Metamodel Usage Analysis for Identifying Metamodel Improvements

Markus Herrmannsdoerfer, Daniel Ratiu, and Maximilian Koegel

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, ratiu, koegel}@in.tum.de

Abstract. Modeling languages raise the abstraction level at which software is built by providing a set of constructs tailored to the needs of their users. Metamodels define their constructs and thereby reflect the expectations of the language developers about the use of the language. In practice, language users often do not use the constructs provided by a metamodel as expected by language developers. In this paper, we advocate that insights about how constructs are used can offer language developers useful information for improving the metamodel. We define a set of usage and improvement patterns to characterize the use of the metamodel by the built models. We present our experience with the analysis of the usage of seven metamodels (EMF, GMF, UNICASE) and a large corpus of models. Our empirical investigation shows that we identify mismatches between the expected and actual use of a language that are useful for metamodel improvements.

1 Introduction

Modeling languages promise to increase productivity of software development by offering their users a restricted set of language constructs that directly address their needs. Many times, modeling languages address a well-defined category of users, since these languages are domain-specific, represent technology niches, or are built in-house. Thereby, the needs of the users are the most important driving force for the existence and evolution of the modeling language. Usable languages have a small set of intuitive constructs, are easy to use, hard to misuse, and easy to learn by novices [8]. The constructs of modeling languages are typically defined by metamodels which reflect the expectations of the language developers about the future use of the language.

The question whether these expectations are fulfilled and whether the language users use the language in the same manner as expected has a central importance for language developers [9, 15]. Too general modeling languages that have a too wide scope hamper the productivity of their users, are more difficult to learn, and are prone to errors. To answer this question, we need direct feedback from the language users. Without feedback about the actual use, the developers can only guess whether the language meets the expectation of its users. The traditional means by which developers get feedback about their languages (e. g.

community forums), are limited to certain kinds of information like e. g. mostly bug reports and usage questions.

In the case when the language developers have access to a relevant set of models built with the language, they can take advantage of this information and learn about how the language is used by investigating its utterances. Once the information about the actual use of the language is available, the language can be improved along two main directions: firstly, restricting the language and eliminating unnecessary constructs, and secondly, by adding new language constructs that language users need. In this paper, we advocate that the analysis of built models can reflect essential information about the use of the language that is otherwise inaccessible. By using our method, we aim to close the loop between language developers and users, since the information about the actual use can serve for language enhancements or even for the elimination of defects.

This paper contains three main contributions to the state of the art:

1. We present a general method by which language developers can obtain feedback about how the modeling language is actually used by analyzing the models conforming to its metamodel.
2. We describe a set of patterns that characterize the usage of the metamodel and which can be used to improve the language by eliminating obsolete or superfluous constructs, adding missing metamodel constraints that were not set by language developers, or by adding new language constructs that better reflect the needs of the users.
3. We present our experience with applying these analyses to seven metamodels, and we show that even in the case of well-known metamodels like EMF we can identify different defects that give rise to metamodel improvements.

Outline. Sec. 2 describes the metamodeling formalism that we considered for defining our analyses. In Sec. 3, we introduce our method as well as a set of usage analyses to identify metamodel improvements. Sec. 4 presents the results of applying the analyses on a corpus of well-known metamodels and their models. In Sec. 5, we discuss related work, before we conclude the paper in Sec. 6.

2 Metamodeling Formalism

Metamodel. To describe metamodels, we use the simplified E-MOF [14] formalism illustrated in Fig. 1 as a class diagram. A metamodel defines a set of *Type*s which can be either primitive (*PrimitiveType*) or complex (*Class*). Primitive types are either *DataType*s—like Boolean, Integer and String—or *Enumeration*s of *Literals*. Classes consist of a set of *features*. They can have *super types* to inherit features and might be *abstract*. The *name* of a feature needs to be unique among all features of a class including the inherited ones. A *Feature* has a multiplicity (*lower bound* and *upper bound*) and maybe *derived*—i. e. its value is derived from the values of other features. A feature is either an *Attribute* or a *Reference*: an attribute is a feature with a primitive *type*, whereas a reference is a

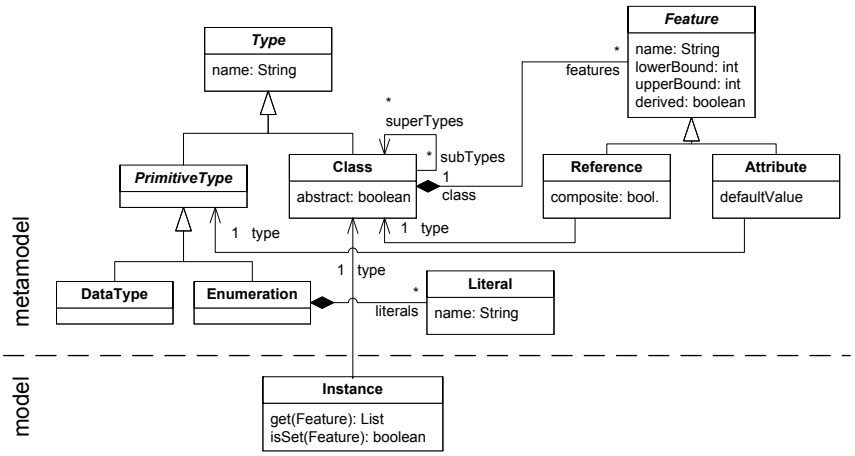

Fig. 1. Metamodeling formalism

feature with a complex *type*. An attribute might also define a *default value* that serves as an initial value for instances. A reference might be *composite*, meaning that an instance can only have one parent via a composite reference.

The classes in Fig. 1 can be interpreted as sets—e. g. *Class* denotes the set of classes defined by a metamodel, and the references can be interpreted as navigation functions on the metamodel—e. g. $c.subTypes$ returns the sub types of $c \in Class$. Additionally, * denotes the transitive closure on navigation functions—e. g. $c.subTypes^*$ returns all sub types of $c \in Class$ including c itself, and $PV(t)$ denotes the possible values of a primitive type $t \in PrimitiveType$.

