007) and leaves the definition of  
203 the abstract syntax to the compiler.

204 *Meta-model-based textual DSLs.* There are also examples  
205 of textual DSLs that are built with meta-model technology.  
206 For example, the Object Management Group (OMG) de-  
207 fines textual DSLs that hook into their meta-model-based  
208 Meta Object Facility (MOF) and Unified Modeling Lan-  
209 guage ecosystems, for example, the Object Constraint Lan-  
210 guage (OCL) (Object Management Group (OMG), 2014)  
211 and the Operational transformation language of the MOF  
212 Query, View, and Transformation (QVTo) (Object Manage-  
213 ment Group, 2016). However, this is done in a cumbersome  
214 way: Both the specifications for OCL and QVTo define a  
215 meta-model specifying the abstract syntax and a grammar  
216 in EBNF specifying the concrete syntax of the DSL. This  
217 grammar, in turn, defines a different set of concepts and,  
218 therefore, a meta-model for the concrete syntax that is  
219 different from the meta-model for the abstract syntax. As  
220 Willink (Willink, 2020) points out, this leads to the awk-  
221 ward fact that the corresponding tool implementations such  
222 as Eclipse OCL (Eclipse Foundation, 2022a) and Eclipse  
223 QVTo (Eclipse Foundation, 2022b) also apply this distinc-  
224 tion. That is, both tool implementations require an abstract  
225 syntax and a concrete syntax meta-model and, due to their  
226 structural divergences, a dedicated transformation between  
227 them. Additionally, both tool implementations provide a  
228 hand-crafted concrete syntax parser, which implements the  
229 actual EBNF grammar. Maintaining these different parts  
230 and updating the manually created ones incurs significant  
231 effort whenever the language should be evolved.

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

```

1 EAXML
2 {
3   topLevelPackage
4   {
5     EAPackage
6     {
7       shortName Structure
8       subPackage
9       {
10        EAPackage
11        {
12          shortName DesignPkg
13          subPackage
14          {
15            EAPackage
16            {
17              shortName FcnDesignArchitecture_new
18              element
19              {
20                DesignFunctionType
21                {
22                  shortName FdAWithController_new
23                  part
24                  {
25                    DesignFunctionPrototype
26                    {
27                      shortName wiperCtrlBasic
28                      type "Structure.DesignPkg.FcnDesignArchitecture_new"
29                    },
30                    DesignFunctionPrototype
31                    {
32                      shortName wiperCtrlBasic1018
33                      type "Structure.DesignPkg.FcnDesignPkg_new.WiperCtrl"
34                    }
35                  }
36                }
37              }
38            }
39          }
40        }
41      }
42    }
43  }
44 }

```

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

241 Xtext supports both grammar-based and meta-model-  
 242 based-evolution in the sense introduced in Section 1. For  
 243 our considered meta-model-based scenario, Xtext’s default  
 244 workflow requires that after each meta-model change, the  
 245 grammar has to be manually synchronized (Bettini, 2016),  
 246 a disadvantage we aim to avoid with our approach. To this  
 247 end, we rely on a built-in feature of Xtext for automati-  
 248 cally deriving a grammar from a meta-model. (we call this  
 249 grammar *generated grammar* in this paper). This creates a  
 250 grammar that contains grammar rules for all meta-model  
 251 elements that are contained in a common root node and  
 252 resolves references, etc., to a degree (see Section 4.3 for de-  
 253 tails). This grammar is typically quite verbose, structured  
 254 extensively using braces, and uses a lot of keywords, as illus-  
 255 trated with the example in Figure 1, depicting an instance  
 256 of the generated grammar for EAST-ADL. Therefore, gener-  
 257 ated grammars are intended to be improved before being  
 258 used in practice (Bettini, 2016). In our approach, we use  
 259 generated grammars as the starting point for recording  
 260 and automatically replaying changes made to the grammar,  
 261 thus avoiding manual synchronization effort.

### 3. Related Work

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

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

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

290 Grammar transformation for DSLs in general is ad-  
 291 dressed by Jouault et al. (2006). They propose an ap-  
 292 proach to specify a syntax for textual, meta-model-based  
 293 DSLs with a dedicated DSL called Textual Concrete Syn-  
 294 tax, which is based on a meta-model. From such a syntax  
 295 specification, a concrete grammar and a parser are gener-  
 296 ated. The approach is similar to a template language  
 297 restricting the language engineer and thereby, as the au-  
 298

299 thors state, lacks the freedom of grammar specifications  
300 in terms of syntax customization options. In contrast, we  
301 argue that the GRAMMARTransformer provides more  
302 syntax customization options to achieve a well-accepted  
303 textual DSL.

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

311 *Designing and Evolving Meta-model-based DSLs.* Many  
312 papers about the design of DSLs focus solely on the con-  
313 struction of the abstract syntax and ignore the concrete  
314 syntaxes (e.g., (Roy Chaudhuri et al., 2019; Frank, 2011)),  
315 or focus exclusively on graphical notations (e.g.,(Frank,  
316 2013; Tolvanen and Kelly, 2018)). In contrast, the guide-  
317 lines proposed by Karsai et al. (2009) contain specific ideas  
318 about concrete syntax design, e.g., to “balance compact-  
319 ness and comprehensibility”. Arguably, the languages au-  
320 tomatically generated by Xtext are neither compact nor  
321 comprehensible and therefore require manual changes.

322 Mernik et al. (2005) acknowledge that DSL design is  
323 not a sequential process. The paper also mentions the im-  
324 portance of textual concrete syntaxes to support common  
325 editing operations as well as the reuse of existing languages.  
326 Likewise, van Amstel et al. (2010) describe DSL devel-  
327 opment as an iterative process and use EMF and Xtext  
328 for the textual syntax of the DSL. They also discuss the  
329 evolution of the language, and that “it is hard to predict  
330 which language features will improve understandability and  
331 modifiability without actually using the language”. Again,  
332 this is an argument for the need to do prototyping when  
333 developing a language. Karaila (2009) broadens the scope  
334 and also argues for the need for evolving DSLs along with  
335 the “engineering environment” they are situated in, in-

cluding editors and code generators. Pizka and Jürgens  
(2007) also acknowledge the “constant need for evolution”  
of DSLs.

339 There is a lot of research supporting different aspects of  
340 language change and evolution. Existing approaches focus  
341 on how diverse artifacts can be co-evolved with evolving  
342 meta-models, namely the models that are instances of the  
343 meta-models (Hebig et al., 2016), OCL constraints that are  
344 used to specify static semantics of the language (Khelladi  
345 et al., 2017, 2016), graphical editors of the language (Ruscio  
346 et al., 2010; Di Ruscio et al., 2011), and model transfor-  
347 mations that consume or produce programs of the lan-  
348 guage (García et al., 2012). Specifically, the evolution of  
349 language instances with evolving meta-models is well sup-  
350 ported by research approaches. For example, Di Ruscio et  
351 al. (Di Ruscio et al., 2011) support language evolution by  
352 using model transformations to simultaneously migrate the  
353 meta-model as well as model instances.

354 Thus, while these approaches cover a lot of requirements,  
355 there is still a need to address the evolution of textual gram-  
356 mars with the change of the meta-model as it happens dur-  
357 ing rapid prototyping or normal language evolution. This  
358 is a challenge, especially since fully generated grammars  
359 are usually not suitable for use in practice. This implies  
360 that upon changing a meta-model, it is necessary to co-  
361 evolve a manually created grammar or a grammar that has  
362 been generated and then manually changed. GRAMMARTransformer  
363 has been created to support prototyping  
364 and evolution of DSLs and is, therefore, able to support  
365 and largely automate these activities.

366 *Other Approaches.* As we mentioned above, besides Xtext,  
367 there are two more approaches that support the generation  
368 of EBNF-based grammars and from these the generation of  
369 the actual parsers. These are EMFText (Heidenreich et al.,  
370 2009) and the Grasland toolkit (Kleppe, 2007), which are  
371 both not maintained anymore.

372 Whereas our work focuses on the Eclipse technology stack

373 based on EMF and Xtext, there are a number of other lan-  
374 guage workbenches and supporting tools that support the  
375 design of DS(M)Ls and their evolution. However, none of  
376 these approaches are able to derive grammars directly from  
377 meta-models, a prerequisite for the approach to language  
378 engineering we propose and the basis of our contribution,  
379 GRAMMARTRANSFORMER. Instead, tools like textX (De-  
380 janović et al., 2017) go the other way around and derive the  
381 meta-model from a grammar. Langium (TypeFox GmbH,  
382 2022) is the self-proclaimed Xtext successor without the  
383 strong binding to Eclipse, but does not support this par-  
384 ticular use case just yet and instead focuses on language  
385 construction based on grammars. MetaEdit+ (Kelly and  
386 Tolvanen, 2018) does not offer a textual syntax for the  
387 languages, but instead a generator to create text out of  
388 diagrams that are modeled using either tables, matrices,  
389 or diagrams. JetBrains MPS (JetBrains, 2022) is based  
390 on projectional editing where concrete syntaxes are projec-  
391 tions of the abstract syntax. However, these projections  
392 are manually defined and not automatically derived from  
393 the meta-model as it is the case with Xtext. Finally, Pizka  
394 and Jürgens (2007) propose an approach to evolve DSLs  
395 including their concrete syntaxes and instances. For that,  
396 they present “evolution languages” that evolve the concrete  
397 syntax separately. However, they focus on DSLs that are  
398 built with classical compilers and not with meta-models.

## 399 4. Methodology

400 In this section, we describe our research methodology,  
401 shown in an overview in Figure 2. Our methodology con-  
402 sists of a number of sequential steps, in particular: selecting  
403 the case languages, preparing metamodels and grammars  
404 (including the exclusion of certain parts of the language),  
405 and two iterations of analysis, including extraction of gram-  
406 mar transformation rules and tool development. We now  
407 describe all of these steps in detail.

### 408 4.1. Selection of Sample DSLs

409 We selected a number of DSLs for which both an expert-  
410 created grammar and a meta-model were available. Our  
411 key idea was that the expert-created grammar serves as a  
412 *ground truth*, specifying what a desirable target of an trans-  
413 formation process would look like. As the starting point  
414 for this transformation process, we considered the Xtext-  
415 generated grammars for the available meta-models. The  
416 goal of our grammar transformation rules was to support  
417 an automated transformation to turn the Xtext-generated  
418 grammar into the expert-created grammar. By selecting a  
419 number of DSLs with a grammar or precise syntax defini-  
420 tion from which we could derive such a ground truth, we  
421 aimed to generalize the grammar transformation rules so  
422 that new languages can be transformed based on rules that  
423 we include in GRAMMARTRANSFORMER.

