Language Evolution in Practice: The History of GMF

Markus Herrmannsdoerfer[1], Daniel Ratiu[1], and Guido Wachsmuth[2]

[1] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, ratiu}@in.tum.de
[2] Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
guwac@gk-metrik.de

Abstract. In consequence of changing requirements and technological progress, software languages are subject to change. The changes affect the language's specification, which in turn affects language processors as well as existing language utterances. Unfortunately, little is known about how software languages evolve in practice. This paper presents a case study on the evolution of four modeling languages provided by the Graphical Modeling Framework. It investigates the following research questions: (1) What is the impact of language changes on related software artifacts?, (2) What activities are performed to implement language changes? and (3) What kinds of adaptations capture the language changes? We found out that the language changes affect various kinds of related artifacts; the distribution of the activities performed to evolve the languages mirrors the classical software maintenance activities, and most language changes can be captured by a small suite of operators that can also be used to migrate the language utterances.

1 Introduction

Software languages change [1]. A software language, as any other piece of software, is designed, developed, tested, and maintained. Requirements, purpose, and scope of software languages change, and they have to be adapted to these changes. This applies particularly to domain-specific languages that are specialized to a specific problem domain, as their specialization causes them to be vulnerable with respect to changes of the domain. But general-purpose languages like Java or the UML evolve, too. Typically, their evolution is quite slow and driven by heavy-weighted community processes.

Software language evolution implicates a threat for language erosion [2]. Typically, language processors and tools do no longer comply with a changing language. But we do not want to build language processors and tools from scratch every time a language changes. Thus, appropriate co-evolution strategies are required. In a similar way, language utterances like programs or models might become inconsistent with a changing language. But these utterances are valuable assets for language users making their co-evolution a serious issue.

Software language engineering [3, 4] evolves as a discipline to the application of a systematic approach to the design, development, maintenance, and evolution of languages. It concerns various technological spaces [5]. Language evolution affects all these spaces: Grammars evolve in grammarware [6], metamodels evolve in modelware [2], schemas evolve in XMLware [7] and dataware [8], ontologies evolve [9], and APIs evolve [10], too.

In this paper, we focus on the technological space of modelware. There is an ever increasing variety of domain-specific modeling languages each developed by a small group of programmers. These languages evolve frequently to meet the requests of their users. Figure 1 illustrates the status quo: modeling languages come with a series of artifacts (e. g. editors, translators, code generators) centered around a metamodel that defines the language syntax. The ever increasing number of language users (usually decoupled from language developers) build many models by using these languages. As new features need to be incorporated, languages evolve, requiring the co-evolution of existing models.

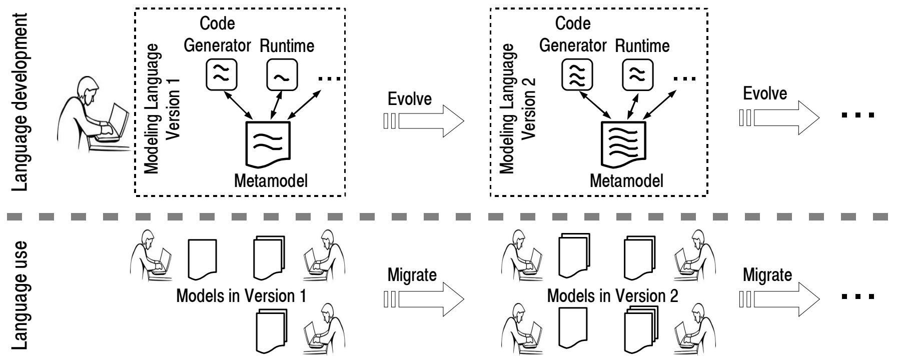


Fig. 1. Development and evolution of modeling languages.

In this paper, we investigate the evolution of modeling languages by reengineering the evolution of their metamodels and the migration of related software artifacts. *Our motivation is to identify requirements for tools that support the (semi-)automatic coupled evolution of modeling languages and related artifacts in a way that avoids the language erosion and minimizes the handwritten code for migration.* As a case study we investigated the evolution of the four modeling languages provided by the Graphical Modeling Framework (GMF). We focus on the following research questions:

– *RQ1) What is the impact of language changes on related software artifacts?* As the metamodel is in the center of the language definition, we are interested to understand how other artifacts change, when the metamodel changes.

– *RQ2) What activities are performed to implement language changes?* We investigate the distribution of the activities performed to implement metamodel changes in order to examine the similarities between the evolution of programs and the evolution of languages.
– *RQ3) What kinds of adaptations capture the language changes?* We are interested to describe the metamodel changes based on a set of canonical adaptations, and thereby to investigate the measure in which these adaptations can be used to migrate the models.

Outline. In Section 2, we introduce the Graphical Modeling Framework as our case study. We present our approach to retrace the evolution of metamodels in Section 3. In Section 4, we answer the research questions from both a quantitative and qualitative point of view. We interpret and discuss the results of the case study in Section 5 by focusing on lessons learned and threats to the study's validity. In Section 6, we present work related to the investigation of language evolution, before we conclude in Section 7.

2 Graphical Modeling Framework

The Graphical Modeling Framework (GMF)[3] is a widely used open source framework for the model-driven development of diagram editors. GMF is a prime example for a Model-Driven Architecture (MDA) [11], as it strictly separates platform-independent models (PIM), platform-specific models (PSM) and code. GMF is implemented on top of the Eclipse Modeling Framework (EMF)[4] and the Graphical Editing Framework (GEF)[5].

2.1 Editor Models

In GMF, a diagram editor is defined by models from which editor code can be generated automatically. For this purpose, GMF provides four modeling languages, a transformator that maps PIMs to PSMs, a code generator that turns PSMs into code, and a runtime platform on which the generated code relies.

The lower part of Fig. 2 illustrates the different kinds of GMF editor models. On the platform-independent level, a diagram editor is modeled from four different views. The domain model focuses on the abstract syntax of diagrams. The graphical definition model defines the graphical elements like nodes and edges in the diagram. The tool definition model defines the tools available to author a diagram. In the mapping model, the first three views are combined to an overall view which maps the graphical elements from the graphical definition model and the tools from the tool definition model onto the domain model elements from the domain model.