Model. A model consists of a set of *Instance*s each of which have a class from the metamodel as *type*. To define our analyses in Sec. 3.2, we require an instance to provide two methods. First, the method $i.get(f)$ returns the value of a feature $f \in Feature$ for a certain instance $i \in Instance$. The value is returned as a list even in the case of single-valued features to simplify the formulas. Second, the method $i.isSet(f)$ returns true if the value of a feature $f \in Feature$ is set for a certain instance $i \in Instance$. A feature is set if and only if the value of the feature is different from the empty list and different from the *default value*, in case the feature is an attribute.

3 Metamodel Usage Analysis

In this section, we present our approach to identify metamodel improvements by analyzing how the metamodel is used by the built models. Sec. 3.1 introduces templates to define usage analyses. Sec. 3.2 lists a number of analyses defined using these templates. Sec. 3.3 presents the implementation of the approach.

3.1 Templates for defining Usage Analyses

Our ultimate goal is to recommend metamodel changes to improve the usability of the language. The metamodel can be changed by applying refactorings, constructors, or destructors [20]. *Refactorings* restructure the metamodel, but do not change the expressiveness of the language defined by the metamodel. By contrast, *destructors* and *constructors* decrease or increase the expressiveness of the language, respectively. By analyzing only the metamodel, we can recommend only refactorings of the metamodel. If we also take the models built with a metamodel into account, we can also recommend destructors and constructors. Destructors can be identified by finding metamodel constructs that are not used in models, and that can thereby be safely removed without affecting the existing models. Constructors can be identified by finding metamodel constructs that are used to encode constructs currently not available in the language. By enriching the metamodel with the needed constructs, we support the users to employ the language in a more direct manner.

Collecting usage data. Before we can identify metamodel improvements, we need to collect usage data from the models built with the metamodel. This usage data collection has to fulfill three requirements: (1) We need to collect data from a significant number of models built with the metamodel. In the best case, we should collect usage data from every built model. If this is not possible, we should analyze a significant number of models to be sure that the results of our analyses are relevant and can be generalized for the actual use of the language. Generally, the higher the ratio of the existing models that are analyzed, the more relevant our analyses. (2) We need to collect the appropriate amount of data necessary for identifying metamodel improvements. If we collect too much data, we might violate the intellectual property of the owners of the analyzed models. If we collect too few data, we might not be able to extract meaningful information from the usage data. (3) The usage data from individual models needs to be collected in a way that it can be composed without losing information.

To specify the collection of usage data, we employ the following template:

Context: the kind of metamodel element for which the usage data is collected. The context can be used as a pattern to apply the usage data collection to metamodel elements of the kind.

Confidence: the number of model elements from which the usage data is collected. The higher this number, the more confidence we can have in the collected data. In the following, we say that we are not confident if this number is zero, i. e. we do not have usage data that can be analyzed.

Specification: a function to specify how the usage data is collected. There may be different result types for the data that is collected. In the following, we use numbers and functions that map elements to numbers.

Analyzing usage data. To identify metamodel improvements, we need to analyze the usage data collected from the models. The analysis is based on

an expectation that we have for the usage data. If the expectation about the usage of a metamodel construct is not fulfilled, the construct is a candidate for improvement. To specify expectations and the identification of metamodel improvements from these expectations, we employ the following template:

Expectation: a boolean formula to specify the expectation that the usage data needs to fulfill. If the formula evaluates to true, then the expectation is fulfilled, otherwise we can propose an improvement. Certain expectations can be automatically derived from the metamodel, e. g. we expect that a non-abstract class is used in models. Other expectations can only be defined manually by the developer of the metamodel, e. g. that certain classes are more often used than other classes. In the following, we focus mostly on expectations that can be automatically derived from the metamodel, as they can be applied to any metamodel without additional information.

Improvement: the metamodel changes that can be recommended if the expectation is not fulfilled. The improvement is specified as operations that can be applied to the metamodel. As described above, the improvements can restrict the language by removing existing constructs, or enlarge the language by adding new constructs. If we have collected data from all models built with a metamodel, then we can also be sure that the restrictions can be safely applied, i. e. without breaking the existing models.

3.2 Towards a Catalog of Usage Analyses

In this section, we present a catalog of analyses of the usage of metamodels. Each subsection presents a category of analyses, each analysis being essentially a question about how the metamodel is actually used. We use the templates defined in the previous section to define the analyses in a uniform manner. This catalog of analyses is by no means complete, but rather represents a set of basic analyses. We only define analyses that are used in Sec. 4 as part of the study.

Class Usage Analysis. If metamodels are seen as basis for the definition of the syntax of languages, a non-abstract class represents a construct of the language. Thereby, the measure in which a language construct is used can be investigated by analyzing the number of instances $CU(c)$ of the non-abstract class c defined by the metamodel:

Context: $c \in Class$, $\neg c.abstract$
Confidence: $\|Instance\|$
Specification: $CU(c) := \|\{i \in Instance | i.type = c\}\|$

Q1) Which classes are not used? We expect that the number of instances for a non-abstract class is greater than zero. Classes with no instance represent language constructs that are not needed in practice, or the fact that language users did not know or understand how to use these constructs:

Expectation: $CU(c) > 0$
Improvement: delete the class, make the class abstract

Classes with no instances that have subclasses can be made abstract, otherwise, classes without subclasses might be superfluous and thereby are candidates to be deleted from the metamodel. Both metamodel changes reduce the number of constructs available to the user, making the language easier to use and learn. Furthermore, deleting a class results in a smaller metamodel implementation which is easier to maintain by the language developers. Non-abstract classes that the developers forgot to make abstract can be seen as metamodel bugs.