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

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

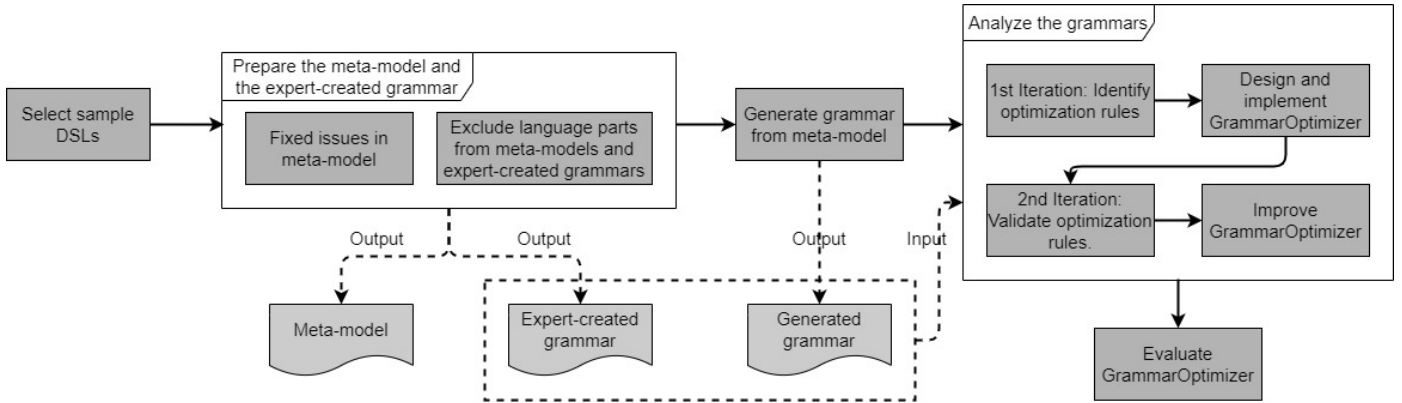


Figure 2: Overview of our methodology.

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

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

465 Therefore, we used the Xtext grammar that parses the  
 466 same language as DOT’s expert-created grammar to derive  
 467 a meta-model (itemis AG, 2020). Xtext grammars include  
 468 more information than an EBNF grammar, such as infor-  
 469 mation about references between concepts of the language.  
 470 Thus, the fact that the DOT grammar was already formu-  
 471 lated in Xtext allowed us to directly generate DOT’s Ecore

472 meta-model from this Xtext grammar. This meta-model  
 473 acquisition method is an exception in this paper. Since  
 474 this paper focuses on how to transform the generated gram-  
 475 mar, we consider this way of obtaining the meta-model  
 476 acceptable for this one case.

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

#### 4.2. Exclusion of Language Parts for Low-level Expressions 491

492 Two of the analyzed languages encompass language parts  
 493 for expressions, which describe low-level concepts like bi-  
 494 nary expressions (e.g., addition). We excluded such lan-  
 495 guage parts in ATL and in SML due to several aspects.  
 496 Both languages distinguish the actual language part and  
 497 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.

Iteration	DSL	Meta-model		Expert-created Grammar		Generated Grammar		
		Source	Classes <sup>1</sup>	Source	Rules	lines	rules	calls
1st	ATL <sup>2</sup>	Atlantic Zoo (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 <sup>3</sup>	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

<sup>1</sup> After adaptations, containing both classes and enumerations.

<sup>2</sup> Excluding embedded OCL rules.

<sup>3</sup> Excluding embedded SML expressions rules.

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)<sup>1</sup>.

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

This dummy grammar rule is then called everywhere where a rule of the excluded language part would have been called.

### 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

<sup>1</sup>See folder “Section\_4\_Methodology”

534 material (Zhang et al., 2024)<sup>2</sup>.

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

#### 538 4.4. Comparing EBNF and Xtext grammars

539 As a prerequisite for our analysis of grammars, we  
540 present a strategy for dealing with a noteworthy aspect  
541 of our methodology: in several cases, we dealt with lan-  
542 guages where the expert-created grammar was available  
543 in EBNF, whereas our contribution targets Xtext, which  
544 augments EBNF with additional technicalities, such as  
545 cross-references and datatypes. Hence, to validate whether  
546 our approach indeed produces grammars that are equiv-  
547 alent to expert-created ones, we needed a concept that  
548 allows comparing EBNF to Xtext grammars.

549 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.

558 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.

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

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

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

```
1 EdgeStmtNode returns EdgeStmtNode:  
2     {EdgeStmtNode}  
3     node=NodeId  
4     (edgeRHS+=EdgeRhs)+  
5     (attrLists+=AttrList)*  
6     ;  
7  
8 EdgeStmtSubgraph returns EdgeStmtSubgraph:  
9     {EdgeStmtSubgraph}  
10    subgraph=Subgraph  
11    (edgeRHS+=EdgeRhs)+  
12    (attrLists+=AttrList)*  
13    ;
```

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.

#### 578 4.5. Analysis of Grammars

579 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.

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

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

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

- 607 • Remove all the braces in the grammar rule `NodeStmt`.
- 608 • Remove all the keywords in the grammar rule  
609 `NodeStmt`.
- 610 • Remove the optionality from all the attributes in the  
611 grammar rule `NodeStmt`.
- 612 • Change the multiplicity of the attribute `attrLists`  
613 from `1..*` to `0..*`.

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

#### 619 4.5.1. First Iteration: Identify Transformation Rules

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

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

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

```
1 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+=
7             AttrList ( "," attrLists+=
                AttrList)* '}' )?
            '}' ;
```

2. derive grammar transformation rules that can be ap- 624  
 625 plied to change the generated grammar so that the  
 626 transformed grammar parses the same language as the  
 627 expert-created grammar.

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

635 As a result of the analysis, we obtained an initial set of  
 636 grammar transformation rules, which contained a total of  
 637 58 candidate transformation rules. Table 2 summarizes in  
 638 the second column the number of identified rule candidates  
 639 and in the second row the number for the first iteration.  
 640 Since the initial set of grammar transformation rules was a  
 641 result of an analysis done by multiple authors, it included  
 642 rules that were partially overlapping and rules that turned  
 643 out to only affect the grammar’s formatting, but not the  
 644 language specified by the grammar. Thus, we filtered rules  
 645 that belong to the latter case. For rule candidates that  
 646 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

```

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

```

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

721 We configured the preliminary GRAMMARTRANS-  
722 FORMER for the new languages by specifying which trans-  
723 formation rules to apply on the generated grammar. The  
724 execution results showed that the existing transformation  
725 rules were sufficient to change the generated grammar of  
726 Xenia to imitate the expert-created grammar used as the  
727 ground truth. However, we could not fully transform the  
728 generated grammar of Xcore and Spectra with the prelimi-  
729 nary set of 46 transformation rules from the first iteration.  
730 For example, Listing 5 shows two attributes `unordered` and  
731 `unique` in the grammar rule `XOperation` in the generated  
732 grammar for Xcore. However, in the expert-created gram-  
733 mar, the rule portions for the two attributes each refer to  
734 the other attribute in a way that allows using the keywords  
735 in several possible orders, as shown in Listing 6. This trans-  
736 formation could not be performed with the transformation  
737 rules from the first iteration.

738 Based on the non-transformed parts of the grammars  
739 of Xcore and Spectra, we identified another ten transfor-  
740 mation rules for the GRAMMARTRANSFORMER. These  
741 ten newly identified transformation rules transform all the  
742 non-transformed parts of the grammar of Xcore, including,  
743 e.g., transforming the grammar in Listing 5 to Listing 6.

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

```

1 ...
2         unordered?='unordered' unique?='
3         unique'? |
4         unique?='unique' unordered?='
5         unordered'?
6 ...

```

744 These new transformation rules also transform part of the  
745 non-transformed parts of the grammar of Spectra. We  
746 will interpret the remaining non-transformed parts in the  
747 *Evaluation* section. In the end, after two iterations, we  
748 identified a total of 56 transformation rules (which will be  
749 implemented by a total of 67 rule variants) (cf. fourth row  
750 in Table 2).

## 751 5. Identified Transformation Rules

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

760 Table 3 shows some examples of the transformation rules.  
761 The rules we implemented can be categorized by the primi-  
762 tives they manipulate: grammar rules, attributes keywords,  
763 braces, multiplicities, optionality (a special form of multi-  
764 plicities), grammar rule calls, import statements, symbols,  
765 primitive types, and lines. They either ‘add’ things (e.g.,  
766 *AddKeywordToRule*), ‘remove’ things (e.g., *RemoveOption-*  
767 *ality*), or ‘change’ things (e.g., *ChangeCalledRule*). All  
768 transformation rules ensure that the resulting changed  
769 grammar is still valid and syntactically correct Xtext.