[3] see GMF website `http://www.eclipse.org/modeling/gmf`
[4] see EMF website `http://www.eclipse.org/modeling/emf`
[5] see GEF website `http://www.eclipse.org/gef`

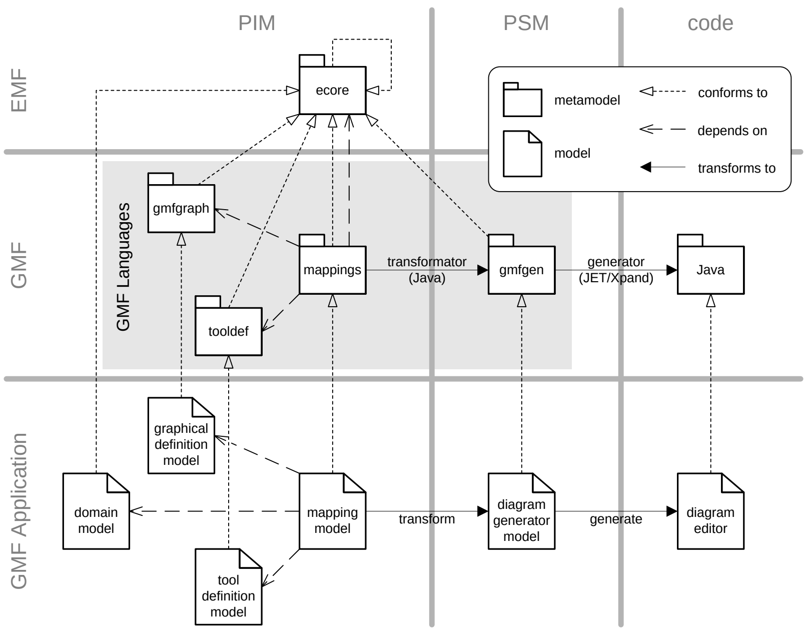


Fig. 2. Languages involved in the Graphical Modeling Framework.

The platform-independent mapping model is transformed into a platform-specific diagram generator model. This model can be altered to customize the code generation.

2.2 Modeling Languages

We can distinguish two kinds of languages involved in GMF. First, GMF provides domain-specific languages for the modeling of diagram editors. Each of these languages comes with a metamodel defining its abstract syntax and a simple tree-based model editor integrated in Eclipse. The upper part of Fig. 2 shows the metamodels involved in GMF. These are ecore for domain models, gmfgraph for graphical definition models, tooldef for tool definition models, mappings for mapping models, and gmfgen for diagram generator models. The mappings metamodel refers to elements in the ecore, gmfgraph, and tooldef metamodels. This kind of dependency is typical for multi-view modeling languages. For example, there are similar dependencies between the metamodel packages defining the various sublanguages of the UML.

Second, GMF itself is implemented in various languages. All metamodels are expressed in ecore, the metamodeling language provided by EMF. EMF is an implementation of Essential MOF which is the basic metamodeling standard proposed by the Object Management Group (OMG) [12]. Notably, the ecore metamodel conforms to itself. Additionally, the metamodels contain context constraints which are attached as textual annotations to the metamodel elements to which they apply. These constraints are expressed in the Object Constraint Language (OCL) [13]. The transformator from a mapping model to a generator model is implemented in Java. For model access, it relies on the APIs generated from the metamodels of the GMF modeling languages. The generator generates code from a generator model. It was formerly implemented in Java Emitter Templates (JET)[6], which was later changed in favor of Xpand[7]. The generated code conforms to the Java programming language, and is based on the GMF runtime platform.

2.3 Metamodel Evolution

With a code base of more than 600k lines of code, GMF is a framework of considerable size. GMF is implemented by 13 developers from 3 different countries using an agile process with small development cycles. Since starting the project, the GMF developers had to adapt the metamodels a significant number of times. As a number of metamodel changes were breaking the existing models, the developers had to manually implement a migrator. Figure 3 quantifies the metamodel evolution for the two release cycles we studied, each taking one year. The figures show the number of metamodel elements for each revision of each GMF metamodel. During the evolution from release 1.0 to release 2.1, the number of classes defined by all metamodels e. g. increased from 201 to 252. We chose GMF as a case study, because the evolution is extensive, publicly available, and well documented by means of commit comments and change requests. However, the evolution is only available in the form of revisions from the version control system, and its documentation is only informal.

3 Investigating the Evolution

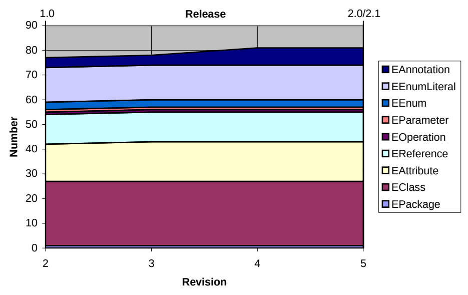
Due to the considerable size of the GMF metamodels, we developed a systematic approach to investigate its evolution as presented in the following subsections.

3.1 Modeling the History

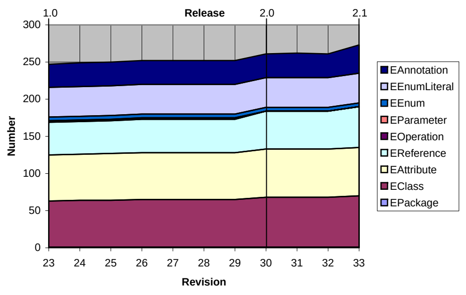
To investigate the evolution of modeling languages, we model the history of their metamodels. In the history model, we capture the evolution of metamodels as sequences of metamodel adaptations [14, 15]. A metamodel adaptation is a well-understood transformation step on metamodels.

We provide a metamodel for history models as depicted in Figure 4. The History of a modeling language is subdivided into a number of releases. A Release denotes a version of the modeling language which has been deployed, and for which models can thus exist. Modeling languages are released at a certain date, and are tagged by a certain version number. A Release is further subdivided into

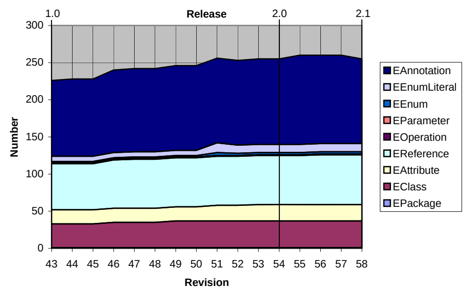
[6] see JET website `http://www.eclipse.org/modeling/m2t`
[7] see Xpand website `http://www.openarchitectureware.org`



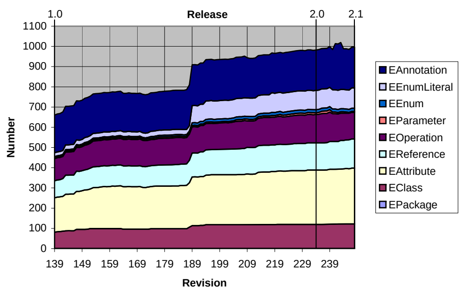
(a) tooldef metamodel.