Q2) What are the most widely used classes? We expect that the more central a class of the metamodel is, the higher the number of its instances. If a class is more widely used than we expect, this might hint at a misuse of the class by the users or the need for additional constructs:

Expectation: the more central the construct, the higher its use frequency
Improvement: the classes that are used more frequently than expected are potential sources for language extensions

This analysis can only be performed manually, since the expectation cannot be derived automatically from the metamodel in a straightforward manner.

Feature Usage Analysis. If metamodels are seen as basis for the definition of a language, features are typically used to define how the constructs of a modeling language can be combined (references) and parameterized (attributes). As derived features cannot be set by users, we investigate only the use of non-derived features:

Context: $f \in Feature, \neg f.derived$
Confidence: $\|FI(f)\|$, $FI(f) := \{i \in Instance | i.type \in f.class.subTypes^*\}$
Specification: $FU(f) := \|\{i \in FI(f) | i.isSet(f)\}\|$

We can only be confident for the cases when there exist instances $FI(f)$ of classes in which the feature f could possibly be set, i. e. in all sub classes of the class in which the feature is defined.

Q3) Which features are not used? We expect that the number of times a non-derived feature is set is greater than zero. Otherwise, we can make it derived or even delete it from the metamodel:

Expectation: $FU(f) > 0$
Improvement: delete the feature, make the feature derived

If we delete a feature from the metamodel or make it derived, it can no longer be set by the users, thus simplifying the usage of the modeling language. Features that are not derived but need to be made derived can be seen as a bug in the metamodel, since the value set by the language user is ignored by the language interpreters.

Feature Multiplicity Analysis. Multiplicities are typically used to define how many constructs can be referred to from another construct. Again, we are only interested in non-derived features, and we can only be confident for a feature, in case there are instances in which the feature could possibly be set:

Context: $f \in Feature, \neg f.derived$
Confidence: $\|FU(f)\|$
Specification: $FM(f) : \mathbb{N} \to \mathbb{N}$, $FM(f,n) := \|\{i \in FU(f) | \|i.get(f)\| = n\}\|$

Q4) Which features are not used to their full multiplicity? We would expect that the distribution of used multiplicities covers the possible multiplicities of a feature. More specifically, we are interested in the following two cases: First, if the lower bound of the feature is 0, there should be instances with no value for the feature—otherwise, we might be able to increase the lower bound:

Expectation: $f.lowerBound = 0 \Rightarrow FM(f,0) > 0$
Improvement: increase the lower bound

A lower bound greater than 0 explicitly states that the feature should be set, thus avoiding possible errors when using the metamodel. Second, if the upper bound of the feature is greater 1, there should be instances with more than one value for the feature – otherwise, we might be able to decrease the upper bound:

Expectation: $f.upperBound > 1 \Rightarrow \max_{n \in \mathbb{N}} FM(f,n) > 1$
Improvement: decrease the upper bound

Decreasing the upper bound reduces the number of possible combinations of constructs and thereby simplifies the usage of the language.

Attribute Value Analysis. The type of an attribute defines the values that an instance can use. The measure in which the possible values are covered can be investigated by determining how often a certain value is used. Again, we are only interested in non-derived attributes, and we can only be confident for an attribute, if there are instances in which the attribute could possibly be set:

Context: $a \in Attribute$, $\neg a.derived$
Confidence: $\|FI(a)\|$
Specification: $AVU(a) : PV(a.type) \to \mathbb{N}$,
$AVU(a,v) := \|\{i \in FI(a) | v \in i.get(a)\}\|$

Q5) Which attributes are not used in terms of their values? We expect that all the possible values of an attribute are used. In case of attributes that have a finite number of possible values (e. g. Boolean, Enumeration), we require them to be all used. In case of attributes with a (practically) infinite domain (e. g. Integer, String), we require that more than 10 different values are used. Otherwise, we might be able to specialize the type of the attribute:

Expectation: $(\|PV(a.type)\| < \infty \Rightarrow \|PV(a.type)\| = \|VU(a)\|) \ \wedge$
$(\|PV(a.type)\| = \infty \Rightarrow \|VU(a)\| > 10)$,
where $VU(a) = \{v \in PV(a.type) | AVU(a, v) > 0\}$
Improvement: specialize the attribute type

More restricted attributes can give users better guidance about how to fill its values in models, thus increasing usability. Additionally, such attributes are easier to implement for developers, since the implementation has to cover less cases.

Q6) Which attributes do not have the most used value as default value? In many cases, the language developers set as default value of attributes the values that they think are most often used. In these cases, the value that is actually most widely used should be set as default value of the attribute:

Expectation: $a.upperBound = 1 \Rightarrow a.defaultValue = [mvu \in PV(a.type) : \max_{v \in PV(a.type)} AVU(a, v) = AVU(a, mvu)]$
Improvement: change the default value

If this is not the case, and users use other values, the new ones can be set as default.

Q7) Which attributes have often used values? We expect that no attribute value is used too often. Otherwise, we might be able to make the value a first-class construct of the metamodel, e. g. create a subclass for the value. A value is used too often if its usage share is at least 10%:

Expectation: $\|PV(a.type)\| = \infty \Rightarrow \forall v \in PV(a.type) : AVU(a, v) < 10\% \cdot FU(a)$
Improvement: lift the value to a first-class construct

Lifting the value to an explicit construct, makes the construct easier to use for users as well as easier to implement for developers.

3.3 Prototypical Implementation

We have implemented the approach based on the Eclipse Modeling Framework (EMF) [18] which is one of the most widely used modeling frameworks. The collecting of usage data is implemented as a batch tool that traverses all the model elements. The results are stored in a model that conforms to a simple metamodel for usage data. The batch tool could be easily integrated into the modeling tool itself and automatically send the data to a server where it can be accessed by the language developers. Since the usage data is required to be composeable, it can be easily aggregated.