770 Most transformation rules are ‘scoped’ which means that  
771 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 ;

```

772 In other cases, the scope is configurable, depending on  
773 the parameters of the transformation rule. For instance,  
774 the *RenameKeyword* rule takes a grammar rule and an  
775 attribute as a parameter. If both are set, the scope is the  
776 given attribute in the given rule. If no attribute is set, the  
777 scope is the given grammar rule. If none of the parameters  
778 is set, the scope is the entire grammar (“Global”). All  
779 occurrences of the given keyword are then renamed inside  
780 the respective scope.

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

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

798 For the example in Listing 4, this means that the neces-  
799 sary changes to reach the same language defined in Listing 3  
800 can be implemented using the following `GRAMMARTrans-`

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

FORMER 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

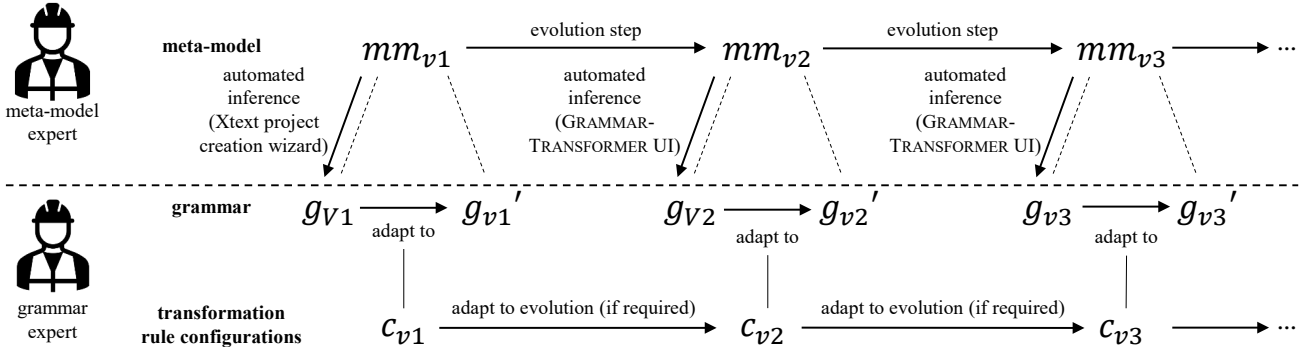


Figure 3: Co-evolution workflow with GRAMMARTransformer. Dashed lines indicate grammar/meta-model conformance.

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 adaptations 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

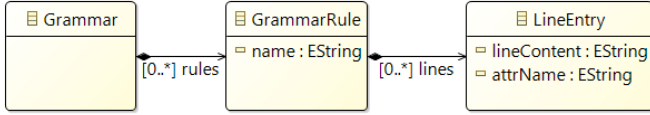


Figure 4: The class design for representing grammar rules.

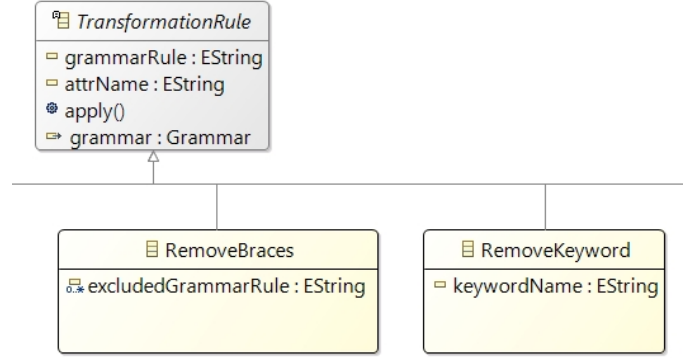


Figure 5: Excerpt of the class diagram for transformation rules.

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

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

### 883 6.2. Grammar Representation

884 We designed GRAMMARTransformer to parse an  
 885 Xtext grammar into an internal data structure which is then  
 886 modified and written out again. This internal representa-  
 887 tion of the grammar follows the structure depicted in Fig-  
 888 ure 4. A **Grammar** contains a number of **GrammarRules** that  
 889 can be identified by their names. In turn, a **GrammarRule**  
 890 consists of a sorted list of **LineEntries** with their textual  
 891 **lineContent** and an optional **attrName** that contains the  
 892 name of the attribute defined in the line. Note that we  
 893 utilize the fact that Xtext generates a new line for each  
 894 attribute.

### 895 6.3. Transformation Rule Design

896 Internally, all transformation rules derive from the ab-  
 897 stract class **TransformationRule** as shown in Figure 5.  
 898 Derived classes overwrite the **apply()**-method to perform  
 899 the specific text modifications for this rule. By doing so, the  
 900 specific rule can access the necessary information through  
 901 the class members: **grammar** (i.e., the entire grammar rep-  
 902 resentation as explained in Section 6.2 and depicted in  
 903 Figure 4), **grammarRuleName** (i.e., the name of the speci-  
 904 fied grammar rule that a user wants to transform exclu-  
 905 sively), and **attrName** (i.e., the name of an attribute that a

user wants to transform exclusively). Sub-classes can also  
 906 add additional members if necessary. This architecture  
 907 makes the GRAMMARTransformer extensible, as new  
 908 transformation rules can easily be defined in the future.  
 909

We built the transformation rules in a model-based man-  
 910 ner by first creating the meta-model shown in Figure 5 and  
 911 then using EMF to automatically generate the class bodies  
 912 of the transformation rules. This way we only needed to  
 913 overwrite the **apply()**-method for the concrete rules. Inter-  
 914 nally, the **apply()**-methods of our transformation rules are  
 915 implemented using regular expressions. Each transforma-  
 916 tion rule takes a number of parameters, e.g., the name of  
 917 the grammar rule to work on or an attribute name to iden-  
 918 tify the line to work on. In addition, some transformation  
 919 rules take a list of exceptions to the scope. For example,  
 920 the transformation rule to remove braces can be applied  
 921 to a global scope (i.e., all grammar rules) while excluding  
 922 a list of specific grammar rules from the processing. This  
 923 allows to configure transformation rule applications in a  
 924 more efficient way. We implemented all identified trans-  
 925 formation rules.<sup>3</sup> For testing, we built a comprehensive  
 926 test suite, based on the transformed grammars considered  
 927 in our design methodology. We created one test case per  
 928 scenario, to ensure that the grammar produced by our  
 929 implementation after applying a full given configuration to  
 930

<sup>3</sup>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.



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

#### 935 6.4. Configuration

936 The language engineer has to configure what transfor-  
937 mation rules the GRAMMARTRANSFORMER should apply  
938 and how. This is supported by the API offered by GRAM-  
939 MARTRANSFORMER. Listing 8 shows an example of how to  
940 configure the transformation rule applications in a method  
941 `executeTransformation()`, where the configuration revis-  
942 its the DOT grammar transformation example transforming  
943 Listing 4 into Listing 7. Lines 3 to 6 configure transfor-  
944 mation rule applications. For example, line 3 removes all  
945 curly braces in the grammar rule `NodeStmt`. The value of  
946 the first parameter is set to “NodeStmt”, which means that  
947 the operation of removing curly braces will occur in the  
948 grammar rule `NodeStmt`. If this first parameter is set to  
949 “null”, the operation would be executed for all grammar  
950 rules in the grammar. The second parameter is used to in-  
951 dicate the target attribute. Since it is set to “null”, all lines  
952 in the targeted grammar rule will be affected. However, if  
953 the parameter is set to a name of an attribute, only curly  
954 braces in the line containing that attribute will be removed.  
955 Finally, the third parameter can be used to indicate names  
956 of attributes for which the braces should not be removed.  
957 This can be used in case the second parameter is set to  
958 “null”.

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

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 }
```

removed. This can be used in case the second parameter is 968  
set to “null”. 969

Line 5 is used to remove the optionality from all lines 970  
in the grammar rule `NodeStmt`. If the second parameter 971  
gets an argument that carries the name of an attribute, 972  
the optionality is removed exclusively from the grammar 973  
line specifying the syntax for this attribute. 974

Finally, line 6 changes the multiplicity of the attribute 975  
`attrLists` in the grammar rule `NodeStmt` from `1..*` to 976  
`0..*`. If the second parameter would get the argument 977  
“null”, this adaptation would have been executed to all 978  
lines representing the respective attributes. 979

#### 980 6.5. Execution

981 Once the language engineer has configured GRAM-  
982 MARTRANSFORMER, they can invoke the tool using  
983 `GrammarTransformerRunner` on the command line and  
984 providing the paths to the input and output grammars  
985 there. Alternatively, instead of invoking GRAMMAR-  
986 TRANSFORMER via the command line and modifying  
987 `executeTransformation()`, it is also possible to use JUnit  
988 test cases to access the API and transform grammars in  
989 known locations. This is the approach we have followed in  
990 order to generate the results presented in this paper.

991 Figure 6 uses the first transformation operation from List-  
992 ing 8 removing curly braces as an example to depict how

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

1000 In **Step 1 (initialization)**, GRAMMARTRANSFORMER  
1001 builds a data structure out of the grammar initially gener-  
1002 ated by Xtext. That is, it builds a `:Grammar` object contain-  
1003 ing multiple `:GrammarRule` objects, with each of them con-  
1004 taining several `:LineEntry` objects in an ordered list. For  
1005 example, the `:Grammar` object contains a `:GrammarRule`  
1006 object with the name `"NodeStmt"`. This `:GrammarRule`  
1007 object contains seven `:LineEntry` objects, which represent  
1008 the seven lines of the grammar rule in Listing 4. Three of  
1009 these `:LineEntry` objects contain at least one curly brace  
1010 (" `{` " or " `}` "). These lines are explicitly repre-  
1011 sented in order to later map relevant transformation rules  
1012 to them. Figure 6 shows an excerpt of the object structure  
1013 created for the example with the three line objects for the  
1014 `NodeStmt` rule.

1015 In **Step 2 (per Transformation Rule)** each trans-  
1016 formation rule application is processed by executing the  
1017 `apply()`-method. For our example, the transformation rule  
1018 `removeBraces` is applied via the GRAMMARTRANSFORMER  
1019 API as configured in line 3 of Listing 8.

1020 In **Step 2a (localization of affected grammar rules  
1021 and lines)**, the grammar rule and lines that need to be  
1022 changed are located, based on the configuration of the  
1023 transformation rule application. In the case of our exam-  
1024 ple, the grammar rule `NodeStmt` (cf. line 1 in Listing 4) is  
1025 identified. Then, all lines of that grammar rule are iden-  
1026 tified that include a curly brace. For example, the lines  
1027 represented by `:LineEntry` objects as shown in Figure 6  
1028 are identified.

1029 In **Step 2b (change)**, the code uses regular expressions  
1030 for character-level matching and searching. If it finds curly

braces surrounded by single quotes (i.e., " `{` " and " 1031  
`}` "), it removes them. 1032

Finally, in **Step 3 (finalization)**, the GRAMMARTRANS- 1033  
FORMER writes the complete data structure containing the 1034  
transformed grammar rules to a new file by means of the 1035  
call `setFileText(...)`. 1036

After the execution of these steps, the transformed ver- 1037  
sions of the grammar is ready for use. The typical next step 1038  
is to re-generate the parser, textual editor and other arti- 1039  
facts for the grammar via Xtext. We recommend that the 1040  
language engineer should systematically test the resulting 1041  
grammar to check whether it matches their expectations, 1042  
based on the generated artifacts and a test suite with di- 1043  
verse language instances. After evolution steps, previously 1044  
developed tests can act as regression tests. 1045

## 6.6. Post-Processing vs. Changing Grammar Generation 1046

GRAMMARTRANSFORMER is designed to modify gram- 1047  
mars that Xtext generated out of meta-models. An alter- 1048  
native to this post-processing approach is to directly 1049  
modify the Xtext grammar generator as, e.g., in `XMLText` 1050  
(Neubauer et al., 2015, 2017). However, we deliberately 1051  
chose a post-processing approach, because the application 1052  
of conventional regular expressions enables the transfer- 1053  
ability to other recent language development frameworks 1054  
like Langium (TypeFox GmbH, 2022) or `textX` (Dejanović 1055  
et al., 2017), if they support the grammar generation from 1056  
a meta-model in a future point in time. While the trans- 1057  
formation rules implemented in grammar transformer are 1058  
currently tailored to the structure of Xtext grammars, 1059  
GRAMMARTRANSFORMER does not technically depend on 1060  
Xtext and the rules could easily be adapted to a different 1061  
grammar language. Furthermore, as the implementation 1062  
of an Xtext grammar generator necessarily depends on 1063  
many version-specific internal aspects of Xtext, the post- 1064  
processing approach using regular expressions is consider- 1065  
ably more maintainable. 1066