(b) gmfgraph metamodel.



(c) mappings metamodel.



(d) gmfgen metamodel.

Fig. 3. Statistics of metamodel evolution.

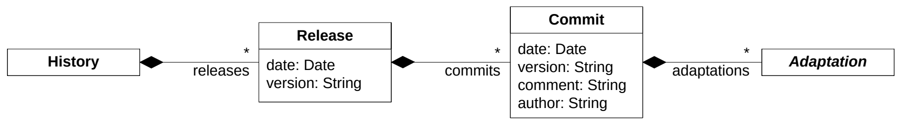


Fig. 4. Modeling language history.

a number of commits. A Commit denotes a version of the modeling language which has been committed to the version control system. Modeling languages are committed at a certain date, by a certain author, with a certain comment, and are tagged by a certain version number. A Commit consists of the sequence of adaptations which have been performed since the last Commit.

3.2 Operator Suite

The metamodel for history models includes an operator suite for stepwise metamodel adaptation. As is depicted in Figure 5, each operator subclasses the abstract class Adaptation. Furthermore, we classify each operator according to four different criteria:

Granularity. Similar to [16], we distinguish primitive and compound operators. A Primitive supports a metamodel adaptation that can not be decomposed into

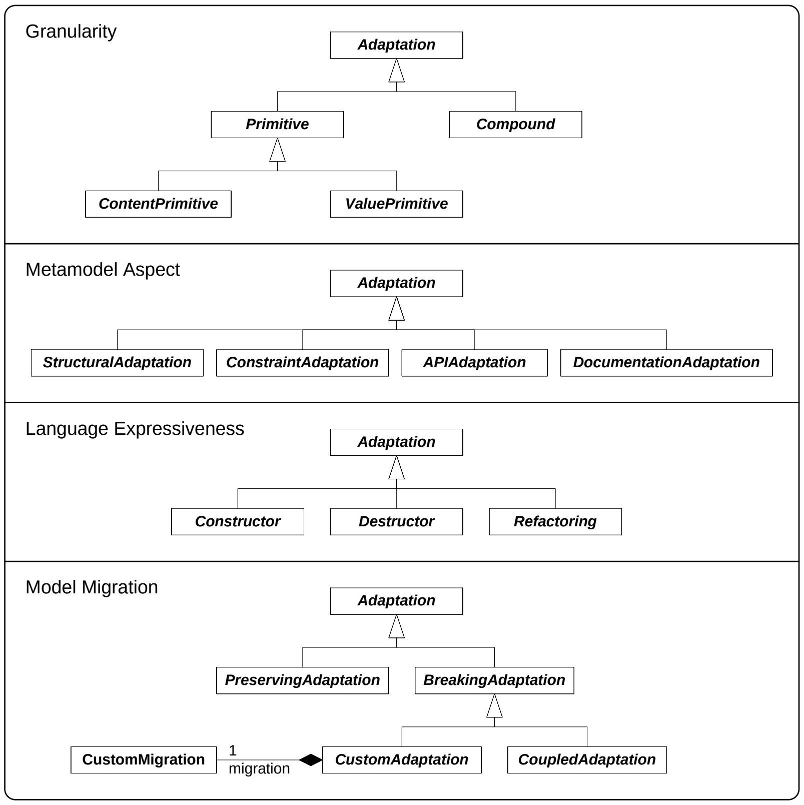


Fig. 5. Classification of operators for metamodel adaptation.

smaller adaptation steps. In contrast, a Compound adaptation can be decomposed into a sequence of Primitives. The required kinds of Primitive operators can be derived from the meta-metamodel. There are two basic kinds of primitive changes: ContentPrimitives and ValuePrimitives. A ContentPrimitive modifies the structure of a metamodel, i. e. creates or deletes a metamodel element. We thus need ContentPrimitives for each kind of metamodel element defined by the meta-metamodel. For classes, e.g., we need ContentPrimitives to create a class in a package and to delete it from its package. A ValuePrimitive modifies an existing metamodel element, i. e. changes a feature of a metamodel element. We thus need ValuePrimitives for each feature defined by the meta-metamodel. For classes, e.g., we need a ValuePrimitive to rename a class, and we need ValuePrimitives to add and remove a superclass. The set of primitive operators already offers a complete operator suite in the sense that every metamodel adaptation can be described by composing them.

Metamodel aspects. We classify an operator according to the metamodel aspect which it addresses. The different classes can be derived from the constructs provided by the meta-metamodel to which the metamodels have to conform. An operator concerns either the structure of models, constraints on models, the API to access models, or the documentation of metamodel elements. A StructuralAdaptation like extracting a superclass affects the abstract syntax defined by the metamodel. A ConstraintAdaptation adds, deletes, moves, or changes constraints in the metamodel. An APIAdaptation concerns the additional access methods defined in the metamodel. This includes volatile features and operations. A DocumentationAdaptation adds, deletes, moves, or changes documentation annotations to metamodel elements.

Language expressiveness. According to [14], we can distinguish three kinds of operators with respect to the expressiveness of the modeling language. By expressiveness of a modeling language, we refer to the set of valid models we can express in the modeling language. Constructors increase this set, i. e. in the new version of the language we can express new models. In contrast, Destructors decrease the set, i. e. in the old version we could express models which we cannot express in the new version of the language. Finally, Refactorings preserve the set of valid models, i. e. we can express all models in the old and the new version of the language.

Model migration. According to [17], we can determine for each operator to what extent model migration can be automated. PreservingAdaptations do not require the migration of models. BreakingAdaptations break the instance relationship between models and the adapted metamodel. In this case, we need to provide a migration for possibly existing models. For a CoupledAdaptation, the migration does not depend on a specific metamodel. Thus it can be specified as a generic couple of metamodel adaptation and model migration. In contrast, a CustomAdaptation is so specific to a certain metamodel that it cannot be composed of generic coupled adaptation steps. Consequently, it can only be covered by a sequence of adaptation steps and a reconciling CustomMigration[8].

As mentioned above, three of the criteria have its origin in existing publications, while metamodel aspects is kind of a natural criterion. There might be other criteria which are interesting in the context of modeling language evolution. Given the sequence of adaptations, it is however easy to classify them according to other criteria.