The usage data can be loaded into the metamodel editor which proposes improvements based on the usage data. The expectations are implemented as constraints that can access the usage data. Fig. 2 shows how violations of these

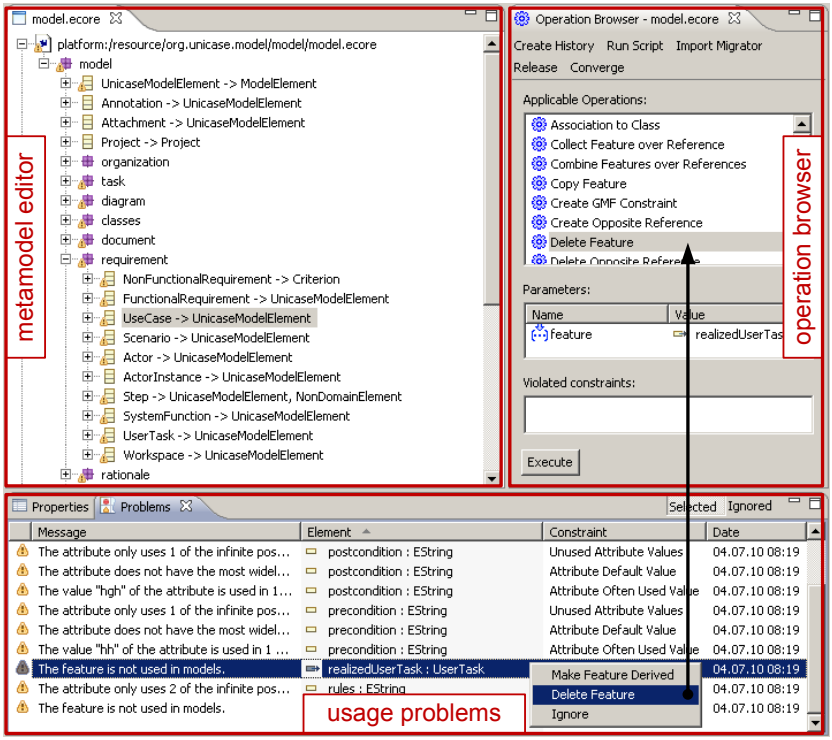

Fig. 2. Proposing metamodel improvements

expectations are presented to the user in the *metamodel editor*. Overlay icons indicate the metamodel elements to which the violations apply, and a view provides a list of all *usage problems*. The constraints have been extended to be able to propose operations for metamodel improvements. The operations are implemented using our tool COPE [7] whose purpose is to automate the model migration in response to metamodel evolution. The proposed operations are shown in the context menu and can be executed via COPE's *operation browser*.

4 Empirical Study

We have performed an empirical study to validate whether the identified metamodel improvements would really lead to changes of the metamodel. Sec. 4.1 presents the method that we have applied, and Sec. 4.2 the metamodels and models that we have analyzed. The results of the empirical study are explained in Sec. 4.3 and discussed in Sec. 4.4.

4.1 Study Method

To perform our analyses, we performed the following steps:

1. *Mine models*: We obtained as many models as possible that conform to a certain metamodel. In the case of an in-house metamodel, we asked the metamodel developers to provide us with the models known to them. For the published metamodels, we iterated through several open repositories (CVS and SVN) and downloaded all models conforming to these metamodels. As far as possible, we removed duplicated models.
2. *Perform usage analysis*: We applied the approach presented in Sec. 3 on the mined models. For each of the analyzed metamodels, this results in a set of usage problems.
3. *Interpret usage problems*: To determine whether the usage problems really help us to identify possible metamodel improvements, we tried to find explanations for the problems. In order to do so, we investigated the documentation of the metamodels as well as the interpreters of the modeling languages, and, if possible, we interviewed the metamodel developers.

4.2 Study Objects

Metamodels. To perform our experiments, we have chosen 7 metamodels whose usage we have analyzed. Table 1 shows the number of elements of these metamodels. Two metamodels are part of the Eclipse Modeling Framework (EMF)[1] which is used to develop the abstract syntax of a modeling language: The ecore metamodel defines the abstract syntax from which an API for model access and a structural editor can be generated; and the genmodel allows to customize the code generation. Four metamodels are part of the Graphical Modeling Framework (GMF)[2] which can be used to develop the diagrammatic, concrete syntax of a modeling language: the graphdef model defines the graphical elements like nodes and edges in the diagram; the tooldef model defines the tools available to author a diagram; the mappings model maps the nodes and edges from the graphdef model and the tools from the tooldef model onto the metamodel elements from the ecore model; and the mappings model is transformed into a gmfgen model which can be altered to customize the generation of a diagram editor. Finally, the last metamodel (unicase) is part of the tool UNICASE[3] which can be used for UML modeling, project planning and change management.

Table 1. A quantitative overview over the analyzed metamodels

#	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
Class	20	14	72	26	36	137	77
Attribute	33	110	78	16	22	302	88
Reference	48	34	57	12	68	160	161

[1] see EMF web site: `http://www.eclipse.org/emf`
[2] see GMF web site: `http://www.eclipse.org/gmf`
[3] see UNICASE web site: `http://unicase.org`

Table 2. A quantitative overview over the analyzed models

repository	ecore		genmodel		graphdef		tooldef		mappings		gmfgen	
	files	elem.s	files	elem.s	files	elem.s	files	elem.s	files	elem.s	files	elem.s
AUTOSAR	18	384,685	18	16,189	11	1,835	11	436	11	538	13	2,373
Eclipse	1,834	250,107	818	69,361	105	6,026	58	1,769	72	5,040	52	11,043
GForge	106	26,736	94	41,997	12	806	10	241	11	480	11	1,680
Google	50	9,266	59	3,786	69	7,627	74	2,421	76	4,028	78	18,710
Atlantic Zoo	278	68,116	–	–	–	–	–	–	–	–	–	–
altogether	2,286	738,910	989	131,333	197	16,294	153	4,867	170	10,086	154	33,806

Models. For each metamodel, we have mined models from different repositories. Table 2 shows the repositories as well as the number of files and elements which have been obtained from them. Models that conform to the first 6 metamodels have been obtained from the AUTOSAR development partnership[4], from the Eclipse[5] and GForge[6] open source repositories, by querying the Google Code Search[7] and from the Atlantic Zoo[8]. The Atlantic Zoo only contains ecore models, while the other repositories contain models for all EMF and GMF metamodels. For the unicase metamodel, its developers provided us with 3 files consisting of 8,213 model elements.