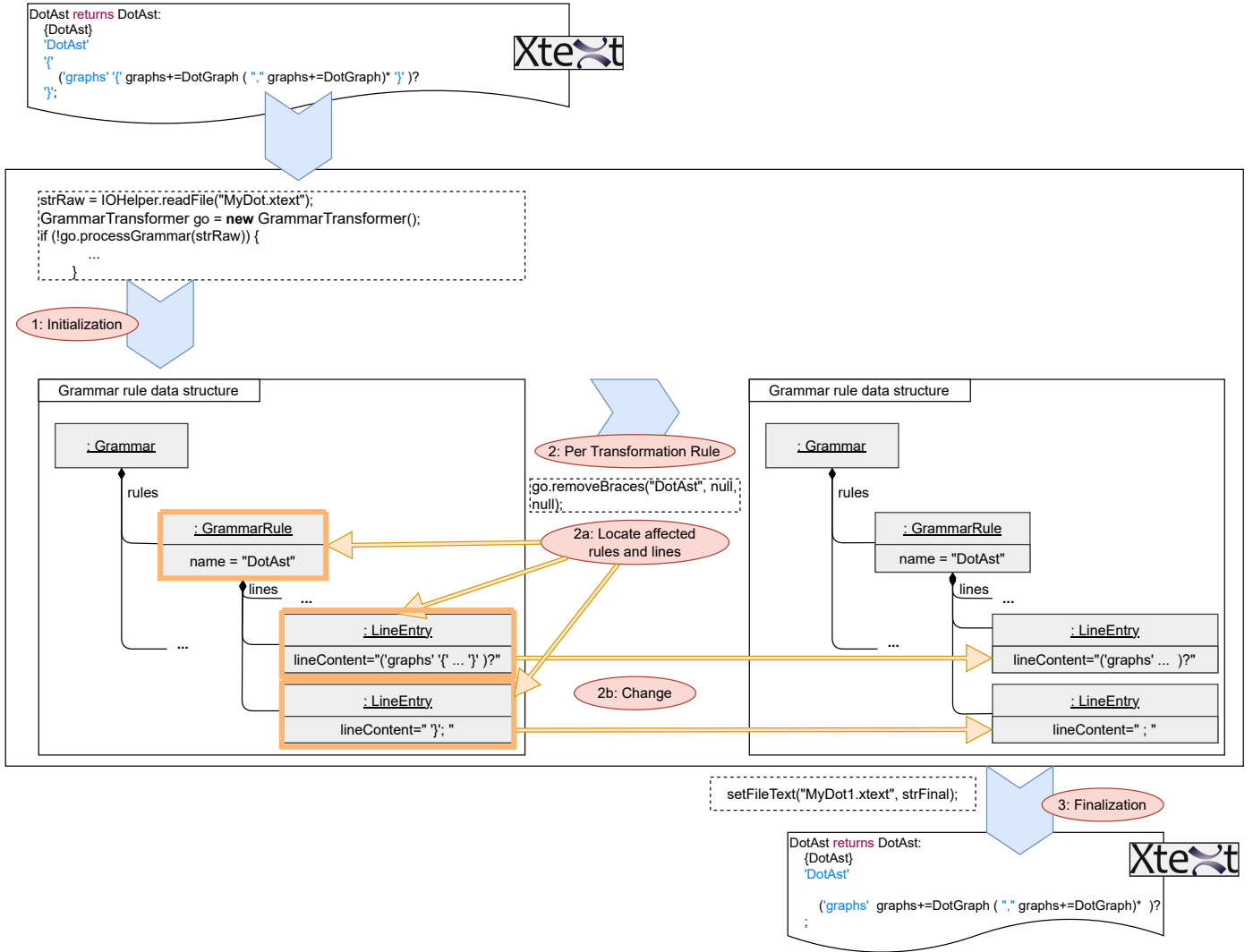


Figure 6: Exemplary Interplay of the Building Blocks of the GRAMMARTRANSFORMER

### 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 adaptations 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 back-

1093 wards comparability to previous grammar versions—in  
1094 general, after rule applications, instances of the previous,  
1095 un-transformed grammar can no longer be parsed. This  
1096 lack of backwards compatibility is generally desirable, as  
1097 the alternative would be support for a mixing of old and  
1098 new grammar elements (e.g., changed keywords and paran-  
1099 theses styles) in the same instance, which would generally  
1100 be confusing to the user, and lead to issues with parsing  
1101 and other tool support. However, to reduce manual effort  
1102 in cases where legacy grammar instances exist, automated  
1103 co-evolution of grammar instances after grammar changes  
1104 is generally possible and leads to a promising future work  
1105 direction (discussed in Section 8.4).

## 1106 7. Evaluation

1107 In this evaluation, we focus on two research questions:

- 1108 • *RQ1: Can our solution be used to adapt generated*  
1109 *grammars so that they produce the same language as*  
1110 *available expert-created grammars?*

1111 The goal of this question is to validate the claim that  
1112 our approach can automatically perform the changes  
1113 that an expert would need to do manually. To this  
1114 end, we consider languages for which an expert-created  
1115 grammar exists, and validate the capability of our  
1116 approach to re-create an equivalent grammar.

- 1117 • *RQ2: Can our solution support the co-evolution of*  
1118 *generated grammars when the meta-model evolves?*

1119 Our original motivation for the work was to enable  
1120 evolution and rapid prototyping for textual languages  
1121 built with a meta-model. The aim here is to evaluate  
1122 whether our approach is suitable for supporting these  
1123 evolution scenarios.

1124 In the following, we address both questions. Our supple-  
1125 mental material (Zhang et al., 2024) contains the source  
1126 code of the implementation as well as all experiments.

### 7.1. Grammar Adaptation (RQ1) 1127

To address the first question, we evaluate the GRAMMAR- 1128  
TRANSFORMER by transforming the generated grammars 1129  
of the seven DSLs, so that they parse the same syntax as 1130  
the expert-created grammars. 1131

#### 7.1.1. Cases 1132

Our goal is to evaluate whether the GRAMMARTRANS- 1133  
FORMER can be used to transform the generated grammars 1134  
so that their rules imitate the rules of the expert-created 1135  
grammars. We reused the meta-model adaptations and 1136  
generated grammars from Section 4.3. Furthermore, we 1137  
continued working with the versions of ATL and SML in 1138  
which parts of their languages were excluded as described 1139  
in Section 4.2. 1140

#### 7.1.2. Method 1141

For each DSL, we wrote a configuration for the final 1142  
version of GRAMMARTRANSFORMER which was the result 1143  
of the work described in Sections 4 to 6. The goal was 1144  
to transform the generated grammar so as to ‘imitate’ as 1145  
many grammar rules as possible from the expert-created 1146  
grammar of the DSL. Note that this was an iterative pro- 1147  
cess in which we incrementally added new transformation 1148  
rule applications to the GRAMMARTRANSFORMER’s con- 1149  
figuration, using the expert-created grammar as a ground 1150  
truth and using our notion of ‘imitation’ (cf. Section 4.4) 1151  
as the gold standard. Essentially, we updated the GRAM- 1152  
MARTRANSFORMER configuration and then ran the tool 1153  
before analysing the transformed grammar for imitation 1154  
of the original. We repeated the process and adjusted 1155  
the GRAMMARTRANSFORMER configuration until the test 1156  
grammar’s rules ‘imitated’ the expert-created grammar. 1157  
Note that in the case of *Spectra*, we did not reach that 1158  
point. We explain this in more detail in Section 7.1.4. For 1159  
all experiments, we used the set of 56 transformation rules 1160  
that were identified after the two iterations described in 1161  
Section 4 and as summarized in Section 5. 1162

1163 To verify whether the transformed grammar imitates the  
 1164 expert-created grammar, we adopted a manual verification  
 1165 method, in which we systematically compared the gram-  
 1166 mar rules in the transformed grammar with the grammar  
 1167 rules in the expert-created grammar. An expert-created  
 1168 grammar is imitated by an transformed grammar if every  
 1169 grammar rule in it is imitated by one (or several) grammar  
 1170 rules from the transformed grammar. The procedure and  
 1171 results of this step are documented in our supplementary  
 1172 materials (Zhang et al., 2024).<sup>4</sup>

### 1173 7.1.3. Metrics

1174 To evaluate the transformation results of the GRAMMAR-  
 1175 TRANSFORMER on the case DSLs, we assessed the following  
 1176 metrics.

1177 **#*GORA*** Number of GRAMMARTransformer rule ap-  
 1178 plications used for the configuration.

1179 **Grammar rules** The changes in grammar rules per-  
 1180 formed by the GRAMMARTransformer when adapt-  
 1181 ing the generated grammar towards the expert-created  
 1182 grammar. We measure these changes in terms of

- 1183 • mod: Number of modified grammar rules
- 1184 • add: Number of added grammar rules
- 1185 • del: Number of deleted grammar rules

1186 **Grammar lines** The changes in the lines of the gram-  
 1187 mar performed by the GRAMMARTransformer when  
 1188 adapting the generated grammar towards the expert-  
 1189 created grammar. We measure these changes in terms  
 1190 of

- 1191 • mod: Number of modified lines
- 1192 • add: Number of added lines
- 1193 • del: Number of deleted lines

1194 **Transformed grammar** Metrics about the resulting  
 1195 transformed grammar. We assess

- lines: Number of overall lines 1196
- rules: Number of grammar rules 1197
- calls: Number of calls between grammar rules 1198

**#*iGR*** Number of grammar rules in the expert-created 1199  
 grammar that were successfully *imitated* by the trans- 1200  
 formed grammar. 1201

**#*niGR*** Number of grammar rules in the expert-created 1202  
 grammar that were not *imitated* by the transformed 1203  
 grammar. 1204

### 1205 7.1.4. Results

1206 Table 4 shows the results of applying the GRAMMAR-  
 1207 TRANSFORMER to the seven DSLs. See Table 1 for the  
 1208 corresponding metrics of the initially generated grammars.

1209 *Imitation.* For all case DSLs in the first two iterations  
 1210 except *Spectra*, we were able to achieve a complete adap-  
 1211 tation, i.e., we were able to modify the grammar by using  
 1212 GRAMMARTransformer so that the grammar rules of  
 1213 the transformed grammar *imitate* all grammar rules of the  
 1214 expert-created grammar.

1215 *Limitation regarding Spectra.* For one of the languages,  
 1216 Spectra, we were able to come very close to the expert-  
 1217 created grammar. Many grammar rules of Spectra could  
 1218 be nearly imitated. However, we did not implement all  
 1219 grammar rules that would have been necessary to allow  
 1220 the full transformation of Spectra. Listing 9 shows the  
 1221 grammar rule `TemporalPrimaryExpr` in Spectra’s gener-  
 1222 ated grammar, while Listing 10 shows what that grammar  
 1223 rule looks like in the expert-created grammar. In order to  
 1224 transform the grammar rule `TemporalPrimaryExpr` from  
 1225 Listing 9 to Listing 10, we need to configure the GRAMMAR-  
 1226 TRANSFORMER so that it combines the attribute `pointer`  
 1227 and `operator` multiple times, and the default value of the  
 1228 attribute `operator` is different each time. The language en-  
 1229 gineers using the GRAMMARTransformer need to input

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

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

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

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

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

```

1 TemporalPrimaryExpr returns
   TemporalPrimaryExpr:
2 {TemporalPrimaryExpr}
3 'TemporalPrimaryExpr'
4 '{'
5 ('operator' operator=EString)?
6 ('predPatt' predPatt=[
   PredicateOrPatternReferrable|EString])?
7 ('pointer' pointer=[Referrable|EString])?
8 ('regexpPointer' regexpPointer=[
   DefineRegExpDecl|EString])?
9 ('predPattParams' '{' predPattParams+=
   TemporalExpression ( "," predPattParams
   +=TemporalExpression)* '}' )?
10 ('tpe' tpe=TemporalExpression)?
11 ('index' '{' index+=TemporalExpression ( ","
   index+=TemporalExpression)* '}' )?
12 ('temporalExpression' temporalExpression=
   TemporalExpression)?
13 ('regexp' regexp=RegExp)?
14 '}' ;