The presented criteria are orthogonal to each other to a large extent. Granularity is orthogonal to all other criteria and vice versa, as we can think of example operators from each granularity for all these criteria. Additionally, language expressiveness and model migration are orthogonal to each other: the first concerns the difference in cardinality between the sets of valid models before and

[8] The categories from [17] were renamed to be more conforming to the literature: *metamodel-only change* was renamed to PreservingAdaptation, *coupled change* to BreakingAdaptation, *metamodel-independent coupled change* to CoupledAdaptation, and *metamodel-specific coupled change* to CustomAdaptation.

after adaptation, whereas the second concerns the correct migration of a model from one set to the other. However, language expressiveness and model migration both focus on the impact on models, and are thus only orthogonal to the metamodel aspects StructuralAdaptation and ConstraintAdaptation. This is due to the fact that operators concerning APIAdaptation and DocumentationAdaptation do not affect models. Consequently, these operators are always Refactorings and PreservingAdaptations.

The operator suite necessary for our case study is depicted in Figure 9. We classify each operator in the operator suite according to the categories presented before. For example, the operator Extract Superclass creates a new common superclass for a number of classes. This operator is a Compound, since we can express the same metamodel adaptation by the primitive operators Create Class and Add Superclass. The operator is a StructuralAdaptation, since it affects the abstract syntax defined by the metamodel. It is a Constructor, because we can instantiate the introduced superclass in the new language version. Finally, it is a PreservingAdaptation, since no migration of old models to the new language version is required.

3.3 Reverse engineering the GMF history

Procedure. We applied the following steps to reconstruct a history model for GMF based on the available information:
Step 1. *Extracting the log:* We extracted the log information for the whole GMF repository. The log information lists the revisions of each file maintained in the repository.
Step 2. *Detecting the commits:* We grouped revisions of files which were committed together with high probability. Two revisions of different files were grouped, in case they were committed within the same time interval and with the same commit comment.
Step 3. *Filtering the commits:* We filtered out all commits which do not include a revision of one of the metamodels.
Step 4. *Clustering the revisions:* We clustered the files which were committed together into more abstract artifacts like metamodels, transformator, code generator, and migrator. This step was performed to reduce the information, as the implementation of each of the artifacts may be modularized into several files. The information available at this point can be used to answer *RQ1*.
Step 5. *Classifying the commits:* We classified the commits according to the software maintenance categories (i. e. perfective, adaptive, preventive, and corrective) [18] based on the commit comments and change requests. The information available at this point can be used to answer *RQ2*.
Step 6. *Extracting the metamodel revisions:* We extracted the metamodel revisions from the GMF repository.
Step 7. *Comparing the metamodel revisions:* We compared subsequent metamodel revisions with each other resulting in a difference model. The difference model consists of a number of primitive changes between subsequent metamodel revisions.

Step 8. *Detecting the adaptation sequence:* We detected the adaptations necessary to bridge the difference between the metamodel revisions. In contrast to the difference model, the adaptations also combine related primitive changes and are ordered as a sequence. To find the most plausible adaptations, we also analyzed commit comments, change requests, and the co-adaptation of other artifacts. The information available at this point can be used to answer *RQ3*.
Step 9. *Validating the adaptation sequence:* We validated the resulting adaptation sequence by applying it to migrate the existing models for testing the handcrafted migrator. We set up a number of test cases each of which consists of a model before migration and the expected model after migration.

Tool Support. We employed a number of helper tools to perform the study. statCVS[9] was employed to parse the log information into a model which is processed further by a handcrafted model transformation (steps 1-4). The difference models between two subsequent metamodel revisions were generated with the help of EMF Compare[10] (step 7). To bridge the difference between subsequent metamodel revisions, we employed the existing tool COPE[11] [15] whose user interface is depicted in Figure 6 (step 8). COPE allows the language developer to directly execute the operators in the *metamodel editor* and automatically records them in a history model [19]. Generic CoupledAdaptations can be invoked through an *operator browser* which offers all such available operators. To perform a CustomAdaptation, a custom migration needs to be attached to metamodel changes recorded in the metamodel editor. For the study, we extended COPE to support its user in letting the metamodel converge to a *target metamodel* by displaying the *difference model* as obtained from EMF Compare. From the recorded history model, a migrator can be generated which was employed for validating the adaptation sequence (step 9). The handcrafted migrator that comes with GMF was used to generate the expected models for validation.

4 Result

In this section, we present the results of our case study in an aggregated manner. However, the complete history can be obtained from our web site[12].

RQ1) What is the impact of language changes on related software artifacts? To answer this question, we determined for each commit which other artifacts were committed together with the metamodels. Figure 7 shows how many of the overall 124 commits had an impact on a certain artifact.

The first four columns denote the metamodels that were changed in a commit, and the fifth column denotes the number of commits. For instance, row 6 means that the metamodels mappings and gmfgen changed together in 6 commits. The

[9] see statCVS website `http://statcvs.sourceforge.net`
[10] see EMF Compare website `http://www.eclipse.org/emft/projects/compare`
[11] Available as open source at `http://cope.in.tum.de`
[12] Available at `http://cope.in.tum.de/pmwiki.php?n=Documentation.GMF`

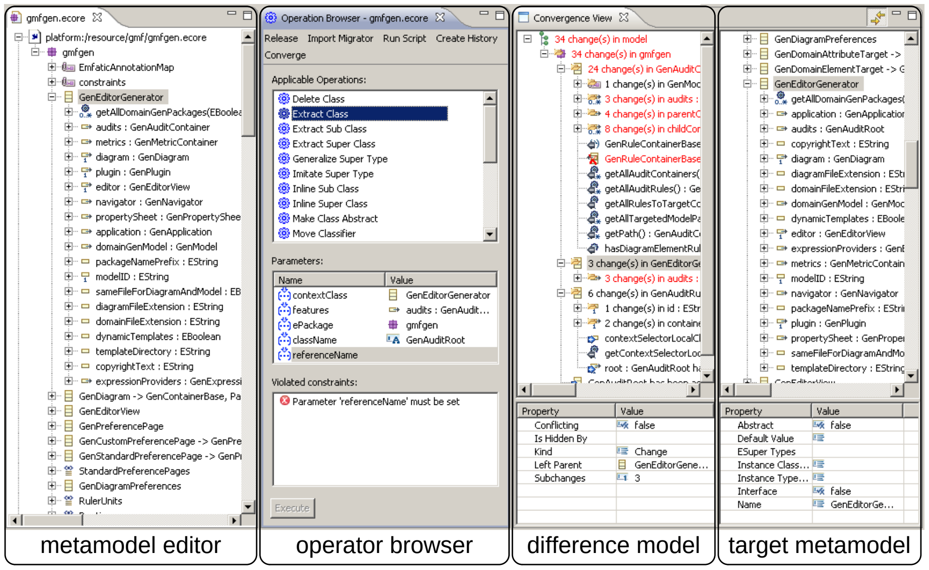


Fig. 6. COPE User Interface.