4.3 Study Results

In this section, we present the study results separately for each question mentioned in Sec. 3.2. To facilitate understanding the explanations for the identified usage problems, we clustered them according to high-level explanations.

Q1) Which classes are not used? Table 3 quantitatively illustrates for each metamodel the number of used and not used classes in the overall number of non-abstract classes. The second last row shows the number of classes that are used, whereas the other rows classify the unused classes according to the explanations why they are not used. As presented in Sec. 4.1, we derived these explanations by manually analyzing the documentation and implementation of the metamodels or interviewing the developers.

Classes that are obsolete, not implemented, or that logically belong to another metamodel can be removed. A class *is obsolete* if it is not intended to be used in the future. For example, in unicase, the 4 classes to define UML stereotypes are no longer required. A class *is not implemented* if it is not used by the interpreters of the modeling language. For example, the tooldef metamodel defines 7 classes to specify menus and toolbars from which, according to [4],

[4] see AUTOSAR web site: `http://www.autosar.org`
[5] see Eclipse Repository web site: `http://dev.eclipse.org/viewcvs/index.cgi`
[6] see GForge web site: `http://gforge.enseeiht.fr`
[7] see Google Code Search web site: `http://www.google.com/codesearch`
[8] see Atlantic Zoo web site: `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`

Table 3. Usage of classes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
is obsolete							4
is not implemented				7			1
should be moved				1			2
should be abstract	1						
should be transient	1					2	1
is too new			3			6	5
should be used			1		1	6	9
is used	13	11	51	11	24	83	42
altogether	15	11	55	19	25	97	64

Table 4. Usage of ecore classes

#	Class	Number	Share
1	EStringToStringMapEntry	328,920	44.51%
2	EGenericType	141,043	19.09%
3	EAnnotation	102,863	13.92%
4	EReference	53,177	7.20%
5	EClass	41,506	5.62%
6	EAttribute	36,357	4.92%
7	EEnumLiteral	10,643	1.44%
8	EOperation	7,158	0.97%
9	EParameter	7,060	0.96%
10	EPackage	4,530	0.61%
11	EDataType	2,747	0.37%
12	EEnum	2,513	0.34%
13	ETypeParameter	226	0.03%
14	EObject	0	0.00%
15	EFactory	0	0.00%

currently no code can be generated. A class *should be moved* to another metamodel if it logically belongs to the other metamodel. For example, in tooldef, the class GenericStyleSelector should be moved to mappings which contains also a composite reference targeting this class. Another example is unicase where 2 classes should be moved to a different metamodel of another organization that also uses the framework underlying UNICASE.

Our manual investigations revealed that other classes that are not used should be abstract or transient. A class *should be abstract* if it is not intended to be instantiated, and is used only to define common features inherited by its subclasses. For instance, in ecore, EObject—the implicit common superclass of all classes—should be made abstract. A class *should be transient* if its instances are not expected to be made persistent—such a class does not represent a language construct. However, the employed metamodeling formalism currently does not support to specify that a class is transient. For instance, in ecore, EFactory—a helper class to create instances of the specified metamodel—should be transient.

Finally, there are non-used classes which do not require any change, since they are either *too new* or *should be used*—i. e. we could not find any plausible explanation why they are not used. A class is *too new* if it was recently added and thus is not yet instantiated in models. For the GMF metamodels graphdef and gmfgen, we identified 9 new classes, while for the unicase metamodel, we found 5 classes that are too new to be used.

Q2) What are the most widely used classes in ecore? Due to space constraints, we focus on the ecore metamodel to answer this question. Table 4 shows its classes and their corresponding number of instances. Interestingly, the number of instances of the ecore classes has an exponential distribution, a phaenomenon observed also in case of the other metamodels. Hence, each metamodel contains a few constructs which are very central to its users. In the case of ecore, we expect that classes, references and attributes are the central constructs. However, the most widely used ecore class is—with more than 44,5% of the analyzed instances—EStringToStringMapEntry which defines key-value-pairs for EAnnotations making up 13,9% of the instances. The fact that the annotation mechanism

Table 5. Usage of features

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
is obsolete						9	11
is not implemented		3		2	2	5	
should be moved							11
should be derived							2
is too new			6			16	6
should be used	1	7	2			55	13
is not confident			6	6	1	23	35
is used	51	125	118	20	84	333	163
altogether	52	135	132	28	87	441	241

Table 6. Usage of multiplicity by features

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should increase ↩ the lower bound		2	1	2	1	21	5
is not implemented				3	1		
should have a ↩ derived default		5	1			48	3
is too new			5				2
should be used		5	4		4	12	3
is not confident			6	6	1	23	35
is used	52	123	115	17	80	337	193
altogether	52	135	132	28	87	441	241

which is used to extend the ecore metamodel is more widely used than the first-class constructs, suggests the need for increasing ecore's expressiveness. As we show in Q7, some often encountered annotations could be lifted to first-class constructs. Another ecore class that is used very often is EGenericType. This is surprising, since we would expect that generic types are very rarely used in metamodels. The investigation of how this class is exactly used, revealed the fact that only 1,8% (2,476) of EGenericType's instances do not define default values for their features and thereby these instances really represent generic types. In the other 98%, the EGenericType is only used as an indirection to the non-generic type, i. e. in a degenerated manner.