```

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

```

1 TemporalPrimaryExpr returns
   TemporalExpression:
2 Constant | '(' QuantifierExpr ')' | {
   TemporalPrimaryExpr}
3 (predPatt=[PredicateOrPatternReferrable]
4 ('(' predPattParams+=TemporalInExpr ( ','
   predPattParams+=TemporalInExpr)* ')' |
   (')') |
5 operator=('-'|'!' ) tpe=TemporalPrimaryExpr |
6 pointer=[Referrable] ('[' index+=
   TemporalInExpr ']' )* |
7 operator='next' '(' temporalExpression=
   TemporalInExpr ')' |
8 operator='regexp' '(' (regexp=RegExp |
   regexpPointer=[DefineRegExpDecl] ) ')' |
9 pointer=[Referrable] operator='.all' |
10 pointer=[Referrable] operator='.any' |
11 pointer=[Referrable] operator='.prod' |
12 pointer=[Referrable] operator='.sum' |
13 pointer=[Referrable] operator='.min' |
14 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,

1246 added, and deleted lines is also an underestimation of the  
1247 amount of necessary changes, as many lines will be changed  
1248 in multiple ways, e.g., by changing keywords and braces in  
1249 the same line. This explains why for some languages the  
1250 number of transformation rule applications is bigger than  
1251 the number of changed lines (e.g., for SML we specified  
1252 421 transformation rule applications which changed, added,  
1253 and deleted together 287 lines in the grammar).

1254 *Effort for the Language Engineer.* We acknowledge that  
1255 the number of transformation rule applications that are  
1256 necessary to adapt a generated grammar to imitate the  
1257 expert-created grammar indicates that it is more effort  
1258 to configure GRAMMARTRANSFORMER than to apply the  
1259 desired change in the grammar manually once. However,  
1260 even with that assumption, we argue that the effort of  
1261 configuring GRAMMARTRANSFORMER is in the same order  
1262 of magnitude as the effort of applying the changes manually  
1263 to the grammar.

1264 Furthermore, we argue that it is more efficient to con-  
1265 figure GRAMMARTRANSFORMER once than to manually  
1266 rewrite grammar rules every time the language changes –  
1267 under the assumption that the configuration can be reused  
1268 for new versions of the grammar. In that case, the effort  
1269 invested in configuring GRAMMARTRANSFORMER would  
1270 quickly pay off when a language is going through changes,  
1271 e.g., while rapidly prototyping modifications or when the  
1272 language is evolving. In the next section (Section 7.2), we  
1273 evaluate this assumption.

1274 In terms of reusability of the configurable transforma-  
1275 tion rules, we observe that most of the languages we cover  
1276 require at least one *unique* transformation rule that is not  
1277 needed by any other language. This applies to DOT, Bib-  
1278 TeX, ATL with one unique transformation rule, each. Spec-  
1279 tra was our most complicated case with six unique rules,  
1280 whereas Xcore requires four and SML requires five unique  
1281 rules. This indicates that using GRAMMARTRANSFORMER  
1282 for a new language might require effort by implementing

a few new transformation rules. However, we argue that 1283  
this effort will be reduced as more transformation rules are 1284  
added to GRAMMARTRANSFORMER and that, in particular 1285  
for evolving languages, the small investment to create a 1286  
new transformation rule will pay off quickly. 1287

## 1288 7.2. Supporting Evolution (RQ2)

To address the second question, we evaluate the GRAM- 1289  
MARTRANSFORMER on two languages’ evolution histories: 1290  
The industrial case of EAST-ADL and the evolution of the 1291  
DSL QVTo. We focus on the question to what degree a 1292  
configuration of the GRAMMARTRANSFORMER that was 1293  
made for one language version can be applied to a new 1294  
version of the language. 1295

### 1296 7.2.1. Cases

The two cases we are using to evaluate how GRAMMAR- 1297  
TRANSFORMER supports the evolution of a DSL are a 1298  
textual variant of EAST-ADL (EAST-ADL Association, 1299  
2021) and QVT Operational (QVTo) (Object Management 1300  
Group, 2016). 1301

*EAST-ADL.* EAST-ADL is an architecture description 1302  
language used in the automotive domain (EAST-ADL As- 1303  
sociation, 2021). Together with an industrial language 1304  
engineer for EAST-ADL, we are currently developing a 1305  
textual notation for version 2.2 of the language (Holtmann 1306  
et al., 2023). We started this work with a simplified version 1307  
of the meta-model to limit the complexity of the resulting 1308  
grammar. In a later step, we switched to the full meta- 1309  
model. We treat this switch as an evolution step here. The 1310  
meta-model of EAST-ADL is taken from the EATOP repos- 1311  
itory (EAST-ADL Association, 2022). The meta-model of 1312  
the simplified version contains 91 classes and enumerations, 1313  
and the meta-model of the full version contains 291 classes 1314  
and enumerations. 1315

*QVTo.* QVTo is one of the languages in the OMG QVT 1316  
standard (Object Management Group, 2016). We use the 1317

1318 original meta-models available in Ecore format on the OMG  
1319 website (Object Management Group, 2016). The baseline  
1320 version is QVTo 1.0 (Object Management Group, 2008)  
1321 and we simulate evolution to version 1.1 (Object Man-  
1322 agement Group, 2011), 1.2 (Object Management Group,  
1323 2015) and 1.3 (Object Management Group, 2016). Our  
1324 original intention was to use the Eclipse reference imple-  
1325 mentation of QVTo (Eclipse Foundation, 2022b), but due  
1326 to the differences in abstract syntax and concrete syntax  
1327 (see Section 2), we chose to use the official meta-models  
1328 instead. We analyzed four versions of QVTo’s OMG offi-  
1329 cial Ecore meta-model. There are 50 differences between  
1330 the meta-models of version 1.0 and 1.1, 29 of which are  
1331 parts that do not contain OCL (as for ATL as described  
1332 in Section 4.2, we exclude OCL in our solution for QVTo).  
1333 These 29 differences include different types, for example, 1)  
1334 the same set of attributes has different arrangement orders  
1335 in the same class in different versions of the meta-model;  
1336 2) the same class has different superclasses in different  
1337 versions; 3) the same attribute has different multiplicities  
1338 in different versions, etc. There are 3 differences between  
1339 versions 1.1 and 1.2, all of which are from the OCL part.  
1340 There is only one difference between versions 1.2 and 1.3,  
1341 and it is about the same attribute having a different lower  
1342 bound for the multiplicity in the same class in the two  
1343 versions. Altogether we observed 54 meta-model differ-  
1344 ences in QVTo between the different versions (cf. the file  
1345 “Comparison of QVTo metamodel versions” in the folder  
1346 “Section\_7\_Evaluation/Subsection\_7.2\_Support” lists all  
1347 the metamodel differences).

1348 The OMG website provides an EBNF grammar for each  
1349 version of QVTo, which is the basis for our imitations of  
1350 the QVTo languages. Among them, versions 1.0, 1.1, and  
1351 1.2 share the same EBNF grammar for the QVTo part  
1352 except for the OCL parts, despite the differences in the  
1353 meta-model. The EBNF grammar of QVTo in version 1.3  
1354 is different from the other three versions.

### 7.2.2. Preparation of the QVTo Case 1355

1356 In contrast to the EAST-ADL case, we needed to perform  
1357 some preparations of the grammar and the meta-model to  
1358 study the QVTo case. All adaptations were done the same  
1359 way on all versions of QVTo.

1360 *Exclusion of OCL.* As described in detail in Section 4.2,  
1361 we excluded the embedded OCL language part from QVTo.  
1362 For the meta-model, we introduced a dummy class for  
1363 OCL, changed all calls to OCL types into calls to that  
1364 dummy class, and removed the OCL metaclasses from the  
1365 meta-model.

1366 As described in Section 4.2, excluding a language part  
1367 such as the embedded OCL from the scope of the investi-  
1368 gation also implies that we need to exclude this language  
1369 part when it comes to judging whether a grammar is imi-  
1370 tated. Therefore, we substituted all grammar rules from  
1371 the excluded OCL part with a placeholder grammar rule  
1372 called `ExpressionG0` where an OCL grammar rule would  
1373 have been called. This change allows us to compare the  
1374 expert-created grammar of the different QVTo versions to  
1375 the transformed grammar versions.

1376 *QVTo Meta-model Adaptations.* We found that some non-  
1377 terminals of QVTo’s EBNF grammar are missing in the  
1378 QVTo meta-model provided by OMG. For example, there  
1379 is a non-terminal `<top_level1>` in the EBNF grammar, but  
1380 there is no counterpart for it in the meta-model. Therefore,  
1381 we need to adapt the meta-model to ensure that it contains  
1382 all the non-terminals in the EBNF grammar. To ensure  
1383 that the adaptation of the meta-model is done systemat-  
1384 ically, we defined seven general adaptation rules that we  
1385 followed when adapting the meta-models of the different  
1386 versions. We list these adaptation rules in the supplemental  
1387 material (Zhang et al., 2024).

1388 As a result, we added 62 classes and enumerations with  
1389 their corresponding references to each version of the meta-  
1390 model. Note that this number is high compared to the  
1391 original number of classes in the meta-model (24 classes).



1392 This massive change was necessary, because the available  
1393 Ecore meta-models were too abstract to cover all elements  
1394 of the language. The original meta-model did contain most  
1395 key concepts, but would not allow to actually specify a  
1396 complete QVTo transformation. For example, with the  
1397 original meta-model, it was not possible to represent the  
1398 scope of a mapping or helper.

1399 These changes enable us to imitate the QVTo gram-  
1400 mar. However, they do not bias the results concerning  
1401 the effects of the observed meta-model evolution as, with  
1402 exception of a single case, these evolutionary differences  
1403 are neither erased nor increased by the changes we per-  
1404 formed to the meta-model. The exception is a meta-model  
1405 evolution change between version 1.0 and 1.1 where the  
1406 class `MappingOperation` has super types `Operation` and  
1407 `NamedElement`, while the same class in V1.1 does not. The  
1408 meta-model change performed by us removes the superclass  
1409 `Operation` from `MappingOperation` in version 1.0. We did  
1410 this change to prevent conflicts as the attribute `name` would  
1411 have been inherited multiple times by `MappingOperation`.  
1412 This in turn would cause problems in the generation pro-  
1413 cess. Thus, only two of the 54 meta-model evolutionary  
1414 differences could not be studied. The differences and their  
1415 analysis can be found in the supplemental material (Zhang  
1416 et al., 2024).

### 1417 7.2.3. Method

1418 To evaluate how GRAMMARTRANSFORMER supports the  
1419 evolution of meta-models we look at the effort that is  
1420 required to update the transformation rule applications  
1421 after an update of the meta-models of EAST-ADL and  
1422 QVTo.

1423 *Baseline GRAMMARTRANSFORMER Configuration.* First,  
1424 we generated the grammar for the initial version of a lan-  
1425 guage’s meta-model (i.e., the simple version for EAST-ADL  
1426 and version 1.0 for QVTo). Then we defined the configu-  
1427 ration of transformation rule applications that allows the  
1428 GRAMMARTRANSFORMER to modify the generated gram-

1429 mar so that its grammar rules *imitate* the expert-created  
1430 grammar for each case. Doing so confirmed the obser-  
1431 vation from the first part of the evaluation that a new  