	Metamodels					Transfor-mator	Genera-tor	Migrator
	gmfgraph	tooldef	mappings	gmfgen	changes			
1	■				7	3	3	
2		■			2	1		
3			■		5	3		2
4				■	100	23	67	9
5		■	■		1			1
6			■	■	6	4	2	1
7	■		■	■	3	1	1	
	10	3	15	109	124	35	73	13

Fig. 7. Correlation between commits of metamodels and related artifacts.

last three columns denote the number of commits in which other artifacts, like transformator, code generator and migrator, were changed. In the example row, the transformator was changed 4 times, the generator 2 times, and the migrator had to be changed once.

In a nutshell, metamodel changes are very likely to impact artifacts which are directly related to them. For instance, the changes to mappings and gmfgen propagated to the transformator from mappings to gmfgen, and to the generator from gmfgen to code. Additionally, metamodel changes are not always carried out on a single metamodel, but are sometimes related to other metamodels.

RQ2) What activities are performed to implement language changes? To answer this question, we classified the commits into the well-known categories of maintenance activities, and we investigated their distribution over these categories. Figure 8 shows the number of commits for each category. Note that several commits could not be uniquely associated to one category, and thus had to be

Perfective	**45**	**Adaptive**	**33**	**Preventive**	**36**	**Corrective**	**16**
Model navigator	13	Transition to Xpand	25	Separation	16	Bug report	7
Rich client platform	6	Ecore constraints	5	Simplification	10	Rename	3
Diagram preferences	4	Namespace URI	2	Unused elements	8	Revert changes	3
Diagram partitioning	2	OCL parser	1	Documentation	2	Wrong constraint	3
Element properties	2						
Individual features	18						

Fig. 8. Classification of metamodel commits according to maintenance categories.

assigned to several categories. However, all commits could be classified into at least one of the four categories.

We classified 45 of the commits as perfective maintenance, i. e. add new features to enhance GMF. Besides a number of individual commits, there are a few features whose introduction spanned several commits. The generated diagram editor was extended with a model navigator, to run as a rich client, to set preferences for diagrams, to partition diagrams, and to set properties of diagram elements. We classified 33 of the commits as adaptive maintenance, i. e. adapt GMF to a changing environment. These commits were either due to the transition from JET to Xpand, adapted to changes to the constraints of ecore, were due to releasing GMF, or adapted the constraints to changes of the OCL parser. We classified 36 of the commits as preventive maintenance, i. e. refactor GMF to prevent faults in the future. These commits either separated concerns to better modularize the generated code, simplified the metamodels to make the transformations more straightforward, removed metamodel elements no longer used by transformations, or added documentation to make the metamodel more understandable. We classified 16 of the commits as corrective maintenance, i. e. correct faults discovered in GMF. These commits either fixed bugs reported by GMF users, corrected incorrectly spelled element names, reverted changes carried out earlier, or corrected invalid OCL constraints.

In a nutshell, the typical activities known from software maintenance also apply to metamodel maintenance [18]. Furthermore, similar to the development of software, the number of perfective activities (34,6%) outranges the preventive (27,7%) and adaptive (25,4%) activities which are double the number of corrective activities (12,3%).

RQ3) What kinds of adaptations capture the language changes? To answer this question, we classified the operators which describe the metamodel evolution. Figure 9 shows the number and classification of each operator occurred during the evolution of each metamodel. The operators are grouped by their granularity and the metamodel aspects to which they apply.

Most of the changes could be covered by Primitive adaptations: we found 379 (51,8%) ContentPrimitive adaptations, 279 (38,2%) ValuePrimitive adaptations and 73 (10,0%) Compound adaptations. Only half of the adaptations affected the structure defined by a metamodel: we identified 361 (49,4%) StructuralAdaptations, 303 (41,5%) APIAdaptations, 36 (4,9%) DocumentationAdaptations, and 31 (4,2%) ConstraintAdaptations. Most of the changes are refactorings which do