Q3) Which features are not used? Table 5 shows for each metamodel the number of used and unused features in the overall number of non-derived features. In the table, we observe a correlation between the number of unused features and the overall number of features. In most cases, the more features a metamodel defines, the less features are actually used. The only exception to this conjecture is the usage of the tooldef metamodel, in which most features are however not implemented as explained below. The table also classifies the unused features according to explanations why they are not used.

Features that are obsolete, not implemented, or logically belong to another metamodel can be removed from the metamodel. A feature *is obsolete* if it is not intended to be used in the future. If none of the analyzed models uses that feature, it is a good candidate to be removed. We identified the 9 obsolete features of gmfgen by investigating the available migrator. Surprisingly, the migrator removes their values, but the developers forgot to remove the features from gmfgen. In the case of the unicase metamodel, 11 unused features are actually obsolete. A feature is classified as *not implemented* if it is not used by the interpreters of the modeling language. We have identified 12 not implemented features of the EMF metamodel genmodel and the GMF metamodels tooldef, mappings and gmfgen by checking whether they are accessed by the code generators. A feature *should be moved* to another metamodel if it logically belongs to the other metamodel.

The features of the unicase metamodel that have to be moved target the classes that should be moved to another metamodel as found by question Q1 in Sec. 3.2.

Another set of non-used features can be changed into derived features, since their values are calculated based on other features. As they are anyway overwritten in the metamodel implementation, setting them explicitly is a mistake, making the language easy to misuse. For example, for the unicase metamodel, we identified 2 features that *should be derived*.

Finally, there are non-used features which do not require changes at all, since they are either too new or our investigation could not identify why the feature is not used. Again, a feature *is too new* to be instantiated, if it was recently added. Like for classes, the 28 too new features were identified by investigating the metamodel histories. The only feature of the ecore metamodel which is not set in any model but *should be used* allows to define generic exceptions for operations. Apparently, exceptions rarely need to have type parameters defined or assigned. For both genmodel and gmfgen, more than 5% of the features *should be used* but are not used. The manual investigation revealed that these are customizations of the code generation that are not used at all. We are *not confident* about the usage of a feature if there are no instances in which the feature could be set. This category thus indicates how many features cannot be used, because the classes in which they are defined are not instantiated.

Q4) Which features are not used to their full multiplicity? Table 6 shows as *is used* the number of used features fulfilling these expectations. Again, the violations are classified according to different explanations that we derived after manual investigation of the metamodel documentation and implementation.

A not completely used feature which can be restricted either *should increase the lower bound* or *is not implemented* yet. Even though we found features whose usage did not completely use the upper bound, we could not find an explanation for any of them. However, we found that some features *should increase lower bound* from 0 to 1. For the EMF and GMF metamodels, we found such features by analyzing whether the code generator does not check whether the feature is set, thereby producing an error if the feature is not set. For the unicase metamodel, the too low lower bounds date back from the days when its developers used an object-to-relational mapping to store data. When doing so, a lower bound of 1 was transformed into a database constraint which required to set the feature already when creating a model element. To avoid this restriction, the lower bounds were decreased to 0, when in effect, they should have been 1. As they no longer store the models in a database, the lower bounds could easily be increased. Again, a feature *is not implemented* if the interpreter does not use the feature. In the tooldef metamodel, we found 3 such features which are not interpreted by the code generator, making them also superfluous.

A not completely used feature which requires to extend the metamodeling formalism *should have a derived default*. A feature *should have a derived default* if it has lower bound 0, but in case it is not set, a default value is derived. This technique is mostly used by the code generation metamodels genmodel and

Table 7. Usage of complete values by attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be specialized		1				3	3
is too new				1		3	
should be used	1	14	14		3	18	4
is not confident		12	44	7	4	164	37
is used	25	82	19	8	14	102	39
altogether	26	109	77	16	21	290	83

Table 8. Usage of default values of attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be changed		2	5			3	
should be set		4	13	1		6	
should not be changed	2	1	1		1		2
should have no default	6	16	7	1	3	38	9
is too new				2			
is not confident		10	19	7	4	93	37
is used	18	76	32	5	13	150	35
altogether	26	109	77	16	21	290	83

gmfgen to be able to customize a value which is otherwise derived from other features. It is implemented by overwriting the API to access the models which is generated from the metamodel. However, the used metamodeling formalism does not provide a means to specify that a feature has a derived default.

A not completely used feature which does not require changes either *is too new* or *should be used*.

Q5) Which attributes are not used in terms of their values? Table 7 illustrates for each metamodel the number of attributes whose values are completely used or not. The table also classifies the not completely used attributes according to different explanations.

A not completely used attribute that can be changed *should be specialized* by restricting its type. In the unicase metamodel, three attributes which use String as domain for UML association multiplicities and UML attribute types can be specialized. We found 4 more such attributes in the genmodel and gmfgen metamodels.

Finally, there are not completely used attributes which do not require changes at all, since they are either too new, should be used, or we do not have enough models to be confident about the result. We are only confident if the attribute is set sufficiently often to cover all its values (finite) or 10 values (infinite). In all metamodels, most of the findings fall into one of these categories.

Q6) Which attributes do not have the most used value as default value? Table 8 illustrates for each metamodel the number of attributes whose value is the same as the default value (*is used*) and those that have different values. Note that in many cases the language developers successfully anticipated the most often used values of attributes by setting the appropriate default value. The table also classifies the deviations according to different explanations.