1432 language of sufficient complexity requires at least some new  
1433 transformation rules (see Section 7.1.4). Consequently, we  
1434 identified the need for four additional transformation rules  
1435 for QVTo, which we implemented accordingly as part of  
1436 the GRAMMARTRANSFORMER (this is also summarized in  
1437 Section 5 in Table 2). This step provided us with a baseline  
1438 configuration for the GRAMMARTRANSFORMER.

*Evolution.* For the following language versions, i.e., the full  
1439 version of EAST-ADL and QVTo 1.1, we then generated  
1440 the grammar from the corresponding version of the meta-  
1441 model and applied the GRAMMARTRANSFORMER with the  
1442 configuration of the previous version (i.e., simple EAST-  
1443 ADL and QVTo 1.0). We then identified whether this  
1444 was already sufficient to *imitate* the language’s grammar  
1445 or whether changes and additions to the transformation  
1446 rule applications were required. We continued adjusting  
1447 the transformation rule applications accordingly to gain a  
1448 GRAMMARTRANSFORMER configuration valid for the new  
1449 version (full EAST-ADL and QVTo 1.1, respectively). For  
1450 QVTo, we repeated that process two more times: For QVTo  
1451 1.2, we took the configuration of QVTo 1.1 as a baseline,  
1452 and for QVTo 1.3, we took the configuration of QVTo 1.2  
1453 as a baseline.  
1454

### 1455 7.2.4. Metrics

1456 We documented the metrics used in Section 7.1.3 for  
1457 EAST-ADL and QVTo in their different versions. In addi-  
1458 tion, we also documented the following metric:

**#cORA** The number of changed, added, and deleted  
1459 transformation rule applications compared to the pre-  
1460 vious language version.  
1461

### 1462 7.2.5. Results

1463 Table 5 shows the results of the evolution cases.

1464 *EAST-ADL*. Compared with the simplified version of  
1465 *EAST-ADL*, the full version is much larger. It contains  
1466 291 metaclasses, i.e., 200 metaclasses more than the simple  
1467 version of *EAST-ADL*, which leads to a generated grammar  
1468 with 291 grammar rules and 2,839 non-blank lines in the  
1469 generated grammar file (cf. Table 5).

1470 The 22 transformation rule applications for the simple  
1471 version of *EAST-ADL* already change the grammar sig-  
1472 nificantly, causing modifications of all 91 grammar rules  
1473 and changes in nearly every line of the grammar. This  
1474 also illustrates how massive the changes to the generated  
1475 grammar are to reach the desired grammar. The number of  
1476 changes is even larger with the full version of *EAST-ADL*.

1477 We only needed to change and add a total of 10 grammar  
1478 transformation rule applications to complete the transfor-  
1479 mation of the grammar of full *EAST-ADL*. For example,  
1480 we excluded the primary type `String0` from the full ver-  
1481 sion of the *EAST-ADL* grammar, which led us to add  
1482 a line of configuration `go.removeRule(String0)`. While  
1483 this is increasing the `GRAMMARTransformer` configura-  
1484 tion from the simple *EAST-ADL* version quite a bit (from  
1485 22 transformation rule applications to 31 transformation  
1486 rule applications), the increase is fairly small given that  
1487 the meta-model increased massively (with 200 additional  
1488 metaclasses).

1489 The reason is that our grammar transformation require-  
1490 ments for the simplified version and the full version of  
1491 *EAST-ADL* are almost the same. This transformation  
1492 requirement is mainly based on the look and feel of the  
1493 language and is provided by an industrial partner. These  
1494 transformation rule applications have been configured for  
1495 the simplified version. When we applied them to the gener-  
1496 ated grammar of the full version of *EAST-ADL*, we found  
1497 that we can reuse all of these transformation rule applica-  
1498 tions. Furthermore, we benefit from the fact that many  
1499 transformation rule applications are formulated for the  
1500 scope of the whole grammar and thus can also influence  
1501 grammar rules added during the evolution step. We do not

list a number of grammar rules in a expert-created gram- 1502  
mar of *EAST-ADL* in Table 5, because there is no “original” 1503  
text grammar of *EAST-ADL*. Instead, we transform the 1504  
generated grammar of *EAST-ADL* according to our in- 1505  
dustrial partner’s requirements for *EAST-ADL*’s textual 1506  
concrete syntax. 1507

*QVTo*. The baseline configuration of the `GRAMMARTransformer` 1508  
for *QVTo* includes 733 transformation rule 1509  
applications, which is a lot given that the expert-created 1510  
grammar of *QVTo* 1.0 has 115 non-terminals. Note that the 1511  
transformed grammar has even fewer grammar rules (77) as 1512  
some of the rules in the transformed grammar *imitate* mul- 1513  
tiple rules from the expert-created grammar at once. This 1514  
again is a testament to how different the expert-created 1515  
grammar is from the generated one (over 228 lines in the 1516  
grammar are modified, 2 lines are added, and 580 lines are 1517  
deleted by these 733 transformation rule applications). 1518

However, if we look at the evolution towards versions 1519  
1.1, 1.2, and 1.3 we witness that very few changes to the 1520  
`GRAMMARTransformer` configuration are required. In 1521  
fact, only between 0 and 2 out of the 733 transformation 1522  
rule applications needed adjustments. This significantly 1523  
reduces the effort required compared to manually modifying 1524  
a grammar generated from a new version of the *QVTo* 1525  
metamodel, which would require modifying hundreds of 1526  
lines. The reason is that, even though there are many 1527  
differences between different versions of the *QVTo* meta- 1528  
model, there are only 0 to 2 differences that affect the 1529  
transformation rule applications. 1530

For example, version 1.0 of the *QVTo* meta-model has an 1531  
attribute called `bindParameter` in the class `VarParameter`, 1532  
whereas it is called `representedParameter` in version 1.1. 1533  
This attribute is not needed according to the expert-created 1534  
grammars, so the `GRAMMARTransformer` configuration 1535  
includes a call to the transformation rule `RemoveAttribute` 1536  
to remove the grammar line that was generated based on 1537  
that attribute. The second parameter of the transforma- 1538

1539 tion rule *RemoveAttribute* needs to specify the name of  
1540 the attribute. As a consequence of the evolution, we had  
1541 to change that name in the transformation rule applica-  
1542 tion. Another example concerns the class `TypeDef`, which  
1543 contains an attribute `typedef_condition` in version 1.2 of  
1544 the QVTo meta-model. We added square brackets to it by  
1545 applying the transformation rule *AddSquareBracketsToAttr*  
1546 in the grammar transformation. However, in version 1.3 of  
1547 the QVTo meta-model, the class `TypeDef` does not contain  
1548 such an attribute, so the transformation rule application  
1549 *AddSquareBracketsToAttr* was unnecessary.

1550 Most of the differences between different versions of the  
1551 meta-model do not lead to changes in the transformation  
1552 rule applications. For example, the multiplicity of the  
1553 attribute `when` in the class `MappingOperation` is different  
1554 in version 1.0 and 1.1. We used *RemoveAttribute* to remove  
1555 the attribute during the transformation of grammar version  
1556 1.0. The same command can still be used in version 1.1,  
1557 as the removal operation does not need to consider the  
1558 multiplicity of an attribute. Therefore, this difference  
1559 does not affect the configuration of transformation rule  
1560 applications.

## 1561 8. Discussion

1562 In the following, we discuss the threats to validity of  
1563 the evaluation, different aspects of the GRAMMARTRANS-  
1564 FORMER, and future work implied by the current limita-  
1565 tions.

### 1566 8.1. Threats to Validity

1567 The threats to validity structured according to the taxon-  
1568 omy of Runeson et al. (Runeson and Höst, 2008; Runeson  
1569 et al., 2012) are as follows.

#### 1570 8.1.1. Construct Validity

1571 We limited our analysis to languages for which we could  
1572 find meta-models in the Ecore format. Some of these meta-  
1573 models were not “official”, in the sense that they had been

1574 reconstructed from a language in order to include them  
1575 in one of the “zoos”. An example of that is the meta-  
1576 model for BibTeX we used in our study. In the case of the  
1577 DOT language, we reconstructed the meta-model from an  
1578 Xtext grammar we found online. We adopted a reverse-  
1579 engineering strategy where we generated the meta-model  
1580 from the expert-created grammar and then generated a  
1581 new grammar out of this meta-model. This poses a threat  
1582 to validity since many of the languages we looked at can  
1583 be considered “artificial” in the sense that they were not  
1584 developed based on meta-models. However, we do not  
1585 think this affects the construct validity of our analysis  
1586 since our purpose is to analyze what changes need to be  
1587 made from an Xtext grammar file that has been generated.  
1588 In addition, we address this threat to validity by also  
1589 including a number of languages (e.g., Xenia and Xcore)  
1590 that are based on meta-models and using the meta-models  
1591 provided by the developers of the language.

1592 Furthermore, we had to make some changes to some of  
1593 the meta-models to be able to generate Xtext grammars  
1594 out of them at all (cf. Section 4.3) or to introduce cer-  
1595 tain language constructs required by the textual concrete  
1596 syntax (cf. Section 7.2.2). These meta-model adaptations  
1597 might have introduced biased changes and thereby impose  
1598 a threat to construct validity. However, we reduced these  
1599 adaptations to a minimum as far as possible to mitigate  
1600 this threat and documented all of them in our supplemental  
1601 material (Zhang et al., 2024) to ensure their reproducibility.

#### 1602 8.1.2. Internal Validity

1603 In the evaluation (cf. Section 7), we set up and quantita-  
1604 tively evaluate size and complexity metrics regarding the  
1605 considered meta-models and grammars as well as regard-  
1606 ing the GRAMMARTRANSFORMER configurations for the  
1607 use cases of one-time grammar adaptations and language  
1608 evolution. Based on that, we conclude and argue in Sec-  
1609 tions 7.1.4 and 8.2 about the effort required for creating and  
1610 evolving languages as well as the effort to create and re-use

Table 5: Result of supporting evolution (RQ2)

DSL	Meta-m.	Generated grammar			Transformed grammar			Grammar rules			Lines in Grammar			#GORA	#cORA
	Classes <sup>1</sup>	lines	rules	calls	lines	rules	calls <sup>2</sup>	mod	add	del	mod	add	del		
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

<sup>1</sup> The number is after adaptation, and it contains both classes and enumerations.

<sup>2</sup> The number includes the calls to dummy OCL and dummy SML expressions.

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

1663 described in Section 6. This procedure imposes multiple  
1664 threats to reliability. For example, analyzing a different  
1665 set of languages could have led to a different set of iden-  
1666 tified transformation rules, which then would have led to  
1667 a different implementation. Furthermore, analyzing the  
1668 languages in a different order or as part of different itera-  
1669 tions could have led to a different abstraction level of the  
1670 rules and thereby a different number of rule. Finally, the  
1671 design decisions that we made during the identification  
1672 of the conceptual transformation rules and during their  