Classification		Operator	Classification		Number of Adaptations				
Granularity	Metamodel Aspect		Language Expressiv.	Model Migration	gmf-graph	tool-def	gmf-map	gmf-gen	all
Content-Primitive	Structural	**Create Class**	Constructor	Preserving	4		2	37	**43**
		Create Enum	Constructor	Preserving			3	13	**16**
		Create Optional Attribute	Constructor	Preserving	1	1	3	63	**68**
		Create Optional Reference	Constructor	Preserving	2		3	14	**19**
		Create Required Attribute	Constructor	Custom				1	**1**
		Create Required Reference	Constructor	Custom				1	**1**
		Delete Feature	Destructor	Coupled	4			14	**18**
		New Opposite Reference	Refactoring	Preserving				14	**14**
	Constraint	**Create Constraint Annotation**	Destructor	Preserving			5	3	**8**
		Delete Constraint Annotation	Constructor	Preserving			2	1	**3**
	API	**Create Deprecated Annotation**	Refactoring	Preserving				3	**3**
		Create Setter Visibility Annotation	Refactoring	Preserving	1		5	30	**36**
		Create Volatile Attribute	Refactoring	Preserving	1			4	**5**
		Create Operation	Refactoring	Preserving				53	**53**
		Create Volatile Reference	Refactoring	Preserving				1	**1**
		Delete Setter Visibility Annotation	Refactoring	Preserving	1		5	31	**37**
		Delete Operation	Refactoring	Preserving	1			34	**35**
	Document.	**Create Documentation Annotation**	Refactoring	Preserving	3	3	2	8	**16**
		Delete Documentation Annotation	Refactoring	Preserving				2	**2**
Value-Primitive	Structural	**Abstract Class to Interface**	Refactoring	Preserving			1	2	**3**
		Add Superclass	Constructor	Preserving	10				**10**
		Change Attribute Type	Refactoring	Coupled		1			**1**
		Drop Attribute ID	Constructor	Preserving	1				**1**
		Drop Class Abstract	Constructor	Preserving				1	**1**
		Drop Class Interface	Constructor	Preserving			1	2	**3**
		Drop Reference Opposite	Constructor	Preserving				1	**1**
		Make Class Abstract when Interface	Refactoring	Preserving	14	4	9	22	**49**
		Make Class Interface when Abstract	Refactoring	Preserving				2	**2**
		Make Feature Required	Destructor	Custom	2				**2**
		Make Feature Volatile	Destructor	Coupled			1	5	**6**
		Make Reference Containment	Constructor	Coupled	1				**1**
		Remove Superclass	Destructor	Coupled	10			1	**11**
		Rename Class	Refactoring	Coupled				1	**1**
		Rename Feature	Refactoring	Coupled				10	**10**
		Rename Literal	Refactoring	Coupled				7	**7**
		Set Package Namespace URI	Refactoring	Coupled	1		3	3	**7**
		Specialize Reference Type	Refactoring	Preserving	4				**4**
	Constraint	**Modify Constraint Annotation**	Destructor	Preserving			8	9	**17**
	API	**Rename Volatile Feature**	Refactoring	Preserving				2	**2**
		Rename Operation	Refactoring	Preserving				9	**9**
		Set Feature Changeable	Refactoring	Preserving	2		11	63	**76**
		Set Reference Resolve Proxies	Refactoring	Preserving	1		5	31	**37**
	Document.	**Modify Documentation Annotation**	Refactoring	Preserving	2		7	9	**18**
Compound	Structural	**Collect Feature**	Destructor	Coupled				4	**4**
		Combine Feature	Refactoring	Coupled				1	**1**
		Complex Restructuring	Refactoring	Custom	1			1	**2**
		Extract and Group Attribute	Refactoring	Coupled				1	**1**
		Extract Class	Refactoring	Coupled				1	**1**
		Extract Subclass	Constructor	Coupled			1		**1**
		Extract Superclass	Constructor	Preserving	3		1	5	**9**
		Flatten Containment Hierarchy	Destructor	Coupled				1	**1**
		Generalize Attribute	Constructor	Preserving	1				**1**
		Generalize Reference	Constructor	Preserving	1			5	**6**
		Imitate Superclass	Refactoring	Preserving	1				**1**
		Inline Superclass	Destructor	Coupled	1			2	**3**
		Move Feature	Refactoring	Coupled				3	**3**
		Propagate Feature	Refactoring	Coupled				1	**1**
		Pull up Feature	Constructor	Preserving				3	**3**
		Push down Feature	Destructor	Coupled	4		2	1	**7**
		Replace Class	Destructor	Coupled				2	**2**
		Replace Enum	Destructor	Coupled			2	2	**4**
		Replace Inheritance by Delegation	Refactoring	Coupled			1	2	**3**
		Replace Literal	Destructor	Coupled				1	**1**
		Specialize Superclass	Constructor	Preserving	6				**6**
	Constraint	**Move Constraint Annotation**	Refactoring	Preserving			1	2	**3**
	API	**Move Operation**	Refactoring	Preserving				1	**1**
		Operation to Volatile Feature	Refactoring	Preserving				3	**3**
		Pull up Operation	Refactoring	Preserving				3	**3**
		Push down Operation	Refactoring	Preserving				1	**1**
		Volatile to Opposite Reference	Refactoring	Preserving				1	**1**
					84	**9**	**84**	**554**	**731**

Fig. 9. Classification of operators occurred during metamodel adaptation.

not change the expressiveness of the modeling language: we found 453 (62,0%) Refactorings, 194 (26,5%) Constructors, and 84 (11,5%) Destructors. Only very few changes cannot be covered by generic coupled operators which are able to automatically migrate models: we identified 630 (86,2%) PreservingAdaptations, 95 (13,0%) CoupledAdaptations, and 6 (0,8%) CustomAdaptations. As can be seen in Figure 9, a custom migration was necessary 4 times to initialize a new mandatory feature or a feature that was made mandatory. In these cases, the migration is associated to one Primitive, and consists of 10 to 20 lines of handwritten code. Additionally, 2 custom migrations were necessary to perform a complex restructuring of the model. In these cases, the migration is associated to a sequence of 11 and 13 Primitives, and consists of 60 and 70 lines of handwritten code.

In a nutshell, a large fraction of changes can be captured by primitive changes or operators which are independent of the metamodel. A significant number of operations are known from object-oriented refactoring. Only very few changes were specific to the metamodel, denoting more complex evolution.

5 Discussion

We interpret and discuss the results of the case study by focusing on lessons learned and threats to the study's validity.

5.1 Lessons learned

Based on the results of our case study, we learned a number of lessons about the evolution of modeling languages in practice.

Metamodels evolve due to user requests and technological changes. On the one hand, a metamodel defines the abstract syntax of a language, and thereby metamodels evolve when the requirements of the language change. In GMF, user requests for new features imposed many of such changes to the GMF modeling languages. On the other hand, an API for model access is intimately related to a metamodel, and thereby metamodels evolve when requirements for model access change. In GMF, particularly the shift from JET to XPand as the language to implement the generator imposed many of such changes in the gmfgen metamodel. Since a metamodel captures the abstract syntax as well as the API for model access, language and API evolution interact. Changes in the abstract syntax clearly lead to changes in the API. But API changes can also require to change the abstract syntax of the underlying language: in GMF, we found several cases where the abstract syntax was changed to simplify model access.

Other artifacts need to be migrated. The migration is not restricted to models, but also concerns other language development artifacts, e. g. transformators and code generators. During the evolution of GMF, these artifacts needed to be migrated manually. In contrast to models, these artifacts are mostly under control of the language developers, and thereby their migration is not necessarily required to be automated. However, automating the migration of these artifacts would further reduce the effort involved in language evolution. The model-driven development of metamodels with EMF facilitated the identification of changes between two different versions of the metamodel. In contrast, the specification of transformators and code generators as Java code made it hard to trace the evolution. We thus need a more structured and appropriate means to describe the other artifacts depending on the metamodels. Language development could benefit from the same advantages as model-driven software development.