In case the default value is intended to represent the most widely used value of an attribute, and we found that the users use other default values, the default value *should be changed*. In this way, the new value better anticipates the actual use, and thereby the effort of language users to change the attribute value is

Table 9. Usage of values often used by attributes

Explanation	ecore	genmodel	graphdef	tooldef	mappings	gmfgen	unicase
should be lifted	3	1	3			3	2
should be reused		1					4
should not be changed	6	18	22	1	3	41	3
is not implemented		1					
is too new				1			
is not confident		11	19	7		93	37
is used	17	77	33	7	18	153	37
altogether	26	109	77	16	21	290	83

Table 10. Most widely used annotations in ecore

#	Source	Number	Share
1	http://www.eclipse.org/emf/↩ 2002/GenModel	33,056	32.15%
2	http:///org/eclipse/emf/↩ ecore/util/ExtendedMetaData	26,039	25.32%
3	TaggedValues	16,304	15.86%
4	MetaData	10,169	9.89%
5	Stereotype	4,628	4.50%
6	subsets	1,724	1.68%
	...		

necessary in less cases. We found 8 attributes whose default value needs to be updated in the metamodels genmodel, graphdef and gmfgen. An attribute has a default value that *should be set* if it does not currently have a default value, but the usage analysis identifies a recurrent use of certain values. By setting a meaningful default value, the language users are helped. In nearly each metamodel, we found attributes whose default value needs to be set.

The unexpected default value of an attribute which does not require change either *should have no default*, *should not be changed*, *is too new* or *is not confident*. An attribute *should have no default* value if we cannot define a constant default value for the attribute. In each metamodel, we are able to find such attributes which are usually of type String or Integer. A default value *should not be changed* if we could not find a plausible explanation for setting or changing the default value. Two attributes from the unicase metamodel whose default value should not be changed denote whether a job is done. Most of the attribute values are true—denoting a completed job, but in the beginning the job should not be done. We are *not confident* about an attribute if it is not set at least 10 times.

Q7) Which attributes have often used values? Table 9 shows the number of attributes which have often used values or not. The table also classifies the attribute with recurring values according to explanations.

A recurring value which requires metamodel changes either *should be lifted* or *should be reused*. A value *should be lifted* if the value should be represented by a new class in the metamodel. In ecore, we find often used values for the source of an annotation as well as for the key of an annotation entry. Table 10 illustrates the 6 most widely used values of the annotation source. The GenModel annotations customize the code generation, even though there is a separate generator model. Thereby, some of these annotations can be lifted to first-class constructs of the genmodel metamodel. The ExtendedMetaData extends ecore to accommodate additional constructs of XML Schemas which can be imported. This shows the lack of expressiveness of ecore in comparison to XML Schema. The next three sources represent stereotypes and result from the import of metamodels from UML class diagrams. The high number of these cases are evidence that many ecore models originate from UML class diagrams. The last source extends

ecore which is an implementation of E-MOF with a subset relation between features which is only available in C-MOF. This shows that for many use cases the expressiveness of E-MOF is not enough. Furthermore, the key of an annotation entry is used in 10% of the cases to specify the documentation of a metamodel element. However, it would be better to make the documentation an explicit attribute of EModelElement—the base class for each metamodel element in ecore. A value *should be reused* if—instead of recurrently using the value—we refer to a single definition of the value as instance of a new class. In unicase, instead of defining types of UML attributes as Strings, it would be better to group them into separate instances of reusable data types.

An attribute with recurring values which does not require change either *should not be changed*, *is not implemented*, *is too new* or we are *not confident*. For a lot of attributes, we have not found a plausible explanation, and thus conservatively assumed that they *should not be changed*.

4.4 Discussion

Lessons learned. Based on the results of our analyses, we learned a number of lessons about the usage of metamodels by existing models.

Metamodels can be more restrictive. In all investigated cases, the set of models covers only a subset of the set of all possible models that can be built with a metamodel. We discovered that not all classes are instantiated (Q1), not all features are used (Q3), and the range of the cardinality of many features does not reflect their definition from the metamodel (Q4). In all these cases, the metamodels can be improved by restricting the number of admissible models.

Metamodels can contain latent defects. During our experiments, we discovered in several cases defects of the metamodels—e. g. classes that should not be instantiated and are not declared abstract (Q1), classes and features that are not implemented or that are overwritten by the generator (Q1, Q3, Q4, Q6). Moreover, in other cases (Q2), we discovered misuses of metamodel constructs.

Metamodels can be extended. Each of the analyzed metamodels offers their users the possibility to add new information (e. g. as annotations, or sets of key-value pairs). The analysis of the manners in which the metamodels are actually extended reveal that the users recurrently used several annotation patterns. These patterns reveal the need to extend the metamodel with new explicit constructs that capture their intended use (Q2, Q7). Moreover, the specification of some attributes can be extended with the definition of default values (Q6).

Metamodeling formalism can be improved. Our metamodeling formalism is very similar to ecore. The results indicate that in certain cases the metamodeling formalism is not expressive enough to capture certain constraints. For instance, we would have required to mark a class as transient (Q1) and to state that a feature has a default value derived from other features (Q4). Consequently, we have also identified improvements concerning the metamodeling formalism.

Limitations. We are aware of the following limitations concerning our results.

Validity of explanations. We presented a set of explanations for the deviations between expectation and usage. In the case of UNICASE, we had direct access to the language developers and hence could ask them directly about their explanations for the usage problems. In the case of the other metamodels, we interpreted the analysis results based only on the documentation and implementation. Consequently, some of our explanations can be mistaken.

Relevance and number of analyzed models. We analyzed only a subset of the entire number of existing models. This fact can make our results rather questionable. In the case of UNICASE, we asked the developers to provide us with a representative sample of existing models. In the case of the other metamodels, we mined as many models as possible from both public and private (AUTOSAR) repositories to obtain a representative sample of existing models.

5 Related Work

Investigating language utterances. In the case of general-purpose languages, investigating their use is especially demanding, as a huge number of programs exist. There are some landmark works that investigate the use of general-purpose languages [3, 17, 11]. Gil et al. present a catalog of Java micro patterns which capture common programming practice on the class-level [3]. By automatically searching these patterns in a number of projects, they show that 75% of the classes are modeled after these patterns. Singer et al. present a catalog of Java nano patterns which capture common programming practice on the method-level [17]. There is also work on investigating the usage of domain-specific languages. Lämmel et al. analyze the usage of XML schemas by applying a number of metrics to a large corpus [10]. Lämmel and Pek present their experience with analyzing the usage of the W3C's P3P language [11]. Our empirical results—that many language constructs are more often used than others—are consistent with all these results. We use a similar method for investigating the language utterances, but our work is focused more on identifying improvements for languages. Tolvanen proposes a similar approach for the metamodeling formalism provided by MetaCase [19]. However, even if the sets of analyses are overlapping, our set contains analyses not addressed by Tolvanen's approach (Q2, Q5 and Q7), and provides a validation through a large-scale empirical study that usage analyses do help to identify metamodel improvements.