1673 implementation could also have led to different kinds of  
1674 rules or of the implementation. However, we discussed all  
1675 of these aspects repeatedly amongst all authors to miti-  
1676 gate this threat and documented the results as part of our  
1677 supplemental material (Zhang et al., 2024) to ensure their  
1678 reproducibility.

## 1679 8.2. The Effort of Creating and Evolving a Language with 1680 the GRAMMARTransformer

1681 The results of our evaluation show three things. First,  
1682 the expert-created grammars of all studied languages differ  
1683 greatly in appearance from the generated grammars. Thus,  
1684 in most cases, creating a DSL with Xtext will require the  
1685 language engineer to perform big changes to the generated  
1686 grammar. Second, in the case of complex changes, manu-  
1687 ally writing a GRAMMARTransformer configuration can  
1688 lead to considerably less effort for the language engineer  
1689 compared to manually adapting the grammar. Third, there  
1690 seems to be a large potential for the reuse of GRAMMAR-  
1691 TRANSFORMER configurations between different versions  
1692 of a language, thus supporting the evolution of textual  
1693 languages.

1694 These observations can be combined with the experience  
1695 that most languages evolve with time and that especially  
1696 DSLs go through a rapid prototyping phase at the be-  
1697 ginning where language versions are built for practical  
1698 evaluation (Wang and Gupta, 2005). Therefore, we con-  
1699 clude that the GRAMMARTransformer has big potential

to save manual effort when it comes to developing DSLs. 1700

1701 Additionally, a topic worth mentioning is how the in-  
1702 volvement of different people and their skill sets affect  
1703 the effort when creating and reusing transformation rule  
1704 configurations. For example, in case that updates to an  
1705 existing configuration are needed after an evolution step,  
1706 the maintainers need to understand the transformation rule  
1707 configuration of the previous version, which could take a  
1708 new contributor more time than the original contributor.  
1709 Assessing the impact of this aspect is a subject for future  
1710 work.

## 1711 8.3. Implications for Practitioners and Researchers

1712 Our results have several implications for language engi-  
1713 neers and researchers.

1714 *Blended Modeling.* Ciccozzi et al. (Ciccozzi et al., 2019)  
1715 coin the term *blended modeling* for the activity of interact-  
1716 ing with one model through multiple notations (e.g., both  
1717 textual and graphical notations), which would increase the  
1718 usability and flexibility for different kinds of model stake-  
1719 holders. However, enabling blended modeling shifts more  
1720 effort to language engineers. This is due to the fact that the  
1721 realization of the different editors for the different notations  
1722 requires many manual steps when using conventional mod-  
1723 eling frameworks. In this context, Ciccozzi and colleagues  
1724 particularly stress the issue of the manual customization of  
1725 grammars in the case of meta-model evolution. Thus, as  
1726 one research direction to enable blended modeling, Ciccozzi  
1727 et al. formulate the need to automatically generate the dif-  
1728 ferent editors from a given meta-model. Our work serves as  
1729 one building block toward realizing this research direction  
1730 and opens up the possibility to develop and evolve blended  
1731 modeling languages that include textual versions.

1732 A relevant question is to which extent our approach  
1733 enables cost savings in a larger context, as the cost for  
1734 evolving the existing tools and applications working with  
1735 existing languages might be higher than the cost for evol-  
1736 ving the languages themselves. We benefit from the exten-

1737 sive tool support offered by Xtext, which can automatically  
1738 re-generate large parts of the available textual editor af-  
1739 ter changes of the underlying grammar, including features  
1740 such as, e.g., auto-formatting, auto-completion, and syntax  
1741 highlighting. In consequence, by supporting automated  
1742 grammar changes (in particular, after evolution steps), we  
1743 also save effort for the overall adaptation of the textual  
1744 editor. However, in MDE contexts, other applications  
1745 and tools typically refer to the metamodel, instead of the  
1746 grammar, and hence, are outside our scope.

1747 *Prevention of Language Flaws.* Willink (Willink, 2020)  
1748 reflects on the version history of the Object Constraint  
1749 Language (OCL) and the flaws that were introduced dur-  
1750 ing the development of the different OCL 2.x specifications  
1751 by the Object Management Group (Object Management  
1752 Group (OMG), 2014). Particularly, he points out that the  
1753 lack of a parser for the proposed grammar led to several  
1754 grammar inaccuracies and thereby to ambiguities in the  
1755 concrete textual syntax. This in turn led to the fact that  
1756 the concrete syntax and the abstract syntax in the Eclipse  
1757 OCL implementation (Eclipse Foundation, 2022a) are so  
1758 divergent that two distinct meta-models with a dedicated  
1759 transformation between both are required, which also holds  
1760 for the QVTo specification and its Eclipse implementation  
1761 (Willink, 2020) (cf. Section 2). The GRAMMARTRANS-  
1762 FORMER will help to prevent and bridge such flaws in  
1763 language engineering in the future. Xtext already enables  
1764 the generation of the complete infrastructure for a textual  
1765 concrete syntax from an abstract syntax represented by a  
1766 meta-model. Our approach adds the ability to transform  
1767 the grammar (i.e., the concrete syntax), as we show in  
1768 the evaluation by deriving an applicable parser with an  
1769 transformed grammar from the QVTo specification meta-  
1770 models.

#### 1771 8.4. Future Work

1772 The GRAMMARTRANSFORMER is a first step in the di-  
1773 rection of supporting the evolution of textual grammars for

1774 DSLs. However, there are, of course, still open questions  
1775 and challenges that we discuss in the following.

*Name Changes to Meta-model Elements.* In the GRAM-  
1776 MARTRANSFORMER configurations, we currently reference  
1777 the grammar concepts derived from the meta-model classes  
1778 and attributes by means of the class and attribute names  
1779 (cf. Listing 8). Thus, if a meta-model evolution involves  
1780 many name changes, likewise many changes to transforma-  
1781 tion rule applications are required. Consequently, we plan  
1782 as future work to improve the GRAMMARTRANSFORMER  
1783 with a more flexible concept, in which we more closely  
1784 align the grammar transformation rule applications with  
1785 the meta-model based on name-independent references.  
1786

*More Efficient Rules and Libraries.* We think that there is  
1787 a lot of potential to make the available set of transforma-  
1788 tion rules more efficient. This could for example be done by  
1789 providing libraries of more complex, recurring changes that  
1790 can be reused. Such a library can contain a default set of  
1791 transformation rule configurations to make the generated  
1792 grammar follow a particular style (e.g., mimicking an exist-  
1793 ing language, to be appealing for users of that language).  
1794 Language engineers can use it as a basis and with mini-  
1795 mal effort define transformation rule configurations that  
1796 perform DSL-specific changes. Such a change might make  
1797 the application of the GRAMMARTRANSFORMER attractive  
1798 even in those cases where no evolution of the language is  
1799 expected. While this use-case still requires effort for defin-  
1800 ing configurations, the overall effort compared to manual  
1801 editing can be reduced especially in cases with applicable  
1802 large-scoped rules that, e.g., globally change the parenthesis  
1803 style in the grammar.  
1804

1805 In addition, the API of GRAMMARTRANSFORMER could  
1806 be changed to a fluent version where the transformation  
1807 rule application is configured via method calls before they  
1808 are executed instead of using the current API that contains  
1809 many `null` parameters. This could also lead to a reduction  
1810 of the number of grammar transformation rule applications

1811 that need to be executed since some executions could be  
1812 performed at the same time.

1813 Another interesting idea would be to use artificial in-  
1814 telligence to learn existing examples of grammar transfor-  
1815 mations in existing languages to provide transformation  
1816 suggestions for new languages and even automatically cre-  
1817 ate configurations for the GRAMMARTRANSFORMER.

1818 *Expression Languages.* In this paper, we excluded the ex-  
1819 pression language parts (e.g., OCL) of two of the exam-  
1820 ple languages (cf. Section 4.2). However, expression lan-  
1821 guages define low-level concepts and have different kinds of  
1822 grammars and underlying meta-models than conventional  
1823 languages. In future work, we want to further explore  
1824 expression languages specifically, in order to ensure that  
1825 the GRAMMARTRANSFORMER can be used for these types  
1826 of syntaxes as well.

1827 *Visualization of Configuration.* Currently, we configure the  
1828 GRAMMARTRANSFORMER by calling the methods of trans-  
1829 formation rules, which is a code-based way of working. In  
1830 the future, we intend to improve the tooling for GRAM-  
1831 MARTRANSFORMER and embed the current library into  
1832 a more sophisticated workbench that allows the language  
1833 engineer to select and parameterize transformation rule  
1834 applications either using a DSL or a graphical user interface  
1835 and provides previews of the modified grammar as well as  
1836 a view of what valid instances of the language look like.

1837 *Co-evolving Model Instances.* We also intend to couple  
1838 GRAMMARTRANSFORMER with an approach for language  
1839 evolution that also addresses the model instances. In prin-  
1840 ciple, a model instance represented by a textual grammar  
1841 instance can be read using the old grammar and parsed  
1842 into an instance of the old meta-model. It can then be  
1843 transformed, e.g., using QVTo to conform to the new meta-  
1844 model, and then be serialized again using the new grammar.  
1845 However, following this approach means that formatting  
1846 and comments can be lost. Instead, we intend to derive a

1847 textual transformation from the differences in the gram-  
1848 mers and the transformation rule applications that can be  
1849 applied to the model instances and maintain formatting  
1850 and comments as much as possible.

*Alternative implementation strategy.* Our implementation  
1851 strategy relies on the format of textual grammars produced  
1852 by Xtext, which is stable across recent versions of Xtext.  
1853 This implementation strategy was suitable for positively  
1854 answering our evaluation questions and thus, substantiating  
1855 the scientific contribution of our paper. An alternative,  
1856 arguably more elegant implementation strategy would be  
1857 to use Xtext’s abstract syntax tree representation of the  
1858 grammar. A benefit of such an implementation would  
1859 be that it would be more robust in case that the output  
1860 format of Xtext changes, rendering it a desirable direction  
1861 for future work.  
1862

## 1863 9. Conclusion

1864 In this paper, we have presented GRAMMARTRANS-  
1865 FORMER, a tool that supports language engineers in the  
1866 rapid prototyping and evolution of textual domain-specific  
1867 languages which are based on meta-models. GRAMMAR-  