Language evolution is similar to software evolution. This hypothesis was postulated by Favre in [1]. The answers to *RQ2* and *RQ3* provide evidence that the hypothesis holds. First, the distribution of activities performed by the developers of GMF to implement language changes mirrors the distribution of classical software maintenance activities (i. e. perfective and adaptive maintenance activities being the most frequent) [18]. Second, many operators to adapt the metamodels (Figure 9) are similar to operators known from object-oriented refactoring [20] (e. g. Extract Superclass). Like software evolution, the time scale for language evolution can be quite small. In the first year of the investigated evolution of GMF, the metamodels were changed 107 times, i. e. on average every four days. However, in the second year the number of metamodel changes decreased to 17, i. e. the stability of GMF increased over time. Apparently, the time scale in which the changes happen increases with the language's maturity. The same phenomenon applies to the relation between the metamodels and the meta-metamodel, as the evolution of ecore required the migration of the GMF metamodels. However, the more abstract the level, the less frequent the changes: we identified two changes in the meta-metamodel of the investigated evolution of GMF.

Operator-based coupled evolution of metamodels and models is feasible. The developers of GMF provided a migrator to automatically migrate the already existing models. This migrator allows the GMF developers to make changes that are not backward compatible, and are essential as the kinds and number of built models is not under control of the language developers. We reverse engineered the evolution of the GMF metamodels by sequencing operators. Most of the metamodel evolution can be covered by operators which are independent of the specific metamodel. Only a few custom operators were required to capture the remaining changes. The employed operators can be used to migrate the models as well. In addition, the case study provides evidence for the suitability of operator-based metamodel evolution in forward engineering like proposed in [14, 15]. Operator-based forward engineering of modeling languages documents changes on a high level of abstraction which allows for a better understanding of language evolution.

5.2 Threats to validity

We are aware that our results can be influenced by threats to construct, internal and external validity.

Construct validity. The results might be influenced by the measurement we used for our case study. For our measurements, we assumed that a commit represents exactly one language change. However, a commit might encapsulate several language changes, and one language change might be implemented by several commits. This interpretation is a threat to the results for both *RQ1* and *RQ2*. Other case studies are required to investigate these research questions in more detail, and to increase the confidence and generality of our results. However, our results are consistent with the view that languages evolve like software, which was postulated and tacitly accepted as a fact [1].

Internal validity. The results might be influenced by the method applied for investigating the evolution. The algorithm to detect the commits (step 2) might miss artifacts which were also committed together. To mitigate this threat, we have manually validated the commits by looking into the temporal neighborhood. By filtering out the commits which did not change the metamodel (step 3), we might miss language changes not affecting the metamodel. Such changes might be changes to the language semantics defined by code generators and transformators. However, the model migration defined by the handcrafted migrator could be fully assigned to metamodel adaptations. We might have misclassified some commits, when classifying the commits according to the maintenance categories (step 5). However, the results are in line with the literature on software evolution [18]. When detecting the adaptation sequence (step 8), the picked operators might have a different intention than the developers had when performing the changes. To mitigate this threat, we have automatically validated the model migration by means of test cases. Furthermore, we have manually validated the migration of all artifacts by taking their co-adaptation into account.

External validity. The results might be influenced by the fact that we investigated a single data point. The modeling languages provided by GMF are among the many modeling languages that are developed using EMF. The relevance of our results obtained by analyzing GMF can be affected when analyzing languages developed with other technologies. Our results are however in line with the literature on grammar evolution [21, 6], and this increases our confidence on the fact that the defined operators are valid for many other languages. Furthermore, our past studies on the evolution of metamodels [17, 15] revealed similar results.

6 Related Work

Work related to language evolution can be found in several technological spaces of software language engineering [5]. This includes grammar evolution in grammarware, metamodel evolution in modelware, schema evolution in dataware, and API evolution.

Grammar evolution has been studied in the context of grammar engineering [3]. Lämmel proposes a comprehensive suite of grammar transformation operators for the incremental adaptation of context-free grammars [16]. The proposed operators are based on sound, formal preservation properties that allow reasoning about the relationship between grammars. The operator suite proved to be valuable for semi-automatic recovery of the COBOL grammar from an informal specification [21]. Based on similar operators, Lämmel proposes a lightweight verification method called *grammar convergence* for establishing and maintaining the correspondence between grammars ingrained in different software artifacts [22]. Grammar convergence proved to be useful for establishing the relationship between grammars from different releases of the Java grammar [6]. The approach presented in this paper transfers these ideas to the technological space of modelware. In contrast to the Java case study, the GMF case study provides us with intermediate revisions of the metamodels. Taking these revisions into account allows us to investigate how languages changes are actually implemented.

Metamodel evolution has been mostly studied from the angle of model migration. To specify and automate the migration of models, Sprinkle introduces a visual graph-transformation-based language [23, 24]. However, this language does not provide a mechanism to reuse migration specifications across metamodels. To reuse migration specifications, there are two kinds of approaches: difference-based and operator-based. Difference-based approaches try to automatically derive a model migration from the difference between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and unresolvable changes [25, 26]. Based on this classification, they propose to automatically derive a migration for non-breaking and resolvable changes, and envision to support the developer in specifying a migration for unresolvable changes. Cichetti et al. go even one step further and try to detect compound changes in the difference between metamodel versions [27]. However, Sprinkle et al. claim that in the general case it is undecidable to automatically synthesize a model migration that preserves the semantics of the models [28]. To avoid the loss of intention during evolution, we follow an operator-based approach where the developers can perform the operators encapsulating the intended model migration [14, 15]. The GMF case study continues and extends our earlier studies [17, 15] which focused solely on the automatability of the model migration. Beyond that, the presented study shows that an operator-based approach can be useful in a reverse engineering process to reveal and document the intention of language evolution on a high level of abstraction. Furthermore, it provides evidence that operator-based metamodel adaptation should be used in forward engineering in order to control and document language evolution. In contrast, difference-based approaches still lack a proof of concept by means of real-life case studies both for forward and reverse engineering.

Schema evolution has been a field of study for several decades, yielding a substantial body of research [29, 30]. For the ORION database system, Banerjee et al. propose a fixed set of change primitives that perform coupled evolution of the schema and data [31]. While reusing migration knowledge in case of these primitives, their approach is limited to local schema restructuring. To allow for non-local changes, Ferrandina et al. propose separate languages for schema and instance data migration for the O_2 database system [32]. While more expressive, their approach does not allow for reuse of coupled transformation knowledge. In order to reuse recurring coupled transformations, SERF – as proposed by Claypool et al. – offers a mechanism to define arbitrary new high-level primitives [33], providing both reuse and expressiveness. However, the last two approaches never found their way into practice, as it is difficult to perform complex migration without taking the database offline. As a consequence, it is hard to find real-world case studies which include complex restructuring.