Language improvements. Once the information about the language use is available, it can serve for language improvements. Atkinson and Kuehne present techniques for language improvements like restricting the language to the used subset or adding new constructs that reflect directly the needs of the language users [1]. Sen et al. present an algorithm that prunes a metamodel [16]: it takes a metamodel as input as well as a subset of its classes and features, and outputs a restricted metamodel that contains only the desired classes and features. By doing this, we obtain a restricted metamodel that contains only necessary constructs. Our work on mining the metamodel usage can serve as input for

the pruning algorithm that would generate a language more appropriate to the expectations of its users. Once recurrent patterns are identified, they can serve as source for new language constructs [2]. The results in this paper demonstrate that the analysis of built programs is a feasible way to identify language improvements. Lange et al. show—by performing an experiment—that modeling conventions have a positive impact on the quality of UML models [12]. Henderson-Sellers and Gonzalez-Perez report on different uses and misuses of the stereotype mechanism provided by UML [6]. By analyzing the built models, we might be able to identify certain types of conventions as well as misuses of language extension mechanisms.

Tool usage. There is work on analyzing the language use by recording and analyzing the interactions with the language interpreter. Li et al. [13] present a study on the usage of the Alloy analyzer tool. From the way how the analyzer tool was used, they identified a number of techniques to improve analysis performance. Hage and Keeken [5] present the framework Neon to analyze usage of a programming environment. Neon records data about every compile performed by a programmer and provides a query to analyze the data. To evaluate Neon, the authors executed analyses showing how student programmers improve over time in using the language. The purpose of our approach is not to improve the tools for using the language, but to improve the language itself.

6 Conclusions and Future Work

This paper is part of a more general research direction that we investigate, namely how to analyze the use of modeling languages in order to assess their quality. The focus of this paper is to derive possible improvements of the metamodel by analyzing the language use. We are convinced that the analysis of models built with a modeling language is interesting for any language developer. We showed that—even in the case of mature languages—the analysis of models can reveal issues with the modeling language. Due to the promising results, we plan to further improve the approach presented in this paper. First, we intend to refine the presented analyses by fine-tuning them according to the results from the empirical study. Second, we plan to add new analyses which are currently missing in the catalog. Third, we intend to apply the presented approach as part of a method for the evolutionary development of modeling languages.

For languages used by different organizations, the analysis of models is often impossible due to the intellectual property that the models carry. We envision as a future direction of research the definition of techniques that anonymize the content of the model and send the language developers only a limited information needed for analyzing the language use.

Acknowledgments. This work was partly funded by the German Federal Ministry of Education and Research (BMBF), grants “SPES2020, 01IS08045A” and “Quamoco, 01IS08023B”.

References

1. Atkinson, C., Kühne, T.: A tour of language customization concepts. Advances in Computers 70, 105–161 (2007)
2. Bosch, J.: Design patterns as language constructs. JOOP - Journal of Object-Oriented Programming 11(2), 18–32 (1998)
3. Gil, J., Maman, I.: Micro patterns in Java code. In: OOPSLA '05: Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 97–116. ACM (2005)
4. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley (2009)
5. Hage, J., van Keeken, P.: Neon: A library for language usage analysis. In: SLE '08. LNCS, vol. 5452, pp. 35–53. Springer (2008)
6. Henderson-Sellers, B., Gonzalez-Perez, C.: Uses and abuses of the stereotype mechanism in UML 1.x and 2.0. In: MoDELS '06. LNCS, vol. 4199, pp. 16–26. Springer (2006)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: ECOOP '09. LNCS, vol. 5653, pp. 52–76. Springer (2009)
8. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkl, S.: Design guidelines for domain specific languages. In: The 9th OOPSLA Workshop on Domain-Specific Modeling (2009)
9. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Software 26(4), 22–29 (2009)
10. Lämmel, R., Kitsis, S., Remy, D.: Analysis of XML schema usage. In: Conference Proceedings XML 2005 (Nov 2005)
11. Lämmel, R., Pek, E.: Vivisection of a non-executable, domain-specific language; understanding (the usage of) the P3P language. In: ICPC '10: 18th International Conference on Program Comprehension. IEEE (2010)
12. Lange, C.F.J., DuBois, B., Chaudron, M.R.V., Demeyer, S.: An experimental investigation of UML modeling conventions. In: MoDELS '06. LNCS, vol. 4199, pp. 27–41. Springer (2006)
13. Li, X., Shannon, D., Walker, J., Khurshid, S., Marinov, D.: Analyzing the uses of a software modeling tool. In: LDTA '06: 6th Workshop on Language Descriptions, Tools, and Applications. pp. 3 – 18. Elsevier (2006)
14. Object Management Group: Meta Object Facility (MOF) core specification version 2.0. `http://www.omg.org/spec/MOF/2.0/` (2006)
15. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for modeling language design. Information and Software Technology 42(10), 665 – 675 (2000)
16. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model pruning. In: MODELS '09. pp. 32–46. No. 5795 in LNCS, Springer (2009)
17. Singer, J., Brown, G., Lujan, M., Pocock, A., Yiapanis, P.: Fundamental nanopatterns to characterize and classify java methods. In: LDTA '09: 9th Workshop on Language Descriptions, Tools and Applications. pp. 204–218 (2009)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley (2009)
19. Tolvanen, J.P.: Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. Ph.D. thesis, University of Jyvskyl (1998)
20. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP '07. LNCS, vol. 4609, pp. 600–624 (2007)