1868 TRANSFORMER uses a number of transformation rules to  
1869 modify a grammar generated by Xtext from a meta-model.  
1870 These transformation rules have been derived from an anal-  
1871 ysis of the difference between the actual and the generated  
1872 grammars of seven DSLs.

1873 We have shown how GRAMMARTRANSFORMER can be  
1874 used to modify grammars generated by Xtext based on  
1875 these transformation rules. This automation is particularly  
1876 useful while a language is being developed to allow for  
1877 rapid prototyping without cumbersome manual configura-  
1878 tion of grammars and when the language evolves. We have  
1879 evaluated GRAMMARTRANSFORMER on seven grammars  
1880 to gauge the feasibility and effort required for defining the  
1881 transformation rules. We have also shown how GRAMMAR-  
1882 TRANSFORMER supports evolution with the examples of

1883 EAST-ADL and QVTo.

1884 Overall, our tool enables language engineers to use a  
1885 meta-model-based language engineering workflow and still  
1886 produce high-quality grammars that are very close in qual-  
1887 ity to hand-crafted ones. We believe that this will reduce  
1888 the development time and effort for domain-specific lan-  
1889 guages and will allow language engineers and users to lever-  
1890 age the advantages of using meta-models, e.g., in terms of  
1891 modifiability and documentation.

1892 In future work, we plan to extend GRAMMARTRANS-  
1893 FORMER into a more full-fledged language workbench that  
1894 supports advanced features like refactoring of meta-models,  
1895 a “what you see is what you get” view of the transforma-  
1896 tion of the grammar, and the ability to co-evolve model  
1897 instances alongside the underlying language. We will also  
1898 explore the integration into workflows that generate graphi-  
1899 cal editors to enable blended modelling.

## 1900 Acknowledgements

1901 This work has been sponsored by Vinnova under grant  
1902 number 2019-02382 as part of the ITEA 4 project *BUM-*  
1903 *BLE*.

## 1904 References

1905 A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: An  
1906 annotated bibliography, *ACM Sigplan Notices* 35 (2000) 26–36.  
1907 Object Management Group, QVT – MOF Query/View/Transforma-  
1908 tion Specification, 2016. URL: <https://www.omg.org/spec/QVT/>,  
1909 Accessed February, 2023.  
1910 Eclipse Foundation, ATL Syntax, 2018. URL: [https://wiki.eclipse.](https://wiki.eclipse.org/M2M/ATL/Syntax)  
1911 [org/M2M/ATL/Syntax](https://wiki.eclipse.org/M2M/ATL/Syntax), Accessed February, 2023.  
1912 Paperpile, A complete guide to the BibTeX format, 2022. URL:  
1913 <https://www.bibtex.com/g/bibtex-format/>, Accessed February,  
1914 2023.  
1915 Graphviz Authors, Dot language, 2022. URL: [https://graphviz.](https://graphviz.org/doc/info/lang.html)  
1916 [org/doc/info/lang.html](https://graphviz.org/doc/info/lang.html), Accessed February, 2023.  
1917 J. Greenyer, Scenario Modeling Language (SML) Repository, 2018.  
1918 URL: [https://bitbucket.org/jgreenyer/scenariotools-sml/](https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/)  
1919 [src/master/](https://bitbucket.org/jgreenyer/scenariotools-sml/src/master/), Accessed February, 2023.  
1920 Spectra Authors, Spectra, 2021. URL: [https://github.com/](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab)  
1921 [SpectraSynthesizer/spectra-lang/blob/master/tau.smlab](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab).

[syntech.Spectra/src/tau/smlab/syntech/Spectra.xtext](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab), Ac- 1922  
1923 cessed February, 2023.

Eclipse Foundation, Eclipse xcore wiki, 2018. URL: [https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/](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) 1924  
1925 [org.eclipse.emf.ecore.xcore/src/org/eclipse/emf/ecore/](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)  
1926 [xcore/Xcore.xtext](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. 1927

Xenia Authors, Xenia xtext, 2019. URL: [https://github.com/](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext) 1928  
1929 [rodchenk/xenia/blob/master/com.foliage.xenia/src/com/](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext)  
1930 [foliage/xenia/Xenia.xtext](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/src/com/foliage/xenia/Xenia.xtext), Accessed February, 2023. 1930

S. Roy Chaudhuri, S. Natarajan, A. Banerjee, V. Choppella, Method- 1931  
1932 ology to develop domain specific modeling languages, in: Pro-  
1933 ceedings of the 17th ACM SIGPLAN International Workshop on  
1934 Domain-Specific Modeling, ACM SIGPLAN, 2019, pp. 1–10. 1934

U. Frank, Domain-specific modeling languages: requirements analysis 1935  
1936 and design guidelines, in: *Domain engineering*, Springer, 2013, pp.  
1937 133–157. 1937

M. Mernik, J. Heering, A. M. Sloane, When and how to develop 1938  
1939 domain-specific languages, *ACM computing surveys (CSUR)* 37  
1940 (2005) 316–344. 1940

L. Bettini, Implementing domain-specific languages with Xtext and 1941  
1942 Xtend, 2 ed., Packt Publishing Ltd, 2016. 1942

D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse 1943  
1944 Modeling Framework, Addison-Wesley Professional, 2008. 1944

F. Ciccozzi, M. Tichy, H. Vangheluwe, D. Weyns, Blended modelling— 1945  
1946 what, why and how, in: 1<sup>st</sup> Intl. Workshop on Multi-Paradigm  
1947 Modelling for Cyber-Physical Systems (MPM4CPS), IEEE, 2019, 1947  
1948 pp. 425–430. doi:10.1109/MODELS-C.2019.00068. 1948

W. Zhang, R. Hebig, D. Strüber, J.-P. Steghöfer, Automated extrac- 1949  
1950 tion of grammar optimization rule configurations for metamodel-  
1951 grammar co-evolution, in: 16th ACM SIGPLAN International  
1952 Conference on Software Language Engineering (SLE’23), 2023. 1952

EAST-ADL Association, East-adl, 2021. URL: [https://www.](https://www.east-adl.info/) 1953  
1954 [east-adl.info/](https://www.east-adl.info/), Accessed February, 2023. 1954

J. Holtmann, J.-P. Steghöfer, W. Zhang, Exploiting meta-model 1955  
1956 structures in the generation of Xtext editors, in: 11th Intl. Conf.  
1957 on Model-Based Software and Systems Engineering (MODELS-  
1958 WARD), 2023, pp. 218–225. doi:10.5220/0011745900003402. 1958

R. F. Paige, D. S. Kolovos, F. A. Polack, A tutorial on metamodelling 1959  
1960 for grammar researchers, *Science of Computer Programming*  
1961 96 (2014) 396–416. doi:10.1016/j.scico.2014.05.007, selected  
1962 Papers from the Fifth Intl. Conf. on Software Language Engineering  
1963 (SLE 2012). 1963

International Organization for Standardization (ISO), Information 1964  
1965 technology—Syntactic metalanguage—Extended BNF (ISO/IEC  
1966 14977:1996), 1996. 1966

A. Kleppe, A language description is more than a metamodel, in: 1967  
1968 4th International Workshop on Language Engineering, 2007. 1968

Object Management Group (OMG), Object constraint language 2.x 1969



- 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.oc1>, 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. Vadera, G. Milosavljević, Ž. Vuković, Textx: A python tool for domain-specific languages implementation, *Knowledge-Based Systems* 115 (2017) 1–4. doi:10.1016/j.knsys.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,

- 2066 2023.
- 2067 AtlanMod Team, Atlantic zoo, 2019. URL: <https://github.com/atlanmod/atlantic-zoo>, Accessed February, 2023. 2114
- 2068
- 2069 A. Nordmann, N. Hochgeschwender, D. Wigand, S. Wrede, An overview of domain-specific languages in robotics, 2020. URL: 2115
- 2070 <https://corlab.github.io/dslzoo/all.html>, Accessed February, 2116
- 2071 2023. 2117
- 2072
- 2073 I. Wikimedia Foundation, Wikipedia page of domain specific language, 2118
- 2074 2023. URL: [https://en.wikipedia.org/wiki/Domain-specific\\_](https://en.wikipedia.org/wiki/Domain-specific_language) 2119
- 2075 [language](https://en.wikipedia.org/wiki/Domain-specific_language), Accessed February, 2023. 2120
- 2076 M. Barash, Zoo of domain-specific languages, 2020. URL: [http://](http://dsl-course.org/) 2121
- 2077 [dsl-course.org/](http://dsl-course.org/), Accessed February, 2023. 2122
- 2078 I. Semantic Designs, Domain specific languages, 2021. URL: [http:](http://www.semdesigns.com/products/DMS/DomainSpecificLanguage.html) 2123
- 2079 [//www.semdesigns.com/products/DMS/DomainSpecificLanguage.](http://www.semdesigns.com/products/DMS/DomainSpecificLanguage.html) 2124
- 2080 [html](http://www.semdesigns.com/products/DMS/DomainSpecificLanguage.html), Accessed February, 2023. 2125
- 2081 D. Community, Financial domain-specific language listing, 2021. URL: 2126
- 2082 <http://dslfin.org/resources.html>, Accessed February, 2023. 2127
- 2083 itemis AG, Dot xtext grammar, 2020. URL: [https:](https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext) 2128
- 2084 [//github.com/eclipse/gef/blob/master/org.eclipse.gef.](https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext) 2129
- 2085 [dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext](https://github.com/eclipse/gef/blob/master/org.eclipse.gef.dot/src/org/eclipse/gef/dot/internal/language/Dot.xtext), 2130
- 2086 Accessed February, 2023. 2131
- 2087 V. Zaytsev, Grammarware bibtex metamodel, 2013. URL: 2132
- 2088 [https://github.com/grammarware/slps/blob/master/topics/](https://github.com/grammarware/slps/blob/master/topics/grammars/bibtex/bibtex-1/BibTeX.ecore) 2133
- 2089 [grammars/bibtex/bibtex-1/BibTeX.ecore](https://github.com/grammarware/slps/blob/master/topics/grammars/bibtex/bibtex-1/BibTeX.ecore), Accessed February, 2134
- 2090 2023. 2135
- 2091 Spectra Authors, Spectra metamodel, 2021. URL: [https:](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/model/generated/Spectra.ecore) 2136
- 2092 [//github.com/SpectraSynthesizer/spectra-lang/blob/master/](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/model/generated/Spectra.ecore) 2137
- 2093 [tau.smlab.syntech.Spectra/model/generated/Spectra.ecore](https://github.com/SpectraSynthesizer/spectra-lang/blob/master/tau.smlab.syntech.Spectra/model/generated/Spectra.ecore), 2138
- 2094 Accessed February, 2023. 2139
- 2095 Eclipse Foundation, Xcore metamodel, 2012. URL: [https:](https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore) 2140
- 2096 [//git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/](https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore) 2141
- 2097 [org.eclipse.emf.ecore.xcore/model/Xcore.ecore](https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.xcore/model/Xcore.ecore), Accessed 2142
- 2098 February, 2023. 2143
- 2099 Xenia Authors, Xenia metamodel, 2019. URL: [https:](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore) 2144
- 2100 [//github.com/rodchenk/xenia/blob/master/com.foliage.](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore) 2145
- 2101 [xenia/model/generated/Xenia.ecore](https://github.com/rodchenk/xenia/blob/master/com.foliage.xenia/model/generated/Xenia.ecore), Accessed February, 2023. 2146
- 2102 W. Zhang, J. Holtmann, D. Strüber, R. Hebig, J.-P. Steghöfer, Gram- 2147
- 2103 martransformer\_data: Formal release, 2024. doi:10.5281/zenodo. 2148
- 2104 10683827, Accessed February 23, 2024. 2149
- 2105 J. E. Hopcroft, On the equivalence and containment problems for 2150
- 2106 context-free languages, *Mathematical systems theory* 3 (1969) 2151
- 2107 119–124. 2152
- 2108 EAST-ADL Association, EATOP Repository, 2022. URL: [https:](https://bitbucket.org/east-adl/east-adl/src/Revision/) 2153
- 2109 [//bitbucket.org/east-adl/east-adl/src/Revision/](https://bitbucket.org/east-adl/east-adl/src/Revision/), Accessed 2154
- 2110 February, 2023. 2155
- 2111 Object Management Group, QVT – MOF Query/View/Transformation 2156
- 2112 Specification Version 1.0, 2008. URL: [https://www.omg.org/](https://www.omg.org/spec/QVT/1.0/) 2157
- 2113 [spec/QVT/1.0/](https://www.omg.org/spec/QVT/1.0/), Accessed February, 2023. 2158