Framework evolution can be automated by refactorings which encapsulate the changes to both the API and its clients [20]. Dig and Johnson present a case study to investigate how object-oriented APIs evolve in practice [10]. They found out that a significant number of API changes can be covered by refactoring operators. In the GMF case study, we found that metamodel evolution is not restricted to the syntax of models, but also includes evolution of APIs to access models. For the migration of client code relying on those APIs, existing work on framework evolution should provide a good starting point.

7 Conclusion

In this paper, we presented a method to investigate the evolution of modeling languages. Our approach is based on retracing the evolution of the metamodel as the central artifact of the language. For this purpose, we provide an operator suite for the stepwise transformation of metamodels from old to new versions. The operators allow us to state clearly the changes made to the language metamodel on a high level of abstraction, and to capture the intention behind the change. Furthermore, these operators can be used to accurately describe the impact of the metamodel changes on related models, and to hint at the possible effects on the related language development artifacts. Thus, we can qualify a certain change with respect to its impact on the other artifacts. This can be in turn used to predict, detect, and prevent language erosion. In the future, the operators could also support the (semi-)automatic migration of co-evolving artifacts other than models.

There is an increasing amount of related work proposing alternative approaches to metamodel evolution and model co-evolution. Real-life case studies are needed to evaluate these approaches. In [17], we presented an industrial case study for operator-based metamodel adaptation. However, the studied evolution is not publicly available due to a non-disclosure agreement. In this paper, we studied the evolution of metamodels in GMF as another extensive case study. GMF's evolution is publicly available through a version control system. The evolution is well-documented in terms of commit comments made by developers, and change requests made by users. Consequently, GMF is a good target to study different approaches to metamodel evolution either on its own (as we did in this paper) or in camparison to each other.

But GMF is not only a case study for metamodel evolution. We consider it as a general case study on software language evolution and the integration of different technological spaces in software language engineering. Not only evolve the modeling languages provided by the framework, but also do APIs. We revealed that a huge amount of GMF metamodel changes were changes to the API for accessing GMF editor models. Further work is needed to investigate the relationships between metamodel evolution and API evolution in frameworks.

Another interesting topic for future work would be a comparison of operator-based approaches in software language engineering. As mentioned in the section on related work, there are many operator-based approaches to software language engineering in different technological spaces, e. g. for grammar evolution, metamodel evolution, schema evolution, and API evolution. It's worth to investigate their common properties, facilities, and restrictions.

Acknowledgement. The work of the first two authors is supported by grants from the BMBF (Federal Ministry of Education and Research, Innovationsallianz SPES 2020), and the work of the third author is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK).

References

1. Favre, J.M.: Languages evolve too! changing the software time scale. In: IWPSE '05: 8th Int. Workshop on Principles of Software Evolution, IEEE (2005) 33–44
2. Favre, J.M.: Meta-model and model co-evolution within the 3D software space. In: ELISA: Workshop on Evolution of Large-scale Industrial Software Applications. (2003) 98–109
3. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3) (2005) 331–380
4. Bézivin, J., Heckel, R.: Guest editorial to the special issue on language engineering for model-driven software development. Software and Systems Modeling **5**(3) (2006) 231–232
5. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track. (2002)
6. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. In: 9th Int. Working Conference on Source Code Analysis and Manipulation, IEEE (2009)
7. Lämmel, R., Lohmann, W.: Format Evolution. In: RETIS '01: 7th Int. Conference on Reverse Engineering for Information Systems. Volume 155 of books@ocg.at., OCG (2001) 113–134
8. Meyer, B.: Schema evolution: Concepts, terminology, and solutions. IEEE Computer **29**(10) (1996) 119–121
9. Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G.: Ontology change: Classification and survey. Knowl. Eng. Rev. **23**(2) (2008) 117–152
10. Dig, D., Johnson, R.: How do apis evolve? a story of refactoring: Research articles. J. Softw. Maint. Evol. **18**(2) (2006) 83–107
11. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
12. Object Management Group: Meta Object Facility, Core Spec., v2.0 (2006)

13. Object Management Group: Object Constraint Language, Spec., v2.0 (2006)
14. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP '07. Volume 4609 of LNCS., Springer (2007) 600–624
15. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP '09. Volume 5653 of LNCS., Springer (2009) 52–76
16. Lämmel, R.: Grammar adaptation. In: FME '01. Volume 2021 of LNCS., Springer (2001) 550–570
17. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: MoDELS '08. Volume 5301 of LNCS., Springer (2008) 645–659
18. Lientz, B.P., Swanson, E.B.: Software Maintenance Management. Addison-Wesley (1980)
19. Herrmannsdoerfer, M.: Operation-based versioning of metamodels with COPE. In: CVSM '09: Int. Workshop on Comparison and Versioning of Software Models, IEEE (2009) 49–54
20. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley (1999)
21. Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. **31**(15) (2001) 1395–1448
22. Lämmel, R., Zaytsev, V.: An introduction to grammar convergence. In: IFM '09. Volume 5423 of LNCS., Springer (2009) 246–260
23. Sprinkle, J.M.: Metamodel driven model migration. PhD thesis, Vanderbilt University, Nashville, TN, USA (2003)
24. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. **15**(3-4) (2004) 291–307
25. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: MSI '07: 1st Workshop MDD, SOA und IT-Management, GiTO-Verlag (2007) 35–46
26. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Int. Workshop on Model-Driven Software Evolution. (2007)
27. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC '08: 12th Int. IEEE Enterprise Distributed Object Computing Conference, IEEE (2008) 222–231
28. Sprinkle, J., Gray, J., Mernik, M.: Fundamental limitations in domain-specific language evolution. `http://www.ece.arizona.edu/~sprinkjm/wiki/uploads/Publications/sprinkle-tse2009-domainevolution-submitted.pdf` (2009)
29. Li, X.: A survey of schema evolution in object-oriented databases. In: TOOLS '99: 31st Int. Conference on Technology of Object-Oriented Language and Systems, IEEE (1999) 362
30. Rahm, E., Bernstein, P.A.: An online bibliography on schema evolution. SIGMOD Rec. **35**(4) (2006) 30–31
31. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: SIGMOD '87: ACM SIGMOD Int. conference on Management of data, ACM (1987) 311–322
32. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O2 object database system. In: VLDB '95: 21th Int. Conference on Very Large Data Bases, Morgan Kaufmann (1995) 170–181
33. Claypool, K.T., Jin, J., Rundensteiner, E.A.: SERF: schema evolution through an extensible, re-usable and flexible framework. In: CIKM '98: 7th Int. Conference on Information and knowledge management, ACM (1998) 314–321